Lecture Slides for Programming in C++ [The C++ Language, Libraries, Tools, and Other Topics]

(Version: 2018-02-15)

Current with the C++17 Standard

Michael D. Adams

Department of Electrical and Computer Engineering
University of Victoria
Victoria, British Columbia, Canada

University of Victoria

For additional information and resources related to these lecture slides (including errata and ***lecture videos*** covering the material on many of these slides), please visit:
`http://www.ece.uvic.ca/~mdadams/cppbook`

If you like these lecture slides, ***please show your support*** by posting a review of them on Google Play:
`https://play.google.com/store/search?q=Michael%20D%20Adams%20C%2B%2B&c=books`

Published by the University of Victoria, Victoria, British Columbia, Canada

This document was typeset with LaTeX.

ISBN 978-1-55058-624-4 (print)
ISBN 978-1-55058-625-1 (PDF)

```
   broadcasts, or other works or subject matter other than works listed
   in Section 1(f) below, which, by reason of the selection and
   arrangement of their contents, constitute intellectual creations, in
   which the Work is included in its entirety in unmodified form along
   with one or more other contributions, each constituting separate and
   independent works in themselves, which together are assembled into a
   collective whole. A work that constitutes a Collection will not be
   considered an Adaptation (as defined above) for the purposes of this
   License.
c. "Distribute" means to make available to the public the original and
   copies of the Work through sale or other transfer of ownership.
d. "Licensor" means the individual, individuals, entity or entities that
   offer(s) the Work under the terms of this License.
e. "Original Author" means, in the case of a literary or artistic work,
   the individual, individuals, entity or entities who created the Work
   or if no individual or entity can be identified, the publisher; and in
   addition (i) in the case of a performance the actors, singers,
   musicians, dancers, and other persons who act, sing, deliver, declaim,
   play in, interpret or otherwise perform literary or artistic works or
   expressions of folklore; (ii) in the case of a phonogram the producer
   being the person or legal entity who first fixes the sounds of a
   performance or other sounds; and, (iii) in the case of broadcasts, the
   organization that transmits the broadcast.
f. "Work" means the literary and/or artistic work offered under the terms
   of this License including without limitation any production in the
   literary, scientific and artistic domain, whatever may be the mode or
   form of its expression including digital form, such as a book,
   pamphlet and other writing; a lecture, address, sermon or other work
   of the same nature; a dramatic or dramatico-musical work; a
   choreographic work or entertainment in dumb show; a musical
   composition with or without words; a cinematographic work to which are
   assimilated works expressed by a process analogous to cinematography;
   a work of drawing, painting, architecture, sculpture, engraving or
   lithography; a photographic work to which are assimilated works
   expressed by a process analogous to photography; a work of applied
   art; an illustration, map, plan, sketch or three-dimensional work
   relative to geography, topography, architecture or science; a
   performance; a broadcast; a phonogram; a compilation of data to the
   extent it is protected as a copyright work; or a work performed by
   a variety or circus performer to the extent it is not otherwise
   considered a literary or artistic work.
```

License III

```
 g. "You" means an individual or entity exercising rights under this
    License who has not previously violated the terms of this License with
    respect to the Work, or who has received express permission from the
    Licensor to exercise rights under this License despite a previous
    violation.
 h. "Publicly Perform" means to perform public recitations of the Work and
    to communicate to the public those public recitations, by any means or
    process, including by wire or wireless means or public digital
    performances; to make available to the public Works in such a way that
    members of the public may access these Works from a place and at a
    place individually chosen by them; to perform the Work to the public
    by any means or process and the communication to the public of the
    performances of the Work, including by public digital performance; to
    broadcast and rebroadcast the Work by any means including signs,
    sounds or images.
 i. "Reproduce" means to make copies of the Work by any means including
    without limitation by sound or visual recordings and the right of
    fixation and reproducing fixations of the Work, including storage of a
    protected performance or phonogram in digital form or other electronic
    medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce,
limit, or restrict any uses free from copyright or rights arising from
limitations or exceptions that are provided for in connection with the
copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License,
Licensor hereby grants You a worldwide, royalty-free, non-exclusive,
perpetual (for the duration of the applicable copyright) license to
exercise the rights in the Work as stated below:

 a. to Reproduce the Work, to incorporate the Work into one or more
    Collections, and to Reproduce the Work as incorporated in the
    Collections; and,
 b. to Distribute and Publicly Perform the Work including as incorporated
    in Collections.

The above rights may be exercised in all media and formats whether now
known or hereafter devised. The above rights include the right to make
such modifications as are technically necessary to exercise the rights in
other media and formats, but otherwise you have no rights to make
```

```
   attribution ("Attribution Parties") in Licensor's copyright notice,
   terms of service or by other reasonable means, the name of such party
   or parties; (ii) the title of the Work if supplied; (iii) to the
   extent reasonably practicable, the URI, if any, that Licensor
   specifies to be associated with the Work, unless such URI does not
   refer to the copyright notice or licensing information for the Work.
   The credit required by this Section 4(c) may be implemented in any
   reasonable manner; provided, however, that in the case of a
   Collection, at a minimum such credit will appear, if a credit for all
   contributing authors of Collection appears, then as part of these
   credits and in a manner at least as prominent as the credits for the
   other contributing authors. For the avoidance of doubt, You may only
   use the credit required by this Section for the purpose of attribution
   in the manner set out above and, by exercising Your rights under this
   License, You may not implicitly or explicitly assert or imply any
   connection with, sponsorship or endorsement by the Original Author,
   Licensor and/or Attribution Parties, as appropriate, of You or Your
   use of the Work, without the separate, express prior written
   permission of the Original Author, Licensor and/or Attribution
   Parties.
d. For the avoidance of doubt:

    i. Non-waivable Compulsory License Schemes. In those jurisdictions in
       which the right to collect royalties through any statutory or
       compulsory licensing scheme cannot be waived, the Licensor
       reserves the exclusive right to collect such royalties for any
       exercise by You of the rights granted under this License;
   ii. Waivable Compulsory License Schemes. In those jurisdictions in
       which the right to collect royalties through any statutory or
       compulsory licensing scheme can be waived, the Licensor reserves
       the exclusive right to collect such royalties for any exercise by
       You of the rights granted under this License if Your exercise of
       such rights is for a purpose or use which is otherwise than
       noncommercial as permitted under Section 4(b) and otherwise waives
       the right to collect royalties through any statutory or compulsory
       licensing scheme; and,
  iii. Voluntary License Schemes. The Licensor reserves the right to
       collect royalties, whether individually or, in the event that the
       Licensor is a member of a collecting society that administers
       voluntary licensing schemes, via that society, from any exercise
       by You of the rights granted under this License that is for a
```

```
      purpose or use which is otherwise than noncommercial as permitted
      under Section 4(b).
 e. Except as otherwise agreed in writing by the Licensor or as may be
    otherwise permitted by applicable law, if You Reproduce, Distribute or
    Publicly Perform the Work either by itself or as part of any
    Collections, You must not distort, mutilate, modify or take other
    derogatory action in relation to the Work which would be prejudicial
    to the Original Author's honor or reputation.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR
OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY
KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE,
INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY,
FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF
LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS,
WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION
OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE
LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR
ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES
ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS
BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

 a. This License and the rights granted hereunder will terminate
    automatically upon any breach by You of the terms of this License.
    Individuals or entities who have received Collections from You under
    this License, however, will not have their licenses terminated
    provided such individuals or entities remain in full compliance with
    those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any
    termination of this License.
 b. Subject to the above terms and conditions, the license granted here is
    perpetual (for the duration of the applicable copyright in the Work).
    Notwithstanding the above, Licensor reserves the right to release the
    Work under different license terms or to stop distributing the Work at
    any time; provided, however that any such election will not serve to
    withdraw this License (or any other license that has been, or is
```

```
   required to be, granted under the terms of this License), and this
   License will continue in full force and effect unless terminated as
   stated above.

8. Miscellaneous

 a. Each time You Distribute or Publicly Perform the Work or a Collection,
    the Licensor offers to the recipient a license to the Work on the same
    terms and conditions as the license granted to You under this License.
 b. If any provision of this License is invalid or unenforceable under
    applicable law, it shall not affect the validity or enforceability of
    the remainder of the terms of this License, and without further action
    by the parties to this agreement, such provision shall be reformed to
    the minimum extent necessary to make such provision valid and
    enforceable.
 c. No term or provision of this License shall be deemed waived and no
    breach consented to unless such waiver or consent shall be in writing
    and signed by the party to be charged with such waiver or consent.
 d. This License constitutes the entire agreement between the parties with
    respect to the Work licensed here. There are no understandings,
    agreements or representations with respect to the Work not specified
    here. Licensor shall not be bound by any additional provisions that
    may appear in any communication from You. This License may not be
    modified without the mutual written agreement of the Licensor and You.
 e. The rights granted under, and the subject matter referenced, in this
    License were drafted utilizing the terminology of the Berne Convention
    for the Protection of Literary and Artistic Works (as amended on
    September 28, 1979), the Rome Convention of 1961, the WIPO Copyright
    Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996
    and the Universal Copyright Convention (as revised on July 24, 1971).
    These rights and subject matter take effect in the relevant
    jurisdiction in which the License terms are sought to be enforced
    according to the corresponding provisions of the implementation of
    those treaty provisions in the applicable national law. If the
    standard suite of rights granted under applicable copyright law
    includes additional rights not granted under this License, such
    additional rights are deemed to be included in the License; this
    License is not intended to restrict the license of any rights under
    applicable law.
```

License VIII

```
Creative Commons Notice

    Creative Commons is not a party to this License, and makes no warranty
    whatsoever in connection with the Work. Creative Commons will not be
    liable to You or any party on any legal theory for any damages
    whatsoever, including without limitation any general, special,
    incidental or consequential damages arising in connection to this
    license. Notwithstanding the foregoing two (2) sentences, if Creative
    Commons has expressly identified itself as the Licensor hereunder, it
    shall have all rights and obligations of Licensor.

    Except for the limited purpose of indicating to the public that the
    Work is licensed under the CCPL, Creative Commons does not authorize
    the use by either party of the trademark "Creative Commons" or any
    related trademark or logo of Creative Commons without the prior
    written consent of Creative Commons. Any permitted use will be in
    compliance with Creative Commons' then-current trademark usage
    guidelines, as may be published on its website or otherwise made
    available upon request from time to time. For the avoidance of doubt,
    this trademark restriction does not form part of this License.

    Creative Commons may be contacted at http://creativecommons.org/.
```

Other Textbooks and Lecture Slides by the Author I

1. M. D. Adams, *Multiresolution Signal and Geometry Processing: Filter Banks, Wavelets, and Subdivision (Version 2013-09-26)*, University of Victoria, Victoria, BC, Canada, Sept. 2013, xxxviii + 538 pages, ISBN 978-1-55058-507-0 (print), ISBN 978-1-55058-508-7 (PDF). Available from Google Books, Google Play Books, University of Victoria Bookstore, and author's web site `http://www.ece.uvic.ca/~mdadams/waveletbook`.
2. M. D. Adams, *Lecture Slides for Multiresolution Signal and Geometry Processing (Version 2015-02-03)*, University of Victoria, Victoria, BC, Canada, Feb. 2015, xi + 587 slides, ISBN 978-1-55058-535-3 (print), ISBN 978-1-55058-536-0 (PDF). Available from Google Books, Google Play Books, University of Victoria Bookstore, and author's web site `http://www.ece.uvic.ca/~mdadams/waveletbook`.

Other Textbooks and Lecture Slides by the Author II

3. M. D. Adams, *Continuous-Time Signals and Systems (Version 2013-09-11)*, University of Victoria, Victoria, BC, Canada, Sept. 2013, xxx + 308 pages, ISBN 978-1-55058-495-0 (print), ISBN 978-1-55058-506-3 (PDF). Available from Google Books, Google Play Books, University of Victoria Bookstore, and author's web site `http://www.ece.uvic.ca/~mdadams/sigsysbook`.
4. M. D. Adams, *Lecture Slides for Continuous-Time Signals and Systems (Version 2013-09-11)*, University of Victoria, Victoria, BC, Canada, Dec. 2013, 286 slides, ISBN 978-1-55058-517-9 (print), ISBN 978-1-55058-518-6 (PDF). Available from Google Books, Google Play Books, University of Victoria Bookstore, and author's web site `http://www.ece.uvic.ca/~mdadams/sigsysbook`.

5. M. D. Adams, *Lecture Slides for Signals and Systems (Version 2016-01-25)*, University of Victoria, Victoria, BC, Canada, Jan. 2016, xvi + 481 slides, ISBN 978-1-55058-584-1 (print), ISBN 978-1-55058-585-8 (PDF). Available from Google Books, Google Play Books, University of Victoria Bookstore, and author's web site `http://www.ece.uvic.ca/~mdadams/sigsysbook`.

Part 0

Preface

- This document constitutes a detailed set of lecture slides on the C++ programming language and is current with the *C++17* standard.
- Many aspects of the C++ language are covered from introductory to more advanced.
- Some aspects of the C++ standard library are also introduced.
- In addition, various general programming-related topics are considered.

Acknowledgments

- The author would like to thank Robert Leahy for reviewing various drafts of many of these slides and providing many useful comments that allowed the quality of these materials to be improved significantly.

Disclaimer

- Many code examples are included throughout these slides.
- Often, in order to make an example short enough to fit on a slide, compromises had to be made in terms of good programming style.
- These deviations from good style include (but are not limited to) such things as:
    1. frequently formatting source code in unusual ways to conserve vertical space in listings;
    2. not fully documenting source code with comments;
    3. using short meaningless identifier names; and
    4. engaging other evil behavior such as using many global variables and employing constructs like "`using namespace std;`".

Typesetting Conventions

- In a definition, the term being defined is often typeset in a font **like this**.
- To emphasize particular words, the words are typeset in a font ***like this***.

Part 1

Software

Why Is Software Important?

- almost all electronic devices run some software
- automobile engine control system, implantable medical devices, remote controls, office machines (e.g., photocopiers), appliances (e.g., televisions, refrigerators, washers/dryers, dishwashers, air conditioner), power tools, toys, mobile phones, media players, computers, printers, photocopies, disk drives, scanners, webcams, MRI machines

Why Software-Based Solutions?

- more cost effective to implement functionality in software than hardware
- software bugs easy to fix, give customer new software upgrade
- hardware bugs extremely costly to repair, customer sends in old device and manufacturer sends replacement
- systems increasingly complex, bugs unavoidable
- allows new features to be added later
- implement only absolute minimal functionality in hardware, do the rest in software

Software-Related Jobs

- many more software jobs than hardware jobs
- relatively small team of hardware designers produce platform like iPhone
- thousands of companies develop applications for platform
- only implement directly in hardware when absolutely necessary (e.g., for performance reasons)

Which Language to Learn?

- C, C++, Fortran, Java, MATLAB, C#, Objective C
- programming language popularity
- `http://www.tiobe.com/` TIOBE Software Programming Community Index Jan 2011 all in top four: Java, C, C++ MATLAB (23rd) Fortran (27th)
- Programming Language Popularity Normalized Comparison `http://www.langpop.com/` top three languages: C, Java, C++
- international standard
- vendor neutral

C

- created by Dennis Ritchie, AT&T Bell Labs in 1970s
- international standard ISO/IEC 9899:2011 (informally known as “C11”)
- available on wide range of platforms, from microcontrollers to supercomputers; very few platforms for which C compiler not available
- procedural, provides language constructs that map efficiently to machine instructions
- does not directly support object-oriented or generic programming
- application domains: system software, device drivers, embedded applications, application software
- greatly influenced development of C++
- when something lasts in computer industry for more than 40 years (outliving its creator), must be good

- created by Bjarne Stroustrup, Bell Labs
- originally C with Classes, renamed as C++ in 1983
- most recent specification of language in ISO/IEC 14882:2017 (informally known as "C++17")
- procedural
- loosely speaking is superset of C
- directly supports object-oriented and generic programming
- maintains efficiency of C
- application domains: systems software, application software, device drivers, embedded software, high-performance server and client applications, entertainment software such as video games, native code for Android applications
- greatly influenced development of C# and Java

Java

- developed in 1990s by James Gosling at Sun Microsystems (later bought by Oracle Corporation)
- de facto standard but not international standard
- usually less efficient than C and C++
- simplified memory management (with garbage collection)
- direct support for object-oriented programming
- application domains: web applications, Android applications

MATLAB

- proprietary language, developed by The MathWorks
- not general-purpose programming language
- application domain: numerical computing
- used to design and simulate systems
- not used to implement real-world systems

Fortran

- designed by John Backus, IBM, in 1950s
- international standard ISO/IEC 1539-1:2010 (informally known as ”Fortran 2008”)
- application domain: scientific and engineering applications, intensive supercomputing tasks such as weather and climate modelling, finite element analysis, computational fluid dynamics, computational physics, computational chemistry

C#

- developed by Microsoft, team led by Anders Hejlsberg
- ECMA-334 and ISO/IEC 23270:2006
- most recent language specifications not standardized by ECMA or ISO/IEC
- intellectual property concerns over Microsoft patents
- object oriented

Objective C

- developed by Tom Love and Brad Cox of Stepstone (later bought by NeXT and subsequently Apple)
- used primarily on Apple Mac OS X and iOS
- strict superset of C
- no official standard that describes Objective C
- authoritative manual on Objective-C 2.0 available from Apple

Why Learn C++?

- vendor neutral
- international standard
- general purpose
- powerful yet efficient
- loosely speaking, includes C as subset; so can learn two languages (C++ and C) for price of one
- easy to move from C++ to other languages but often not in other direction
- many other popular languages inspired by C++

Part 2

C++

Section 2.1

History of C++

Motivation

- developed by Bjarne Stroustrup starting in 1979 at Computing Science Research Center of Bell Laboratories, Murray Hill, NJ, USA
- doctoral work in Computing Laboratory of University of Cambridge, Cambridge, UK
- study alternatives for organization of system software for distributed systems
- required development of relatively large and detailed simulator
- dissertation:

  B. Stroustrup. *Communication and Control in Distributed Computer Systems.*
  PhD thesis, University of Cambridge, Cambridge, UK, 1979.

- in 1979, joined Bell Laboratories after having finished doctorate
- work started with attempt to analyze UNIX kernel to determine to what extent it could be distributed over network of computers connected by LAN
- needed way to model module structure of system and pattern of communication between modules
- no suitable tools available

Objectives

- had bad experiences writing simulator during Ph.D. studies; originally used Simula for simulator; later forced to rewrite in BCPL for speed; more low level than C; BCPL was horrible to use
- notion of what properties good tool would have motivated by these experiences
- suitable tool for projects like simulator, operating system, other systems programming tasks should:
  - support for effective program organization (like in Simula) (i.e., classes, some form of class hierarchies, some form of support for concurrency, strong checking of type system based on classes)
  - produce programs that run fast (like with BCPL)
  - be able to easily combine separately compilable units into program (like with BCPL)
  - have simple linkage convention, essential for combining units written in languages such as C, Algol68, Fortran, BCPL, assembler into single program
  - allow highly portable implementations (only very limited ties to operating system)

Timeline for C with Classes (1979–1983) I

May 1979 work on C with Classes starts

Oct 1979 initial version of Cpre, preprocessor that added Simula-like classes to C; language accepted by preprocessor later started being referred to as C with Classes

Mar 1980 Cpre supported one real project and several experiments (used on about 16 systems)

Apr 1980 first internal Bell Labs paper on C with Classes published (later to appear in ACM SIGPLAN Notices in Jan. 1982)

> B. Stroustrup. Classes: An abstract data type facility for the C language.
> Bell Laboratories Computer Science Technical Report CSTR-84, Apr. 1980.

Timeline for C with Classes (1979–1983) II

1980 initial 1980 implementation had following features:

- classes
- derived classes
- public/private access control
- constructors and destructors
- call and return functions (call function implicitly called before every call of every member function; return function implicitly called after every return from every member function; can be used for synchronization)
- friend classes
- type checking and conversion of function arguments

1981 in 1981, added:

- inline functions
- default arguments
- overloading of assignment operator

Jan 1982 first external paper on C with Classes published

Timeline for C with Classes (1979–1983) III

B. Stroustrup. Classes: An abstract data type facility for the C language.
ACM SIGPLAN Notices, 17(1):42–51, Jan. 1982.

Feb 1983 more detailed paper on C with Classes published

B. Stroustrup. Adding classes to the C language: An exercise in language evolution.
Software: Practice and Experience, 13(2):139–161, Feb. 1983.

- C with Classes proved very successful; generated considerable interest
- first real application of C with Classes was network simulators

Timeline for C84 to C++98 (1982–1998) I

- started to work on cleaned up and extended successor to C with Classes, initially called C84 and later renamed C++

Spring 1982 started work on Cfront compiler front-end for C84; initially written in C with Classes and then transcribed to C84; traditional compiler front-end performing complete check of syntax and semantics of language, building internal representation of input, analyzing and rearranging representation, and finally producing output for some code generator; generated C code as output; difficult to bootstrap on machine without C84 compiler; Cfront software included special "half-processed" version of C code resulting from compiling Cfront, which could be compiled with native C compiler and resulting executable then used to compile Cfront

Dec 1983 C84 (C with Classes) renamed C++;
name used in following paper prepared in Dec. 1983

> B. Stroustrup. Data abstraction in C.
> *Bell Labs Technical Journal*, 63(8):1701–1732, Oct. 1984.

(name C++ suggested by Rick Mascitti)

1983 virtual functions added

Note: going from C with Classes to C84 added: virtual functions, function name and operator overloading, references, constants (`const`), user-controlled free-store memory control, improved type checking

Jan 1984 first C++ manual

> B. Stroustrup. The C++ reference manual.
> AT&T Bell Labs Computer Science Technical Report No. 108, Jan. 1984.

Sep 1984 paper describing operator overloading published

B. Stroustrup. Operator overloading in C++.
In *Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience & Assessment*, Sept. 1984.

1984 stream I/O library first implemented and later presented in

B. Stroustrup. An extensible I/O facility for C++.
In *Proc. of Summer 1985 USENIX Conference*, pages 57–70, June 1985.

Feb 1985 Cfront Release E (first external release); “E” for “Educational”; available to universities

Oct 1985 Cfront Release 1.0 (first commercial release)

Oct 1985 first edition of C++PL written

B. Stroustrup. *The C++ Programming Language*.
Addison Wesley, 1986.

(Cfront Release 1.0 corresponded to language as defined in this book)

Oct 1985 tutorial paper on C++

> B. Stroustrup. A C++ tutorial.
> In *Proceedings of the ACM annual conference on the range of computing: mid-80's perspective*, pages 56–64, Oct. 1985.

Jun 1986 Cfront Release 1.1; mainly bug fix release

Aug 1986 first exposition of set of techniques for which C++ was aiming to provide support (rather than what features are already implemented and in use)

> B. Stroustrup. What is object-oriented programming?
> In *Proc. of 14th Association of Simula Users Conference*, Stockholm, Sweden, Aug. 1986.

Timeline for C84 to C++98 (1982–1998) V

Sep 1986 first Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) conference (start of OO hype centered on Smalltalk)

Nov 1986 first commercial Cfront PC port (Cfront 1.1, Glockenspiel [in Ireland])

Feb 1987 Cfront Release 1.2; primarily bug fixes but also added:

- pointers to members
- protected members

Nov 1987 first conference devoted to C++:
USENIX C++ conference (Santa Fe, NM, USA)

Dec 1987 first GNU C++ release (1.13)

Jan 1988 first Oregon Software (a.k.a. TauMetric) C++ release

Jun 1988 first Zortech C++ release

Oct 1988 first presented templates at USENIX C++ conference (Denver, CO, USA) in paper:

Timeline for C84 to C++98 (1982–1998) VI

B. Stroustrup. Parameterized types for C++. In *Proc. of USENIX C++ Conference*, pages 1–18, Denver, CO, USA, Oct. 1988.

Oct 1988 first USENIX C++ implementers workshop (Estes Park, CO, USA)

Jan 1989 first C++ journal "The C++ Report" (from SIGS publications) started publishing

Jun 1989 Cfront Release 2.0 major cleanup; new features included:

- multiple inheritance
- type-safe linkage
- better resolution of overloaded functions
- recursive definition of assignment and initialization
- better facilities for user-defined memory management
- abstract classes
- static member functions
- const member functions

- protected member functions (first provided in release 1.2)
- overloading of operator `->`
- pointers to members (first provided in release 1.2)

1989 main features of Cfront 2.0 summarized in

B. Stroustrup. The evolution of C++: 1985–1989. *USENIX Computer Systems*, 2(3), Summer 1989.

first presented in

B. Stroustrup. The evolution of C++: 1985–1987. In *Proc. of USENIX C++ Conference*, pages 1–22, Santa Fe, NM, USA, Nov. 1987.

Nov 1989 paper describing exceptions published

A. Koenig and B. Stroustrup. Exception handling for C++. In *Proc. of “C++ at Work” Conference*, Nov. 1989.

followed up by

A. Koenig and B. Stroustrup. Exception handling for C++. In *Proc. of USENIX C++ Conference*, Apr. 1990.

- **Dec 1989** ANSI X3J16 organizational meeting (Washington, DC, USA)
- **Mar 1990** first ANSI X3J16 technical meeting (Somerset, NJ, USA)
- **Apr 1990** Cfront Release 2.1; bug fix release to bring Cfront mostly into line with ARM
- **May 1990** annotated reference manual (ARM) published

  M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, May 1990.

  (formed basis for ANSI standardization)
- **May 1990** first Borland C++ release
- **Jul 1990** templates accepted (Seattle, WA, USA)
- **Nov 1990** exceptions accepted (Palo Alto, CA, USA)

Timeline for C84 to C++98 (1982–1998) IX

Jun 1991 second edition of C++PL published

> B. Stroustrup. *The C++ Programming Language*.
> Addison Wesley, 2nd edition, June 1991.

Jun 1991 first ISO WG21 meeting (Lund, Sweden)

Sep 1991 Cfront Release 3.0; added templates (as specified in ARM)

Oct 1991 estimated number of C++ users 400,000

Feb 1992 first DEC C++ release (including templates and exceptions)

Mar 1992 run-time type identification (RTTI) described in

> B. Stroustrup and D. Lenkov. Run-time type identification for C++.
> *The C++ Report*, Mar. 1992.

(RTTI in C++ based on this paper)

Mar 1992 first Microsoft C++ release (did not support templates or exceptions)

May 1992	first IBM C++ release (including templates and exceptions)
Mar 1993	RTTI accepted (Portland, OR, USA)
Jul 1993	namespaces accepted (Munich, Germany)
1993	further work on Cfront Release 4.0 abandoned after failed attempt to add exception support
Aug 1994	ANSI/ISO Committee Draft registered
Aug 1994	Standard Template Library (STL) accepted (Waterloo, ON, CA); described in

A. Stepanov and M. Lee. The standard template library. Technical Report HPL-94-34 (R.1), HP Labs, Aug. 1994.

Aug 1996	**`export`** accepted (Stockholm, Sweden)
1997	third edition of C++PL published

B. Stroustrup. *The C++ Programming Language*. Addison Wesley Longman, Reading, MA, USA, 3rd edition, 1997.

Timeline for C84 to C++98 (1982–1998) XI

Nov 1997 final committee vote on complete standard (Morristown, NJ, USA)

Jul 1998 Microsoft releases VC++ 6.0, first Microsoft compiler to provide close-to-complete set of ISO C++

Sep 1998 ISO/IEC 14882:1998 (informally known as C++98) published

> ISO/IEC 14882:1998 — programming languages — C++, Sept. 1998.

1998 Beman Dawes starts Boost (provides peer-reviewed portable C++ source libraries)

Feb 2000 special edition of C++PL published

> B. Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, MA, USA, special edition, Feb. 2000.

Timeline After C++98 (1998–Present) I

- **Apr 2001** motion passed to request new work item: technical report on libraries (Copenhagen, Denmark); later to become ISO/IEC TR 19768:2007
- **Oct 2003** ISO/IEC 14882:2003 (informally known as C++03) published; essentially bug fix release; no changes to language from programmer's point of view

  > ISO/IEC 14882:2003 — programming languages — C++, Oct. 2003.

- **2003** work on C++0x (now known as C++11) starts
- **Oct 2004** estimated number of C++ users 3,270,000
- **Apr 2005** first votes on features for C++0x (Lillehammer, Norway)
- **2005** `auto`, `static_assert`, and rvalue references accepted in principle
- **Apr 2006** first full committee (official) votes on features for C++0x (Berlin, Germany)

Timeline After C++98 (1998–Present) II

Sep 2006 performance technical report (TR 18015) published:

> ISO/IEC TR 18015:2006 — information technology — programming languages, their environments and system software interfaces — technical report on C++ performance, Sept. 2006.

work spurred by earlier proposal to standardize subset of C++ for embedded systems called Embedded C++ (or just EC++); EC++ motivated by performance concerns

Apr 2006 decision to move special mathematical functions to separate ISO standard (Berlin, Germany); deemed too specialized for most programmers

Nov 2007 ISO/IEC TR 19768:2007 (informally known as C++TR1) published;

> ISO/IEC TR 19768:2007 — information technology — programming languages — technical report on C++ library extensions, Nov. 2007.

specifies series of library extensions to be considered for adoption later in C++

2009 another particularly notable book on C++ published

B. Stroustrup. *Programming: Principles and Practice Using C++.*
Addison Wesley, Upper Saddle River, NJ, USA, 2009.

Aug 2011 ISO/IEC 14882:2011 (informally known as C++11) ratified

ISO/IEC 14882:2011 — information technology — programming languages — C++, Sept. 2011.

2013 fourth edition of C++PL published

B. Stroustrup. *The C++ Programming Language.*
Addison Wesley, 4th edition, 2013.

2014 ISO/IEC 14882:2014 (informally known as C++14) ratified

ISO/IEC 14882:2014 — information technology — programming languages — C++, Dec. 2014.

Timeline After C++98 (1998–Present) IV

2017 ISO/IEC 14882:2017 (informally known as C++17) ratified

ISO/IEC 14882:2017 — information technology — programming languages — C++, Dec. 2017.

Additional Comments

- reasons for using C as starting point:
  - flexibility (can be used for most application areas)
  - efficiency
  - availability (C compilers available for most platforms)
  - portability (source code relatively portable from one platform to another)
- main sources for ideas for C++ (aside from C) were Simula, Algol68, BCPL, Ada, Clu, ML; in particular:
  - Simula gave classes
  - Algol68 gave operator overloading, references, ability to declare variables anywhere in block
  - BCPL gave `//` comments
  - exceptions influenced by ML
  - templates influenced by generics in Ada and parameterized modules in Clu

C++ User Population

Time	Estimated Number of Users
Oct 1979	1
Oct 1980	16
Oct 1981	38
Oct 1982	85
Oct 1983	??+2 (no Cpre count)
Oct 1984	??+50 (no Cpre count)
Oct 1985	500
Oct 1986	2,000
Oct 1987	4,000
Oct 1988	15,000
Oct 1989	50,000
Oct 1990	150,000
Oct 1991	400,000
Oct 2004	over 3,270,000

- above numbers are conservative
- 1979 to 1991: C++ user population doubled approximately every 7.5 months
- stable growth thereafter

Success of C++

- C++ very successful programming language
- not luck or solely because based on C
- efficient, provides low-level access to hardware, but also supports abstraction
- non-proprietary: in 1989, all rights to language transferred to standards bodies (first ANSI and later ISO) from AT&T
- multi-paradigm language, supporting procedural, object-oriented, generic, and functional (e.g., lambda functions) programming
- does not force particular programming style
- reasonably portable
- has continued to evolve, incorporating new ideas (e.g., templates, exceptions, STL)
- stable: high degree of compatibility with earlier versions of language
- very strong bias towards providing general-purpose facilities rather than more application-specific ones

Application Areas

- banking and financial (funds transfer, financial modelling, teller machines)
- classical systems programming (compilers, operating systems, device drivers, network layers, editors, database systems)
- small business applications (inventory systems)
- desktop publishing (document viewers/editors, image editing)
- embedded systems (cameras, cell phones, airplanes, medical systems, appliances)
- entertainment (games)
- GUI
- hardware design and verification
- scientific and numeric computation (physics, engineering, simulations, data analysis, geometry processing)
- servers (web servers, billing systems)
- telecommunication systems (phones, networking, monitoring, billing, operations systems)

Section 2.1.1

References

- B. Stroustrup. A history of C++: 1979–1991.
  In *Proc. of ACM History of Programming Languages Conference*, pages 271–298, Mar. 1993
- B. Stroustrup. *The Design and Evolution of C++*.
  Addison Wesley, Mar. 1994.
- B. Stroustrup. Evolving a language in and for the real world: C++ 1991–2006.
  In *Proc. of the ACM SIGPLAN Conference on History of Programming Languages*, pages 4–1–4–59, 2007.
- Cfront software available from Computer History Museum's Software Preservation Group `http://www.softwarepreservation.org`.
  (See `http://www.softwarepreservation.org/projects/c_plus_plus/cfront`).
- ISO JTC1/SC22/WG21 web site. `http://www.open-std.org/jtc1/sc22/wg21/`.

Standards Documents I

- ISO/IEC 14882:1998 — programming languages — C++, Sept. 1998.
- ISO/IEC 14882:2003 — programming languages — C++, Oct. 2003.
- ISO/IEC TR 18015:2006 — information technology — programming languages, their environments and system software interfaces — technical report on C++ performance, Sept. 2006.
- ISO/IEC TR 19768:2007 — information technology — programming languages — technical report on C++ library extensions, Nov. 2007.
- ISO/IEC 14882:2011 — information technology — programming languages — C++, Sept. 2011.
- ISO/IEC 14882:2014 — information technology — programming languages — C++, Dec. 2014.
- ISO/IEC TS 18822:2015 — programming languages — C++ — file system technical specification, July 2015.

Standards Documents II

- ISO/IEC TS 19570:2015 — programming languages — technical specification for C++ extensions for parallelism, July 2015.
- ISO/IEC TS 19841:2015 — technical specification for C++ extensions for transactional memory, Oct. 2015.
- ISO/IEC TS 19217:2015 — programming languages — C++ extensions for concepts, Nov. 2015.
- ISO/IEC TS 19571:2016 — programming languages — technical specification for C++ extensions for concurrency, Feb. 2016.
- ISO/IEC TS 19568:2017 — programming languages — C++ extensions for library fundamentals, Mar. 2017.
- ISO/IEC TS 21425:2017 — programming languages — C++ extensions for ranges, Nov. 2017.
- ISO/IEC 14882:2017 — information technology — programming languages — C++, Dec. 2017.

- ISO JTC1/SC22/WG21 web site. `http://www.open-std.org/jtc1/sc22/wg21/`.

Section 2.2

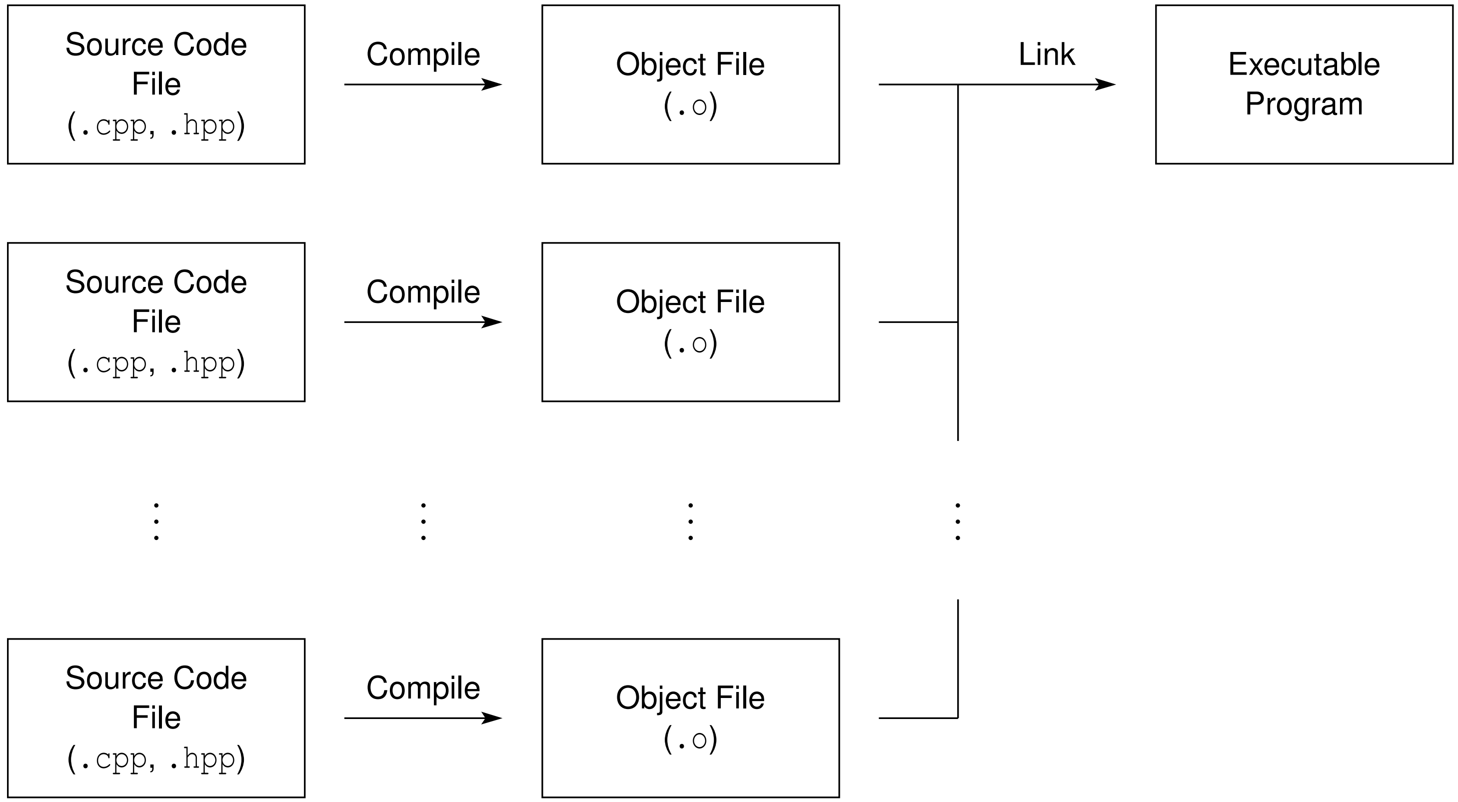
Getting Started

`hello` Program: `hello.cpp`

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

- program prints message “Hello, world!” and then exits
- starting point for execution of C++ program is function called `main`; every C++ program must define function called `main`
- `#include` preprocessor directive to include complete contents of file
- `iostream` standard header file that defines various types and variables related to I/O
- `std::cout` is standard output stream (defaults to user’s terminal)
- operator `<<` is used for output

Software Build Process



- start with C++ source code files (`.cpp`, `.hpp`)
- compile: convert source code to object code
- object code stored in object file (`.o`)
- link: combine contents of one or more object files (and possibly some libraries) to produce executable program
- executable program can then be run directly

GNU Compiler Collection (GCC) C++ Compiler

- `g++` command provides both compiling and linking functionality
- command-line usage:

  `g++` [*options*] *input_file* ...
- many command-line options are supported
- some particularly useful command-line options listed on next slide
- compile C++ source file *file.cpp* to produce object code file *file.o*:

  `g++ -c` *file.cpp*
- link object files *file_1.o*, *file_2.o*, ... to produce executable file *executable*:

  `g++ -o` *executable file_1.o file_2.o* ...
- web site:

  `http://www.gnu.org/software/gcc`
- C++ standards support in GCC:

  `https://gcc.gnu.org/projects/cxx-status.html`

Common g++ Command-Line Options

- `-c`
  - compile only (i.e., do not link)
- `-o` *file*
  - use file *file* for output
- `-g`
  - include debugging information
- `-O`*n*
  - set optimization level to n (0 almost none; 3 full)
- `-std=c++17`
  - conform to C++17 standard
- `-I`*dir*
  - specify additional directory *dir* to search for include files
- `-L`*dir*
  - specify additional directory *dir* to search for libraries
- `-l`*lib*
  - link with library *lib*

Common `g++` Command-Line Options (Continued 1)

- `-pthread`
  - enable concurrency support (via pthreads library)
- `-pedantic-errors`
  - strictly enforce compliance with standard
- `-Wall`
  - enable most warning messages
- `-Wextra`
  - enable some extra warning messages not enabled by `-Wall`
- `-Wpedantic`
  - warn about deviations from strict standard compliance
- `-Werror`
  - treat all warnings as errors
- `-fno-elide-constructors`
  - in contexts where standard allows (but does not require) optimization that omits creation of temporary, do not attempt to perform this optimization

Common `g++` Command-Line Options (Continued 2)

- `-fconstexpr-loop-limit=`*n*
  - set maximum number of iterations for loop in constexpr functions to n
- `-fconstexpr-depth=`*n*
  - set maximum nested evaluation depth for constexpr functions to n

Clang C++ Compiler

- `clang++` command provides both compiling and linking functionality
- command-line usage:

  `clang++` [*options*] *input_file* ...

- many command-line options are supported
- command-line interface is largely compatible with that of GCC `g++` command
- web site:

  `http://clang.llvm.org`

- C++ standards support in Clang:

  `http://clang.llvm.org/cxx_status.html`

Common `clang++` Command-Line Options

- many of more frequently used command-line options for `clang++` identical to those for `g++`
- consequently, only small number of `clang++` options given below
- `-fconstexpr-steps=`n
  - sets maximum number of computation steps in constexpr functions to n
- `-fconstexpr-depth=`n
  - sets maximum nested evaluation depth for constexpr functions to n

Manually Building `hello` Program

- numerous ways in which `hello` program could be built
- often advantageous to compile each source file separately
- can compile and link as follows:
  1. compile source code file `hello.cpp` to produce object file `hello.o`:

     ```
     g++ -c hello.cpp
     ```
  2. link object file `hello.o` to produce executable program `hello`:

     ```
     g++ -o hello hello.o
     ```
- generally, manual building of program is quite tedious, especially when program consists of multiple source files and additional compiler options need to be specified
- in practice, we use tools to automate build process (e.g., CMake and Make)

Section 2.3

C++ Basics

The C++ Programming Language

- created by Bjarne Stroustrup of Bell Labs
- originally known as C with Classes; renamed as C++ in 1983
- most recent specification of language in ISO/IEC 14882:2017 (informally known as "C++17")
- next version of standard expected in approximately 2020 (informally known as "C++20")
- procedural
- loosely speaking is superset of C
- directly supports object-oriented and generic programming
- maintains efficiency of C
- application domains: systems software, application software, device drivers, embedded software, high-performance server and client applications, entertainment software such as video games, native code for Android applications
- greatly influenced development of C# and Java

Comments

- two styles of comments provided
- comment starts with // and proceeds to end of line
- comment starts with /* and proceeds to first */

```
// This is an example of a comment.
/* This is another example of a comment. */
/* This is an example of a comment that
   spans
   multiple lines. */
```

- comments of /* ··· */ style ***do not nest***

```
/*
/* This sentence is part of a comment. */
This sentence is not part of any comment and
will probably cause a compile error.
*/
```

- identifiers used to name entities such as: types, objects (i.e., variables), and functions
- valid identifier is sequence of one or more letters, digits, and underscore characters that does not begin with a digit
- identifiers that begin with underscore (in many cases) or contain double underscores are reserved for use by C++ implementation and should be avoided
- examples of valid identifiers:
  - `event_counter`
  - `eventCounter`
  - `sqrt_2`
  - `f_o_o_b_a_r_4_2`
- identifiers are case sensitive (e.g., `counter` and `cOuNtEr` are distinct identifiers)
- identifiers cannot be any of reserved keywords (see next slide)
- **scope** of identifier is context in which identifier is valid (e.g., block, function, global)

Reserved Keywords

```
alignas        default           noexcept           this
alignof        delete            not                thread_local
and            do                not_eq             throw
and_eq         double            nullptr            true
asm            dynamic_cast      operator           try
auto           else              or                 typedef
bitand         enum              or_eq              typeid
bitor          explicit          private            typename
bool           export            protected          union
break          extern            public             unsigned
case           false             register           using
catch          float             reinterpret_cast   virtual
char           for               return             void
char16_t       friend            short              volatile
char32_t       goto              signed             wchar_t
class          if                sizeof             while
compl          inline            static             xor
const          int               static_assert      xor_eq
constexpr      long              static_cast        override*
const_cast     mutable           struct             final*
continue       namespace         switch
decltype       new               template
```

*Note: context sensitive

Section 2.3.1

Preprocessor

The Preprocessor

- prior to compliation, source code transformed by preprocessor
- preprocessor output then passed to compiler for compilation
- preprocessor behavior can be controlled by preprocessor directives
- preprocessor directive occupies single line and consists of:
  1. hash character (i.e., "`#`")
  2. preprocessor instruction (i.e., `define`, `undef`, `include`, `if`, `ifdef`, `ifndef`, `else`, `elif`, `endif`, `line`, `error`, and `pragma`)
  3. arguments (depending on instruction)
  4. line break
- preprocessor can be used to:
  - conditionally compile parts of source file
  - define macros and perform macro expansion
  - include other files
  - force error to be generated

Source-File Inclusion

- can include contents of another file in source using preprocessor **`#include`** directive
- syntax:

  `#include` <*path_specifier*>

  or

  `#include` "*path_specifier*"

- *path_specifier* is pathname (which may include directory) identifying file whose content is to be substituted in place of include directive
- typically, angle brackets used for system header files and double quotes used otherwise
- example:

```
#include <iostream>
#include <boost/tokenizer.hpp>
#include "my_header_file.hpp"
#include "some_directory/my_header_file.hpp"
```

Defining Macros

- can define macros using **`#define`** directive
- syntax:

  `#define` *name value*

- *name* is name of macro and *value* is value of macro
- example:

```
#define DEBUG_LEVEL 10
```

- macros can also take arguments
- generally, macros should be avoided when possible (i.e., when other better mechanisms are avaliable to achieve desired effect)
- for example, although macros can be used as way to accomplish inlining of functions, such usage should be avoided since language mechanism exists for specifying inline functions

Conditional Compilation

- can conditionally include code through use of if-elif-else construct
- conditional preprocessing block consists of following (in order)
  1. **`#if`**, **`#ifdef`**, or **`#ifndef`** directive
  2. optionally any number of **`#elif`** directives
  3. at most one **`#else`** directive
  4. **`#endif`** directive
- code in taken branch of if-elif-else construct passed to compiler, while code in other branches discarded
- example:

```
#if DEBUG_LEVEL == 1
// ...
#elif DEBUG_LEVEL == 2
// ...
#else
// ...
#endif
```

Preprocessor Predicate __has_include

- preprocessor predicate `__has_include` can be used in expressions for preprocessor to test for existence of header files
- example:

```
#ifdef __has_include
#   if __has_include(<optional>)
#       include <optional>
#       define have_optional 1
#   elif __has_include(<experimental/optional>)
#       include <experimental/optional>
#       define have_optional 1
#       define experimental_optional
#   else
#       define have_optional 0
#   endif
#endif
```

Section 2.3.2

Objects, Types, and Values

Fundamental Types

- boolean type: **`bool`**
- character types:
  - **`char`** (may be signed or unsigned)
  - **`signed char`**
  - **`unsigned char`**
  - **`char16_t`**
  - **`char32_t`**
  - **`wchar_t`**
- **`char`** is distinct type from **`signed char`** and **`unsigned char`**
- standard signed integer types:
  - **`signed char`**
  - **`signed short int`**
  - **`signed int`**
  - **`signed long int`**
  - **`signed long long int`**
- standard unsigned integer types:
  - **`unsigned char`**
  - **`unsigned short int`**
  - **`unsigned int`**
  - **`unsigned long int`**
  - **`unsigned long long int`**

Fundamental Types (Continued)

- "**`int`**" may be omitted from names of (non-character) integer types (e.g., "**`unsigned`**" equivalent to "**`unsigned int`**" and "**`signed`**" equivalent to "**`signed int`**")
- "**`signed`**" may be omitted from names of signed integer types, excluding **`signed char`** (e.g., "**`int`**" equivalent to "**`signed int`**")
- boolean, character, and (signed and unsigned) integer types collectively called **integral types**
- integral types must use binary positional representation; two's complement, one's complement, and sign magnitude representations permitted
- floating-point types:
  - **`float`**
  - **`double`**
  - **`long double`**
- void (i.e., incomplete/valueless) type: **`void`**
- null pointer type: `std::nullptr_t` (defined in header file `cstddef`)

Literals

- **literal** (a.k.a. literal constant) is value written exactly as it is meant to be interpreted
- examples of literals:

```
"Hello, world"
"Bjarne"
'a'
'A'
123
123U
1'000'000'000
3.1415
1.0L
1.23456789e-10
```

Character Literals

- character literal consists of optional prefix followed by one or more characters enclosed in single quotes
- type of character literal determined by prefix (or lack thereof) as follows:

Prefix	Literal	Type
None	ordinary	normally **`char`** (in special cases **`int`**)
`u8`	UTF-8	**`char`**
`u`	UCS-2	**`char16_t`**
`U`	UCS-4	**`char32_t`**
`L`	wide	**`wchar_t`**

- special characters can be represented by escape sequence:

Character	Escape Sequence
newline (LF)	`\n`
horizontal tab (HT)	`\t`
vertical tab (VT)	`\v`
backspace (BS)	`\b`
carriage return (CR)	`\r`
form feed (FF)	`\f`
alert (BEL)	`\a`
backslash (`\`)	`\\`

Character	Escape Sequence
question mark (`?`)	`\?`
single quote (`'`)	`\'`
double quote (`"`)	`\"`
octal number ooo	`\`ooo
hex number hhh	`\x`hhh
code point nnnn	`\u`nnnn
code point nnnnnnnn	`\U`nnnnnnnn

- examples of character literals:

```
'a'   '1'   '!'   '\n'   u'a'   U'a'   L'a'   u8'a'
```

- decimal digit characters guaranteed to be consecutive in value (e.g., `'1'` must equal `'0' + 1`)
- in case of ordinary character literals, alphabetic characters are ***not*** guaranteed to be consecutive in value (e.g., `'b'` is not necessarily `'a' + 1`)

String Literals

- (non-raw) string literal consists of optional prefix followed by zero or more characters enclosed in double quotes
- string literal has character array type
- type of string literal determined by prefix (or lack thereof) as follows:

Prefix	Literal	Type
None	narrow	**const char**[]
`u8`	UTF-8	**const char**[]
`u`	UTF-16	**const char16_t**[]
`U`	UTF-32	**const char32_t**[]
`L`	wide	**const wchar_t**[]

- examples of string literals:

```
"Hello, World!\n"
"123"
"ABCDEFG"
```

- adjacent string literals are concatenated (e.g., `"Hel" "lo"` equivalent to `"Hello"`)
- string literals implicitly terminated by null character (i.e., `'\0'`)
- so, for example, `"Hi"` means `'H'` followed by `'i'` followed by `'\0'`

Raw String Literals

- interpretation of escape sequences (e.g., "`\n`") inside string literal can be avoided by using raw literal
- raw literal has form:
  - *prefix* `R"`*delimiter*`(`*raw_characters*`)`*delimiter*`"`
- optional *prefix* is string-literal prefix (e.g., `u8`)
- optional *delimiter* is sequence of characters used to assist in delimiting string
- *raw_characters* is sequence of characters comprising string
- escape sequences not processed inside raw literal
- raw literal can also contain newline characters
- examples of raw string literals:

```
R"(He said, "No.")"
u8R"(He said, "No.")"
R"foo(The answer is 42.)foo"
R"((+|-)?[[:digit:]]+)"
```

Integer Literals

- can be specified in decimal, binary, hexadecimal, and octal
- number base indicated by prefix (or lack thereof) as follows:

Prefix	Number Base
None	decimal
Leading `0`	octal
`0b` or `0B`	binary
`0x` or `0X`	hexadecimal

- various suffixes can be specified to control type of literal:
  - `u` or `U`
  - `l` or `L`
  - both `u` or `U` and `l` or `L`
  - `ll` or `LL`
  - both `u` or `U` and `ll` or `LL`
- can use single quote as digit separator (e.g., `1'000'000`)
- examples of integer literals:

```
42
1'000'000'000'000ULL
0xdeadU
```

- integer literal always nonnegative; so, for example, `-1` is integer literal `1` with negation operation applied

Integer Literals (Continued)

Suffix	Decimal Literal	Non-Decimal Literal
None	**`int`** **`long int`** **`long long int`**	**`int`** **`unsigned int`** **`long int`** **`unsigned long int`** **`long long int`** **`unsigned long long int`**
`u` or `U`	**`unsigned int`** **`unsigned long int`** **`unsigned long long int`**	**`unsigned int`** **`unsigned long int`** **`unsigned long long int`**
`l` or `L`	**`long int`** **`long long int`**	**`long int`** **`unsigned long int`** **`long long int`** **`unsigned long long int`**
Both `u` or `U` and `l` or `L`	**`unsigned long int`** **`unsigned long long int`**	**`unsigned long int`** **`unsigned long long int`**
`ll` or `LL`	**`long long int`**	**`long long int`** **`unsigned long long int`**
Both `u` or `U` and `ll` or `LL`	**`unsigned long long int`**	**`unsigned long long int`**

Floating-Point Literals

- type of literal indicated by suffix (or lack thereof) as follows:

Suffix	Type
None	**`double`**
`f` or `F`	**`float`**
`l` or `L`	**`long double`**

- examples of **`double`** literals:

```
1.414
1.25e-8
```

- examples of **`float`** literals:

```
1.414f
1.25e-8f
```

- examples of **`long double`** literals:

```
1.5L
1.25e-20L
```

- floating-point literals always nonnegative; so, for example, `-1.0` is literal `1.0` with negation operator applied

Hexadecimal Floating-Point Literals

- hexadecimal floating-point literal has general form:
  1. prefix `0x` or `0X`
  2. hexadecimal digits for integer part of number (optional if at least one digit after radix point)
  3. period character (i.e., radix point)
  4. hexadecimal digits for fractional part of number (optional if at least one digit before radix point)
  5. `p` character (which designates exponent to follow)
  6. one or more decimal digits for base-16 exponent
  7. optional floating-point literal suffix (e.g., `f` or `l`)
- examples of hexadecimal floating-point literals:

Literal	Type	Value (Decimal)
`0x.8p0`	**`double`**	0.5
`0x10.cp0`	**`double`**	16.75
`0x.8p0f`	**`float`**	0.5
`0xf.fp0f`	**`float`**	15.9375
`0x1p10L`	**`long double`**	1024

Boolean and Pointer Literals

- boolean literals:

```
true
false
```

- pointer literal:

```
nullptr
```

Declarations and Definitions

- **declaration** introduces identifier for type, object (i.e., variable), or function (without necessarily providing full information about identifier)
  - in case of object, specifies type (of object)
  - in case of function, specifies number of parameters, type of each parameter, and type of return value (if not automatically deduced)
- each identifier must be declared before it can be used (i.e., referenced)
- **definition** provides full information about identifier and causes entity associated with identifier (if any) to be created
  - in case of type, provides full details about type
  - in case of object, causes storage to be allocated for object and object to be created
  - in case of function, provides code for function body
- in case of objects, in most (but not all) contexts, declaring object also defines it
- can declare identifier multiple times but can define only once
- above terminology often abused, with “declaration” and “definition” being used interchangeably

Examples of Declarations and Definitions

```
int count; // declare and define count
extern double alpha; // (only) declare alpha

void func() { // declare and define func
    int n; // declare and define n
    double x = 1.0; // declare and define x
    // ...
}

bool isOdd(int); // declare isOdd
bool isOdd(int x); // declare isOdd (x ignored)

bool isOdd(int x) { // declare and define isOdd
    return x % 2;
}

struct Thing; // declare Thing

struct Vector2 { // declare and define Vector2
    double x;
    double y;
};
```

Variable Declarations and Definitions

- **variable declaration** (a.k.a. object declaration) introduces identifier that names object and specifies type of object
- **variable definition** (a.k.a. object definition) provides all information included in variable declaration and also causes object to be created (e.g., storage allocated for object)
- example:

```
int count;
  // declare and define count
double alpha;
  // declare and define alpha
extern double gamma;
  // declare (but do not define) gamma
```

Arrays

- **array** is collection of one or more objects of ***same*** type that are stored ***contiguously*** in memory
- each element in array identified by (unique) integer index, with indices starting from ***zero***
- array denoted by `[]`
- example:

```
double x[10]; // array of 10 doubles
int data[512][512]; // 512 by 512 array of ints
```

- elements of array accessed using subscripting operator `[]`
- example:

```
int x[10];
// elements of arrays are x[0], x[1], ..., x[9]
```

- often preferable to use user-defined type for representing array instead of array type
- for example, `std::array` and `std::vector` types (to be discussed later) have numerous practical advantages over array types

Array Example

- code:

```
int a[4] = {1, 2, 3, 4};
```

- assumptions (for some completely fictitious C++ language implementation):
  - `sizeof(int)` is 4
  - array `a` starts at address `1000`
- memory layout:

Address		Name
1000	1	a[0]
1004	2	a[1]
1008	3	a[2]
1012	4	a[3]

Pointers

- **pointer** is object whose value is address in memory where another object is stored
- pointer to object of type `T` denoted by `T*`
- **null pointer** is special pointer value that does not refer to any valid memory location
- null pointer value provided by **`nullptr`** keyword
- accessing object to which pointer refers called **dereferencing**
- dereferencing pointer performed by ***indirection operator*** (i.e., "`*`")
- if `p` is pointer, `*p` is object to which pointer refers
- if `x` is object of type `T`, `&x` is (normally) address of object, which has type `T*`
- example:

```
char c;
char* cp = nullptr; // cp is pointer to char
char* cp2 = &c; // cp2 is pointer to char
```

Pointer Example

- code:

```
int i = 42;
int* p = &i;
assert(*p == 42);
```

- assumptions (for some completely fictitious C++ language implementation):
  - `sizeof(int)` is 4
  - `sizeof(int*)` is 4
  - `&i` is `((int*)1000)`
  - `&p` is `((int*)1004)`
- memory layout:

Address	Value	Name
1000	42	i
1004	1000	p

References

- **reference** is alias (i.e., nickname) for ***already existing*** object
- two kinds of references:
  1. lvalue reference
  2. rvalue reference
- lvalue reference to object of type `T` denoted by `T&`
- rvalue reference to object of type `T` denoted by `T&&`
- initializing reference called **reference binding**
- lvalue and rvalue references differ in their binding properties (i.e., to what kinds of objects reference can be bound)
- in most contexts, lvalue references usually needed
- rvalue references used in context of move constructors and move assignment operators (to be discussed later)
- example:

```
int x;
int& y = x; // y is lvalue reference to int
int&& tmp = 3; // tmp is rvalue reference to int
```

References Example

- code:

```
int i = 42;
int& j = i;
assert(j == 42);
```

- assumptions (for some completely fictitious C++ language implementation):
  - `sizeof(int)` is 4
  - `&i` is `((int*)1000)`
- memory layout:

Address		Name
`1000`	`42`	`i, j`

References Versus Pointers

- references and pointers similar in that both can be used to refer to some other entity (e.g., object or function)
- two key differences between references and pointers:
  1. reference must refer to something, while pointer can have null value (**`nullptr`**)
  2. references cannot be rebound, while pointers can be changed to point to different entity
- references have cleaner syntax than pointers, since pointers must be dereferenced upon each use (and dereference operations tend to clutter code)
- use of pointers often implies need for memory management (i.e., memory allocation, deallocation, etc.), and memory management can introduce numerous kinds of bugs when done incorrectly
- often faced with decision of using pointer or reference in code
- generally advisable to prefer use of references over use of pointers unless compelling reason to do otherwise, such as:
  - must be able to handle case of referring to nothing
  - must be able to change entity being referred to

Unscoped Enumerations

- **enumerated type** provides way to describe range of values that are represented by named constants called **enumerators**
- object of enumerated type can take any one of enumerators as value
- enumerator values represented by some ***integral type***
- enumerator can be assigned specific value (which may be negative)
- if enumerator not assigned specific value, value defaults to zero if first enumerator in enumeration and one greater than value for previous enumerator otherwise
- example:

```
enum Suit {
    Clubs, Diamonds, Hearts, Spades
};

Suit suit = Clubs;
```

- example:

```
enum Suit {
    Clubs = 1, Diamonds = 2, Hearts = 4, Spades = 8
};
```

Scoped Enumerations

- scoped enumeration similar to unscoped enumeration, except
  - all enumerators are placed in scope of enumeration itself
  - integral type to used to hold enumerator values can be explicitly specified
  - conversions involving scoped enumerations are stricter (i.e., more type safe)
- **`class`** or **`struct`** added after **`enum`** keyword to make enumeration scoped
- scope resolution operator (i.e., "`::`") used to access enumerators
- scoped enumerations should probably be preferred to unscoped ones
- example:

```
enum struct Season {
    spring, summer, fall, winter
};

enum struct Suit : unsigned char {
    clubs, diamonds, hearts, spades
};

Season season = Season::summer;
Suit suit = Suit::spades;
```

Type Aliases with `typedef` Keyword

- **`typedef`** keyword used to create alias for existing type
- example:

```
typedef long long BigInt;
BigInt i; // i has type long long

typedef char* CharPtr;
CharPtr p; // p has type char*
```

Type Aliases with **using** Statement

- **using** statement can be used to create alias for existing type
- probably preferable to use **using** statement over **typedef**
- example:

```
using BigInt = long long;
BigInt i; // i has type long long

using CharPtr = char*;
CharPtr p; // p has type char*
```

The `extern` Keyword

- **translation unit**: basic unit of compilation in C++ (i.e., single source code file plus all of its directly and indirectly included header files)
- `extern` keyword used to declare object/function in separate translation unit
- example:

```
extern int evil_global_variable;
  // declaration only
  // actual definition in another file
```

The `const` Qualifier

- **`const`** qualifier specifies that object has value that is ***constant*** (i.e., cannot be changed)
- qualifier that applies to object itself said to be **top level**
- following defines `x` as **`int`** with value 42 that cannot be modified:

```
const int x = 42;
```

- example:

```
const int x = 42;
x = 13; // ERROR: x is const
const int& x1 = x; // OK
const int* p1 = &x; // OK
int& x2 = x; // ERROR: x const, x2 not const
int* p2 = &x; // ERROR: x const, *p2 not const
```

- example:

```
int x = 0;
const int& y = x;
x = 42; // OK
// y also changed to 42 since y refers to x
// y cannot be used to change x, however
// i.e., the following would cause compile error:
// y = 24; // ERROR: y is const
```

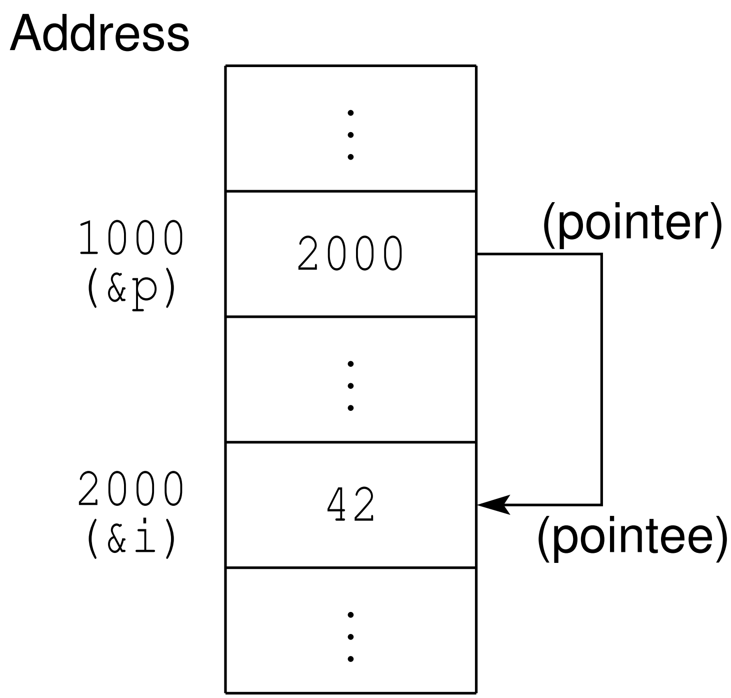
Example: `const` Qualifier and Non-Pointer/Non-Reference Types

```
// with types that are not pointer or reference types, const
//  can only be applied to object itself (i.e., top level)
// object itself may be const or non-const

int i = 0; // non-const int object
const int ci = 0; // const int object

i = 42; // OK: can modify non-const object
ci = 42; // ERROR: cannot modify const object

i = ci; // OK: can modify non-const object
ci = i; // ERROR: cannot modify const object
```

The `const` Qualifier and Pointer Types

- every pointer is associated with two objects: pointer itself and pointee (i.e., object to which pointer points)
- `const` qualifier can be applied to each of pointer (i.e., top-level qualifier) and pointee

```
int i = 42; // pointee

// p is pointer to int i
// for example:
// int* p = &i;
// const int* p = &i;
// int* const p = &i;
// const int* const p = &i;
```



Example: `const` Qualifier and Pointer Types

```
// with pointer types, const can be applied to each of:
//  pointer and pointee
// pointer itself may be const or non-const (top-level)
// pointee may be const or non-const

int i = 0;
int j = 0;

int* pi = &i; // non-const pointer to a non-const int
pi = &j; // OK: can modify non-const pointer
*pi = 42; // OK: can modify non-const pointee

const int* pci = &i; // non-const pointer to a const int
// equivalently: int const* pci = &i;
pci = &j; // OK: can modify non-const pointer
*pci = 42; // ERROR: cannot modify const pointee

int* const cpi = &i; // const pointer to a non-const int
cpi = &j; // ERROR: cannot modify const pointer
*cpi = 42; // OK: can modify non-const pointee

const int* const cpci = &i; // const pointer to a const int
// equivalently: int const* const cpci = &i;
cpci = &j; // ERROR: cannot modify const pointer
*cpci = 42; // ERROR: cannot modify const pointee

pci = pi; // OK: adds const to pointee
pi = pci; // ERROR: discards const from pointee
```

The `const` Qualifier and Reference Types

- reference is name that refers to object (i.e., referee)
- in principle, **`const`** qualifier can be applied to reference itself (i.e., top-level qualifier) or referee
- since reference cannot be rebound, reference itself is effectively always constant
- for this reason, does not make sense to explicitly apply **`const`** as top-level qualifier for reference type and language disallows this
- **`const`** qualifier can only be applied to referee

Example: `const` Qualifier and Reference Types

```
// with reference types, const can only be applied to referee
// reference itself cannot be rebound (i.e., is constant)
// referee may be const or non-const

int i = 0; const int ci = 0;
int i1 = 0; const int ci1 = 0;

// reference to non-const int
int& ri = i;
ri = ci; // OK: can modify non-const referee
int& ri = i1; // ERROR: cannot redefine/rebind reference

// reference to const int
const int& rci = ci;
rci = i; // ERROR: cannot modify const referee
const int& rci = ci1;
  // ERROR: cannot redefine/rebind reference

// ERROR: reference itself cannot be const qualified
int& const cri = i; // ERROR: invalid const qualifier

// ERROR: reference itself cannot be const qualified
const int& const crci = ci; // ERROR: invalid const qualifier
// also: int const& const crci = ci; // ERROR

const int& r1 = ci; // OK: adds const to referee
int& r2 = ci; // ERROR: discards const from referee
```

The `const` Qualifier and Pointer-to-Pointer Types

- for given type `T`, cannot implicitly convert `T**` to `const T**`
- although such conversion looks okay at first glance, actually would create backdoor for changing const objects
- can, however, implicitly convert `T**` to `const T* const*`
- for example, code like that shown below could be used to change const objects if `T**` to `const T**` were valid conversion:

```
const int i = 42;
int* p;
const int** q = &p;
  // Fortunately, this line is not valid code.
  // ERROR: cannot convert int** to const int**
*q = &i;
  // Change p (to which q points) to point to i.
  // OK: *q is not const (only **q is const)
*p = 0;
  // Set i (to which p points) to 0.
  // OK: *p is not const
  // This line would change i, which is const.
```

The **volatile** Qualifier

- **volatile** qualifier used to indicate that object can change due to agent *external to program* (e.g., memory-mapped device, signal handler)
- compiler cannot optimize away read and write operations on **volatile** objects (e.g., repeated reads without intervening writes cannot be optimized away)
- **volatile** qualifier typically used when object:
  - corresponds to register of memory-mapped device
  - may be modified by signal handler (namely, object of type **volatile** std::sig_atomic_t)
- example:

```
volatile int x;
volatile unsigned char* deviceStatus;
```

The `auto` Keyword

- in various contexts, **`auto`** keyword can be used as place holder for type
- in such contexts, implication is that compiler must deduce type
- example:

```
auto i = 3; // i has type int
auto j = i; // j has type int
auto& k = i; // k has type int&
const auto& n = i; // n has type const int&
auto x = 3.14; // x has type double
```

- very useful in generic programming (covered later) when types not always easy to determine
- can potentially save typing long type names
- can lead to more readable code (if well used)
- if overused, can lead to bugs (sometimes very subtle ones) and difficult to read code

Inline Variables

- **inline variable**: variable that may be defined in multiple translation units as long as all definitions are identical
- potential for multiple definitions avoided by having linker simply choose one of identical definitions and discard others (if more than one exists)
- can request that variable be made inline by including `inline` qualifier in variable declaration
- inline variable must have static storage duration (e.g., static class member or namespace-scope variable)
- inline variable typically used to allow definition of variable to be placed in header file without danger of multiple definitions
- inline variable has same address in all translation units

Inline Variable: Example

inline_variable_1_1.hpp

```
inline int magic = 42;
```

main.cpp

```
#include <iostream>
#include "inline_variable_1_1.hpp"
int main() {
    std::cout << magic << "\n";
}
```

other.cpp

```
#include "inline_variable_1_1.hpp"
void func() {/* ... */}
```

Section 2.3.3

Operators and Expressions

Arithmetic Operators

Operator Name	Syntax
addition	`a + b`
subtraction	`a - b`
unary plus	`+a`
unary minus	`-a`
multiplication	`a * b`
division	`a / b`
modulo (i.e., remainder)	`a % b`
pre-increment	`++a`
post-increment	`a++`
pre-decrement	`--a`
post-decrement	`a--`

Bitwise Operators

Operator Name	Syntax
bitwise NOT	`~a`
bitwise AND	`a & b`
bitwise OR	`a | b`
bitwise XOR	`a ^ b`
arithmetic left shift	`a << b`
arithmetic right shift	`a >> b`

Operators (Continued 1)

Assignment and
Compound-Assignment Operators

Operator Name	Syntax
assignment	`a = b`
addition assignment	`a += b`
subtraction assignment	`a -= b`
multiplication assignment	`a *= b`
division assignment	`a /= b`
modulo assignment	`a %= b`
bitwise AND assignment	`a &= b`
bitwise OR assignment	`a |= b`
bitwise XOR assignment	`a ^= b`
arithmetic left shift assignment	`a <<= b`
arithmetic right shift assignment	`a >>= b`

Operators (Continued 2)

Logical/Relational Operators

Operator Name	Syntax
equal	`a == b`
not equal	`a != b`
greater than	`a > b`
less than	`a < b`
greater than or equal	`a >= b`
less than or equal	`a <= b`
logical negation	`!a`
logical AND	`a && b`
logical OR	`a || b`

Member and Pointer Operators

Operator Name	Syntax
array subscript	`a[b]`
indirection	`*a`
address of	`&a`
member selection	`a.b`
member selection	`a->b`
member selection	`a.*b`
member selection	`a->*b`

Operators (Continued 3)

Other Operators

Operator Name	Syntax
function call	`a(...)`
comma	`a, b`
ternary conditional	`a ? b : c`
scope resolution	`a::b`
sizeof	**sizeof**`(a)`
parameter-pack sizeof	**sizeof...**`(a)`
alignof	**alignof**`(T)`
allocate storage	**new** `T`
allocate storage (array)	**new** `T[a]`
deallocate storage	**delete** `a`
deallocate storage (array)	**delete**`[] a`

Other Operators (Continued)

Operator Name	Syntax
type ID	**typeid**(a)
type cast	(T) a
const cast	**const_cast**<T>(a)
static cast	**static_cast**<T>(a)
dynamic cast	**dynamic_cast**<T>(a)
reinterpret cast	**reinterpret_cast**<T>(a)
throw	**throw** a
noexcept	**noexcept**(e)

Operator Precedence

Precedence	Operator	Name	Associativity
1	`::`	scope resolution	none
2	`.`	member selection (object)	left to right
	`->`	member selection (pointer)	
	`[]`	subscripting	
	`()`	function call	
	`++`	postfix increment	
	`--`	postfix decrement	

Operator Precedence (Continued 1)

Precedence	Operator	Name	Associativity
3	**sizeof**	size of object/type	right to left
	++	prefix increment	
	--	prefix decrement	
	~	bitwise NOT	
	!	logical NOT	
	-	unary minus	
	+	unary plus	
	&	address of	
	*	indirection	
	new	allocate storage	
	new[]	allocate storage (array)	
	delete	deallocate storage	
	delete[]	deallocate storage (array)	
	()	cast	

Operator Precedence (Continued 2)

Precedence	Operator	Name	Associativity
4	`.*`	member selection (objects)	left to right
	`->*`	member selection (pointers)	
5	`*`	multiplication	left to right
	`/`	division	
	`%`	modulus	
6	`+`	addition	left to right
	`-`	subtraction	
7	`<<`	left shift	left to right
	`>>`	right shift	
8	`<`	less than	left to right
	`<=`	less than or equal	
	`>`	greater than	
	`>=`	greater than or equal	
9	`==`	equality	left to right
	`!=`	inequality	

Operator Precedence (Continued 3)

Precedence	Operator	Name	Associativity
10	`&`	bitwise AND	left to right
11	`^`	bitwise XOR	left to right
12	`|`	bitwise OR	left to right
13	`&&`	logical AND	left to right
14	`||`	logical OR	left to right
15	`? :`	ternary conditional	right to left

Operator Precedence (Continued 4)

Precedence	Operator	Name	Associativity
16	`=`	assignment	right to left
	`*=`	multiplication assignment	
	`/=`	division assignment	
	`%=`	modulus assignment	
	`+=`	addition assignment	
	`-=`	subtraction assignment	
	`<<=`	left shift assignment	
	`>>=`	right shift assignment	
	`&=`	bitwise AND assignment	
	`|=`	bitwise OR assignment	
	`^=`	bitwise XOR assignment	
17	**`throw`**	throw exception	right to left
18	`,`	comma	left to right

Alternative Tokens

Alternative	Primary
`and`	`&&`
`bitor`	`|`
`or`	`||`
`xor`	`^`
`compl`	`~`
`bitand`	`&`
`and_eq`	`&=`
`or_eq`	`|=`
`xor_eq`	`^=`
`not`	`!`
`not_eq`	`!=`

- alternative tokens above probably best avoided as they lead to more verbose code

- An **expression** is a sequence of operators and operands that specifies a computation.
- An expression has a type and, if the type is not `void`, a value.
- A **constant expression** is an expression that can be evaluated at compile time (e.g., `1 + 1`).
- Example:

```
int x = 0;
int y = 0;
int* p = &x;
double d = 0.0;
// Evaluate some
// expressions here.
```

Expression	Type	Value
`x`	`int`	`0`
`y = x`	`int&`	reference to `y`
`x + 1`	`int`	`1`
`x * x + 2 * x`	`int`	`0`
`y = x * x`	`int&`	reference to `y`
`x == 42`	`bool`	`false`
`*p`	`int&`	reference to `x`
`p == &x`	`bool`	`true`
`x > 2 * y`	`bool`	`false`
`std::sin(d)`	`double`	`0.0`

Operator Precedence/Associativity Example

<table>
<tr><th>Expression</th><th>Fully-Parenthesized Expression</th></tr>
<tr><td><code>a + b + c</code></td><td><code>((a + b) + c)</code></td></tr>
<tr><td><code>a = b = c</code></td><td><code>(a = (b = c))</code></td></tr>
<tr><td><code>c = a + b</code></td><td><code>(c = (a + b))</code></td></tr>
<tr><td><code>d = a && !b || c</code></td><td><code>(d = ((a && (!b)) || c))</code></td></tr>
<tr><td><code>++*p++</code></td><td><code>(++(*(p++)))</code></td></tr>
<tr><td><code>a | ~b & c ^ d</code></td><td><code>(a | (((~b) & c) ^ d))</code></td></tr>
<tr><td><code>a[0]++ + a[1]++</code></td><td><code>(((a[0])++) + ((a[1])++))</code></td></tr>
<tr><td><code>a + b * c / d % - g</code></td><td><code>(a + (((b * c) / d) % (-g)))</code></td></tr>
<tr><td><code>++p[i]</code></td><td><code>(++(p[i]))</code></td></tr>
<tr><td><code>--*++p</code></td><td><code>(--(*(++p)))</code></td></tr>
<tr><td><code>a += b += c += d</code></td><td><code>(a += (b += (c += d)))</code></td></tr>
<tr><td><code>z = a == b ? ++c : --d</code></td><td><code>(z = ((a == b) ? (++c) : (--d)))</code></td></tr>
</table>

Short-Circuit Evaluation

- logical-and operator (i.e., `&&`):
  - groups left-to-right
  - result true if both operands are true, and false otherwise
  - second operand is ***not evaluated*** if first operand is false (in case of built-in logical-and operator)
- logical-or operator (i.e., `||`):
  - groups left-to-right
  - result is true if either operand is true, and false otherwise
  - second operand is ***not evaluated*** if first operand is true (in case of built-in logical-or operator)
- example:

```
int x = 0;
bool b = (x == 0 || ++x == 1);
// b equals true; x equals 0
b = (x != 0 && ++x == 1);
// b equals false; x equals 0
```

- above behavior referred to as short circuit evaluation

The `static_assert` Statement

- **`static_assert`** allows testing of boolean condition at compile time
- used to test sanity of code or test validity of assumptions made by code
- **`static_assert`** has two arguments:
  1. boolean constant expression (condition to test)
  2. string literal for error message to print if boolean expression not true
- second argument is optional
- failed static assertion results in compile error
- example:

```
static_assert(sizeof(int) >= 4, "int is too small");
static_assert(1 + 1 == 2, "compiler is buggy");
```

The `sizeof` Operator

- **`sizeof`** operator is used to query size of object or object type (i.e., amount of storage required)
- for object type `T`, `sizeof(T)` yields size of `T` in bytes (e.g., `sizeof(int)`, `sizeof(int[10])`)
- for expression `e`, `sizeof e` yields size of object required to hold result of `e` in bytes (e.g., `sizeof(&x)` where `x` is some object)
- `sizeof(char)`, `sizeof(signed char)`, and `sizeof(unsigned char)` guaranteed to be 1
- byte is at least 8 bits (usually exactly 8 bits except on more exotic platforms)

The `alignof` Operator

- object type can have restriction on address at which object of type can start called **alignment requirement**
- for given object type `T`, starting address for objects of type `T` must be integer multiple of N bytes, where integer N is called **alignment** of type
- alignment of 1 corresponds to no restriction on alignment (since starting address of object can be any address in memory)
- alignment of 2 restricts starting address of object to be even (i.e., integer multiple of 2)
- for efficiency reasons and due to restrictions imposed by hardware, alignment of particular type may be greater than 1
- **`alignof`** operator is used to query alignment of type
- for object type `T`, `alignof(T)` yields alignment used for objects of this type
- `alignof(char)`, `alignof(signed char)`, and `alignof(unsigned char)` guaranteed to be `1`
- fundamental types of size greater than 1 often have alignment greater than 1

- when declaring variable, can specify its alignment in memory with **`alignas`** specifier
- example:

```
alignas(4096) static char x[8192];
static_assert(alignof(x) == 4096);
  // x is aligned on 4096-byte boundary
alignas(double) float f;
static_assert(alignof(f) == alignof(double));
  // f has same alignment as double
```

The `constexpr` Qualifier for Variables

- **`constexpr`** qualifier indicates object has value that is ***constant expression*** (i.e., can be evaluated at compile time)
- **`constexpr`** implies **`const`** (but converse not necessarily true)
- following defines `x` as constant expression with type **`const int`** and value 42:

```
constexpr int x = 42;
```

- example:

```
constexpr int x = 42;
int y = 1;
x = 0; // ERROR: x is const
const int& x1 = x; // OK
const int* p1 = &x; // OK
int& x2 = x; // ERROR: x const, x2 not const
int* p2 = &x; // ERROR: x const, *p2 not const
int a1[x]; // OK: x is constexpr
int a2[y]; // ERROR: y is not constexpr
```

Section 2.3.4

Control-Flow Constructs: Selection and Looping

The `if` Statement

- allows conditional execution of code
- syntax has form:

  `if` (*expression*)
      $statement_1$
  `else`
      $statement_2$

- if expression *expression* is true, execute statement $statement_1$; otherwise, execute statement $statement_2$
- **`else`** clause can be omitted leading to simpler form:

  `if` (*expression*)
      $statement_1$

- conditional execution based on more than one condition can be achieved using construct like:

  `if` ($expression_1$)
      $statement_1$
  `else if` ($expression_2$)
      $statement_2$
  ...
  `else`
      $statement_n$

The **`if`** Statement (Continued 1)

- to include multiple statements in branch of **`if`**, ***must group statements*** into single statement using brace brackets

  if (*expression*) {
  $\quad statement_{1,1}$
  $\quad statement_{1,2}$
  $\quad statement_{1,3}$
  $\quad \ldots$
  } **else** {
  $\quad statement_{2,1}$
  $\quad statement_{2,2}$
  $\quad statement_{2,3}$
  $\quad \ldots$
  }

- advisable to ***always include brace brackets*** even when not necessary, as this avoids potential bugs caused by forgetting to include brackets later when more statements added to branch of **`if`**

The `if` Statement (Continued 2)

- if statement may include initializer:

  if (*initializer*; *expression*)
      $statement_1$;
  else
      $statement_2$;

- above construct equivalent to:

  {
      initializer;
      if (*expression*)
          $statement_1$;
      else
          $statement_2$;
  }

- if condition in if statement is constant expression, **`constexpr`** keyword can be added after **`if`** keyword to yield what is called constexpr-if statement
- constexpr-if statement is evaluated at compile time and branch of if statement that is not taken is discarded

The `if` Statement: Example

- example with **`else`** clause:

```
int x = someValue;
if (x % 2 == 0) {
    std::cout << "x is even\n";
} else {
    std::cout << "x is odd\n";
}
```

- example without **`else`** clause:

```
int x = someValue;
if (x % 2 == 0) {
    std::cout << "x is divisible by 2\n";
}
```

- example that tests for more than one condition:

```
int x = someValue;
if (x > 0) {
    std::cout << "x is positive\n";
} else if (x < 0) {
    std::cout << "x is negative\n";
} else {
    std::cout << "x is zero\n";
}
```

The `if` Statement: Example

- example with initializer:

```
int execute_command();
if (int ret = execute_command(); ret == 0) {
    std::cout << "command successful\n";
} else {
    std::cout << "command failed with status " <<
      ret << '\n';
}
```

- example constexpr-if statement:

```
constexpr int x = 10;
if constexpr (x < 0) {
    std::cout << "negative\n";
} else if constexpr(x > 0) {
    std::cout << "positive\n";
} else {
    std::cout << "zero\n";
}
```

The `switch` Statement

- allows conditional execution of code based on integral/enumeration value
- syntax has form:

  switch $(expression)$ {
  case $const\_expr_1$ **:**
  $\quad statements_1$
  case $const\_expr_2$ **:**
  $\quad statements_2$
  ...
  case $const\_expr_n$ **:**
  $\quad statements_n$
  default:
  $\quad statements$
  }

- $expression$ is expression of integral or enumeration type or implicitly convertible to such type; $const\_expr_i$ is constant expression of same type as $expression$ after conversions/promotions
- if expression $expression$ equals $const\_expr_i$, jump to beginning of statements $statements_i$; if expression $expr$ does not equal $const\_expr_i$ for any i, jump to beginning of statements $statements$
- then, continue executing statements until **`break`** statement is encountered

The `switch` Statement (Continued)

- switch statement can also include initializer:

  `switch` (*initializer*; *expression*)
  statement

- above construct equivalent to:

  {
  initializer;
  `switch` (*expression*)
  statement
  }

The `switch` Statement: Example

- example without initializer:

```
int x = someValue;
switch (x) {
case 0:
    // Note that there is no break here.
case 1:
    std::cout << "x is 0 or 1\n";
    break;
case 2:
    std::cout << "x is 2\n";
    break;
default:
    std::cout << "x is not 0, 1, or 2\n";
    break;
}
```

The `switch` Statement: Example (Continued)

- example with initializer:

```
int get_value();
switch (int x = get_value(); x) {
case 0:
case 1:
    std::cout << "x is 0 or 1\n";
    break;
case 2:
    std::cout << "x is 2\n";
    break;
default:
    std::cout << "x is not 0, 1, or 2\n";
    break;
}
```

The `while` Statement

- looping construct
- syntax has form:

$$\texttt{while}\ (\textit{expression})$$
$$\quad\textit{statement}$$

- if expression *expression* is true, statement *statement* is executed; this process repeats until expression *expression* becomes false
- to allow multiple statements to be executed in loop body, ***must group multiple statements*** into single statement with brace brackets

$$\begin{array}{l}\texttt{while}\ (\textit{expression})\ \{\\ \quad \textit{statement}_1\\ \quad \textit{statement}_2\\ \quad \textit{statement}_3\\ \quad \ldots\\ \}\end{array}$$

- advisable to ***always use brace brackets***, even when loop body consists of only one statement

The `while` Statement: Example

```
// print hello 10 times
int n = 10;
while (n > 0) {
    std::cout << "hello\n";
    --n;
}

// loop forever, printing hello
while (true) {
    std::cout << "hello\n";
}
```

The `for` Statement

- looping construct
- has following syntax:

  $\texttt{for}\ (\textit{statement}_1;\ \textit{expression};\ \textit{statement}_2)$
  $\quad\textit{statement}_3$

- first, execute statement $\textit{statement}_1$; then, while expression $\textit{expression}$ is true, execute statement $\textit{statement}_3$ followed by statement $\textit{statement}_2$
- $\textit{statement}_1$ and $\textit{statement}_2$ may be omitted; $\textit{expression}$ treated as **`true`** if omitted
- to include multiple statements in loop body, ***must group multiple statements*** into single statement using brace brackets; advisable to ***always use brace brackets***, even when loop body consists of only one statement:

  $\texttt{for}\ (\textit{statement}_1;\ \textit{expression};\ \textit{statement}_2)\ \{$
  $\quad\textit{statement}_{3,1}$
  $\quad\textit{statement}_{3,2}$
  $\quad\ldots$
  $\}$

- any objects declared in $\textit{statement}_1$ go out of scope as soon as **`for`** loop ends

The `for` Statement (Continued)

- consider **`for`** loop:

  `for` (*statement*$_1$; *expression* ; *statement*$_2$)
      statement$_3$

- above **`for`** loop can be equivalently expressed in terms of **`while`** loop as follows (except for behavior of **`continue`** statement, yet to be discussed):

  {
      statement$_1$;
      `while` (*expression*) {
          statement$_3$
          statement$_2$;
      }
  }

The `for` Statement: Example

- example with single statement in loop body:

```
// Print the integers from 0 to 9 inclusive.
for (int i = 0; i < 10; ++i)
    std::cout << i << '\n';
```

- example with multiple statements in loop body:

```
int values[10];
// ...
int sum = 0;
for (int i = 0; i < 10; ++i) {
    // Stop if value is negative.
    if (values[i] < 0) {
        break;
    }
    sum += values[i];
}
```

- example with error in assumption about scoping rules:

```
for (int i = 0; i < 10; ++i) {
    std::cout << i << '\n';
}
++i; // ERROR: i no longer exists
```

Range-Based for Statement

- variant of for loop for iterating over elements in range
- example:

```
int array[4] = {1, 2, 3, 4};
// Triple the value of each element in the array.
for (auto&& x : array) {
    x *= 3;
}
```

- range-based for loop nice in that it clearly expresses programmer intent (i.e., iterate over each element of collection)

The do Statement

- looping construct
- has following general syntax:

```
do
    statement
while (expression);
```

- statement *statement* executed;
  then, expression *expression* evaluated;
  if expression *expression* is true, entire process repeats from beginning
- to execute multiple statements in body of loop, must group multiple statements into single statement using brace brackets

```
do {
    statement1
    statement2
    ...
} while (expression);
```

- advisable to ***always use brace brackets***, even when loop body consists of only one statement

The do Statement: Example

- example with single statement in loop body:

```
// delay by looping 10000 times
int n = 0;
do
    ++n;
while (n < 10000);
```

- example with multiple statements in loop body:

```
// print integers from 0 to 9 inclusive
int n = 0;
do {
    std::cout << n << '\n';
    ++n;
} while (n < 10);
```

The **break** Statement

- **break** statement causes enclosing loop or switch to be terminated immediately
- example:

```
// Read integers from standard input until an
// error or end-of-file is encountered or a
// negative integer is read.
int x;
while (std::cin >> x) {
    if (x < 0) {
        break;
    }
    std::cout << x << '\n';
}
```

The `continue` Statement

- **`continue`** statement causes next iteration of enclosing loop to be started immediately
- example:

```
int values[10];
...
// Print the nonzero elements of the array.
for (int i = 0; i < 10; ++i) {
    if (values[i] == 0) {
        // Skip over zero elements.
        continue;
    }
    // Print the (nonzero) element.
    std::cout << values[i] << '\n';
}
```

The goto Statement

- **goto** statement transfers control to another statement specified by label
- should generally try to ***avoid use of goto statement***
- well written code rarely has legitimate use for **goto** statement
- example:

```
int i = 0;
loop: // label for goto statement
do {
    if (i == 3) {
        ++i;
        goto loop;
    }
    std::cout << i << '\n';
    ++i;
} while (i < 10);
```

- some restrictions on use of **goto** (e.g., cannot jump over initialization in same block as **goto**)

```
goto skip; // ERROR
int i = 0;
skip:
++i;
```

Section 2.3.5

Functions

Function Parameters, Arguments, and Return Values

- **argument** (a.k.a. **actual parameter**): argument is value supplied to function by caller; appears in parentheses of function-call operator
- **parameter** (a.k.a. **formal parameter**): parameter is object/reference declared as part of function that acquires value on entry to function; appears in function definition/declaration
- although abuse of terminology, parameter and argument often used interchangeably
- **return value**: result passed from function back to caller

```
int square(int i) { // i is parameter
    return i * i; // return value is i * i
}

void compute() {
    int i = 3;
    int j = square(i); // i is argument
}
```

Function Declarations and Definitions

- **function declaration** introduces identifier that names function and specifies following properties of function:
  - number of parameters
  - type of each parameter
  - type of return value (if not automatically deduced)
- example:

```
bool isOdd(int); // declare isOdd
bool isOdd(int x); // declare isOdd (x ignored)
```

- **function definition** provides all information included in function declaration as well as code for body of function
- example:

```
bool isOdd(int x) { // declare and define isOdd
    return x % 2;
}
```

Basic Syntax (Leading Return Type)

- most basic syntax for function declarations and definitions places return type at start (i.e., leading return-type syntax)
- basic syntax for function declaration:

  return_type function_name (*parameter_declarations*) ;

- examples of function declarations:

```
int min(int, int);
double square(double);
```

- basic syntax for function definition:

  return_type function_name (*parameter_declarations*)
  {
  statements
  }

- examples of function definitions:

```
int min(int x, int y) {return x < y ? x : y;}
double square(double x) {return x * x;}
```

Trailing Return-Type Syntax

- with trailing return-type syntax, return type comes after parameter declarations and **auto** used as placeholder for where return type would normally be placed
- trailing return-type syntax for function declaration:

  auto *function_name* (*parameter_declarations*) -> *return_type*;

- examples of function declarations:

```
auto min(int, int) -> int;
auto square(double) -> double;
```

- trailing return-type syntax for function definition:

  auto *function_name* (*parameter_declarations*) -> *return_type*
  {
  statements
  }

- examples of function definitions:

```
auto min(int x, int y) -> int
  {return x < y ? x : y;}
auto square(double x) -> double {return x * x;}
```

The **return** Statement

- **return** statement used to exit function, passing specified return value (if any) back to caller
- code in function executes until **return** statement is reached or execution falls off end of function
- if function return type is not **void**, **return** statement takes single parameter indicating value to be returned
- if function return type is **void**, function does not return any value and **return** statement takes either no parameter or expression of type **void**
- falling off end of function equivalent to executing **return** statement with no value
- example:

```
double unit_step(double x) {
    if (x >= 0.0) {
        return 1.0; // exit with return value 1.0
    }
    return 0.0; // exit with return value 0.0
}
```

Automatic Return-Type Deduction

- with both leading and trailing return-type syntax, can specify return type as **auto**
- in this case, return type of function will be automatically deduced
- if function definition has no **return** statement, return type deduced to be **void**
- otherwise, return type deduced to match type in expression of **return** statement or, if **return** statement has no expression, as **void**
- if multiple return statements, must use same type for all **return** expressions
- when return-type deduction used, function definition must be visible in order to call function (since return type cannot be determined otherwise)
- example:

```
auto square(double x) {
    return x * x;
      // x * x has type double
      // deduced return type is double

}
```

The `main` Function

- entry point to program is always function called `main`
- has return type of **`int`**
- can be declared to take either no arguments or two arguments as follows (although other possibilities may also be supported by implementation):

```
int main();
int main(int argc, char* argv[]);
```

- two-argument variant allows arbitrary number of C-style strings to be passed to program from environment in which program run
- `argc`: number of C-style strings provided to program
- `argv`: array of pointers to C-style strings
- `argv[0]` is name by which program invoked
- `argv[argc]` is guaranteed to be 0 (i.e., null pointer)
- `argv[1]`, `argv[2]`, ..., `argv[argc - 1]` typically correspond to command line options

The `main` Function (Continued)

- suppose that following command line given to shell:

```
program one two three
```

- `main` function would be invoked as follows:

```
int argc = 4;
char* argv[] = {
    "program", "one", "two", "three", 0
};
main(argc, argv);
```

- return value of `main` typically passed back to operating system
- can also use function `void exit(int)` to terminate program, passing integer return value back to operating system
- return statement in `main` is optional
- if control reaches end of `main` without encountering return statement, effect is that of executing "`return 0;`"

Lifetime

- **lifetime** of object is period of time in which object exists (e.g., block, function, global)

```
int x;

void wasteTime()
{
    int j = 10000;
    while (j > 0) {
        --j;
    }
    for (int i = 0; i < 10000; ++i) {
    }
}
```

- in above example: `x` global scope and lifetime; `j` function scope and lifetime; `i` block scope and lifetime

Parameter Passing

- function parameter can be passed by value or by reference
- **pass by value**: function given copy of object from caller
- **pass by reference**: function given reference to object from caller
- to pass parameter by reference, use ***reference type*** for parameter
- example:

```
void increment(int& x)
  // x is passed by reference
{
    ++x;
}

double square(double x)
  // x is passed by value
{
    return x * x;
}
```

Pass-By-Value Versus Pass-By-Reference

- if function needs to ***change value of object in caller***, must pass by reference
- for example:

```
void increment(int& x)
  // x refers to object in caller
{
    ++x;
}
```

- if object being passed to function is ***expensive to copy*** (e.g., a very large data type), always faster to pass by reference
- for example:

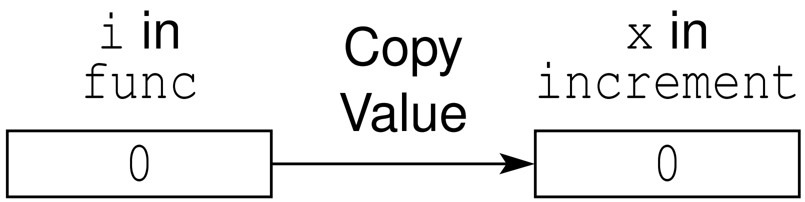
```
double compute(const std::vector<double>& x)
  // x refers to object in caller
  // object is not copied
{
    double result;
    // ... (initialize result with value computed from x)
    return result;
}
```

Increment Example: Incorrectly Using Pass By Value

- consider code:

```
void increment(int x) {
    ++x;
}

void func() {
    int i = 0;
    increment(i); // i is not modified
    // i is still 0
}
```

- when `func` calls `increment`, parameter passing copies value of `i` in `func` to local variable `x` in `increment`:



- when code in `increment` executes, local variable `x` is incremented (not `i` in `func`):

Increment Example: Correctly Using Pass By Reference

- consider code:

```
void increment(int& x) {
    ++x;
}

void func() {
    int i = 0;
    increment(i); // i is incremented
    // i is now 1
}
```

- when `func` calls `increment`, reference `x` in `increment` is bound to object `i` in `func` (i.e., `x` becomes alias for `i`):

`i` in `func`
and
`x` in `increment`

$\boxed{0}$

- when code in `increment` executes, `x` is incremented, which is alias for `i` in `func`:

`i` in `func`
and
`x` in `increment`

$\boxed{1}$

The `const` Qualifier and Functions

- `const` qualifier can be used in function declaration to make promises about what non-local objects will not be modified by function
- for function parameter of pointer type, const-ness of pointed-to object (i.e., pointee) extremely important
- if pointee is const, function promises not to change pointee; for example:

```
int strlen(const char*); // get string length
```

- for function parameter of reference type, const-ness of referred-to object (i.e., referee) extremely important
- if referee is const, function promises not to change referee; for example:

```
std::complex<double>
  square(const std::complex<double>&);
  // compute square of number
```

- not making appropriate choice of const-ness for pointed-to or referred-to object will result in ***fundamentally incorrect*** code
- if function will never modify pointee/referee associated with function parameter, parameter type should be made pointer/reference to const object

String Length Example: Not Const Correct

```
// ERROR: parameter type should be const char*
int string_length(char* s) {
    int n = 0;
    while (*s++ != '\0') {++n;}
    return n;
}

int main() {
    char buf[] = "Goodbye";
    const char* const m1 = "Hello";
    char* const m2 = &buf[0];
    int n1 = string_length(m1);
      // must copy argument m1 to parameter s:
      //    char* s = m1;
      // convert from const char* const to char*
      // ERROR: must discard const from pointee
    int n2 = string_length(m2);
      // must copy argument m2 to parameter s:
      //    char* s = m2;
      // convert from char* const to char*
      // OK: constness of pointee unchanged
}
```

String Length Example: Const Correct

```
// OK: pointee is const
int string_length(const char* s) {
    int n = 0;
    while (*s++ != '\0') {++n;}
    return n;
}

int main() {
    char buf[] = "Goodbye";
    const char* const m1 = "Hello";
    char* const m2 = &buf[0];
    int n1 = string_length(m1);
      // must copy argument m1 to parameter s:
      //    const char* s = m1;
      // convert from const char* const to const char*
      // OK: constness of pointee unchanged
    int n2 = string_length(m2);
      // must copy argument m2 to parameter s:
      //    const char* s = m2;
      // convert from char* const to const char*
      // OK: can add const to pointee
}
```

Square Example: Not Const Correct

```
#include <complex>

using Complex = std::complex<long double>;

// ERROR: parameter type should be reference to const
Complex square(Complex& z) {
    return z * z;
}

int main() {
    const Complex c1(1.0, 2.0);
    Complex c2(1.0, 2.0);
    Complex r1 = square(c1);
      // must bind parameter z to argument c1
      //    Complex& z = c1;
      // convert from const Complex to Complex&
      // ERROR: must discard const from referee
    Complex r2 = square(c2);
      // must bind parameter z to argument c2
      //    Complex& z = c2;
      // convert from Complex to Complex&
      // OK: constness of referee unchanged
}
```

Square Example: Const Correct

```
#include <complex>

using Complex = std::complex<long double>;

// OK: parameter type is reference to const
Complex square(const Complex& z) {
    return z * z;
}

int main() {
    const Complex c1(1.0, 2.0);
    Complex c2(1.0, 2.0);
    Complex r1 = square(c1);
      // must bind parameter z to argument c1
      //    const Complex& z = c1;
      // convert from const Complex to const Complex&
      // OK: constness of referee not discarded
    Complex r2 = square(c2);
      // must bind parameter z to argument c2
      //    const Complex& z = c2;
      // convert from Complex to const Complex&
      // OK: can add const to referee
}
```

Function Types and the `const` Qualifier

```
// top-level qualifiers of parameter types are
// not part of function type and should be omitted
// from function declaration

// BAD: const not part of function type
// (nothing here to which const can refer)
bool is_even(const unsigned int);

// OK
bool is_odd(unsigned int);

// OK: parameter with top-level const qualifier
// is ok in function definition
bool is_even(const unsigned int x) {
    // cannot change x in function
    return x % 2 == 0;
}

// OK
bool is_odd(unsigned int x) {
    // x can be changed if desired
    return x % 2 != 0;
}
```

Inline Functions

- in general programming sense, **inline function** is function for which compiler copies code from function definition directly into code of calling function rather than creating separate set of instructions in memory
- since code copied directly into calling function, no need to transfer control to separate piece of code and back again to caller, ***eliminating performance overhead*** of function call
- inline typically used for ***very short functions*** (where overhead of calling function is large relative to cost of executing code within function itself)
- can request function be made inline by including `inline` qualifier along with function return type (but compiler may ignore request)
- inline function must be defined in each translation unit in which function is used and all definitions must be identical; this is exception to one-definition rule
- example:

```
inline bool isEven(int x) {
    return x % 2 == 0;
}
```

Inlining of a Function

- inlining of `isEven` function transforms code fragment 1 into code fragment 2
- Code fragment 1:

```
inline bool isEven(int x) {
    return x % 2 == 0;
}

void myFunction() {
    int i = 3;
    bool result = isEven(i);
}
```

- Code fragment 2:

```
void myFunction() {
    int i = 3;
    bool result = (i % 2 == 0);
}
```

The `constexpr` Qualifier for Functions

- **`constexpr`** qualifier indicates return value of function is constant expression (i.e., can be evaluated at compile time) ***provided that all arguments to function are constant expressions***
- constexpr function required to be evaluated at compile time if all arguments are constant expressions and return value ***used in constant expression***
- constexpr functions are implicitly inline
- constexpr function very restricted in what it can do (e.g., no external state, can only call constexpr functions, variables must be initialized)
- example:

```
constexpr int factorial(int n) {
    return n >= 2 ? (n * factorial(n - 1)) : 1;
}

int u[factorial(5)];
  // OK: factorial(5) is constant expression

int x = 5;
int v[factorial(x)];
  // ERROR: factorial(x) is not constant
  // expression
```

Constexpr Function Example: square

```cpp
#include <iostream>

constexpr double square(double x) {
    return x * x;
}

int main() {
    constexpr double a = square(2.0);
      // must be computed at compile time

    double b = square(0.5);
      // might be computed at compile time

    double t;
    if (!(std::cin >> t)) {
        return 1;
    }
    const double c = square(t);
      // must be computed at run time

    std::cout << a << ' ' << b << ' ' << c << '\n';
}
```

Constexpr Function Example: `power_int` (Recursive)

```cpp
#include <iostream>

constexpr double power_int_helper(double x, int n) {
    return (n > 0) ? x * power_int_helper(x, n - 1) : 1;
}

constexpr double power_int(double x, int n) {
    return (n < 0) ? power_int_helper(1.0 / x, -n) :
      power_int_helper(x, n);
}

int main() {
    constexpr double a = power_int(0.5, 8);
      // must be computed at compile time

    double b = power_int(0.5, 8);
      // might be computed at compile time

    double x;
    if (!(std::cin >> x)) {return 1;}
    const double c = power_int(x, 2);
      // must be computed at run time

    std::cout << a << ' ' << b << ' ' << c << '\n';
}
```

Constexpr Function Example: `power_int` (Iterative)

```cpp
#include <iostream>

constexpr double power_int(double x, int n) {
    double result = 1.0;
    if (n < 0) {
        x = 1.0 / x;
        n = -n;
    }
    while (--n >= 0) {
        result *= x;
    }
    return result;
}

int main() {
    constexpr double a = power_int(0.5, 8);
      // must be computed at compile time

    double b = power_int(0.5, 8);
      // might be computed at compile time

    double x;
    if (!(std::cin >> x)) {return 1;}
    const double c = power_int(x, 2);
      // must be computed at run time

    std::cout << a << ' ' << b << ' ' << c << '\n';
}
```

Compile-Time Versus Run-Time Computation

- constexpr variables and constexpr functions provide mechanism for moving computation from run time to compile time
- benefits of compile-time computation include:
  1. no execution-time cost at run-time
  2. can facilitate compiler optimization (e.g., eliminate conditional branch if condition always true/false)
  3. can reduce code size since code used only for compile-time computation does not need to be included in executable
  4. can find errors at compile-time and link-time instead of at run time
  5. no concerns about order of initialization (which is not necessarily true for const objects)
  6. no synchronization concerns (e.g., multiple threads trying to initialize object)
- when floating point is involved, compile-time and run-time computations can yield different results, due to differences in such things as
  - rounding mode in effect
  - processor architecture used for computation (when cross compiling)

Function Overloading

- **function overloading**: multiple functions can have same name as long as they differ in number/type of their arguments
- example:

```
void print(int x) {
    std::cout << "int has value " << x << '\n';
}

void print(double x) {
    std::cout << "double has value " << x << '\n';
}

void demo() {
    int i = 5;
    double d = 1.414;
    print(i); // calls print(int)
    print(d); // calls print(double)
    print(42); // calls print(int)
    print(3.14); // calls print(double)
}
```

Default Arguments

- can specify default values for arguments to functions
- example:

```
// Compute log base b of x.
double logarithm(double x, double b) {
    return std::log(x) / std::log(b);
}

// Declaration of logarithm with a default argument.
double logarithm(double, double = 10.0);

void demo() {
    double x =
      logarithm(100.0); // calls logarithm(100.0, 10.0)
    double y =
      logarithm(4.0, 2.0); // calls logarithm(4.0, 2.0)
}
```

Argument Matching

- call of given function name chooses function that best matches actual arguments
- consider all functions in scope for which set of conversions exists so function could possibly be called
- best match is intersection of sets of functions that best match on each argument
- matches attempted in following order:
  1. exact match with zero or more trivial conversions (e.g., `T` to `T&`, `T&` to `T`, adding **`const`** and/or **`volatile`**); of these, those that do not add **`const`** and/or **`volatile`** to pointer/reference better than those that do
  2. match with promotions (e.g., **`int`** to **`long`**, **`float`** to **`double`**)
  3. match with standard conversions (e.g., **`float`** to **`int`**, **`double`** to **`int`**)
  4. match with user-defined conversions
  5. match with ellipsis
- if set of best matches contains exactly one element, this element chosen as function to call
- if set of best matches is either empty or contains more than one element, function call is invalid (since either no matches found or multiple equally-good matches found)

Argument Matching: Example

```
int max(int, int);
double max(double, double);

int i, j, k;
double a, b, c;

// ...
k = max(i, j);
  // best match on first argument: max(int, int)
  // best match on second argument: max(int, int)
  // best match: max(int, int)
  // OK: calls max(int, int)
c = max(a, b);
  // best match on first argument: max(double, double)
  // best match on second argument: max(double, double)
  // best match: max(double, double)
  // OK: calls max(double, double)
c = max(i, b);
  // best match on first argument: max(int, int)
  // best match on second argument: max(double, double)
  // best match: empty set
  // ERROR: ambiguous function call
```

The `assert` Macro

- `assert` macro allows testing of boolean condition at run time
- typically used to test sanity of code (e.g., test preconditions, postconditions, or other invariants) or test validity of assumptions made by code
- defined in header file `cassert`
- macro takes single argument: boolean expression
- if assertion fails, program is terminated by calling `std::abort`
- if `NDEBUG` preprocessor symbol is defined at time `cassert` header file included, all assertions are disabled (i.e., not checked)
- example:

```
#include <cassert>

double sqrt(double x) {
    assert(x >= 0);
    // ...
}
```

Section 2.3.6

Input/Output (I/O)

Basic I/O

- relevant declarations and such in header file `iostream`
- `std::istream`: stream from which characters/data can be read (i.e., input stream)
- `std::ostream`: stream to which characters/data can be written (i.e., output stream)
- `std::istream std::cin` standard input stream
- `std::ostream std::cout` standard output stream
- `std::ostream std::cerr` standard error stream
- in most environments, above three streams refer to user's terminal by default
- output operator (inserter) <<
- input operator (extractor) >>
- stream can be used as **`bool`** expression; converts to **`true`** if stream has not encountered any errors and **`false`** otherwise (e.g., if invalid data read or I/O error occurred)

Basic I/O Example

```
#include <iostream>

int main() {
    std::cout << "Enter an integer: ";
    int x;
    std::cin >> x;
    if (std::cin) {
        std::cout << "The integer entered was "
          << x << ".\n";
    } else {
        std::cerr <<
          "End-of-file reached or I/O error.\n";
    }
}
```

I/O Manipulators

- manipulators provide way to control formatting of data values written to streams as well as parsing of data values read from streams
- declarations related information for manipulators can be found in header files: `ios`, `iomanip`, `istream`, and `ostream`
- most manipulators used to control output formatting
- focus here on manipulators as they pertain to output
- manipulator may have ***immediate*** effect (e.g., `endl`), only affect ***next*** data value output (e.g., `setw`), or affect ***all*** subsequent data values output (e.g., `setprecision`)

I/O Manipulators (Continued)

Name	Description
`setw`	set field width
`setfill`	set fill character
`endl`	insert newline and flush
`flush`	flush stream
`dec`	use decimal
`hex`	use hexadecimal
`oct`	use octal
`showpos`	show positive sign
`noshowpos`	do not show positive sign
`left`	left align
`right`	right align
`fixed`	write floating-point values in fixed-point notation
`scientific`	write floating-point values in scientific notation
`setprecision`	for default notation, specify maximum number of meaningful digits to display before and after decimal point; for fixed and scientific notations, specify exactly how many digits to display after decimal point (padding with trailing zeros if necessary)

```
#include <iostream>
#include <ios>
#include <iomanip>

int main() {
    constexpr double pi = 3.1415926535;
    constexpr double big = 123456789.0;
    // default notation
    std::cout << pi << ' ' << big << '\n';
    // fixed-point notation
    std::cout << std::fixed << pi << ' ' << big << '\n';
    // scientific notation
    std::cout << std::scientific << pi << ' ' << big << '\n';
    // fixed-point notation with 7 digits after decimal point
    std::cout << std::fixed << std::setprecision(7) << pi << ' '
      << big << '\n';
    // fixed-point notation with precision and width specified
    std::cout << std::setw(8) << std::fixed << std::setprecision(2)
      << pi << ' ' << std::setw(20) << big << '\n';
    // fixed-point notation with precision, width, and fill specified
    std::cout << std::setw(8) << std::setfill('x') << std::fixed
      << std::setprecision(2) << pi << ' ' << std::setw(20) << big << '\n';
}

/* This program produces the following output:
3.14159 1.23457e+08
3.141593 123456789.000000
3.141593e+00 1.234568e+08
3.1415927 123456789.0000000
    3.14         123456789.00
xxxx3.14 xxxxxxxx123456789.00
*/
```

Section 2.3.7

Miscellany

Namespaces

- **namespace** is region that provides scope for identifiers declared inside
- namespace provides mechanism for reducing likelihood of naming conflicts
- syntax for namespace has general form:

  `namespace` *name* {

  body

  }

- *name*: identifier that names namespace
- *body*: body of namespace (i.e., code)
- all identifiers (e.g., names of variables, functions, and types) declared in *body* made to belong to scope associated with namespace *name*
- same identifier can be re-used in different namespaces, since each namespace is separate scope
- scope-resolution operator (i.e., `::`) can be used to explicitly specify namespace to which particular identifier belongs
- **`using`** statement can be used to bring identifiers from other namespaces into current scope

Namespaces: Example

```cpp
#include <iostream>

using std::cout; // bring std::cout into current scope

namespace mike {
    int someValue;
    void initialize() {
        cout << "mike::initialize called\n";
        someValue = 0;
    }
}

namespace fred {
    double someValue;
    void initialize() {
        cout << "fred::initialize called\n";
        someValue = 1.0;
    }
}

void func() {
    mike::initialize(); // call initialize in namespace mike
    fred::initialize(); // call initialize in namespace fred
    using mike::initialize;
      // bring mike::initialize into current scope
    initialize(); // call mike::initialize
}
```

Nested Namespace Definitions

- name given in namespace declaration can be qualified name in order to succinctly specify nested namespace
- consider following namespace declaration:

```
namespace foo {
    namespace bar {
        namespace impl {
            // ...
        }
    }
}
```

- preceding declaration can be written more succinctly as:

```
namespace foo::bar::impl {
    // ...
}
```

Namespace Aliases

- identifier can be introduced as alias for namespace
- syntax has following form:

  namespace *alias_name* = *ns_name*;

- identifier *alias_name* is alias for namespace *ns_name*
- namespace aliases particularly useful for creating short names for deeply-nested namespaces or namespaces with long names
- example:

```
#include <iostream>

namespace foobar {
    namespace miscellany {
        namespace experimental {
            int get_meaning_of_life() {return 42;}
            void greet() {std::cout << "hello\n";};
        }
    }
}

int main() {
    namespace n = foobar::miscellany::experimental;
    n::greet();
    std::cout << n::get_meaning_of_life() << '\n';
}
```

Inline Namespaces

- namespace can be made inline, in which case all identifiers in namespace also visible in enclosing namespace
- inline namespaces useful, for example, for library versioning
- example:

```
#include <cassert>

// some awesome library
namespace awesome {
    // version 1
    namespace v1 {
        int meaning_of_life() {return 41;}
    }
    // new and improved version 2
    // which should be default for library users
    inline namespace v2 {
        int meaning_of_life() {return 42;}
    }
}

int main() {
    assert(awesome::v1::meaning_of_life() == 41);
    assert(awesome::v2::meaning_of_life() == 42);
    assert(awesome::meaning_of_life() == 42);
}
```

Unnamed Namespaces

- can create unnamed namespace (i.e., namespace without name)
- unnamed namespace often referred to as anonymous namespace
- each translation unit may contain its own unique unnamed namespace
- entities defined in unnamed namespace only visible in its associated translation unit (i.e., has internal linkage)
- example:

```
#include <iostream>

namespace {
const int forty_two = 42;
int x;
}

int main() {
    x = forty_two;
    std::cout << x << '\n';
}
```

Memory Allocation: **new** and **delete**

- to allocate memory, use **new** statement
- to deallocate memory allocated with **new** statement, use **delete** statement
- similar to `malloc` and `free` in C
- two forms of allocation: 1) single object (i.e., nonarray case) and 2) array of objects
- array version of new/delete distinguished by `[]`
- example:

```
char* buffer = new char[64]; // allocate
                             // array of 64 chars
delete [] buffer; // deallocate array
double* x = new double; // allocate single double
delete x; // deallocate single object
```

- important to match nonarray and array versions of **new** and **delete**:

```
char* buffer = new char[64]; // allocate
delete buffer; // ERROR: nonarray delete to
               // delete array
               // may compile fine, but crash
```

User-Defined Literals

- C++ has several categories of literals (e.g., character, integer, floating-point, string, boolean, and pointer)
- can define additional literals based on these categories
- identifier used as suffix for user-defined literal must begin with underscore
- suffixes that do not begin with underscore are reserved for use by standard library
- example:

```
#include <iostream>
#include <complex>

std::complex<long double> operator "" _i(long double d) {
    return std::complex<long double>(0.0, d);
}

int main() {
    auto z = 3.14_i;
    std::cout << z << '\n';
}

// Program output:
// (0,3.14)
```

Attributes

- attributes provide unified syntax for implementation-defined language extensions
- attribute can be used almost anywhere in source code and can be applied to almost anything (e.g., types, variables, functions, names, code blocks, and translation units)
- specific types of entities to which attribute can be applied depends on particular attribute in question
- attribute specifiers start with two consecutive left brackets and continue to two consecutive right brackets
- example:

```
[[deprecated]]
void some_very_old_function() {/* ... */};
```

Some Standard Attributes

Name	Description
noreturn	function does not return
deprecated	use of entity is deprecated (i.e., allowed but discouraged)
fallthrough	fall through in switch statement is deliberate
maybe_unused	entity (e.g., variable) may be unused
nodiscard	used to indicate that return value of function should not be ignored

Some GCC and Clang Attributes

GCC C++ Compiler

Name	Description
gnu::noinline	do not inline function
gnu::no_sanitize_address	do not instrument function for address sanitizer
gnu::no_sanitize_undefined	do not instrument function for undefined-behavior sanitizer

Clang C++ Compiler

Name	Description
gnu::noinline	do not inline function
clang::no_sanitize	do not instrument function for sanitizer

Section 2.3.8

References

References I

1. D. Saks. Placing const in declarations.
   Embedded Systems Programming, pages 19–20, June 1998.
2. D. Saks. What const really means.
   Embedded Systems Programming, pages 11–14, Aug. 1998.
3. D. Saks. const T vs. T const.
   Embedded Systems Programming, pages 13–16, Feb. 1999.
4. D. Saks. Top-level cv-qualifiers in function parameters.
   Embedded Systems Programming, pages 63–65, Feb. 2000.

Section 2.4

Classes

Classes

- since fundamental types provided by language are quite limiting, language provides mechanism for defining new (i.e., user-defined) types
- **class** is user-defined type
- class specifies:
  1. how objects of class are ***represented***
  2. ***operations*** that can be performed on objects of class
- not all parts of class are directly accessible to all code
- **interface** is part of class that is directly accessible to its users
- **implementation** is part of class that its users access only indirectly through interface

Section 2.4.1

Members and Access Specifiers

Class Members

- class consists of ***zero or more members***
- three basic kinds of members (excluding enumerators):
  1. data member
  2. function member
  3. type member
- **data members** define representation of class object
- **function members** (also called member functions) provide operations on such objects
- **type members** specify any types associated with class

Access Specifiers

- can control *level of access* that users of class have to its members
- three levels of access:
  1. public
  2. protected
  3. private
- **public**: member can be accessed by any code
- **private**: member can only be accessed by other members of class and friends of class (to be discussed shortly)
- **protected**: relates to inheritance (discussion deferred until later)
- public members constitute class interface
- private members constitute class implementation

Class Example

- class typically has form:

```
class Widget // The class is named Widget.
{
public:
    // public members
    // (i.e., the interface to users)
    // usually functions and types (but not data)
private:
    // private members
    // (i.e., the implementation details only
    // accessible by members of class)
    // usually functions, types, and data
};
```

Default Member Access

- class members are private by default
- two code examples below are exactly equivalent:

```
class Widget {
    // ...
};
```

```
class Widget {
private:
    // ...
};
```

The struct Keyword

- **struct** is class where members public by default
- two code examples below are exactly equivalent:

```
struct Widget {
    // ...
};
```

```
class Widget {
public:
    // ...
};
```

- class example:

```
class Vector_2 { // Two-dimensional vector class.
public:
    double x; // The x component of the vector.
    double y; // The y component of the vector.
};

void func() {
    Vector_2 v;
    v.x = 1.0; // Set data member x to 1.0
    v.y = 2.0; // Set data member y to 2.0
}
```

- above class has data members `x` and `y`
- members accessed by ***member-selection operator*** (i.e., “.”)

Function Members

- class example:

```
class Vector_2 { // Two-dimensional vector class.
public:
    void initialize(double newX, double newY);
    double x; // The x component of the vector.
    double y; // The y component of the vector.
};

void Vector_2::initialize(double newX, double newY) {
    x = newX; // "x" means "this->x"
    y = newY; // "y" means "this->y"
}

void func() {
    Vector_2 v; // Create Vector_2 called v.
    v.initialize(1.0, 2.0); // Initialize v to (1.0, 2.0).
}
```

- above class has member function `initialize`
- to refer to member of class outside of class body must use ***scope-resolution operator*** (i.e., `::`)
- for example, in case of `initialize` function, we use `Vector_2::initialize`
- member function always has ***implicit parameter*** referring to class object

The `this` Keyword

- member function always has ***implicit parameter*** referring to class object
- implicit parameter accessible inside member function via **`this`** keyword
- **`this`** is pointer to object for which member function is being invoked
- data members can be accessed through **`this`** pointer
- since data members can also be referred to directly by their names, explicit use of **`this`** often not needed and normally avoided
- example:

```
class Widget {
public:
    int updateValue(int newValue) {
        int oldValue = value; // "value" means "this->value"
        value = newValue; // "value" means "this->value"
        return oldValue;
    }
private:
    int value;
};

void func() {
    Widget x;
    x.updateValue(5);
      // in Widget::updateValue, variable this equals &x
}
```

`const` Member Functions

- member function has reference to object of class as implicit parameter (i.e., object pointed to by **`this`**)
- need way to indicate if member function can change value of object
- **`const`** member function cannot change value of object

```
class Counter {
public:
    int getCount() const
      {return count;} // count means this->count
    void setCount(int newCount)
      {count = newCount;} // count means this->count
    void incrementCount()
      {++count;} // count means this->count
private:
    int count; // counter value
};

void func() {
    Counter ctr;
    ctr.setCount(0);
    int count = ctr.getCount();
    const Counter& ctr2 = ctr;
    count = ctr2.getCount(); // getCount better be const!
}
```

Definition of Function Members in Class Body

- member function whose definition is provided in body of class is automatically **`inline`**
- two code examples below are exactly equivalent:

```
class MyInteger {
public:
    // Set the value of the integer and return the old value.
    int setValue(int newValue) {
        int oldValue = value;
        value = newValue;
        return oldValue;
    }
private:
    int value;
};
```

```
class MyInteger {
public:
    // Set the value of the integer and return the old value.
    int setValue(int newValue);
private:
    int value;
};

inline int MyInteger::setValue(int newValue) {
    int oldValue = value;
    value = newValue;
    return oldValue;
}
```

Type Members

- example:

```
class Point_2 { // Two-dimensional point class.
public:
    using Coordinate = double; // Coordinate type.
    Coordinate x; // The x coordinate of the point.
    Coordinate y; // The y coordinate of the point.
};

void func() {
    Point_2 p;
    // ...
    Point_2::Coordinate x = p.x;
      // Point_2::Coordinate same as double
}
```

- above class has type member `Coordinate`
- to refer to type member outside of class body, we must use ***scope-resolution operator*** (i.e., `::`)

Friends

- normally, only class has access to its private members
- sometimes, necessary to allow another class or function to have access to private members of class
- friend of class is function/class that is allowed to access private members of class
- to make function or class friend of another class, use **`friend`** statement
- example:

```
class Gadget; // forward declaration of Gadget

class Widget {
    // ...
    friend void myFunc();
      // function myFunc is friend of Widget
    friend class Gadget;
      // class Gadget is friend of Widget
    // ...
};
```

- generally, use of friends should be avoided except when absolutely necessary

Class Example

```
class Widget {
public:
    int setValue(int newValue) { // member function
        int oldValue = value; // save old value
        value = newValue; // change value to new value
        return oldValue; // return old value
    }
private:
    friend void wasteTime();
    void doNothing() {}
    int value; // data member
};

void wasteTime() {
    Widget x;
    x.doNothing(); // OK: friend
    x.value = 5; // OK: friend
}

void func() {
    Widget x; // x is object of type Widget
    x.setValue(5); // call Widget's setValue member
      // sets x.value to 5
    x.value = 5; // ERROR: value is private
    x.doNothing(); // ERROR: doNothing is private
}
```

Section 2.4.2

Constructors and Destructors

Propagating Values: Copying and Moving

- Suppose that we have two objects of the same type and we want to propagate the value of one object (i.e., the source) to the other object (i.e., the destination).
- This can be accomplished in one of two ways: 1) copying or 2) moving.
- **Copying** propagates the value of the source object to the destination object ***without modifying the source object***.
- **Moving** propagates the value of the source object to the destination object and is ***permitted to modify the source object***.
- Moving is always at least as efficient as copying, and for many types, moving is ***more efficient*** than copying.
- For some types, ***copying does not make sense***, while moving does (e.g., `std::ostream`, `std::istream`).

Copying and Moving

- ***Copy operation.*** Propagating the value of the source object `source` to the destination object `destination` by copying.

Before Copy After Copy

- A copy operation ***does not modify*** the value of the source object.

- ***Move operation.*** Propagating the value of the source object `source` to the destination object `destination` by moving.

Before Move After Move

- A move operation is ***not guaranteed to preserve*** the value of the source object. After the move operation, the value of the source object is ***unknown*** (i.e., unspecified but valid).

Constructors

- when new object created usually desirable to immediately initialize it to some known state
- prevents object from accidentally being used before it is initialized
- **constructor** is member function that is ***called automatically*** when object created in order to ***initialize*** its value
- constructor has ***same name as class*** (i.e., constructor for class `T` is function `T::T`)
- constructor has ***no return type*** (not even **`void`**)
- constructor ***cannot be called directly*** (although placement new provides mechanism for achieving similar effect, in rare cases when needed)
- constructor ***can be overloaded***
- before constructor body is entered, all data members of class type are first constructed
- in certain circumstances, constructors may be automatically provided
- sometimes, automatically provided constructors ***will not*** have correct behavior

Default Constructor

- constructor that can be called with no arguments known as **default constructor**
- if ***no constructors*** specified, default constructor ***automatically provided*** that calls default constructor for each data member of class type (does nothing for data member of built-in type)

```
class Vector { // Two-dimensional vector class.
public:
    Vector() // Default constructor.
      {x_ = 0.0; y_ = 0.0;}
    // ...
private:
    double x_; // The x component of the vector.
    double y_; // The y component of the vector.
};

Vector v; // calls Vector::Vector(); v set to (0,0)
Vector x(); // declares function x that returns Vector
```

Copy Constructor

- for class `T`, constructor taking lvalue reference to `T` as first parameter that can be called with one argument known as **copy constructor**
- used to ***create*** object by ***copying*** from already-existing object
- copy constructor for class `T` typically is of form `T(const T&)`
- if ***no copy constructor*** specified (and no move constructor or move assignment operator specified), copy constructor is ***automatically provided*** that copies each data member (using copy constructor for class and bitwise copy for built-in type)

```
class Vector { // Two-dimensional vector class.
public:
    Vector() {x_ = 0.0; y_ = 0.0;} // Default constructor
    Vector(const Vector& v) // Copy constructor.
      {x_ = v.x_; y_ = v.y_;}
    // ...
private:
    double x_; // The x component of the vector.
    double y_; // The y component of the vector.
};

Vector v;
Vector w(v); // calls Vector::Vector(const Vector&)
Vector u = v; // calls Vector::Vector(const Vector&)
```

Move Constructor

- for class `T`, constructor taking rvalue reference to `T` as first parameter that can be called with one argument known as **move constructor**
- used to ***create*** object by ***moving*** from already-existing object
- move constructor for class `T` typically is of form `T(T&&)`
- if ***no move constructor*** specified (and no destructor, copy constructor, or copy/move assignment operator specified), move constructor is ***automatically provided*** that moves each data member (using move for class and bitwise copy for built-in type)

```
class Vector { // Two-dimensional vector class.
public:
    Vector() {x_ = 0.0; y_ = 0.0;} // Default constructor
    Vector(Vector&& v) {x_ = v.x_; y_ = v.y_;} // Move constructor.
    // ...
private:
    double x_; // The x component of the vector.
    double y_; // The y component of the vector.
};
```

```
#include <utility>
Vector v;
Vector w(std::move(v)); // calls Vector::Vector(Vector&&)
Vector x = std::move(w); // calls Vector::Vector(Vector&&)
```

Constructor Example

```
class Vector { // Two-dimensional vector class.
public:
    // Default constructor.
    Vector() {x_ = 0.0; y_ = 0.0;}
    // Copy constructor.
    Vector(const Vector& v) {x_ = v.x_; y_ = v.y_;}
    // Move constructor.
    Vector(Vector&& v) {x_ = v.x_; y_ = v.y_;}
    // Another constructor.
    Vector(double x, double y) {x_ = x; y_ = y;}
    // ...
private:
    double x_; // The x component of the vector.
    double y_; // The y component of the vector.
};
```

- four constructors provided

Constructor Example (Continued 1)

```
// include definition of Vector class here

int main() {
    Vector u;
      // calls default constructor
    Vector v(1.0, 2.0);
      // calls Vector::Vector(double, double)
    Vector w(v);
      // calls copy constructor
    Vector x = u;
      // calls copy constructor
    Vector y = Vector(1.0, 0.0);
      // guaranteed copy/move elision
      // calls Vector::Vector(double, double), directly
      // constructing new object in y
      // does not call move constructor
    Vector z(Vector());
      // guaranteed copy/move elision
      // calls default constructor, directly constructing
      // new object in z
      // does not call move constructor
    Vector f();
      // declares function f that returns Vector
}
```

Constructor Example (Continued 2)

```
#include <utility>
#include <cstdlib>
// include definition of Vector class here

// named RVO not possible
Vector func1() {
    Vector a(1.0, 0.0);
    Vector b(0.0, 1.0);
    if (std::rand() % 2) {return a;}
    else {return b;}
}

// RVO required
Vector func2() {return Vector(1.0, 1.0);}

int main() {
    Vector u(1.0, 1.0);
    Vector v(std::move(u));
      // move constructor invoked to propagate value from u
      // to v
    Vector w = func1();
      // move constructor invoked to propagate value of object
      // in return statement of func1 to object w in main
      // (named RVO not possible)
    Vector x = func2();
      // move constructor not invoked, due to guaranteed
      // copy/move elision (return value of func2 directly
      // constructed in object x in main)
}
```

Initializer Lists

- in constructor of class, often we want to control which constructor is used to initialize each data member
- since all data members are constructed ***before*** body of constructor is entered, this cannot be controlled inside body of constructor
- to allow control over which constructors are used to initialize individual data members, mechanism called **initializer lists** provided
- initializer list forces specific constructors to be used to initialize individual data members before body of constructor is entered
- data members always initialized in ***order of declaration***, regardless of order in initializer list

Initializer List Example

```
class ArrayDouble { // array of doubles class
public:
    ArrayDouble(); // create empty array
    ArrayDouble(int size); // create array of specified size
    // ...
private:
    // ...
};

class Vector { // n-dimensional real vector class
public:
    Vector(int size) : data_(size) {}
    // force data_ to be constructed with
    // ArrayDouble::ArrayDouble(int)
    // ...
private:
    ArrayDouble data_; // elements of vector
};
```

Destructors

- when object reaches end of lifetime, typically some cleanup required before object passes out of existence
- **destructor** is member function that is ***automatically called*** when object reaches end of lifetime in order to perform any necessary cleanup
- often object may have allocated resources associated with it (e.g., memory, files, devices, network connections, processes/threads)
- when object destroyed, must ensure that any resources associated with object are released
- destructors often serve to ***release resources*** associated with object
- destructor for class `T` always has ***name*** `T::~T`
- destructor has ***no return type*** (not even **`void`**)
- destructor ***cannot be overloaded***
- destructor always takes ***no parameters***
- if ***no destructor*** is specified, destructor ***automatically provided*** that calls destructor for each data member of class type
- sometimes, automatically provided destructor ***will not*** have correct behavior

Destructor Example

- example:

```
class Widget {
public:
    Widget(int bufferSize) { // Constructor.
        // allocate some memory for buffer
        bufferPtr_ = new char[bufferSize];
    }
    ~Widget() { // Destructor.
        // free memory previously allocated
        delete [] bufferPtr_;
    }
    // copy constructor, assignment operator, ...
private:
    char* bufferPtr_; // pointer to start of buffer
};
```

- without explicitly-provided destructor (i.e., with destructor automatically provided by compiler), memory associated with `bufferPtr_` would not be freed

Section 2.4.3

Operator Overloading

Operator Overloading

- can specify meaning of operator whose operands are one or more user-defined types through process known as **operator overloading**
- operators that can be overloaded:

<table>
<tr><td>arithmetic</td><td><code>+ - * / %</code></td></tr>
<tr><td>bitwise</td><td><code>^ & | ~ << >></code></td></tr>
<tr><td>logical</td><td><code>! && ||</code></td></tr>
<tr><td>relational</td><td><code>< > <= >= == !=</code></td></tr>
<tr><td>assignment</td><td><code>=</code></td></tr>
<tr><td>compound assignment</td><td><code>+= -= *= /= %= ^= &= |= <<= >>=</code></td></tr>
<tr><td>increment/decrement</td><td><code>++ --</code></td></tr>
<tr><td>subscript</td><td><code>[]</code></td></tr>
<tr><td>function call</td><td><code>()</code></td></tr>
<tr><td>address, indirection</td><td><code>& *</code></td></tr>
<tr><td>others</td><td><code>->* , -></code> <code>new delete</code></td></tr>
</table>

- not possible to change precedence/associativity or syntax of operators
- meaning of operator specified by specially named function

Operator Overloading (Continued 1)

- operator `@` overloaded via special function named `operator@`
- with some exceptions, operator can be overloaded as member function or nonmember function
- if operator overloaded as member function, first operand provided as `*this` and remaining operands, if any, provided as function parameters
- if operator overloaded as nonmember function, all operands provided as function parameters
- postfix unary (increment/decrement) operators take additional dummy parameter of type `int` in order to distinguish from prefix case
- expressions involving overloaded operators interpreted as follows:

		Interpretation As	
Type	Expression	Member Function	Nonmember Function
Binary	`a@b`	`a.operator@(b)`	`operator@(a, b)`
Prefix unary	`@a`	`a.operator@()`	`operator@(a)`
Postfix unary	`a@`	`a.operator@(i)`	`operator@(a, i)`

`i` is dummy parameter of type `int`

Operator Overloading (Continued 2)

- assignment, function-call, subscript, and member-selection operators must be overloaded as member functions
- if member and nonmember functions both defined, argument matching rules determine which is called
- if first operand of overloaded operator not object of class type, must use nonmember function
- for most part, operators can be defined quite arbitrarily for user-defined types
- for example, no requirement that "`++x`", "`x += 1`", and "`x = x + 1`" be equivalent
- of course, probably not advisable to define operators in very counterintuitive ways, as will inevitably lead to bugs in code

- some examples showing how expressions translated into function calls are as follows:

Expression	Member Function	Nonmember Function
`y = x`	`y.operator=(x)`	—
`y += x`	`y.operator+=(x)`	`operator+=(y, x)`
`x + y`	`x.operator+(y)`	`operator+(x, y)`
`++x`	`x.operator++()`	`operator++(x)`
`x++`	`x.operator++(int)`	`operator++(x, int)`
`x == y`	`x.operator==(y)`	`operator==(x, y)`
`x < y`	`x.operator<(y)`	`operator<(x, y)`

Operator Overloading Example: Vector

```
class Vector { // Two-dimensional vector class
public:
    Vector() : x_(0.0), y_(0.0) {}
    Vector(double x, double y) : x_(x), y_(y) {}
    double x() const { return x_; }
    double y() const { return y_; }
private:
    double x_; // The x component
    double y_; // The y component
};

// Vector addition
Vector operator+(const Vector& u, const Vector& v)
  {return Vector(u.x() + v.x(), u.y() + v.y());}

// Dot product
double operator*(const Vector& u, const Vector& v)
  {return u.x() * v.x() + u.y() * v.y();}

void func() {
    Vector u(1.0, 2.0);
    Vector v(u);
    Vector w;
    w = u + v; // w.operator=(operator+(u, v))
    double c = u * v; // calls operator*(u, v)
      // since c is built-in type, assignment operator
      // does not require function call
}
```

Operator Overloading Example: Array10

```
class Array10 { // Ten-element real array class
public:
    Array10() {
        for (int i = 0; i < 10; ++i) { // Zero array
            data_[i] = 0;
        }
    }
    const double& operator[](int index) const {
        return data_[index];
    }
    double& operator[](int index) {
        return data_[index];
    }
private:
    double data_[10]; // array data
};

void func() {
    Array10 v;
    v[1] = 3.5; // calls Array10::operator[](int)
    double c = v[1]; // calls Array10::operator[](int)
    const Array10 u;
    u[1] = 2.5; // ERROR: u[1] is const
    double d = u[1]; // calls Array10::operator[](int) const
}
```

Operator Overloading: Member vs. Nonmember Functions

- some considerations: access to private members; whether first operand has class type

```
class Complex { // Complex number type.
public:
    Complex(double x, double y) : x_(x), y_(y) {}
    double real() const {return x_;}
    double imag() const {return y_;}

    // Alternatively, overload as a member function.
    // Complex operator+(double b) const
    //  {return Complex(real() + b, imag());}
private:
    double x_; // The real part.
    double y_; // The imaginary part.
};

// Overload as a nonmember function.
// (A member function could instead be used. See above.)
Complex operator+(const Complex& a, double b)
  {return Complex(a.real() + b, a.imag());}

// This can only be accomplished with a nonmember function.
Complex operator+(double b, const Complex& a)
  {return Complex(b + a.real(), a.imag());}

void myFunc() {
    Complex a(1.0, 2.0);
    Complex b(1.0, -2.0);
    double r = 2.0;
    Complex c = a + r; // could use nonmember or member function
                       // operator+(a, r) or a.operator+(r)
    Complex d = r + a; // must use nonmember function
                       // operator+(r, a)
                       // since r.operator+(a) will not work
}
```

Copy Assignment Operator

- for class `T`, `T::operator=` having exactly one parameter that is lvalue reference to `T` known as **copy assignment operator**
- used to assign, to already-existing object, value of another object by ***copying***
- if no copy assignment operator specified (and no move constructor or move assignment operator specified), copy assignment operator ***automatically provided*** that copy assigns to each data member (using data member's copy assignment operator for class and bitwise copy for built-in type)
- copy assignment operator for class `T` typically is of form `T& operator=(const T&)` (returning reference to `*this`)
- copy assignment operator returns (nonconstant) reference in order to allow for statements like following to be valid (where `x`, `y`, and `z` are of type `T` and `T::modify` is a non-const member function):

```
x = y = z; // x.operator=(y.operator=(z))
(x = y) = z; // (x.operator=(y)).operator=(z)
(x = y).modify(); // (x.operator=(y)).modify()
```

- be careful to correctly consider case of self-assignment

Self-Assignment Example

- in practice, self assignment typically occurs when references (or pointers) are involved
- example:

```
void doSomething(SomeType& x, SomeType& y) {
    x = y; // self assignment if &x == &y
    // ...
}

void myFunc() {
    SomeType z;
    // ...
    doSomething(z, z); // results in self assignment
    // ...
}
```

Move Assignment Operator

- for class `T`, `T::operator=` having exactly one parameter that is rvalue reference to `T` known as **move assignment operator**
- used to assign, to already-existing object, value of another object by ***moving***
- if no move assignment operator specified (and no destructor, copy/move constructor, or copy assignment operator specified), move assignment operator ***automatically provided*** that move assigns to each data member (using move for class and bitwise copy for built-in type)
- move assignment operator for class `T` typically is of form `T& operator=(T&&)` (returning reference to `*this`)
- move assignment operator returns (nonconstant) reference for same reason as in case of copy assignment operator
- self-assignment should probably not occur in move case (but might be prudent to protect against "insane" code with assertion) (library effectively forbids self-assignment for move)

Copy/Move Assignment Operator Example: `Complex`

```
class Complex {
public:
    Complex(double x = 0.0, double y = 0.0) :
      x_(x), y_(y) {}
    Complex(const Complex& a) : x_(a.x_), y_(a.y_) {}
    Complex(Complex&& a) : x_(a.x_), y_(a.y_) {}
    Complex& operator=(const Complex& a) { // Copy assign
        if (this != &a) {
            x_ = a.x_; y_ = a.y_;
        }
        return *this;
    }
    Complex& operator=(Complex&& a) { // Move assign
        x_ = a.x_; y_ = a.y_;
        return *this;
    }
private:
    double x_; // The real part.
    double y_; // The imaginary part.
};

int main() {
    Complex z(1.0, 2.0);
    Complex v(1.5, 2.5);
    v = z; // v.operator=(z)
    v = Complex(0.0, 1.0); // v.operator=(Complex(0.0, 1.0))
}
```

Assignment Operator Example: Buffer

```
class Buffer { // Character buffer class.
public:
    Buffer(int bufferSize) { // Constructor.
        bufSize_ = bufferSize;
        bufPtr_ = new char[bufferSize];
    }
    Buffer(const Buffer& buffer) { // Copy constructor.
        bufSize_ = buffer.bufSize_;
        bufPtr_ = new char[bufSize_];
        for (int i = 0; i < bufSize_; ++i)
            bufPtr_[i] = buffer.bufPtr_[i];
    }
    ~Buffer() { // Destructor.
        delete [] bufPtr_;
    }
    Buffer& operator=(const Buffer& buffer) { // Copy assignment operator.
        if (this != &buffer) {
            delete [] bufPtr_;
            bufSize_ = buffer.bufSize_;
            bufPtr_ = new char[bufSize_];
            for (int i = 0; i < bufSize_; ++i)
                bufPtr_[i] = buffer.bufPtr_[i];
        }
        return *this;
    }
    // ...
private:
    int bufSize_; // buffer size
    char* bufPtr_; // pointer to start of buffer
};
```

- without explicitly-provided assignment operator (i.e., with assignment operator automatically provided by compiler), memory leaks and memory corruption would result

Section 2.4.4

Miscellany

`std::initializer_list` Class Template

- class template `std::initializer_list` provides lightweight list type
- in order to use `initializer_list`, need to include header file `initializer_list`
- declaration:

```
template <class T> initializer_list;
```

- `T` is type of elements in list
- `initializer_list` is very lightweight
- can query number of elements in list and obtain iterators to access these elements
- `initializer_list` often useful as parameter type for constructor

std::initializer_list Example

```
#include <iostream>
#include <vector>

class Sequence {
public:
    Sequence(std::initializer_list<int> list) {
        for (std::initializer_list<int>::const_iterator i =
          list.begin(); i != list.end(); ++i)
            elements_.push_back(*i);
    }
    void print() const {
        for (std::vector<int>::const_iterator i =
          elements_.begin(); i != elements_.end(); ++i)
            std::cout << *i << '\n';
    }
private:
    std::vector<int> elements_;
};

int main() {
    Sequence seq = {1, 2, 3, 4, 5, 6};
    seq.print();
}
```

Explicit Constructors

- constructor callable with ***single*** argument can be used in implicit conversions (e.g., when attempting to obtain matching type for function parameter in function call)
- often, desirable to prevent constructor from being used for implicit conversions
- to accommodate this, constructor can be marked as explicit
- **explicit constructor** is constructor that cannot be used to perform implicit conversions
- prefixing constructor declaration with `explicit` keyword makes constructor explicit
- example:

```
class Widget {
public:
    explicit Widget(int); // explicit constructor
    // ...
};
```

Example Without Explicit Constructor

```
#include <cstdlib>

// one-dimensional integer array class
class IntArray {
public:
    // create array of int with size elements
    IntArray(std::size_t size) { /* ... */ };
    // ...
};

void processArray(const IntArray& x) {
    // ...
}

int main() {
    // following lines of code almost certain to be
    // incorrect, but valid due to implicit type
    // conversion provided by
    // IntArray::IntArray(std::size_t)
    IntArray a = 42;
      // probably incorrect
      // implicit conversion effectively yields code:
      // IntArray a = IntArray(42);
    processArray(42);
      // probably incorrect
      // implicit conversion effectively yields code:
      // processArray(IntArray(42));
}
```

Example With Explicit Constructor

```
#include <cstdlib>

// one-dimensional integer array class
class IntArray {
public:
    // create array of int with size elements
    explicit IntArray(std::size_t size) { /* ... */ };
    // ...
};

void processArray(const IntArray& x) {
    // ...
}

int main() {
    IntArray a = 42; // ERROR: cannot convert
    processArray(42); // ERROR: cannot convert
}
```

Explicitly Deleted/Defaulted Special Member Functions

- can explicitly default or delete special member functions (i.e., default constructor, copy constructor, move constructor, destructor, copy assignment operator, and move assignment operator)
- can also delete non-special member functions
- example:

```
class Thing {
public:
    Thing() = default;

    // Prevent copying.
    Thing(const Thing&) = delete;
    Thing& operator=(const Thing&) = delete;

    Thing(Thing&&) = default;
    Thing& operator=(Thing&&) = default;
    ~Thing() = default;
    // ...
};
// Thing is movable but not copyable.
```

Delegating Constructors

- sometimes, one constructor of class needs to performs all work of another constructor followed by some additional work
- rather than duplicate common code in both constructors, one constructor can use its initializer list to invoke other constructor (which must be only one in initializer list)
- constructor that invokes another constructor via initializer list called **delegating constructor**
- example:

```
class Widget {
public:
    Widget(char c, int i) : c_(c), i_(i) {}
    Widget(int i) : Widget('a', i) {}
      // delegating constructor
    // ...
private:
    char c_;
    int i_;
};

int main() {
    Widget w('A', 42);
    Widget v(42);
}
```

Static Data Members

- sometimes want to have object that is shared by all objects of class
- data member that is shared by all objects of class is called **static data member**
- to make data member static, declare using `static` qualifier
- static data member must (in most cases) be defined outside body of class
- example:

```
class Widget {
public:
    Widget() {++count_;}
    Widget(const Widget&) {++count_;}
    Widget(Widget&&) {++count_;}
    ~Widget() {--count_;}
    // ...
private:
    static int count_; // total number of Widget
                       // objects in existence
};

// Define (and initialize) count member.
int Widget::count_ = 0;
```

Static Member Functions

- sometimes want to have member function that does not operate on objects of class
- member function of class that does not operate on object of class (i.e., has no **this** variable) called **static member function**
- to make member function static, declare using **static** qualifier
- example:

```
class Widget {
public:
    // ...
    // convert degrees to radians
    static double degToRad(double deg)
      {return (M_PI / 180.0) * deg;}
private:
    // ...
};

void func() {
    Widget x; double rad;
    rad = Widget::degToRad(45.0);
    rad = x.degToRad(45.0); // x is ignored
}
```

`constexpr` Member Functions

- like non-member functions, member functions can also be qualified as `constexpr` to indicate function can be computed ***at compile time*** provided that all arguments to function are constant expressions
- some additional restrictions on constexpr member functions relative to nonmember case (e.g., cannot be virtual)
- constexpr member function ***implicitly inline***
- constexpr member function ***not implicitly const*** (as of C++14)

`constexpr` Constructors

- constructors can also be qualified as **`constexpr`** to indicate object construction can be performed ***at compile time*** provided that all arguments to constructor are constant expressions
- constexpr constructor ***implicitly inline***

Example: Constexpr Constructors and Member Functions

```
#include <cmath>
#include <iostream>

// Two-dimensional vector class.
class Vector {
public:
    constexpr Vector() : x_(0), y_(0) {}
    constexpr Vector(double x, double y) : x_(x), y_(y) {}
    constexpr Vector(const Vector& v) : x_(v.x_), y_(v.y_) {}
    constexpr Vector& operator=(const Vector& v)
      {x_ = v.x_; y_ = v.y_; return *this;}
    constexpr double x() const {return x_;}
    constexpr double y() const {return y_;}
    constexpr double norm() const
      {return std::sqrt(x_ * x_ + y_ * y_);}
    // ...
private:
    double x_; // The x component of the vector.
    double y_; // The y component of the vector.
};

int main() {
    constexpr Vector v(3.0, 4.0);
    static_assert(v.x() == 3.0 && v.y() == 4.0);
    constexpr double d = v.norm();
    std::cout << d << '\n';
}
```

Why Constexpr Member Functions Not Implicitly Const

```
class Widget {
public:
    constexpr Widget() : i_(42) {}
    constexpr const int& get() const {return i_;}
    constexpr int& get() /* what if implicitly const? */
      {return i_;}
    // ...
private:
    int i_;
};

constexpr int i = ++Widget().get();
static_assert(i == 43);
```

- in above code example, we want to have const and non-const overloads of `get` member function that can each be used in constant expressions
- so both overloads of `get` need to be constexpr
- if constexpr member functions were implicitly const, it would be impossible to overload on const in manner we wish to do here, since second overload of `get` would automatically become const member function (resulting in multiple conflicting definitions of const member function `get`)

The **`mutable`** Qualifier

- type for data member can be qualified as **`mutable`** meaning that member does not affect externally visible state of class
- mutable data member can be modified in const member function
- **`mutable`** qualifier often used for mutexes, condition variables, cached values, statistical information for performance analysis or debugging

Example: Mutable Qualifier for Statistical Information

```cpp
#include <iostream>
#include <string>

class Employee {
public:
    Employee(int id, std::string& name, double salary) :
      id_(id), name_(name), salary_(salary), accessCount_(0) {}
    int getId() const {
        ++accessCount_; return id_;
    }
    std::string getName() const {
        ++accessCount_; return name_;
    }
    double getSalary() const {
        ++accessCount_; return salary_;
    }
    // ...
    // for debugging
    void outputDebugInfo(std::ostream& out) const {
        out << accessCount_ << '\n';
    }
private:
    int id_; // employee ID
    std::string name_; // employee name
    double salary_; // employee salary
    mutable unsigned long accessCount_; // for debugging
};
```

Pointers to Members

- pointer to member is offset-like construct that provides means to indirectly refer to member of class
- type corresponding to pointer to member of class `T` written as `T::*`
- pointer to member `m` in class `T` can be obtained by applying address-of operator to `T::m` (i.e., using expression `&T::m`)
- given object `x` of type `T`, can access member through pointer to member `ptm` by applying member-selection operator `.*` to `x` using expression `x.*ptm`
- given pointer `p` to object of type `T`, can access member through pointer to member `ptm` by applying member-selection operator `->*` to `p` using expression `p->*ptm`
- null pointer can be assigned to pointer to member to represent no member
- one can approximately think of pointer to member as offset from start of object to particular member
- pointer to member needs to be used in conjunction with object

Pointers to Members: Example

```
#include <iostream>

class Widget {
public:
    Widget(bool flag) {
        op_ = flag ? &Widget::op_2 : &Widget::op_1;
    }
    void modify() {
        // ...
        (this->*op_)(); // invoke member function
        // ...
    }
    // ...
private:
    void op_1() {std::cout << "op_1 called\n";}
    void op_2() {std::cout << "op_2 called\n";}
    void (Widget::*op_)();
      // pointer to member function of Widget class that
      // takes no parameters and returns no value
    // ...
};

int main() {
    Widget u(false);
    Widget v(true);
    u.modify(); // modify invokes op_1
    v.modify(); // modify invokes op_2
}
```

Stream Inserters

- stream inserters write data to output stream
- overload **operator**<<
- have general form `std::ostream& operator<<(std::ostream&, T)` where type `T` is typically const lvalue reference type
- example:

```
std::ostream& operator<<(std::ostream& outStream,
  const Complex& a)
{
    outStream << a.real() << ' ' << a.imag();
    return outStream;
}
```

- inserter and extractor should use ***compatible formats*** (i.e., what is written by inserter should be readable by extractor)

Stream Extractors

- stream extractors read data from input stream
- overload `operator>>`
- have general form

  `std::istream& operator>>(std::istream&, T)` where type `T` is typically non-const lvalue reference type
- example:

```
std::istream& operator>>(std::istream& inStream,
  Complex& a)
{
    double real = 0.0;
    double imag = 0.0;
    inStream >> real >> imag;
    a = Complex(real, imag);
    return inStream;
}
```

Structured Bindings

- structured bindings allow, with single statement, multiple variables to be declared and initialized with values from pair, tuple, array, or struct
- declaration uses auto keyword
- variables enclosed in brackets
- multiple variables separated by commas

Structured Bindings Example

```
#include <tuple>
#include <array>
#include <cassert>

int main() {
    int a[3] = {1, 2, 3};
    auto [a0, a1, a2] = a;
    assert(a0 == a[0] && a1 == a[1] && a2 == a[2]);

    int b[3] = {0, 2, 3};
    auto& [b0, b1, b2] = b;
    ++b0;
    assert(b[0] == 1);

    std::array<int, 3> c = {1, 2, 3};
    auto [c0, c1, c2] = c;
    assert(c0 == c[0] && c1 == c[1] && c2 == c[2]);

    auto t = std::make_tuple(true, 42, 'A');
    auto [tb, ti, tc] = t;
    assert(tb == true && ti == 42 && tc == 'A');
}
```

Structured Bindings Example

```
#include <map>
#include <string>
#include <iostream>

int main() {
    std::map<std::string, int> m = {
        {"apple", 1},
        {"banana", 2},
        {"orange", 3},
    };
    for (auto&& [key, value] : m) {
        std::cout << key << ' ' << value << '\n';
    }
}
```

Literal Types

- each of following types said to be **literal type**:
  - **`void`**
  - scalar type (e.g., integral, floating point, pointer, enumeration, pointer to member)
  - reference type
  - class type that has all of following properties:
    - has trivial destructor
    - is either: aggregate type; or type with at least one constexpr constructor that is not copy or move constructor; or closure type
    - all nonstatic data members and base classes are of nonvolatile literal types
  - array of literal type
- examples of literal types:
  - `int`, `double[16]`, and `std::complex<double>`
- examples of types that are not literal types:
  - `std::vector<int>` and `std::string`
- literal types important in context of constexpr variables, functions, and constructors

Constexpr Variable Requirements

- constexpr variable must satisfy following requirements:
  - its type must be literal type
  - it must be immediately initialized
  - full expression of its initialization must be constant expression (including all implicit conversions and constructor calls)

Constexpr Function Requirements

- constexpr function must satisfy following requirements:
  - must not be virtual
  - its return type must be literal type
  - each of its parameters must be of literal type
  - function body must be either deleted or defaulted or contain any statements except:
    - asm declaration
    - goto statement
    - statement with label other than **`case`** and **`default`**
    - try block
    - definition of variable of non-literal type
    - definition of variable of static or thread storage duration
    - definition of variable for which no initialization is performed
  - if function is defaulted copy/move assignment, class of which it is member must not have mutable member

Constexpr Constructor Requirements

- constexpr constructor must satisfy following requirements:
  - each of its parameters must be of literal type
  - class must not have any virtual base classes
  - constructor must not have function try block
  - constructor body must be either deleted or defaulted or satisfy following constraints:
    - compound statement of constructor body must satisfy constraints for body of constexpr function
    - every base class sub-object and every non-static data member must be initialized
    - every constructor selected to initialize non-static members and base class must be constexpr constructor
  - if constructor is defaulted copy/move constructor, class of which it is member must not have mutable member

Section 2.4.5

Temporary Objects

Temporary Objects

- A **temporary object** is an unnamed object introduced by the compiler.
- Temporary objects are used during:
  - evaluation of expressions
  - argument passing
  - function returns (that return by value)
  - reference initialization
- It is important to understand when temporary objects can be introduced, since the introduction of temporaries impacts performance.
- Evaluation of expression:

```
std::string s1("Hello ");
std::string s2("World");
std::string s;
s = s1 + s2; // must create temporary
  // std::string _tmp(s1 + s2);
  // s = _tmp;
```

- Argument passing:

```
double func(const double& x);
func(3); // must create temporary
  // double _tmp = 3;
  // func(_tmp);
```

Temporary Objects (Continued)

- Reference initialization:

```
int i = 2;
const double& d = i; // must create temporary
  // double _tmp = i;
  // const double& d = _tmp;
```

- Function return:

```
std::string getMessage();
std::string s;
s = getMessage(); // must create temporary
  // std::string _tmp(getMessage());
  // s = _tmp;
```

- In most (but not all) circumstances, a temporary object is destroyed as the last step in evaluating the full expression that contains the point where the temporary object was created.

Temporary Objects Example

```
class Complex {
public:
    Complex(double re = 0.0, double im = 0.0) : re_(re),
      im_(im) {}
    Complex(const Complex& a) = default;
    Complex(Complex&& a) = default;
    Complex& operator=(const Complex& a) = default;
    Complex& operator=(Complex&& a) = default;
    ~Complex() = default;
    double real() const {return re_;}
    double imag() const {return im_;}
private:
    double re_; // The real part.
    double im_; // The imaginary part.
};

Complex operator+(const Complex& a, const Complex& b) {
    return Complex(a.real() + b.real(), a.imag() + b.imag());
}

int main() {
    Complex a(1.0, 2.0);
    Complex b(1.0, 1.0);
    Complex c;
    // ...
    c = a + b;
}
```

Temporary Objects Example (Continued)

Original code:

```
int main() {
    Complex a(1.0, 2.0);
    Complex b(1.0, 1.0);
    Complex c;
    // ...
    c = a + b;
}
```

Code showing temporaries:

```
int main() {
    Complex a(1.0, 2.0);
    Complex b(1.0, 1.0);
    Complex c;
    // ...
    Complex _tmp(a + b);
    c = _tmp;
}
```

Prefix Versus Postfix Increment/Decrement

```
class Counter {
public:
    Counter() : count_(0) {}
    int getCount() const {return count_;}
    Counter& operator++() { // prefix increment
        ++count_;
        return *this;
    }
    Counter operator++(int) { // postfix increment
        Counter old(*this);
        ++count_;
        return old;
    }
private:
    int count_; // counter value
};

int main() {
    Counter x;
    Counter y;
    y = ++x; // no temporaries, int increment, operator=
    y = x++; // 1 temporary, 1 named, 2 constructors,
             // 2 destructors, int increment, operator=
}
```

Compound Assignment Versus Separate Assignment

```cpp
#include <complex>
using std::complex;

int main() {
    complex<double> a(1.0, 1.0);
    complex<double> b(1.0, -1.0);
    complex<double> z(0.0, 0.0);

    // 2 temporary objects
    // 2 constructors, 2 destructors
    // 1 operator=, 1 operator+, 1 operator*
    z = b * (z + a);

    // no temporary objects
    // only 1 operator+= and 1 operator*=
    z += a;
    z *= b;
}
```

Lifetime of Temporary Objects

- Normally, a temporary object is destroyed as the last step in evaluating the full expression that contains point where temporary object was created.
- First exception: When a default constructor with one or more default arguments is called to initialize an element of an array.
- Second exception: When a ***reference is bound to a temporary*** (or a subobject of a temporary), the lifetime of the temporary is extended to ***match the lifetime*** of the reference, with following ***exceptions***:
  - A temporary bound to a reference member in a constructor initializer list persists until the constructor exits.
  - A temporary bound to a reference parameter in a function call persists until the completion of the full expression containing the call.
  - A temporary bound to the return value of a function in a return statement is not extended, and is destroyed at end of the full expression in the return statement.
  - A temporary bound to a reference in an initializer used in a new-expression persists until the end of the full expression containing that new-expression.

Lifetime of Temporary Objects Examples

- Example:

```
void func() {
    std::string s1("Hello");
    std::string s2(" ");
    std::string s3("World!\n");
    const std::string& s = s1 + s2 + s3;
    std::cout << s; // OK?
}
```

- Example:

```
const std::string& getString() {
    return std::string("Hello");
}
void func() {
    std::cout << getString(); // OK?
}
```

Return Value Optimization (RVO)

- **return value optimization (RVO)** is compiler optimization technique that eliminates copy of return value from ***unnamed*** local object in function to object in caller
- example:

```
SomeType function() {
    return SomeType(); // returns temporary object
}

void caller() {
    SomeType x = function();
}
```

- without RVO: return value of function (which is local to function) is copied to new temporary object in caller (so return value not lost when function returns); then, value of new temporary object copied to object that is to hold return value
- with RVO: return value of function is placed directly in object (in caller) that is to hold return value
- by avoiding need for temporary object to hold return value, eliminates move/copy constructor and destructor call
- as will be seen later, C++ requires this type of optimization to be performed

Named Return Value Optimization (NRVO)

- **named return value optimization (NRVO)** is variation on RVO where return value is named object (i.e., not temporary object)
- example:

```
SomeType function() {
    SomeType result;
    // ...
    return result; // returns named object
}

void caller() {
    SomeType x = function();
}
```

- compiler optimizes away `result` in `function` and return value constructed directly in `x`
- effectively, `result` becomes reference to `x`
- code with NRVO more efficient (i.e., move/copy constructor and destructor calls eliminated)

Copy Elision

- normally, compiler forbidden from applying optimizations to code that would change its observable behavior (i.e., so called “as if” rule)
- one important exception to as-if rule is copy elision
- copy elision is code transformation that omits copy/move operation by constructing object in place to which it would later be copied/moved
- copy elision allows copy/move operations to be eliminated (thus, avoiding cost of copy/move constructors)
- copy elision either allowed or required in several contexts, which relate to:
  - returning by value
  - passing by value
  - throwing and catching exceptions by value

Mandatory Copy Elision

- if prvalue used as initializer of object with same type, object must be initialized directly
- in constant expression and constant initialization, all copy elision is guaranteed
- guaranteed copy elision enables more flexibility in dealing with non-copyable non-movable types

Copy Elision and Returning by Value

- in return statement of function with class return type, when expression is name of non-volatile automatic object (other than function or catch-clause parameter) with same cv-unqualified type as function return type, automatic object can be constructed directly in function's return value
- example:

```
#include <iostream>

class Widget {
public:
    Widget() {}
    Widget(const Widget&) {std::cout << "copy\n";}
    Widget(Widget&&) {std::cout << "move\n";}
    // ...
};

Widget func1() {return Widget();}
Widget func2() {Widget w; return w;}

int main() {
    Widget w = func1(); // required copy elision
    Widget x = func2(); // possible copy elision
}
```

Copy Elision and Passing by Value

- in function call, when temporary class object not bound to reference would be copied/moved to class object with same cv-unqualified type, temporary object can be constructed directly in target of omitted copy/move
- example:

```
#include <iostream>

class Widget {
public:
    Widget() : x_(42) {}
    Widget(const Widget&) {std::cout << "copy\n";}
    Widget(Widget&&) {std::cout << "move\n";}
    int get() const {return x_;}
    // ...
private:
    int x_;
};

void func(Widget w) {std::cout << w.get() << '\n';}

int main() {
    func(Widget()); // required copy elision
}
```

Copy Elision and Throwing by Value

- in throw expression, when operand is name of non-volatile automatic object (other than function or catch-clause parameter) whose scope does not extend beyond end of innermost enclosing try block (if there is one), copy/move operation from operand to exception object can be omitted by constructing automatic object directly into exception object
- example:

```
#include <iostream>

class Widget {
public:
    Widget() {}
    Widget(const Widget &) {std::cout << "copy\n";}
    Widget(Widget&&) {std::cout << "move\n";}
    // ...
};

void f(){
    throw Widget(); // required copy elision
}

int main() {
    try {f();}
    catch (Widget foo) {std::cout << "catch\n";}
}
```

Copy Elision and Catching by Value

- when exception declaration of exception handler declares object of same type (except for cv-qualification) as exception object, copy/move operation can be omitted by treating exception declaration as alias for exception object if meaning of program will be unchanged except for execution of constructors and destructors for object declared by exception declaration
- example:

```
#include <iostream>

class Widget {
public:
    Widget() {}
    Widget(const Widget &) {std::cout << "copy\n";}
    Widget(Widget&&) {std::cout << "move\n";}
    // ...
};

void f(){throw Widget();}

int main() {
    try {f();}
    catch (Widget foo) { // possible copy elision
        std::cout << "catch\n";
    }
}
```

Mandatory Copy Elision Example: Factory Function

```cpp
class Widget {
public:
    Widget() {/* ... */}
    // not copyable
    Widget(const Widget&) = delete;
    Widget& operator=(const Widget&) = delete;
    // not movable
    Widget(Widget&&) = delete;
    Widget& operator=(Widget&&) = delete;
    // ...
};

Widget make_widget() {
    return Widget();
}

int main() {
    Widget w(make_widget());
      // OK: copy elision required
    Widget v(Widget());
      // OK: copy elision required
}
```

Mandatory Copy Elision Example: Constant Expressions

```
#include <iostream>

struct Widget {
    Widget *p;
    constexpr Widget() : p(this) {}
};

constexpr Widget func() {
    Widget w;
    return w; // NOTE: returning named object
}

constexpr Widget a;
static_assert(a.p == &a);

constexpr Widget b = func();
static_assert(b.p == &b);
  // OK: required copy elision (NVRO guaranteed here)

int main() {
    Widget c = func();
      // c.p may point to c or to a temporary
    std::cout << (c.p == &c) << '\n';
}
```

Section 2.4.6

Functors

Functors

- **function object** (also known as **functor**) is object that can be invoked or called as if it were ordinary function
- class that provides member function that overloads `operator()` is called **functor class** and object of that class is **functor**
- functors more flexible than functions as functors are objects and can therefore carry arbitrary state information
- functors are extremely useful, especially in generic programming
- as we will see later, standard library makes heavy use of functors

Functor Example: Less Than

```
struct LessThan { // Functor class
    bool operator()(double x, double y) const {
        return x < y;
    }
};

void myFunc() {
    double a = 1.0;
    double b = 2.0;
    LessThan lessThan; // Functor
    bool result = lessThan(a, b);
      // calls LessThan::operator()(double, double)
      // lessThan is functor, not function
    // result == true
}
```

Functor Example With State

```cpp
class IsGreater { // Functor class
public:
    IsGreater(int threshold) : threshold_(threshold) {}
    bool operator()(int x) const {
        return x > threshold_;
    }
private:
    // state information for functor
    int threshold_; // threshold for comparison
};

void myFunc() {
    IsGreater isGreater(5); // functor
    int x = 3;
    bool result = isGreater(x);
      // calls IsGreater::operator()(int)
    // result == false
}
```

Section 2.5

Templates

Templates

- **generic programming**: algorithms written in terms of types to be specified later (i.e., algorithms are generic in sense of being applicable to any type that meets only some very basic constraints)
- templates facilitate generic programming
- extremely important language feature
- avoids code duplication
- leads to highly efficient and customizable code
- promotes code reuse
- C++ standard library makes very heavy use of templates (actually, most of standard library consists of templates)
- many other libraries make heavy use of templates (e.g., CGAL, Boost)

Section 2.5.1

Function Templates

Motivation for Function Templates

- consider following functions:

```
int max(int x, int y)
  {return x > y ? x : y;}

double max(double x, double y)
  {return x > y ? x : y;}

// more similar-looking max functions...
```

- each of above functions has ***same general form***; that is, for some type `T`, we have:

```
T max(T x, T y)
  {return x > y ? x : y;}
```

- would be nice if we did not have to repeatedly type, debug, test, and maintain nearly identical code
- in effect, would like code to be parameterized on type `T`

Function Templates

- **function template** is family of functions parameterized by one or parameters
- each template parameter can be: non-type (e.g., integral constant), type, template, or parameter pack (in case of variadic template)
- syntax for template function has general form:

  `template` <*parameter_list*> *function_declaration*
- *parameter_list*: parameters on which template function depends
- *function_declaration*: function declaration or definition
- type parameter designated by **`class`** or **`typename`** keyword
- template parameter designated by **`template`** keyword
- template template parameter must use **`class`** keyword
- non-type parameter designed by its type (e.g., **`bool`**, **`int`**)
- example:

```
// declaration of function template
template <class T> T max(T x, T y);

// definition of function template
template <class T> T max(T x, T y)
  {return x > y ? x : y;}
```

Function Templates (Continued)

- to explicitly identify particular instance of template, use syntax:

  function<parameters>

- example:
  for function template declaration:

  ```
  template <class T> T max(T x, T y);
  ```

  `max<int>` refers to `int max(int, int)`
  `max<double>` refers to `double max(double, double)`
- compiler only creates code for function template when it is instantiated (i.e., used)
- therefore, definition of function template must be visible in place where it is instantiated
- consequently, function template definitions usually appear in header file
- template code only needs to pass basic syntax checks, unless actually instantiated

Function Template Examples

```
// compute minimum of two values
template <class T>
T min(T x, T y) {
    return x < y ? x : y;
}

// compute square of value
template <typename T>
T sqr(T x) {
    return x * x;
}

// swap two values
template <class T>
void swap(T& x, T& y) {
    T tmp = x;
    x = y;
    y = tmp;
}

// invoke function/functor multiple times
template <int N = 1, typename F, typename T>
void invoke(F func, const T& value) {
    for (int i = 0; i < N; ++i) {
        func(value);
    }
}
```

Template Function Overload Resolution

- overload resolution proceeds (in order) as follows:
  1. look for an exact match with zero or more trivial conversions on (nontemplate) functions; if found call it
  2. look for function template from which function that can be called with exact match with zero or more trivial conversions can be generated; if found, call it
  3. try ordinary overload resolution for functions; if function found, call it; otherwise, call is error
- in each step, if more than one match found, call is ambiguous and is error
- template function only used in case of exact match, unless explicitly forced
- example:

```
template <class T>
T max(T x, T y) {return x > y ? x : y;}

void func(int i, int j, double x, double y) {
    double z = max(x, y); // calls max<double>
    int k = max(i, j); // calls max<int>
    z = max(i, x); // ERROR: no match
    z = max<double>(i, x); // calls max<double>
}
```

Qualified Names

- **qualified name** is name that specifies scope
- example:

```
#include <iostream>

int main(int argc, char** argv) {
    for (int i = 0; i < 10; ++i) {
        std::cout << "Hello, world!" << std::endl;
    }
}
```

- in above example, names `std::cout` and `std::endl` are qualified, while names `main`, `argc`, `argv`, and `i`, are not qualified

Dependent Names

- **dependent name** is name that depends on template parameter
- example:

```
template <class T>
void func(const T& x) {
    int i = T::magicValue;
    // ...
}
```

- name `T::magicValue` is dependent

Qualified Dependent Names

- to avoid any potential ambiguities, compiler will automatically assume qualified dependent name does not name type unless **`typename`** keyword is used
- must precede qualified dependent name that names type by **`typename`**
- in following example, note use of **`typename`** keyword:

```
#include <vector>

template <class T>
void func(const T& x) {
    std::vector<T> v(42, x);
    // std::vector<T>::const_iterator is
    // qualified dependent name
    for (typename std::vector<T>::const_iterator i =
      v.begin(); i != v.end(); ++i) {
        // std::vector<T>::value_type is
        // qualified dependent name
        typename std::vector<T>::value_type x = *i;
        // ...
    }
    // ...
}
```

Why typename is Needed

```
int x = 42;

template <class T> void func() {
    // The compiler must be able to check syntactic
    // correctness of this template code without
    // knowing T. Without knowing T, however, the
    // meaning of following line of code is ambiguous.
    // Is it a declaration of a variable x or an
    // expression consisting of a binary operator*
    // with operands T::foo and x?
    T::foo* x; // Does T::foo name a type or an object?
    // ...
}

struct ContainsType {
    using foo = int; // foo is type
    // ...
};

struct ContainsValue {
    static int foo; // foo is value
    // ...
};

int main() {
    // Only one of the following lines should be valid.
    func<ContainsValue>();
    func<ContainsType>();
}
```

Example: What is wrong with this code?

```cpp
// templates_1_0.cpp

#include <iostream>
#include <complex>
#include "templates_1_1.hpp"

int main() {
    std::complex a(0.0, 1.0);
    auto b = square(a);
    std::cout << b << '\n';
}
```

```cpp
// templates_1_1.hpp

template <class T>
T square(const T&);
```

```cpp
// templates_1_1.cpp

#include "templates_1_1.hpp"

template <class T>
T square(const T& x) {
    return x * x;
}
```

Section 2.5.2

Class Templates

Motivation for Class Templates

- consider almost identical complex number classes:

```
class ComplexDouble {
public:
    ComplexDouble(double x = 0.0, double y = 0.0) : x_(x), y_(y) {}
    double real() const { return x_; }
    double imag() const { return y_; }
    // ...
private:
    double x_, y_; // real and imaginary parts
};

class ComplexFloat {
public:
    ComplexFloat(float x = 0.0f, float y = 0.0f) : x_(x), y_(y) {}
    float real() const { return x_; }
    float imag() const { return y_; }
    // ...
private:
    float x_, y_; // real and imaginary parts
};
```

- both of above classes are special cases of following class parameterized on type `T`:

```
class Complex {
public:
    Complex(T x = T(0), T y = T(0)) : x_(x), y_(y) {}
    T real() const { return x_; }
    T imag() const { return y_; }
    // ...
private:
    T x_, y_; // real and imaginary parts
};
```

- again, would be nice if we did not have to repeatedly type, debug, test, and maintain nearly identical code

Class Templates

- **class template** is family of classes parameterized on one or more parameters
- each template parameter can be: non-type (e.g., integral constant), type, template, or parameter pack (in case of variadic template)
- syntax has general form:

  template <*parameter_list*> *class_declaration*

- *parameter_list*: parameter list for class
- *class_declaration*: class/struct declaration or definition
- example:

```
// declaration of class template
template <class T, unsigned int size>
class MyArray;

// definition of class template
template <class T, unsigned int size>
class MyArray {
    // ...
    T array_[size];
};

MyArray<double, 100> x;
```

Class Templates (Continued)

- compiler only generates code for class template when it is instantiated (i.e., used)
- since compiler only generates code for class template when it is instantiated, definition of template must be visible at point where instantiated
- consequently, class template code usually placed in header file
- template code only needs to pass basic syntax checks, unless actually instantiated
- compile errors related to class templates can often be very long and difficult to parse (especially, when template class has parameters that are template classes which, in turn, have parameters that are template classes, and so on)

Class Template Example

```
// complex number class template
template <class T>
class Complex {
public:
    Complex(T x = T(0), T y = T(0)) :
      x_(x), y_(y) {}
    T real() const {
        return x_;
    }
    T imag() const {
        return y_;
    }
    // ...
private:
    T x_; // real part
    T y_; // imaginary part
};

Complex<int> zi;
Complex<double> zd;
```

Class-Template Default Parameters

- class template parameters can have ***default values***
- example:

```
template <class T = int, unsigned int size = 2>
struct MyArray {
    T data[size];
};

MyArray<> a; // MyArray<int, 2>
MyArray<double> b; // MyArray<double, 2>
MyArray<double, 10> b; // MyArray<double, 10>
```

Qualified Dependent Names

- qualified dependent name assumed not to name type, unless preceded by **typename** keyword
- in following example, note use of **typename** keyword:

```
#include <vector>

template <class T> class Vector {
public:
    using Coordinate = typename T::Coordinate;
    using Distance = typename T::Distance;
    Vector(const std::vector<Coordinate>& coords) :
      coords_(coords) {}
    Distance squaredLength() const {
        Distance d = Distance(0);
        for (typename
          std::vector<Coordinate>::const_iterator i =
          coords_.begin(); i != coords_.end(); ++i) {
            typename std::vector<Coordinate>::value_type
              x = *i;
            d += x * x;
        }
        return d;
    }
    // ...
private:
    std::vector<Coordinate> coords_;
};
```

Template Template Parameter Example

```
#include <vector>
#include <list>
#include <deque>
#include <memory>

template <template <class, class> class Container, class Value>
class Stack {
public:
    // ...
private:
    Container<Value, std::allocator<Value>> data_;
};

int main() {
    Stack<std::vector, int> s1;
    Stack<std::list, int> s2;
    Stack<std::deque, int> s3;
}
```

Class Template Parameter Deduction

- template parameters for class template can be deduced based on arguments passed to constructor
- example:

```
std::tuple t(42, 'A');
  // OK: deduced as tuple<int, char>
```

- deduction only performed if no template arguments provided
- example:

```
std::tuple<int> t(1, 2);
    // ERROR: missing template parameter, as
    // no template parameter deduction takes place
```

Class Template Parameter Deduction Example

```
#include <vector>
#include <tuple>
#include <set>
#include <string>

using namespace std::string_literals;

auto get_tuple() {
    return std::tuple("Zaphod"s, 42);
      // deduces tuple<std::string, int>
}

int main() {
    std::vector v{1, 2, 3};
      // deduces vector<int>
    std::tuple t(true, 'A', 42);
      // deduces tuple<bool, char, int>
    std::pair p(42, "Hello"s);
      // deduces pair<int, std::string>
    std::set s{0.5, 0.25};
      // deduces set<double>
    //auto ptr = new std::tuple(true, 42);
      // should deduce tuple<bool, int>?
      // fails to compile with GCC 7.1.0
}
```

Template Deduction Guides

- can provide additional rules to be used to determine how class template parameters should be deduced when not provided
- such rules called deduction guides
- deduction guide itself can be either template or non-template
- deduction guides must be introduced in same scope as class template
- example:

```
// class definition
template <class T> smart_ptr {/* ... */};
// deduction guide
template <class T>
smart_ptr(T*) -> smart_ptr<T>;
```

- example:

```
/// class definition
template <class T> name {/* ... */};
// deduction guide
name(const char*) -> name<std::string>;
```

Template Deduction Guide Example

```
#include <string>
#include <type_traits>

using namespace std::string_literals;

template <class T>
class Name {
public:
    Name(T first, T last) : first_(first), last_(last) {}
    // ...
private:
    T first_;
    T last_;
};

// deduction guide
Name(const char*, const char*) -> Name<std::string>;

int main() {
    Name n("Zaphod", "Beeblebrox");
      // deduces Name<std::string> via deduction guide
    static_assert(std::is_same_v<decltype(n), Name<std::string>>);
    Name n2("Jane"s, "Doe"s);
      // deduces Name<std::string> (without deduction guide)
    static_assert(std::is_same_v<decltype(n2), Name<std::string>>);
}
```

Auto Non-Type Template Parameters

- can use **auto** keyword for non-type template parameter
- in such case, type of non-type template parameter will be deduced
- example:

```
template <auto v>
struct constant {
    static constexpr decltype(v) value = v;
};
using forty_two_type = constant<42>;
  // template parameter v deduced to have type int
```

- non-type template parameter type deduction probably most useful for template metaprogramming

Example Without Auto Non-Type Template Parameter

```
#include <cstdlib>
#include <iostream>

template<class T, T v>
struct integral_constant {
    using value_type = T;
    static constexpr value_type value = v;
    using type = integral_constant;
    constexpr operator value_type() const noexcept
      {return value;}
    constexpr value_type operator()() const noexcept
      {return value;}
};

using forty_two_type = integral_constant<int, 42>;

int main() {
    constexpr forty_two_type x;
    constexpr auto v = x.value;
    std::cout << v << '\n';
}
```

Example With Auto Non-Type Template Parameter

```
#include <cstdlib>
#include <iostream>

template<auto v>
struct integral_constant {
    using value_type = decltype(v);
    static constexpr value_type value = v;
    using type = integral_constant;
    constexpr operator value_type() const noexcept
      {return value;}
    constexpr value_type operator()() const noexcept
      {return value;}
};

using forty_two_type = integral_constant<42>;

int main() {
    constexpr forty_two_type x;
    constexpr auto v = x.value;
    std::cout << v << '\n';
}
```

Section 2.5.3

Variable Templates

Variable Templates

- **variable template** is family of variables parameterized on one or more parameters
- each template parameter can be: non-type (e.g., integral constant), type, template, or parameter pack (in case of variadic templates)
- although less frequently used than function and class templates, variable templates quite useful in some situations
- syntax has general form:

  template <*parameter_list*> *variable_declaration*

- *parameter_list*: parameter list for variable template
- *variable_declaration*: variable declaration or definition
- example:

```
template <class T>
T meaning_of_life = T(42);

int x = meaning_of_life<int>;
```

Variable Template Example: pi

```
#include <limits>
#include <complex>
#include <iostream>

template <typename T>
constexpr T pi =
  T(3.14159265358979323846264338327950288419716939937510L);

int main() {
    std::cout.precision(
      std::numeric_limits<long double>::max_digits10);
    std::cout
      << pi<int> << '\n'
      << pi<float> << '\n'
      << pi<double> << '\n'
      << pi<long double> << '\n'
      << pi<std::complex<float>> << '\n'
      << pi<std::complex<double>> << '\n'
      << pi<std::complex<long double>> << '\n';
}
```

Section 2.5.4

Alias Templates

Alias Templates

- **alias template** is family of types parameterized on one or more parameters
- each template parameter can be: non-type (e.g., integral constant), type, template, or parameter pack (in case of variadic templates)
- syntax has general form:

  template <*parameter_list*> *alias_declaration*

- *parameter_list*: parameter list for class
- *alias_declaration*: alias declaration (i.e., with **using**)
- example:

```
template <class Value,
  class Alloc = std::allocator<Value>>
using GreaterMultiSet =
  std::multiset<Value, std::greater<Value>, Alloc>;

GreaterMultiSet<int> x{4, 1, 3, 2};
```

```
#include <iostream>
#include <set>

// alias template for set that employs std::greater for
// comparison
template <typename Value,
  typename Alloc = std::allocator<Value>>
using GreaterSet = std::set<Value,
  std::greater<Value>, Alloc>;

int main() {
    std::set x{1, 4, 3, 2};
    GreaterSet<int> y{1, 4, 3, 2};
    for (auto i : x) {
        std::cout << i << '\n';
    }
    std::cout << '\n';
    for (auto i : y) {
        std::cout << i << '\n';
    }
}
```

Section 2.5.5

Variadic Templates

Variadic Templates

- language provides ability to specify template that can take variable number of arguments
- template that can take variable number of arguments called **variadic template**
- alias templates, class templates, function templates, and variable templates may be variadic
- variable number of arguments specified by using what is called parameter pack
- parameter pack is parameter that accepts (i.e., is placeholder for) zero or more arguments (of same kind)
- parameter pack used in parameter list of template to allow to variable number of template parameters
- ellipsis (i.e., "...") is used in various contexts relating to parameter packs
- ellipsis after designator for kind of template argument in template parameter list designates argument is parameter pack
- ellipsis after parameter pack parameter expands parameter pack in context-sensitive manner

- syntax for ***non-type*** template parameter pack named *Args* and containing elements of type *type* (e.g., **bool**, **int**, **unsigned int**):

  type... *Args*

- example:

```
template <int... Is> /* ... */
```

  `Is` is (non-type) template parameter pack that corresponds to zero or more (compile-time constant) values of type **int**

- syntax for ***type*** template parameter pack named *Args*:

  typename... *Args*

  or equivalently

  class... *Args*

- examples:

```
template <typename... Ts> /* ... */
template <class... Ts> /* ... */
```

  `Ts` is (type) template parameter pack that corresponds to zero or more types

Parameter Packs (Continued 1)

- syntax for ***template*** template parameter pack named *Args*:

  template <*parameter_list*> **typename**... *Args*

  or equivalently

  template <*parameter_list*> **class**... *Args*

- example:

```
template <template <class T> class... Ts>
  /* ... */
```

  `Ts` is (template) template parameter pack that corresponds to zero or more templates

- syntax for ***function*** parameter pack named *args* whose elements have types corresponding to elements of type template parameter pack *Args*:

  Args... *args*

- example:

```
template <class... Ts> void func(Ts... args);
```

  `args` is (function) parameter pack that correponds to zero or more function parameters whose types correspond to elements of type parameter pack `Ts`

Parameter Packs (Continued 2)

- in context where template arguments ***cannot be deduced*** (e.g., primary class templates), ***only last*** template parameter can be parameter pack
- in context where template arguments ***can be deduced*** (e.g., function templates and class template partial specializations), template parameter pack ***need not be last*** template parameter
- example:

```
template <class U, class... Ts> class C1 { /* ... */ };
  // OK: Ts is last template parameter

template <class... Ts, class U> class C2 { /* ... */ };
  // ERROR: Ts not last and U not deduced

template <class... Ts, class U> void f1(Ts... ts)
  { /* ... */ } // NOT OK: Ts not last and U not deduced

template <class... Ts, class U> void f2(Ts... ts, U u)
  { /* ... */ } // OK: Ts not last but U is deduced

int main() {
    f1<int, int, bool>(1, 2, true);
      // ERROR: no matching function call
    f2<int, int>(1, 2, true); // OK
    f2(1, 2, true); // ERROR: one argument expected
}
```

Parameter Pack Expansion

- **parameter pack expansion**: expands pack into its constituent elements
- syntax for parameter pack expansion of expression *pattern*, which must contain parameter pack:

  pattern...

- example:

```
template <class... Ts> void f(Ts... t) { /* ... */ }

template <class... Us> void g(Us... u) {
    f(u...);
      // u... is pack expansion
      // when g is called by main,
      // u... expands to 1, 2.0, 3.0f
}

int main() {
    g(1, 2.0, 3.0f);
}
```

```
#include <tuple>

// variadic alias template
template <class... T>
using My_tuple = std::tuple<bool, T...>;

// variadic class template
template <int... Values>
class Integer_sequence {
    // ...
};

// variadic function template
template <class... Ts>
void print(const Ts&... values) {
    // ...
}

// variadic variable template
template <typename T, T... Values>
constexpr T array[] = {Values...};

int main() {
    Integer_sequence<1, 3, 4, 2> x;
    auto a = array<int, 1, 2, 4, 8>;
    My_tuple<int, double> t(true, 42, 42.0);
    print(1'000'000, 1, 43.2, "Hello");
}
```

Parameter Pack Expansion

- parameter pack expansion allowed in following contexts:
  - inside parentheses of function call operator
  - in template argument list
  - in function parameter list
  - in template parameter list
  - base class specifiers in class declaration
  - member initializer lists
  - braced initializer lists
  - lambda captures
  - fold expressions
  - in using declarations

The `sizeof...` Operator

- `sizeof...` operator yields number of elements in parameter pack
- example:

```
template <int... Values>
constexpr int num_parms = sizeof...(Values);

static_assert(num_parms<1, 2, 3> == 3, "");
static_assert(num_parms<> == 0, "");
```

- example:

```
#include <cassert>

template <typename... Ts>
int number_of_arguments(const Ts&... args) {
    return sizeof...(args);
}

int main() {
    assert(number_of_arguments(1, 2, 3) == 3);
    assert(number_of_arguments() == 0);
}
```

Variadic Function Template: sum

```
#include <iostream>
#include <string>

using namespace std::string_literals;

template <class T>
auto sum(T x) {
    return x;
}

template <class T, class... Args>
auto sum(T x, Args... args) {
    return x + sum(args...);
}

int main() {
    auto x = sum(42.5, -1.0, 0.5f);
    auto y = sum("The "s, "answer "s, "is "s);
    std::cout << y << x << ".\n";
    // sum(); // ERROR: no matching function call
}

/* Output:
The answer is 42.
*/
```

Variadic Function Template: maximum

```cpp
#include <type_traits>
#include <string>
#include <cassert>

using namespace std::string_literals;

template <typename T>
T maximum(const T& a) {return a;}

template <typename T1, typename T2>
typename std::common_type_t<const T1&, const T2&>
maximum(const T1 &a, const T2 &b) {
    return a > b ? a : b;
}

template <typename T1, typename T2, typename... Args>
typename std::common_type_t<const T1&, const T2&,
  const Args&...>
maximum(const T1& a, const T2& b, const Args&... args) {
    return maximum(maximum(a, b), args...);
}

int main() {
    assert(maximum(1) == 1);
    assert(maximum(1, 2, 3, 4, -1.4) == 4);
    assert(maximum(-1'000'000L, -42L, 10, 42.42) == 42.42);
    assert(maximum("apple"s, "zebra"s, "c++"s) == "zebra"s);
}
```

Variadic Function Template With Template Template Parameter: print_container

```
#include <iostream>
#include <vector>
#include <string>
#include <set>

template <template <class, class...>
  class ContainerType, class ValueType, class... Args>
bool print_container(const ContainerType<ValueType, Args...>&
  c) {
    for (auto i = c.begin(); i != c.end();) {
        std::cout << *i;
        if (++i != c.end()) {std::cout << ' ';}
    }
    std::cout << '\n';
    return bool(std::cout);
}

int main() {
    using namespace std::string_literals;
    std::vector vi{1, 2, 3, 4, 5};
    std::set si{5, 4, 3, 2, 1};
    std::set ss{"world"s, "hello"s};
    print_container(vi);
    print_container(si);
    print_container(ss);
}
```

Variadic Class Template: Integer_sequence

```
#include <iostream>
#include <cstdlib>

template <class T, T... Values>
class Integer_sequence {
public:
    using value_type = T;
    using const_iterator = const T*;
    constexpr std::size_t size() const
      {return sizeof...(Values);}
    constexpr T operator[](int i) const {return values_[i];}
    constexpr const_iterator begin() const
      {return &values_[0];}
    constexpr const_iterator end() const
      {return &values_[size()];}
private:
    static constexpr T values_[sizeof...(Values)] =
      {Values...};
};

template <class T, T... Values>
constexpr T
Integer_sequence<T, Values...>::values_[sizeof...(Values)];

int main() {
    Integer_sequence<std::size_t, 1, 2, 4, 8> seq;
    std::cout << seq.size() << '\n' << seq[0] << '\n';
    for (auto i : seq) {std::cout << i << '\n';}
}
```

Variadic Variable Template: `int_array`

```
#include <iostream>

template <int... Args>
constexpr int int_array[] = {Args...};

int main() {
    for (auto i : int_array<1,2,4,8>) {
        std::cout << i << '\n';
    }
}

/* Output:
1
2
4
8
*/
```

```
#include <iostream>
#include <string>
#include <tuple>

template <class... Ts>
using My_tuple = std::tuple<bool, Ts...>;

int main() {
    My_tuple<int, std::string> t(true, 42,
      "meaning of life");
    std::cout << std::get<0>(t) << ' '
      << std::get<1>(t) << ' '
      << std::get<2>(t) << '\n';
}

/* Output:
1 42 meaning of life
*/
```

Fold Expressions

- may want to apply binary operator (such as $+$) across all elements in parameter pack
- fold expression reduces (i.e., folds) parameter pack over binary operator
- op: binary operator
- E: expression that contains unexpanded parameter pack
- I: expression that does not contain unexpanded parameter pack

Fold	Syntax	Expansion
unary left	(... op E)	((E_1 op E_2) op ...) op E_N
unary right	(E op ...)	E_1 op (... op (E_{N-1} op E_N))
binary left	(I op ... op E)	(((I op E_1) op E_2) op ...) op E_N
binary right	(E op ... op I)	E_1 op (... op (E_{N-1} op (E_N op I)))

- unary fold of empty parameter pack:

Operator	Value for Empty Parameter Pack
`&&`	`true`
`||`	`false`
`,`	`void()`

Sum Example Without Fold Expression

```
#include <iostream>
#include <string>

using namespace std::string_literals;

template <class T>
auto sum(T x) {
    return x;
}

template <class T, class... Args>
auto sum(T x, Args... args) {
    return x + sum(args...);
}

int main() {
    auto x = sum(42.5, -1.0, 0.5f);
    auto y = sum("The "s, "answer "s, "is "s);
    std::cout << y << x << ".\n";
    // sum(); // ERROR: no matching function call
}

/* Output:
The answer is 42.
*/
```

Sum Example With Fold Expression

```cpp
#include <iostream>
#include <string>

using namespace std::string_literals;

template <class T, class... Args>
auto sum(T x, Args... args) {
    return x + (... + args);
}

int main() {
    auto x = sum(42.5, -1.0, 0.5f);
    auto y = sum("The "s, "answer "s, "is "s);
    std::cout << y << x << ".\n";
    // sum(); // ERROR: no matching function call
}

/* Output:
The answer is 42.
*/
```

Print Example Without Fold Expression

```
#include <iostream>
#include <string>

using namespace std::string_literals;

std::ostream& print() {return std::cout;}

template <class T>
std::ostream& print(const T& value) {
    return std::cout << value;
}

template <class T, class... Args>
std::ostream& print(const T& value, const Args&... args) {
    if (!(std::cout << value)) {
        return std::cout;
    }
    return print(args...);
}

int main() {
    print("The "s, "answer "s, "is "s, 42, ".\n"s);
    print(); // OK: no-op
}

/* Output:
The answer is 42.
*/
```

Print Example With Fold Expression

```cpp
#include <iostream>
#include <string>

using namespace std::string_literals;

template <class... Args>
std::ostream& print(const Args&... args) {
    return (std::cout << ... << args);
}

int main() {
    print("The "s, "answer "s, "is "s, 42, ".\n"s);
    print(); // OK: no-op
}

/* Output:
The answer is 42.
*/
```

Fold Expression Example: All/Any/One/Even

```
#include <cassert>

template <class... Args>
bool all(Args... args)
  {return (... && args);}

template <class... Args>
bool any(Args... args)
  {return (... || args);}

template <class... Args>
bool one(Args... args)
  {return (0 + ... + args) == 1;}

template <class... Args>
bool even(Args... args)
  {return (1 + ... + args) % 2;}

int main() {
    assert(all(false, true, true) == false);
    assert(all(true, true, true) == true);
    assert(any(false, false, true) == true);
    assert(any(false, false, false) == false);
    assert(one(true, false, false) == true);
    assert(one(true, true, false) == false);
    assert(even(true, true, false) == true);
    assert(even(true, false, false) == false);
    assert(even() == true && one() == false);
}
```

Constexpr-Friendly Heterogeneous List Example

```
#include <iostream>
#include <tuple>

// heterogeneous list of constant values
template <auto... vs> class value_list {
public:
    constexpr value_list() : v_(vs...) {}
    template <int n> constexpr auto get() const
      {return std::get<n>(v_);}
    constexpr int size() const {return sizeof...(vs);}
private:
    std::tuple<decltype(vs)...> v_;
};

int main() {
    constexpr value_list<42, true, 'A'> v;
    constexpr auto n = v.size();
    constexpr auto a = v.get<0>();
    constexpr auto b = v.get<1>();
    constexpr auto c = v.get<2>();
    std::cout << n << ' ' << a << ' ' << b << ' ' << c << '\n';
}
```

Constexpr-Friendly Homogeneous List Example

```
#include <iostream>
#include <tuple>

// homogeneous list of constant values
template <auto v1, decltype(v1)... vs> class value_list {
public:
    constexpr value_list() : v_(v1, vs...) {}
    template <int n> constexpr auto get() const
      {return std::get<n>(v_);}
    constexpr int size() const {return 1 + sizeof...(vs);}
private:
    std::tuple<decltype(v1), decltype(vs)...> v_;
};

int main() {
    constexpr value_list<1, 2, 3> v;
    constexpr auto n = v.size();
    constexpr auto a = v.get<0>();
    constexpr auto b = v.get<1>();
    constexpr auto c = v.get<2>();
    std::cout << n << ' ' << a << ' ' << b << ' ' << c << '\n';
}
```

Section 2.5.6

Template Specialization

Template Specialization

- sometimes can be desirable to provide customized version of template for certain choices of template parameters
- customized version of templates can be specified through language feature known as **template specialization**
- two kinds of specialization: explicit and partial
- **explicit specialization** (less formally known as full specialization): customized version of template where all template parameters are fixed
- **partial specialization**: customized version of template where only some of template parameters are fixed
- class templates, function templates, and variable templates can all be specialized
- alias templates cannot be specialized
- class templates and variable templates can be partially or explicitly specialized
- function templates can only be explicitly specialized (not partially)

Explicit Specialization

- syntax for explicit specialization:

  template <> *declaration*

- *declaration*: declaration of templated entity (e.g., function, class, variable)
- example:

```
// unspecialized template
template <class T, class U>
  void func(T x, U y) { /* ... */ }

// explicit specialization of template
// (for when template parameters are bool, bool)
template <>
  void func<bool, bool>(bool x, bool y) { /* ... */ }
```

Partial Specialization

- syntax for partial specialization of class template:

  template <*parameter_list*> *class_key*
  class_name <*argument_list*> *declaration*

- syntax for partial specialization of variable template:

  template <*parameter_list*> *type_name*
  variable_name <*argument_list*> *declaration*

- *class_key*: class or struct keyword (for class template)
- *class_name*: class being specialized (for class template)
- *type_name*: type of variable (for variable template)
- *variable_name*: variable being specialized (for variable template)
- *argument_list*: template argument list
- *declaration*: declaration of templated entity (e.g., class, variable)
- example:

```
// unspecialized template
template <class T, int N> class Widget { /* ... */ };

// partial specialization of template
// (for when first template parameter is bool)
template <int N> class Widget<bool, N> { /* ... */ };
```

Explicitly-Specialized Function Template: printPointee

```
#include <iostream>

// unspecialized version
template <class T>
typename std::ostream& printPointee(
  typename std::ostream& out, const T* p)
  {return out << *p << '\n';}

// specialization
template <>
typename std::ostream& printPointee<void>(
  typename std::ostream& out, const void* p)
  {return out << *static_cast<const char*>(p) << '\n';}

int main() {
    int i = 42;
    const int* ip = &i;
    char c = 'A';
    const void* vp = &c;
    printPointee(std::cout, ip);
    printPointee(std::cout, vp);
}

/* Output:
42
A
*/
```

Explicitly-Specialized Class Template: is_void

```
template <class T>
struct is_void
  {static constexpr bool value = false;};

template <>
struct is_void<void>
  {static constexpr bool value = true;};

template <>
struct is_void<const void>
  {static constexpr bool value = true;};

template <>
struct is_void<volatile void>
  {static constexpr bool value = true;};

template <>
struct is_void<const volatile void>
  {static constexpr bool value = true;};

static_assert(is_void<int>::value == false, "");
static_assert(is_void<double*>::value == false, "");
static_assert(is_void<void>::value == true, "");
static_assert(is_void<const void>::value == true, "");
static_assert(is_void<volatile void>::value == true , "");
static_assert(is_void<const volatile void>::value == true,
  "");

int main() {}
```

Partially-Specialized Class Template

```
#include <iostream>

// unspecialized version
template <typename T, typename V>
struct Widget {
    Widget() {std::cout << "unspecialized\n";}
};

// partial specialization
template <typename T>
struct Widget<int, T> {
    Widget() {std::cout << "partial\n";}
};

// explicit specialization
template <>
struct Widget<int, int> {
    Widget() {std::cout << "explicit\n";}
};

int main() {
    Widget<double, int> w1; // unspecialized verion
    Widget<int, double> w2; // partial specialization
    Widget<int, int> w3; // explicit specialization
}
```

Partially-Specialized Class Template: std::vector

- std::vector class employs specialization
- consider vector of elements of type T
- most natural way to store elements is as array of T
- if T is **bool**, such an approach makes very inefficient use of memory, since each **bool** object requires one byte of storage
- if T is **bool**, would be much more memory-efficient to use array of, say, **unsigned char** and pack multiple **bool** objects in each byte
- std::vector accomplishes this by providing (partial) specialization for case that T is **bool**
- declaration of base template for std::vector and its partial specialization for case when T is **bool** are as follows:

```
template <class T, class Alloc = allocator<T>>
class vector; // unspecialized version

template <class Alloc>
class vector<bool, Alloc>; // partial specialization
```

Explicitly-Specialized Variable Template: `is_void_v`

```cpp
template <class T>
constexpr bool is_void_v = false;

template <>
constexpr bool is_void_v<void> = true;

template <>
constexpr bool is_void_v<const void> = true;

template <>
constexpr bool is_void_v<volatile void> = true;

template <>
constexpr bool is_void_v<const volatile void> = true;

static_assert(is_void_v<int> == false, "");
static_assert(is_void_v<double*> == false, "");
static_assert(is_void_v<void> == true, "");
static_assert(is_void_v<const void> == true, "");
static_assert(is_void_v<volatile void> == true , "");
static_assert(is_void_v<const volatile void> == true, "");

int main() {}
```

Explicitly-Specialized Variable Template: factorial

```
template <unsigned long long N>
constexpr unsigned long long
  factorial = N * factorial<N - 1>;

template <>
constexpr unsigned long long
  factorial<0> = 1;

int main() {
    static_assert(factorial<5> == 120,
      "factorial<5> failed");
    static_assert(factorial<12> == 479'001'600,
      "factorial<12> failed");
}
```

Partially-Specialized Variable Template: quotient

```
#include <limits>

// unspecialized version
template <int X, int Y>
constexpr int quotient = X / Y;

// partial specialization (which prevents division by zero)
template <int X>
constexpr int quotient<X, 0> = (X < 0) ?
  std::numeric_limits<int>::min() :
  std::numeric_limits<int>::max();

static_assert(quotient<4, 2> == 2, "");
static_assert(quotient<5, 3> == 1, "");
static_assert(quotient<4, 0> ==
  std::numeric_limits<int>::max(), "");
static_assert(quotient<-4, 0> ==
  std::numeric_limits<int>::min(), "");

int main() {}
```

Section 2.5.7

Miscellany

Overload Resolution and Substitution Failure

- when creating candidate set (of functions) for overload resolution, some or all candidates of that set may be result of instantiated templates with template arguments substituted for corresponding template parameters
- process of substituting template arguments for corresponding template parameters can lead to invalid code
- if certain types of invalid code result from substitution in any of following, **substitution failure** said to occur:
  - all types used in function type (i.e., return type and types of all parameters)
  - all types used in template parameter declarations
  - all expressions used in function type
  - all expressions used in template parameter declaration
- substitution failure ***not treated as error***
- instead, substitution failure simply causes overload to be ***removed from candidate set***
- this behavior often referred to by term "***substitution failure is not an error (SFINAE)***"
- SFINAE behavior often exploited in template metaprogramming

Some Kinds of Substitution Failures

- attempting to instantiate pack expansion containing multiple parameter packs of differing lengths
- attempting to create array with element type that is **`void`**, function type, reference type, or abstract class type
- attempting to create array with size that is zero or negative
- attempting to use type that is not class or enumeration type in qualified name
- attempting to use type in nested name specifier of qualified ID, when type does not contain specified member, or
  - specified member is not type where type is required
  - specified member is not template where template is required
  - specified member is not non-type where non-type is required
- attempting to create pointer to reference type
- attempting to create reference to **`void`**

Some Kinds of Substitution Failures (Continued)

- attempting to create pointer to member of `T` when `T` is not class type
- attempting to give invalid type to non-type template parameter
- attempting to perform invalid conversion in either template argument expression, or expression used in function declaration
- attempting to create function type in which parameter has type of **`void`**, or in which return type is function type or array type
- attempting to create function type in which parameter type or return type is abstract class

SFINAE Example: Truncate

```cpp
class Real {
public:
    using rounded_type = long long;
    rounded_type truncate() const {
        rounded_type result;
        // ...
        return result;
    }
    // ...
};

// function 1
template <class T>
typename T::rounded_type truncate(const T& x) {return x.truncate();}
  // NOTE: example would not compile if return type specified as auto

// function 2
int truncate(double x) {return x;}

int main() {
    Real r;
    float f = 3.14f;
    auto rounded_r = truncate(r);
      // calls function 1 (only trivial conversions)
    auto rounded_f = truncate(f);
      // function 2 requires nontrivial conversions
      // function 1 would only require trivial conversions but
      //  substitution failure occurs
      // calls function 2 (with conversions)
}
```

[link: overload resolution]

SFINAE Example: Truncate Revisited

```
class Real {
public:
    using rounded_type = long long;
    rounded_type truncate() const {
        rounded_type result;
        // ...
        return result;
    }
    // ...
};

// function 1
template <class T, class = typename T::rounded_type>
auto truncate(const T& x) {return x.truncate();}

// function 2
int truncate(double x) {return x;}

int main() {
    Real r;
    float f = 3.14f;
    auto rounded_r = truncate(r);
      // calls function 1 (only trivial conversions)
    auto rounded_f = truncate(f);
      // function 2 requires nontrivial conversions
      // function 1 would only require trivial conversions but
      //  substitution failure occurs
      // calls function 2 (with conversions)
}
```

`std::enable_if` and `std::enable_if_t`

- to make SFINAE more convenient to exploit, class template `std::enable_if` and alias template `std::enable_if_t` are provided
- declaration of class template `enable_if`:

```
template <bool B, class T = void>
struct enable_if;
```

- if `B` is **`true`**, class has member type `type` defined as `T`; otherwise, class has no `type` member
- possible implementation of `enable_if`:

```
template <bool B, class T = void>
struct enable_if {};

template <class T>
struct enable_if<true, T> {
    using type = T;
};
```

- declaration of alias template `enable_if_t`:

```
template <bool B, class T = void>
using enable_if_t = typename enable_if<B, T>::type;
```

- if `enable_if_t` is used with its first parameter as **`false`**, substitution failure will result

SFINAE Example: Modulo

```
#include <type_traits>
#include <cassert>
#include <iostream>

// ISO-Pascal modulo operator for signed integral types
template <class T> inline
std::enable_if_t<std::is_integral_v<T> && std::is_signed_v<T>, T>
mod(T x, T y) {
    assert(y > 0);
    if (x < 0) {x += (((-x) / y) + 1) * y;}
    return x % y;
}

// ISO-Pascal modulo operator for unsigned integral types
template <class T> inline
std::enable_if_t<std::is_integral_v<T> && std::is_unsigned_v<T>, T>
mod(T x, T y)
  {return x % y;}

int main() {
    auto si = mod(-4, 3); // uses signed version
    auto ui = mod(5u, 3u); // uses unsigned version
    auto slli = mod(-5ll, 3ll); // uses signed version
    auto ulli = mod(4ull, 3ull); // uses unsigned version
    // auto f = mod(3.0, 4.0);
      // ERROR: no matching function call
    std::cout << si << ' ' << ui << ' ' << slli << ' ' << ulli << '\n';
}
```

Detection Idiom Example

```
#include <iostream>
#include <experimental/type_traits>

class Widget {
public:
    void foo() const {}
    // ...
};

class Gadget {
public:
    void foo() {}
    // ...
};

// helper template for testing if class has member function called
// foo that can be invoked on const object with no arguments.
template <class T>
using has_usable_foo_t = decltype(std::declval<const T&>().foo());

int main() {
    std::cout
      << "Widget "
      << std::experimental::is_detected_v<has_usable_foo_t, Widget>
      << '\n'
      << "Gadget "
      << std::experimental::is_detected_v<has_usable_foo_t, Gadget>
      << '\n';
}
```

Section 2.5.8

References

References I

1. D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison Wesley, 2002.

2. P. Sommerlad. Variadic and variable templates. *Overload*, 126:14–17, Apr. 2015. Available online at http://accu.org/index.php/journals/2087.

3. A. Sutton. Introducing concepts. *Overload*, 129:4–8, Oct. 2015. Available online at http://accu.org/index.php/journals/2157.

4. A. Sutton. Defining concepts. *Overload*, 131:4–8, Feb. 2016. Available online at http://accu.org/index.php/journals/2198.

Talks I

1. Peter Sommerlad. Variadic Templates in C++11/C++14: An Introduction, CppCon, Bellevue, WA, USA, Sept. 21, 2015. Available online at `https://youtu.be/R1G3P5SRXCw`.
2. Arthur O'Dwyer. Template Normal Programming. CppCon, Bellevue, WA, USA, Sept. 19, 2016. Available online at `https://youtu.be/vwrXHznaYLA` and `https://youtu.be/VIz6xBvwYd8`. (This talk is split into two parts.)
3. Arthur O'Dwyer, A Soupcon of SFINAE, CppCon, Bellevue, WA, USA, Sept. 27, 2017. Available online at `https://youtu.be/ybaE9qlhHvw`.
4. Marshall Clow, The Detection Idiom: A Better Way to SFINAE, C++Now, Aspen, CO, USA, May 19, 2017. Available online at `https://youtu.be/U3jGdnRL3KI`.

   Notwithstanding the talk's title, this talk is actually about the functionality in the Library Fundamentals TS related to `is_detected`, `detected_or`, `is_detected_exact`, and `is_detected_convertible`.

Talks II

5. Walter E. Brown, Modern Template Metaprogramming: A Compendium, Part I, CppCon, Bellevue, WA, USA, Sept. 9, 2014. Available online at `https://youtu.be/Am2is2QCvxY`.
6. Walter E. Brown, Modern Template Metaprogramming: A Compendium, Part II, CppCon, Bellevue, WA, USA, Sept. 9, 2014. Available online at `https://youtu.be/a0FliKwcwXE`.

Section 2.6

Lambda Expressions

Motivation for Lambda Expressions

- functor classes extremely useful, especially for generic programming
- writing definitions of functor classes somewhat tedious, especially if many such classes
- functor classes all have same general structure (i.e., constructor, function-call operator, zero or more data members)
- would be nice if functor could be created without need to explicitly write functor-class definition
- lambda expressions provide compact notation for creating functors
- convenience feature (not fundamentally anything new that can be done with lambda expressions that could not already have been done without them)

Lambda Expressions

- lambda expression consists of:
  1. introducer: ***capture list*** in square brackets
  2. declarator: ***parameter list*** in parentheses followed by ***return type*** using trailing return-type syntax
  3. ***compound statement*** in brace brackets
- capture list specifies objects to be captured as data members
- declarator specifies parameter list and return type of function-call operator
- compound statement specifies body of function-call operator
- if no declarator specified, defaults to `()`
- if no return type specified, defaults to type of expression in return statement, or void if no return statement
- when evaluated, lambda expression yields object called **closure** (which is essentially a functor)
- examples:

```
[](double x)->int{return floor(x);}
[](int x, int y){return x < y;}
[]{std::cout << "Hello, World!\n";}
```

Lambda Expressions (Continued)

- closure object is unnamed (temporary object)
- closure type is unnamed
- **`operator()`** is always inline
- **`operator()`** is const member function unless **`mutable`** keyword used
- if closure type is literal type, all members of closure type automatically constexpr
- if no capture, closure type provides conversion function to pointer to function having same parameter and return types as closure type's function call operator; value returned is address of function that, when invoked, has same effect as invoking closure type's function call operator (function pointer not tied to lifetime of closure object)
- although **`operator()`** in closure very similar to case of normal functor, not everything same (e.g., **`operator()`** member in closure type cannot access **`this`** pointer for closure type)

Hello World Program Revisited

```
#include <iostream>

int main() {
    []{std::cout << "Hello, World!\n";}();
}
```

```
#include <iostream>

struct Hello {
    void operator()() const {
        std::cout << "Hello, World!\n";
    }
};

int main() {
    Hello hello;
    hello();
}
```

Comparison Functor Example

```
#include <iostream>
#include <algorithm>
#include <cstdlib>
#include <vector>

int main() {
    std::vector<int> v{-3, 3, 4, 0, -2, -1, 2, 1, -4};
    std::sort(v.begin(), v.end(),
      [](int x, int y) {return std::abs(x) < std::abs(y);});
    for (auto x : v) std::cout << x << '\n';
}
```

```
#include <iostream>
#include <algorithm>
#include <cstdlib>
#include <vector>

struct abs_less {
    bool operator()(int x, int y) const
      {return std::abs(x) < std::abs(y);}
};

int main() {
    std::vector<int> v{-3, 3, 4, 0, -2, -1, 2, 1, -4};
    std::sort(v.begin(), v.end(), abs_less());
    for (auto x : v) std::cout << x << '\n';
}
```

Capturing Objects

- locals only available if captured; non-locals always available
- can capture by value or by reference
- different locals can be captured differently
- can specify default capture mode
- can explicitly list objects to be captured or not
- might be wise to explicitly list all objects to be captured (when practical) to avoid capturing objects accidentally (e.g., due to typos)
- in member function, to capture class object by value, capture `*this`
- in member function, can also capture `this`
- `this` must be captured by value

- (unary version of) `std::transform` applies given (unary) operator to each element in range specified by pair of iterators and writes result to location specified by another iterator
- definition of `std::transform` would typically resemble:

```
template <class InputIterator, class OutputIterator,
  class UnaryOperator>
  OutputIterator transform(InputIterator first,
  InputIterator last, OutputIterator result,
  UnaryOperator op) {
    while (first != last) {
        *result = op(*first);
        ++result;
        ++first;
    }
    return result;
}
```

Modulus Example

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    int m = 2;
    std::vector<int> v{0, 1, 2, 3};
    std::transform(v.begin(), v.end(), v.begin(),
      [m](int x){return x % m;});
    for (auto x : v) std::cout << x << '\n';
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>

class mod {
public:
    mod(int m_) : m(m_) {}
    int operator()(int x) const {return x % m;}
private:
    int m;
};

int main() {
    int m = 2;
    std::vector<int> v{0, 1, 2, 3};
    std::transform(v.begin(), v.end(), v.begin(), mod(m));
    for (auto x : v) std::cout << x << '\n';
}
```

Modulus Example: Without Lambda Expression

```
#include <iostream>
#include <vector>
#include <algorithm>

class mod {
public:
    mod(int m_) : m(m_) {}
    int operator()(int x) const {return x % m;}
private:
    int m;
};

int main() {
    int m = 2;
    std::vector<int> v{0, 1, 2, 3};
    std::transform(v.begin(), v.end(), v.begin(), mod(m));
    for (auto x : v) std::cout << x << '\n';
}
```

- approximately 8.5 lines of code to generate functor

Modulus Example: With Lambda Expression

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    int m = 2;
    std::vector<int> v{0, 1, 2, 3};
    std::transform(v.begin(), v.end(), v.begin(),
      [m](int x){return x % m;});
    for (auto x : v) std::cout << x << '\n';
}
```

- m captured by value
- approximately 0.5 lines of code to generate functor

- `std::for_each` applies given function/functor to each element in range specified by pair of iterators
- definition of `std::for_each` would typically resemble:

```
template<class InputIterator, class Function>
  Function for_each(InputIterator first,
  InputIterator last, Function func) {
    while (first != last) {
        func(*first);
        ++first;
    }
    return move(func);
}
```

Product Example

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v{2, 3, 4};
    int prod = 1;
    std::for_each(v.begin(), v.end(),
      [&prod](int x)->void{prod *= x;});
    std::cout << prod << '\n';
}
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

class cum_prod {
public:
    cum_prod(int& prod_) : prod(prod_) {}
    void operator()(int x) const {prod *= x;}
private:
    int& prod;
};

int main() {
    std::vector<int> v{2, 3, 4};
    int prod = 1;
    std::for_each(v.begin(), v.end(), cum_prod(prod));
    std::cout << prod << '\n';
}
```

Product Example: Without Lambda Expression

```
#include <iostream>
#include <vector>
#include <algorithm>

class cum_prod {
public:
    cum_prod(int& prod_) : prod(prod_) {}
    void operator()(int x) const {prod *= x;}
private:
    int& prod;
};

int main() {
    std::vector<int> v{2, 3, 4};
    int prod = 1;
    std::for_each(v.begin(), v.end(), cum_prod(prod));
    std::cout << prod << '\n';
}
```

- approximately 8.5 lines of code to generate functor

Product Example: With Lambda Expression

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v{2, 3, 4};
    int prod = 1;
    std::for_each(v.begin(), v.end(),
      [&prod](int x)->void{prod *= x;});
    std::cout << prod << '\n';
}
```

- `prod` captured by reference
- approximately 1 line of code to generate functor

More Variations on Capture

```
double a = 2.14;
double b = 3.14;
double c = 42.0;

// capture all objects by reference (i.e., a, b, and c)
[&](double x, double y){return a * x + b * y + c;}

// capture all objects by value (i.e., a, b, and c)
[=](double x, double y){return a * x + b * y + c;}

// capture all objects by value, except a
// which is captured by reference
[=,&a](double x, double y){return a * x + b * y + c;}

// capture all objects by reference, except a
// which is captured by value
[&,a](double x, double y){return a * x + b * y + c;}
```

Generalized Lambda Capture

- can specify name for captured object in closure type

```
int a = 1;
auto f = [x = a]() {return x;};
```

- can capture result of expression (e.g., to perform move instead of copy or to add arbitrary new state to closure type)

```
std::vector<int> v(1000, 1);
auto f = [v = std::move(v)]() ->
  const std::vector<int>& {return v;};
```

Generalized Lambda Capture Example

```
#include <iostream>

int main() {
    int x = 0;
    int y = 1;
    auto f = [&count = x, inc = y + 1](){
        return count += inc;
    };
    std::cout << f() << ' ';
    std::cout << f() << '\n';
}

// output: 2 4
```

Generic Lambda Expressions

- can allow compiler to deduce type of lambda function parameters
- generates closure type with templated function-call operator
- one template type parameter for each occurrence of **`auto`** in lambda expression's parameter declaration clause

Generic Lambda Expression Example [Generic]

```
#include <iostream>
#include <complex>
#include <string>

int main() {
    using namespace std::literals;
    auto add = [](auto x, auto y) {return x + y;};
    std::cout << add(1, 2) << ' ' << add(1.0, 2.0) << ' '
      << add(1.0, 2.0i) << ' ' << add("Jell"s, "o"s) << '\n';
}
```

```
#include <iostream>
#include <complex>
#include <string>

struct Add {
    template <class T, class U>
    auto operator()(T x, U y) {return x + y;};
};

int main() {
    using namespace std::literals;
    Add add;
    std::cout << add(1, 2) << ' ' << add(1.0, 2.0) << ' '
      << add(1.0, 2.0i) << ' ' << add("Jell"s, "o"s) << '\n';
}
```

Generic Lambda Expression Example [Convenience]

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7};
    // sort elements of vector in descending order
    std::sort(v.begin(), v.end(),
      [](auto i, auto j) {return i > j;});
    std::for_each(v.begin(), v.end(),
      [](auto i) {std::cout << i << '\n';});
}
```

Dealing With Unnamed Types

- fact that closure types unnamed causes complications when need arises to refer to closure type
- helpful language features: **`auto`**, **`decltype`**
- helpful library features: `std::function`
- closures can be stored using **`auto`** or `std::function`
- closures that do not capture can be “stored” by assigning to function pointer

Using auto, decltype, and std::function

```cpp
#include <iostream>
#include <functional>

std::function<double(double)> linear(double a, double b) {
    return [=](double x){return a * x + b;};
}

int main() {
    // type of f is std::function<double(double)>
    auto f = linear(2.0, -1.0);
    // g has closure type
    auto g = [](double x){return 2.0 * x - 1.0;};
    double (*u)(double) = [](double x){return 2.0 * x - 1.0;};
    // h has same type as g
    decltype(g) h = g;
    for (double x = 0.0; x < 10.0; x += 1.0) {
        std::cout << x << ' ' << f(x) << ' ' << g(x) <<
          ' ' << h(x) << (*u)(x) << '\n';
    }
}
```

- applying function-call operator to `f` much slower than in case of `g` and `h`
- when `std::function` used, inlining of called function probably not possible
- when functor used directly (via function-call operator) inlining is very likely
- prefer **auto** over `std::function` for storing closures

`operator()` as Non-const Member

```
#include <iostream>

int main()
{
    int count = 5;
    // Must use mutable in order to be able to
    // modify count member.
    auto get_count = [count]() mutable -> int {
        return count++;
    };

    int c;
    while ((c = get_count()) < 10) {
        std::cout << c << '\n';
    }
}
```

- **`operator()`** is declared as const member function unless **`mutable`** keyword used
- const member function cannot change (non-static) data members

Constexpr Lambdas

```
#include <iostream>
#include <array>

template <typename T>
constexpr auto multiply_by(T i) {
    return [i](auto j) {return i * j;};
      // OK: lambda is literal type so members
      // are automatically constexpr
}

int main() {
    constexpr auto mult_by_2 = multiply_by(2);
    std::array<int, mult_by_2(8)> a;
    std::cout << a.size() << '\n';
}
```

Comparison Functors for Containers

```
#include <iostream>
#include <vector>
#include <set>

int main() {
    // The following two lines are the only important ones:
    auto cmp = [](int* x, int* y){return *x < *y;};
    std::set<int*, decltype(cmp)> s(cmp);

    // Just for something to do:
    // Print the elements of v in sorted order with
    // duplicates removed.
    std::vector<int> v = {4, 1, 3, 2, 1, 1, 1, 1};
    for (auto& x : v) {
        s.insert(&x);
    }
    for (auto x : s) {
        std::cout << *x << '\n';
    }
}
```

- note that `s` is not default constructed
- since closure types not default constructible, following would fail:
  `std::set<int*, decltype(cmp)> s;`
- note use of decltype in order to specify type of functor

What Could Possibly Go Wrong?

```cpp
#include <iostream>
#include <vector>
#include <functional>

std::vector<int> vec{2000, 4000, 6000, 8000, 10000};
std::function<int(int)> func;

void do_stuff()
{
    int modulus = 10000;
    func = [&](int x){return x % modulus;};
    for (auto x : vec) {
        std::cout << func(x) << '\n';
    }
}

int main()
{
    do_stuff();
    for (auto x : vec) {
        std::cout << func(x) << '\n';
    }
}
```

- above code has very serious bug; what is it?

Dangling References

- if some objects captured by reference, closure can hold dangling references
- responsibility of programmer to avoid such problems
- if will not cause performance issues, may be advisable to capture by value (to avoid problem of dangling references)
- dangling-reference example:

```
#include <iostream>
#include <functional>

std::function<double(double)> linear(double a, double b) {
    return [&](double x){return a * x + b;};
}

int main() {
    auto f = linear(2.0, -1.0);
    // bad things will happen here
    std::cout << f(1.0) << '\n';
}
```

Section 2.6.1

References

Talks I

1. Herb Sutter. Lambdas, Lambdas Everywhere, Professional Developers Conference (PDC), Redmond, WA, USA, Oct. 27–29, 2010. Available online at `https://youtu.be/rcgRY7sOA58`.
2. Herb Sutter. C++0x Lambda Functions, Northwest C++ Users' Group (NWCPP), Redmond, WA, USA, May 18, 2011. Available online at `https://vimeo.com/23975522`.

Section 2.7

Classes and Inheritance

Section 2.7.1

Derived Classes and Class Hierarchies

Derived Classes

- sometimes, want to express commonality between classes
- want to create new class from existing class by adding new members or replacing (i.e., hiding/overriding) existing members
- can be achieved through language feature known as ***inheritance***
- generate new class with all members of already existing class, excluding special member functions (i.e., constructors, assignment operators, and destructor)
- new class called **derived class** and original class called **base class**
- derived class said to **inherit** from base class
- can add new members (not in base class) to derived class
- can hide or override member functions from base class with new version
- syntax for specifying derived class:

  `class` *derived_class* : *base_class_specifiers*

- *derived_class* is name of derived class; *base_class_specifiers* provide base-class information

Derived Classes (Continued)

- can more clearly express intent by explicitly identifying relationship between classes
- can facilitate code reuse by leverage existing code
- interface inheritance: allow different derived classes to be used interchangeably through interface provided by common base class
- implementation inheritance: save implementation effort by sharing capabilities provided by base class

```
#include <string>

class Person {
public:
    Person(const std::string& family_name,
      const std::string& given_name) :
      family_name_(family_name), given_name_(given_name) {}
    std::string family_name() const {return family_name_;}
    std::string given_name() const {return given_name_;}
    std::string full_name() const
      {return family_name_ + ", " + given_name_;}
    // ...
private:
    std::string family_name_;
    std::string given_name_;
};
```

Student Class Without Inheritance

```cpp
#include <string>

class Student {
public:
    Student(const std::string& family_name,
      const std::string& given_name) :
      family_name_(family_name), given_name_(given_name) {}
      // NEW
    std::string family_name() const {return family_name_;}
    std::string given_name() const {return given_name_;}
    std::string full_name() const
      {return family_name_ + ", " + given_name_;}
    std::string student_id() {return student_id_;} // NEW
private:
    std::string family_name_;
    std::string given_name_;
    std::string student_id_; // NEW
};
```

Student Class With Inheritance

```
// include definition of Person class here

class Student : public Person {
public:
    Student(const std::string& family_name,
      const std::string& given_name,
      const std::string& student_id) :
      Person(family_name, given_name),
      student_id_(student_id) {}
    std::string student_id() {return student_id_;}
private:
    std::string student_id_;
};
```

Complete Inheritance Example

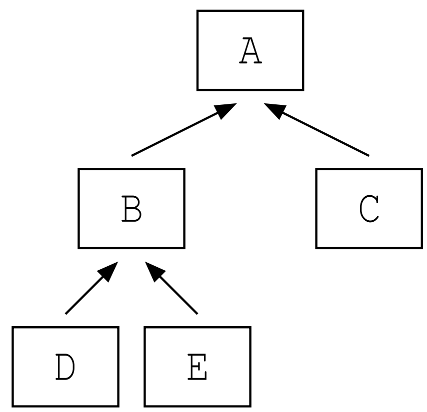
```
#include <string>

class Person {
public:
    Person(const std::string& family_name,
      const std::string& given_name) :
      family_name_(family_name), given_name_(given_name) {}
    std::string family_name() const {return family_name_;}
    std::string given_name() const {return given_name_;}
    std::string full_name() const
      {return family_name_ + ", " + given_name_;}
    // ... (including virtual destructor)
private:
    std::string family_name_;
    std::string given_name_;
};

class Student : public Person {
public:
    Student(const std::string& family_name,
      const std::string& given_name,
      const std::string& student_id) :
      Person(family_name, given_name),
      student_id_(student_id) {}
    std::string student_id() {return student_id_;}
private:
    std::string student_id_;
};
```

Class Hierarchies

- inheritance relationships between classes form what is called **class hierarchy**
- often class hierarchy represented by directed (acyclic) graph, where nodes correspond to classes and edges correspond to inheritance relationships
- class definitions:

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public A { /* ... */ };
class D : public B { /* ... */ };
class E : public B { /* ... */ };
```

- inheritance diagram:

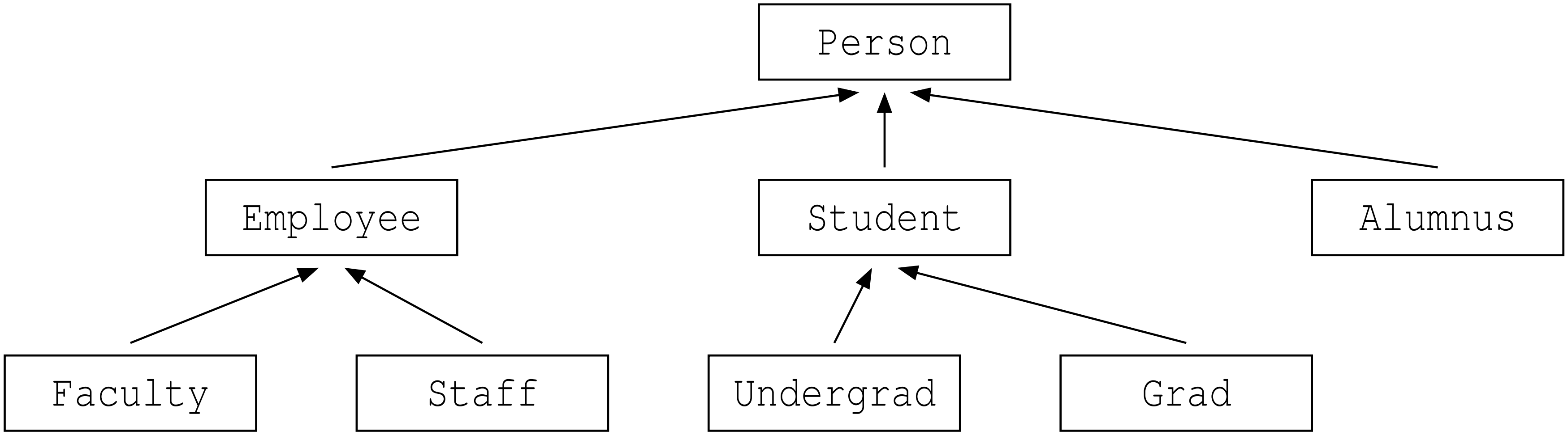


Class Hierarchy Example

- class definitions:

```
class Person { /* ... */ };
class Employee : public Person { /* ... */ };
class Student : public Person { /* ... */ };
class Alumnus : public Person { /* ... */ };
class Faculty : public Employee { /* ... */ };
class Staff : public Employee { /* ... */ };
class Grad : public Student { /* ... */ };
class Undergrad : public Student { /* ... */ };
```

- inheritance diagram:



- each of `Employee`, `Student`, and `Alumnus` is a `Person`; each of `Faculty` and `Staff` is an `Employee`; each of `Undergrad` and `Grad` is a `Student`

Member Access Specifiers: `protected`

- earlier, introduced **`public`** and **`private`** access specifiers for class members
- in context of inheritance, another access specifier becomes relevant, namely, **`protected`**
- member declared in protected section of class can only be accessed by
  - member functions and friends of that class; and
  - by member functions and friends of ***derived classes***
- protected members used to provide developers of derived classes access to some inner workings of base class without exposing such inner workings to everyone
- usually, bad idea to use protected access for data members (for similar reasons that using public access for data members is usually bad)
- protected access usually employed for function members

Types of Inheritance

- three types of inheritance with respect to access protection: public, protected, and private
- these three types of inheritance differ in terms of accessibility, in derived class, of members inherited from base class
- private parts of base class are always inaccessible in derived class, regardless of whether public, protected, or private inheritance used
- if this were not case, all access protection could simply be bypassed by using inheritance
- access specifiers for members accessible in derived class chosen as follows:

Access Specifier in Base Class	Access Specifier in Derived Class: Public Inheritance	Access Specifier in Derived Class: Protected Inheritance	Access Specifier in Derived Class: Private Inheritance
public	public	protected	private
protected	protected	protected	private

Types of Inheritance (Continued)

- for struct, defaults to public inheritance
- for class, defaults to private inheritance
- public and protected/private inheritance have different use cases, as we will see later

Inheritance and Member Access Example

```
class Base {
public:
    void f();
protected:
    void g();
private:
    int x;
};

class Derived_1 : public Base {
    // f is public
    // g is protected
    // x is not accessible from Derived_1
};

class Derived_2 : protected Base {
    // f is protected
    // g is protected
    // x is not accessible from Derived_2
};

class Derived_3 : private Base {
    // f is private
    // g is private
    // x is not accessible from Derived_3
};
```

Public Inheritance Example

```
class Base {
public:
    void func_1();
protected:
    void func_2();
private:
    int x_;
};

class Derived : public Base {
public:
    void func_3() {
        func_1(); // OK
        func_2(); // OK
        x_ = 0; // ERROR: inaccessible
    }
};

struct Widget : public Derived {
    void func_4() { func_2(); } // OK
};

int main() {
    Derived d;
    d.func_1(); // OK
    d.func_2(); // ERROR: inaccessible
    d.x_ = 0; // ERROR: inaccessible
}
```

Protected Inheritance Example

```
class Base {
public:
    void func_1();
protected:
    void func_2();
private:
    int x_;
};

class Derived : protected Base {
public:
    void func_3() {
        func_1(); // OK
        func_2(); // OK
        x_ = 0; // ERROR: inaccessible
    }
};

struct Widget : public Derived {
    void func_4() { func_2(); } // OK
};

int main() {
    Derived d; // OK: defaulted constructor is public
    d.func_1(); // ERROR: inaccessible
    d.func_2(); // ERROR: inaccessible
    d.x_ = 0; // ERROR: inaccessible
}
```

Private Inheritance Example

```
class Base {
public:
    void func_1();
protected:
    void func_2();
private:
    int x_;
};

class Derived : private Base {
public:
    void func_3() {
        func_1(); // OK
        func_2(); // OK
        x_ = 0; // ERROR: inaccessible
    }
};

struct Widget : public Derived {
    void func_4() { func_2(); } // ERROR: inaccessible
};

int main() {
    Derived d; // OK: defaulted constructor is public
    d.func_1(); // ERROR: inaccessible
    d.func_2(); // ERROR: inaccessible
    d.x_ = 0; // ERROR: inaccessible
}
```

Public Inheritance

- public inheritance is inheritance in traditional object-oriented programming sense
- public inheritance models an ***is-a*** relationship (i.e., derived class object is a base class object)
- most common form of inheritance
- inheritance relationship visible to all code

Public Inheritance Example

```
#include <string>

class Person {
public:
    Person(const std::string& family_name, const std::string&
      given_name) : family_name_(family_name),
      given_name_(given_name) {}
    std::string family_name() const
      {return family_name_;}
    std::string given_name() const
      {return given_name_;}
    std::string full_name() const
      {return family_name_ + ", " + given_name_;}
private:
    std::string family_name_;
    std::string given_name_;
};

class Student : public Person {
public:
    Student(const std::string& family_name, const std::string&
      given_name, const std::string& student_id) :
      Person(family_name, given_name), student_id_(student_id) {}
    std::string student_id()
      {return student_id_;}
private:
    std::string student_id_;
};
```

Protected and Private Inheritance

- protected and private inheritance not inheritance in traditional object-oriented programming sense (i.e., no is-a relationship)
- form of implementation inheritance
- ***implemented-in-terms-of*** relationship (i.e., derived class object implemented in terms of a base class object)
- in case of protected inheritance, inheritance relationship only seen by derived classes and their friends and class itself and its friends
- in case of private inheritance, inheritance relationship only seen by class itself and its friends (not derived classes and their friends)
- except in special circumstances, normally bad idea to use inheritance for composition
- one good use case for private/protected inheritance is in policy-based design, which exploits empty base optimization (EBO)

Policy-Based Design Example: Inefficient Memory Usage

```cpp
#include <mutex>

class ThreadSafePolicy {
public:
    void lock() {mutex_.lock();}
    void unlock() {mutex_.unlock();}
private:
    std::mutex mutex_;
};

class ThreadUnsafePolicy {
public:
    void lock() {} // no-op
    void unlock() {} // no-op
};

template<class ThreadSafetyPolicy>
class Widget {
    ThreadSafetyPolicy policy_;
    // ...
};

int main() {
    Widget<ThreadUnsafePolicy> w;
      // w.policy_ has no data members, but
      // sizeof(w.policy_) >= 1
      // inefficient use of memory
}
```

Policy-Based Design Example: Private Inheritance and EBO

```
#include <mutex>

class ThreadSafePolicy {
public:
    void lock() {mutex_.lock();}
    void unlock() {mutex_.unlock();}
private:
    std::mutex mutex_;
};

class ThreadUnsafePolicy {
public:
    void lock() {} // no-op
    void unlock() {} // no-op
};

template<class ThreadSafetyPolicy>
class Widget : ThreadSafetyPolicy {
    // ...
};

int main() {
    Widget<ThreadUnsafePolicy> w;
      // empty-base optimization (EBO) can be applied
      // no memory overhead for no-op thread-safety policy
}
```

Inheritance and Constructors

- by default, constructors not inherited
- often, derived class introduces new data members not in base class
- since base-class constructors cannot initialize derived-class data members, inheriting constructors from base class by default would be bad idea (e.g., could lead to uninitialized data members)
- in some cases, however, base-class constructors may be sufficient to initialize derived-class objects
- in such cases, can inherit all non-special base-class constructors with **`using`** statement
- special constructors (i.e., default, copy, and move constructors) cannot be inherited
- constructors to be inherited with **`using`** statement may still be hidden by constructors in derived class

Inheriting Constructors Example

```
class Base {
public:
    Base() : i_(0.0), j_(0) {}
    Base(int i) : i_(i), j_(0) {}
    Base(int i, int j) : i_(i), j_(j) {}
    // ... (other non-constructor members)
private:
    int i_, j_;
};

class Derived : public Base {
public:
    // inherit non-special constructors from Base
    // (default constructor not inherited)
    using Base::Base;
    // default constructor is implicitly declared and
    // not inherited
};

int main() {
    Derived a;
      // invokes non-inherited Derived::Derived()
    Derived b(42, 42);
      // invokes inherited Base::Base(int, int)
}
```

Inheriting Constructors Example

```
class Base {
public:
    Base() : i_(0), j_(0), k_(0) {}
    Base(int i, int j) : i_(i), j_(j), k_(0) {}
    Base(int i, int j, int k) : i_(i), j_(j), k_(k) {}
    // ... (other non-constructor members)
private:
    int i_, j_, k_;
};

class Derived : public Base {
public:
    // inherit non-special constructors from Base
    // (default constructor not inherited)
    using Base::Base;
    // following constructor hides inherited constructor
    Derived(int i, int j, int k) : Base(-i, -j, -k) {}
    // no implicitly-generated default constructor
};

int main() {
    Derived b(1, 2);
      // invokes inherited Base::Base(int, int)
    Derived c(1, 2, 3);
      // invokes Derived::Derived(int, int, int)
    // following would produce compile-time error:
    // Derived a; // ERROR: no default constructor
}
```

Inheritance, Assignment Operators, and Destructors

- by default, assignment operators not inherited (for similar reasons as in case of constructors)
- can inherit all non-special base-class assignment operators with **`using`** statement
- copy and move assignment operators cannot be inherited
- assignment operators to be inherited with **`using`** statement may still be hidden by assignment operators in derived class
- cannot inherit destructor

Inheriting Assignment Operators Example

```
class Base {
public:
    explicit Base(int i) : i_(i) {}
    Base& operator=(int i) {
        i_ = i;
        return *this;
    }
    // ...
private:
    int i_;
};

class Derived : public Base {
public:
    // inherit non-special constructors
    using Base::Base;
    // inherit non-special assignment operators
    using Base::operator=;
    // ...
};

int main() {
    Derived d(0);
      // invokes inherited Base::Base(int)
    d = 42;
      // invokes inherited Base::operator=(int)
}
```

Construction and Destruction Order

- during construction of object, all of its base class objects constructed first
- order of construction:
  1. ***base class objects*** as listed in type definition left to right
  2. data members as listed in type definition top to bottom
  3. constructor body
- order of destruction is exact reverse of order of construction, namely:
  1. destructor body
  2. data members as listed in type definition bottom to top
  3. ***base class objects*** as listed in type definition right to left

Order of Construction

```cpp
#include <vector>
#include <string>

class Base {
public:
    Base(int n) : v_(n, 0) {}
    // ...
private:
    std::vector<char> v_;
};

class Derived : public Base {
public:
    Derived(const std::string& s) : Base(1024), s_(s)
      { i_ = 0; }
    // ...
private:
    std::string s_;
    int i_;
};

int main() {
    Derived d("hello");
}
```

- construction order for `Derived` constructor: 1) `Base` class object, 2) data member `s_`, 3) `Derived` constructor body (initializes data member `i_`)

Hiding Base-Class Member Functions in Derived Class

- can provide new versions of member functions in derived class to hide original functions in base class

```
#include <iostream>

class Fruit {
public:
    void print() const {std::cout << "fruit\n";}
};

class Apple : public Fruit {
public:
    void print() const {std::cout << "apple\n";}
};

class Banana : public Fruit {
public:
    void print() const {std::cout << "banana\n";}
};

int main() {
    Fruit f;
    Apple a;
    Banana b;
    f.print(); // calls Fruit::print
    a.print(); // calls Apple::print
    b.print(); // calls Banana::print
}
```

Upcasting

- derived-class object always has base-class subobject
- given reference or pointer to derived-class object, may want to find reference or pointer to corresponding base-class object
- **upcasting**: converting derived-class pointer or reference to base-class pointer or reference
- upcasting allows us to treat derived-class object as base-class object
- upcasting always safe in sense that cannot result in incorrect type (since every derived-class object is also a base-class object)
- can upcast without explicit type-cast operator as long as casted-to type is accessible; C-style cast can used to bypass access protection (although not recommended)
- example:

```
class Base { /* ... */ };
class Derived : public Base { /* ... */ };
void func() {
    Derived d;
    Base* bp = &d;
}
```

Downcasting

- **downcasting**: converting base-class pointer or reference to derived-class pointer or reference
- downcasting allows us to force base-class object to be treated as derived-class object
- downcasting is not always safe (since not every base-class object is necessarily also derived-class object)
- must only downcast when known that object actually has derived type (except in case of **`dynamic_cast`**)
- downcasting always requires explicit cast (e.g., **`static_cast`**, **`dynamic_cast`** for dynamically-checked cast in polymorphic case, or C-style cast)
- example:

```
class Base { /* ... (nonpolymorphic) */ };
class Derived : public Base { /* ... */ };
void func() {
    Derived d;
    Base* bp = &d;
    Derived* dp = static_cast<Derived*>(bp);
}
```

Upcasting/Downcasting Example

```
class Base { /* ... (nonpolymorphic) */ };

class Derived : public Base { /* ... */ };

int main() {
    Base b;
    Derived d;
    Base* bp = nullptr;
    Derived* dp = nullptr;
    bp = &d;
      // OK: upcast does not require explicit cast
    dp = bp;
      // ERROR: downcast requires explicit cast
    dp = static_cast<Derived*>(bp);
      // OK: downcast with explicit cast and
      // pointer (bp) refers to Derived object
    Base& br = d;
      // OK: upcast does not require explicit cast
    Derived& dr1 = *bp;
      // ERROR: downcast requires explicit cast
    Derived& dr2 = *static_cast<Derived*>(bp);
      // OK: downcast with explicit cast and
      // object (*bp) is of Derived type
    dp = static_cast<Derived*>(&b);
      // BUG: pointer (&b) does not refer to Derived object
}
```

Upcasting Example

```
class Base { /* ... */ };

class Derived : public Base { /* ... */ };

void func_1(Base& b) { /* ... */ }

void func_2(Base* b) { /* ... */ }

int main() {
    Base b;
    Derived d;
    func_1(b);
    func_1(d); // OK: Derived& upcast to Base&
    func_2(&b);
    func_2(&d); // OK: Derived* upcast to Base*
}
```

Nonpolymorphic Behavior

```
#include <iostream>
#include <string>

class Person {
public:
    Person(const std::string& family, const std::string& given) :
      family_(family), given_(given) {}
    void print() const {std::cout << "person: " << family_ << ',' << given_ << '\n';}
protected:
    std::string family_; // family name
    std::string given_; // given name
};

class Student : public Person {
public:
    Student(const std::string& family, const std::string& given,
      const std::string& id) : Person(family, given), id_(id) {}
    void print() const {
        std::cout << "student: " << family_ << ',' << given_ << ',' << id_ << '\n';
    }
private:
    std::string id_; // student ID
};

void processPerson(const Person& p) {
    p.print(); // always calls Person::print
    // ...
}

int main() {
    Person p("Ritchie", "Dennis");
    Student s("Doe", "John", "12345678");
    processPerson(p); // invokes Person::print
    processPerson(s); // invokes Person::print
}
```

- would be nice if `processPerson` called version of `print` that corresponds to ***actual*** type of object referenced by function parameter `p`

- **slicing**: copying or moving object of derived class to object of base class (e.g., during construction or assignment), losing part of information in so doing
- example:

```
class Base {
    // ...
    int x_;
};

class Derived : public Base {
    // ...
    int y_;
};

int main() {
    Derived d1, d2;
    Base b = d1;
      // slicing occurs
    Base& r = d1;
    r = d2;
      // more treacherous case of slicing
      // slicing occurs
      // d1 now contains mixture of d1 and d2
      // (i.e., base part of d2 and derived part of d1)
}
```

Inheritance and Overloading

- functions do not overload across scopes
- can employ **`using`** statement to bring base members into scope for overloading

Inheritance and Overloading Example

```
#include<iostream>

class Base {
public:
    double f(double d) const {return d;}
    // ...
};

class Derived : public Base {
public:
    int f(int i) const {return i;}
    // ...
};

int main()
{
    Derived d;
    std::cout << d.f(0) << '\n';
      // calls Derived::f(int) const
    std::cout << d.f(0.5) << '\n';
      // calls Derived::f(int) const; probably not intended
    Derived* dp = &d;
    std::cout << dp->f(0) << '\n';
      // calls Derived::f(int) const
    std::cout << dp->f(0.5) << '\n';
      // calls Derived::f(int) const; probably not intended
}
```

Using Base Members Example

```
#include<iostream>

class Base {
public:
    double f(double d) const {return d;}
    // ...
};

class Derived : public Base {
public:
    using Base::f; // bring Base::f into scope
    int f(int i) const {return i;}
    // ...
};

int main()
{
    Derived d;
    std::cout << d.f(0) << '\n';
      // calls Derived::f(int) const
    std::cout << d.f(0.5) << '\n';
      // calls Base::f(double) const
    Derived* dp = &d;
    std::cout << dp->f(0) << '\n';
      // calls Derived::f(int) const
    std::cout << dp->f(0.5) << '\n';
      // calls Base::f(double) const
}
```

Inheritance, Templates, and Name Lookup

- name lookup in templates takes place in two phases:
  1. at template definition time
  2. at template instantiation time
- at template definition time, compiler parses template and looks up any ***nondependent*** names
- result of nondependent name lookup must be ***identical*** in all instantiations of template (since, by definition, nondependent name does not depend on template parameter)
- at template instantiation time, compiler looks up any ***dependent*** names
- results of dependent name lookup can differ from one template instantiation to another (since, by definition, dependent name depends on template parameters)
- two-phase name lookup can interact with inheritance in ways that can sometimes lead to unexpected problems in code
- may need to add "`this->`" or employ **`using`** statement to make name dependent (when it would otherwise be nondependent)

Name Lookup Example (Incorrect Code)

```
#include <iostream>

template <class T>
struct Base {
    using Real = T;
    Base(Real x_ = Real()) : x(x_) {}
    void f() {std::cout << x << "\n";};
    Real x;
};

template <class T>
struct Derived : Base<T> {
    Derived(Real y_ = Real()) : y(y_) {}
      // ERROR: Real (which is nondependent and looked up at
      // template definition time) is assumed to be defined
      // outside class
    void g() {
        x = y;
          // ERROR: x assumed to be object outside class
        f();
          // ERROR: f assumed to be function outside class
    }
    Real y;
};

int main() {
    Derived<double> w(0.0);
    w.g();
}
```

Name Lookup Example (Correct Code)

```
#include <iostream>

template <class T>
struct Base {
    using Real = T;
    Base(Real x_ = Real()) : x(x_) {}
    void f() {std::cout << x << "\n";};
    Real x;
};

template <class T>
struct Derived : Base<T> {
    using Real = typename Base<T>::Real;
      // OK: Base<T>::Real dependent
    Derived(Real y_ = Real()) : y(y_) {}
    void g() {
        this->x = y; // OK: this->x dependent
        this->f(); // OK: this->f() dependent
    }
    Real y;
};

int main() {
    Derived<double> w(0.0);
    w.g();
}
```

Section 2.7.2

Virtual Functions and Run-Time Polymorphism

Run-Time Polymorphism

- **polymorphism** is characteristic of being able to assign different meaning to something in different contexts
- polymorphism that occurs at run time called **run-time polymorphism** (also known as **dynamic polymorphism**)
- in context of inheritance, key type of run-time polymorphism is polymorphic function call (also known as dynamic dispatch)
- when inheritance relationship exists between two classes, type of reference or pointer to object may not correspond to actual dynamic (i.e., run-time) type of object referenced by reference or pointer
- that is, reference or pointer to type `T` may, in fact, refer to object of type `D`, where `D` is either directly or indirectly derived from `T`
- when calling member function through pointer or reference, may want actual function invoked to be determined by ***dynamic*** type of object referenced by pointer or reference
- function call with this property said to be **polymorphic**

Virtual Functions

- in context of class hierarchies, polymorphic function calls achieved through use of virtual functions
- **virtual function** is member function with polymorphic behavior
- when call made to virtual function through reference or pointer, actual function invoked will be determined by ***dynamic*** type of referenced object
- to make member function virtual, add keyword **virtual** to function declaration
- example:

```
class Base {
public:
    virtual void func(); // virtual function
    // ...
};
```

Virtual Functions (Continued)

- once function made virtual, it will ***automatically*** be virtual in all derived classes, regardless of whether `virtual` keyword is used in derived classes
- therefore, not necessary to repeat `virtual` qualifier in derived classes (and perhaps preferable not to do so)
- virtual function must be defined in class where first declared unless pure virtual function (to be discussed shortly)
- derived class inherits definition of each virtual function from its base class, but may override each virtual function with new definition
- function in derived class with same name and same set of argument types as virtual function in base class overrides base class version of virtual function

Virtual Function Example

```
#include <iostream>
#include <string>

class Person {
public:
    Person(const std::string& family, const std::string& given) :
      family_(family), given_(given) {}
    virtual void print() const
      {std::cout << "person: " << family_ << ',' << given_ << '\n';}
protected:
    std::string family_; // family name
    std::string given_; // given name
};

class Student : public Person {
public:
    Student(const std::string& family, const std::string& given,
      const std::string& id) : Person(family, given), id_(id) {}
    void print() const {
        std::cout << "student: " << family_ << ',' << given_ << ',' << id_ << '\n';
    }
private:
    std::string id_; // student ID
};

void processPerson(const Person& p) {
    p.print(); // polymorphic function call
    // ...
}

int main() {
    Person p("Ritchie", "Dennis");
    Student s("Doe", "John", "12345678");
    processPerson(p); // invokes Person::print
    processPerson(s); // invokes Student::print
}
```

Override Control: The `override` Qualifier

- when looking at code for derived class, often not possible to determine if member function intended to override virtual function in base class (or one of its base classes)
- can sometimes lead to bugs where programmer expects member function to override virtual function when function not virtual
- **`override`** qualifier used to indicate that member function is expected to override virtual function in parent class; must come at end of function declaration
- example:

```
class Person {
public:
    virtual void print() const;
    // ...
};

class Employee : public Person {
public:
    void print() const override; // must be virtual
    // ...
};
```

Override Control: The `final` Qualifier

- sometimes, may want to prevent any further overriding of virtual function in any subsequent derived classes
- adding `final` qualifier to declaration of virtual function prevents function from being overridden in any subsequent derived classes
- preventing further overriding can sometimes allow for better optimization by compiler (e.g., via devirtualization)
- example:

```
class A {
public:
    virtual void doStuff();
    // ...
};

class B : public A {
public:
    void doStuff() final; // prevent further overriding
    // ...
};

class C : public B {
public:
    void doStuff(); // ERROR: cannot override
    // ...
};
```

final Qualifier Example

```
class Worker {
public:
    virtual void prepareEnvelope();
    // ...
};

class SpecialWorker : public Worker {
public:
    // prevent overriding function responsible for
    // overall envelope preparation process
    // but allow functions for individual steps in
    // process to be overridden
    void prepareEnvelope() final {
        stuffEnvelope(); // step 1
        lickEnvelope(); // step 2
        sealEnvelope(); // step 3
    }
    virtual void stuffEnvelope();
    virtual void lickEnvelope();
    virtual void sealEnvelope();
    // ...
};
```

Constructors, Destructors, and Virtual Functions

- except in very rare cases, destructors in class hierarchy need to be virtual
- otherwise, invoking destructor through base-class pointer/reference would only destroy base-class part of object, leaving remainder of derived-class object untouched
- normally, bad idea to call virtual function inside constructor or destructor
- dynamic type of object changes during construction and changes again during destruction
- final overrider of virtual function will change depending where in hierarchy virtual function call is made
- when constructor/destructor being executed, object is of exactly that type, never type derived from it
- although semantics of virtual function calls during construction and destruction well defined, easy to write code where actual overrider not what expected (and might even be pure virtual)

Problematic Code with Non-Virtual Destructor

```
class Base {
public:
    Base() {}
    ~Base() {} // non-virtual destructor
    // ...
};

class Derived : public Base {
public:
    Derived() : buffer_(new char[10'000]) {}
    ~Derived() {delete[] buffer_;}
    // ...
private:
    char* buffer_;
};

void process(Base* bp) {
    // ...
    delete bp; // always invokes only Base::~Base
}

int main() {
    process(new Base);
    process(new Derived); // leaks memory
}
```

Corrected Code with Virtual Destructor

```
class Base {
public:
    Base() {}
    virtual ~Base() {} // virtual destructor
    // ...
};

class Derived : public Base {
public:
    Derived() : buffer_(new char[10'000]) {}
    ~Derived() {delete[] buffer_;}
    // ...
private:
    char* buffer_;
};

void process(Base* bp) {
    // ...
    delete bp; // invokes destructor polymorphically
}

int main() {
    process(new Base);
    process(new Derived);
}
```

Preventing Creation of Derived Classes

- in some situations, may want to prevent deriving from class
- language provides means for accomplishing this
- in class/struct declaration, after name of class can add keyword **final** to prevent deriving from class
- example:

```
class Widget final { /* ... */ };
class Gadget : public Widget { /* ... */ };
  // ERROR: cannot derive from Widget
```

- might want to prevent deriving from class with destructor that is not virtual
- preventing derivation can sometimes also facilitate better compiler optimization (e.g., via devirtualization)
- might want to prevent derivation so that objects can be copied safely without fear of slicing

Covariant Return Type

- in some special cases, language allows relaxation of rule that type of overriding function f must be same as type of virtual function f overrides
- in particular, requirement that return type be same is relaxed
- return type of derived-class function is permitted to be type derived (directly or indirectly) from return type of base-class function
- this relaxation of return type more formally known as **covariant return type**
- ***case of pointer return type***: if original return type `B*`, return type of overriding function may be `D*`, provided `B` is public base of `D` (i.e., may return pointer to more derived type)
- ***case of reference return type***: if original return type `B&` (or `B&&`), return type of overriding function may be `D&` (or `D&&`), provided `B` is public base of `D` (i.e., may return reference to more derived type)
- covariant return type can sometimes be exploited in order to avoid need for type casts

Covariant Return Type Example: Cloning

```
class Base {
public:
    virtual Base* clone() const {
        return new Base(*this);
    }
    // ...
};

class Derived : public Base {
public:
    // use covariant return type
    Derived* clone() const override {
        return new Derived(*this);
    }
    // ...
};

int main() {
    Derived* d = new Derived;
    Derived* d2 = d->clone();
      // OK: return type is Derived*
    // without covariant return type, would need cast:
    // Derived* d2 = static_cast<Derived*>(d->clone());
}
```

Pure Virtual Functions

- sometimes desirable to require derived class to override virtual function
- **pure virtual function**: virtual function that must be overridden in every derived class
- to declare virtual function as pure, add "`= 0`" at end of declaration
- example:

```
class Widget {
public:
    virtual void doStuff() = 0; // pure virtual
    // ...
};
```

- pure virtual function can still be defined, although likely only useful in case of virtual destructor

- class with one or more pure virtual functions called **abstract class**
- cannot directly instantiate objects of abstract class (can only use them as base class objects)
- class that derives from abstract class need not override all of its pure virtual methods
- class that does not override all pure virtual methods of abstract base class will also be abstract
- most commonly, abstract classes have no state (i.e., data members) and used to provide interfaces, which can be inherited by other classes
- if class has no pure virtual functions and abstract class is desired, can make destructor pure virtual (but must provide definition of destructor since invoked by derived classes)

Abstract Class Example

```cpp
#include <cmath>

class Shape {
public:
    virtual bool isPolygon() const = 0;
    virtual float area() const = 0;
    virtual ~Shape() {};
};

class Rectangle : public Shape {
public:
    Rectangle(float w, float h) : w_(w), h_(h) {}
    bool isPolygon() const override {return true;}
    float area() const override {return w_ * h_;}
private:
    float w_; // width of rectangle
    float h_; // height of rectangle
};

class Circle : public Shape {
public:
    Circle(float r) : r_(r) {}
    float area() const override {return M_PI * r_ * r_;}
    bool isPolygon() const override {return false;}
private:
    float r_; // radius of circle
};
```

Pure Virtual Destructor Example

```
class Abstract {
public:
    virtual ~Abstract() = 0; // pure virtual destructor
    // ... (no other virtual functions)
};

inline Abstract::~Abstract()
  { /* possibly empty */ }
```

The `dynamic_cast` Operator

- often need to upcast and downcast (as well as cast sideways) in inheritance hierarchy
- **`dynamic_cast`** can be used to safely perform type conversions on pointers and references to classes
- syntax: **`dynamic_cast`**`<T>(`*expr*`)`
- types involved must be ***polymorphic*** (i.e., have at least one virtual function)
- inspects run-time information about types to determine whether cast can be safely performed
- if conversion is valid (i.e., *expr* can validly be cast to `T`), casts *expr* to type `T` and returns result
- if conversion is not valid, cast fails
- if *expr* is of pointer type, **`nullptr`** is returned upon failure
- if *expr* is of reference type, `std::bad_cast` exception is thrown upon failure (where exceptions are discussed later)

dynamic_cast Example

```
#include <cassert>

class Base {
public:
    virtual void doStuff() { /* ... */ };
    // ...
};

class Derived1 : public Base { /* ... */ };
class Derived2 : public Base { /* ... */ };

bool isDerived1(Base& b) {
    return dynamic_cast<Derived1*>(&b) != nullptr;
}

int main() {
    Base b;
    Derived1 d1;
    Derived2 d2;
    assert(isDerived1(b) == false);
    assert(isDerived1(d1) == true);
    assert(isDerived1(d2) == false);
}
```

Cost of Run-Time Polymorphism

- typically, run-time polymorphism does not come without run-time cost in terms of both time and memory
- in some contexts, cost can be significant
- typically, virtual functions implemented using virtual function table
- each polymorphic class has virtual function table containing pointers to all virtual functions for class
- each polymorphic class object has pointer to virtual function table
- memory cost to store virtual function table and pointer to table in each polymorphic object
- in most cases, impossible for compiler to inline virtual function calls since function to be called cannot be known until run time
- each virtual function call is made through pointer, which adds overhead

Curiously-Recurring Template Pattern (CRTP)

- when derived type known at compile time, may want behavior similar to virtual functions but without run-time cost (by performing binding at compile time instead of run time)
- can be achieved with technique known as **curiously-recurring template pattern (CRTP)**
- class `Derived` derives from class template instantiation using `Derived` itself as template argument
- example:

```
template <class Derived>
class Base {
    // ...
};

class Derived : public Base<Derived> {
    // ...
};
```

CRTP Example: Static Polymorphism

```
#include <iostream>

template <class Derived>
class Base {
public:
    void interface() {
        std::cout << "Base::interface called\n";
        static_cast<Derived*>(this)->implementation();
    }
    // ...
};

class Derived : public Base<Derived> {
public:
    void implementation() {
        std::cout << "Derived::implementation called\n";
    }
    // ...
};

int main() {
    Derived d;
    d.interface();
      // calls Base::interface which, in turn, calls
      // Derived::implementation
      // no virtual function call, however

}
```

CRTP Example: Static Polymorphism

```
class TreeNode {
public:
    enum Kind {RED, BLACK}; // kinds of nodes
    TreeNode *left(); // get left child node
    TreeNode *right(); // get right child node
    Kind kind(); // get kind of node
    // ...
};

template <class Derived>
class GenericVisitor {
public:
    void visit_preorder(TreeNode* node) {
        if (node) {
            process_node(node);
            visit_preorder(node->left());
            visit_preorder(node->right());
        }
    }
    void visit_inorder(TreeNode* node) { /* ... */ }
    void visit_postorder(TreeNode* node) { /* ... */ }
    void process_red_node(TreeNode* node) { /* ... */ };
    void process_black_node(TreeNode* node) { /* ... */ };
private:
    Derived& derived() {return *static_cast<Derived*>(this);}
    void process_node(TreeNode* node) {
        if (node->kind() == TreeNode::RED) {
            derived().process_red_node(node);
        } else {
            derived().process_black_node(node);
        }
    }
};

class SpecialVisitor : public GenericVisitor<SpecialVisitor> {
public:
    void process_red_node(TreeNode* node) { /* ... */ }
};

int main() {SpecialVisitor v;}
```

CRTP Example: Comparisons

```
#include <cassert>

template<class Derived>
struct Comparisons {
    friend bool operator==(const Comparisons<Derived>& x,
      const Comparisons<Derived>& y) {
        const Derived& xr = static_cast<const Derived&>(x);
        const Derived& yr = static_cast<const Derived&>(y);
        return !(xr < yr) && !(yr < xr);
    }
    // operator!= and others
};

class Widget : public Comparisons<Widget> {
public:
    Widget(bool b, int i) : b_(b), i_(i) {}
    friend bool operator<(const Widget& x, const Widget& y)
      {return x.i_ < y.i_;}
private:
    bool b_;
    int i_;
};

int main() {
    Widget w1(true, 1);
    Widget w2(false, 1);
    assert(w1 == w2);
}
```

CRTP Example: Object Counting

```
#include <iostream>
#include <cstdlib>

template <class T>
class Counter {
public:
    Counter() {++count_;}
    Counter(const Counter&) {++count_;}
    ~Counter() {--count_;}
    static std::size_t howMany() {return count_;}
private:
    static std::size_t count_;
};

template <class T>
std::size_t Counter<T>::count_ = 0;

// inherit from Counter to count objects
class Widget: private Counter<Widget> {
public:
    using Counter<Widget>::howMany;
    // ...
};

int main() {
    Widget w1; int c1 = Widget::howMany();
    Widget w2, w3; int c2 = Widget::howMany();
    std::cout << c1 << ' ' << c2 << '\n';
}
```

Section 2.7.3

Multiple Inheritance and Virtual Inheritance

Multiple Inheritance

- language allows derived class to inherit from more than one base class
- **multiple inheritance (MI)**: deriving from more than one base class
- although multiple inheritance not best solution for most problems, does have some compelling use cases
- one compelling use case is for inheriting interfaces by deriving from abstract base classes with no data members
- when misused, multiple inheritance can lead to very convoluted code
- in multiple inheritance contexts, ambiguities in naming can arise
- for example, if class `Derived` inherits from classes `Base1` and `Base2`, each of which have member called `x`, name `x` can be ambiguous in some contexts
- scope resolution operator can be used to resolve ambiguous names

Ambiguity Resolution Example

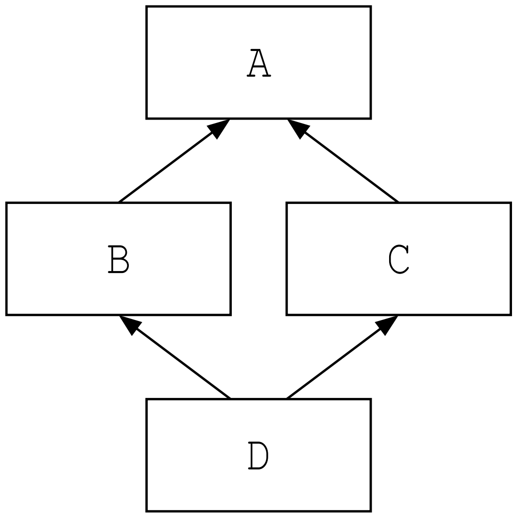
```
class Base1 {
public:
    void func();
    // ...
};

class Base2 {
    void func();
    // ...
};

class Derived : public Base1, public Base2 {
public:
    // ...
};

int main() {
    Derived d;
    d.func(); // ERROR: ambiguous function call
    d.Base1::func(); // OK: invokes Base1::func
    d.Base2::func(); // OK: invokes Base2::func
}
```

Multiple Inheritance Example

```
class Input_stream {
public:
    virtual ~Input_stream() {}
    virtual int read_char() = 0;
    virtual int read(char* buffer, int size) = 0;
    virtual bool is_input_ready() const = 0;
    // ...(all pure virtual, no data)
};

class Output_stream {
public:
    virtual ~Output_stream() {}
    virtual int write_char(char c) = 0;
    virtual int write(char* buffer, int size) = 0;
    virtual int flush_output() = 0;
    // ... (all pure virtual, no data)
};

class Input_output_stream : public Input_stream,
  public Output_stream {
    // ...
};
```

Dreaded Diamond Inheritance Pattern

- use of multiple inheritance can lead to so called dreaded diamond scenario
- **dreaded diamond** inheritance pattern has following form:



- class `D` will have ***two*** subobjects of class `A`, since class `D` (indirectly) inherits twice from class `A`
- situation like one above probably undesirable and often sign of poor design

Dreaded Diamond Example

```cpp
class Base {
public:
    // ...
protected:
    int data_;
};

class D1 : public Base { /* ... */ };

class D2 : public Base { /* ... */ };

class Join : public D1, public D2 {
public:
    void method() {
        data_ = 1; // ERROR: ambiguous
        D1::data_ = 1; // OK: unambiguous
    }
};

int main() {
    Join* j = new Join();
    Base* b;
    b = j; // ERROR: ambiguous
    b = static_cast<D1*>(j); // OK: unambiguous
}
```

Virtual Inheritance

- when using multiple inheritance, may want to ensure that only one instance of base-class object can appear in derived-class object
- **virtual base class**: base class that is only ever included once in derived class, even if derived from multiple times
- **virtual inheritance**: when derived class inherits from base class that is virtual
- virtual inheritance can be used to avoid situations like dreaded diamond pattern
- order of construction: virtual base classes constructed first in depth-first left-to-right traversal of graph of base classes, where left-to-right refers to order of appearance of base class names

Avoiding Dreaded Diamond With Virtual Inheritance

```
class Base {
public:
    // ...
protected:
    int data_;
};

class D1 : public virtual Base { /* ... */ };

class D2 : public virtual Base { /* ... */ };

class Join : public D1, public D2 {
public:
    void method() {
        data_ = 1; // OK: unambiguous
    }
};

int main() {
    Join* j = new Join();
    Base* b = j; // OK: unambiguous
}
```

Section 2.7.4

References

References I

1. N. Meyers. The empty base C++ optimization.
   Dr. Dobb's Journal, Aug. 1997.
   Available online at `http://www.cantrip.org/emptyopt.html`.
2. J. O. Coplien. Curiously recurring template patterns.
   C++ Report, pages 24–27, Feb. 1995.
3. S. Meyers. Counting objects in C++.
   C++ User's Journal, Apr. 1998.
   Available online at `http://www.drdobbs.com/cpp/counting-objects-in-c/184403484`.
4. A. Nasonov. Better encapsulation for the curiously recurring template pattern.
   Overload, 70:11–13, Dec. 2005.

Section 2.8

C++ Standard Library

- C++ standard library provides huge amount of functionality (orders of magnitude more than C standard library)
- uses `std` namespace (to avoid naming conflicts)
- well worth effort to familiarize yourself with all functionality in library in order to avoid writing code unnecessarily

C++ Standard Library (Continued)

- functionality can be grouped into following sublibraries:
  1. language support library (e.g., exceptions, memory management)
  2. diagnostics library (e.g., assertions, exceptions, error codes)
  3. general utilities library (e.g., functors, date/time)
  4. strings library (e.g., C++ and C-style strings)
  5. localization library (e.g., date/time formatting and parsing, character classification)
  6. containers library (e.g., sequence containers and associative containers)
  7. iterators library (e.g., stream iterators)
  8. algorithms library (e.g., searching, sorting, merging, set operations, heap operations, minimum/maximum)
  9. numerics library (e.g., complex numbers, math functions)
  10. input/output (I/O) library (e.g., streams)
  11. regular expressions library (e.g., regular expression matching)
  12. atomic operations library (e.g., atomic types, fences)
  13. thread support library (e.g., threads, mutexes, condition variables, futures)

Commonly-Used Header Files

Language-Support Library

Header File	Description
cstdlib	run-time support, similar to stdlib.h from C (e.g., exit)
limits	properties of fundamental types (e.g., numeric_limits)
exception	exception handling support (e.g., set_terminate, current_exception)
initializer_list	initializer_list class template

Diagnostics Library

Header File	Description
cassert	assertions (e.g., assert)
stdexcept	predefined exception types (e.g., invalid_argument, domain_error, out_of_range)

General-Utilities Library

Header File	Description
`utility`	basic function and class templates (e.g., `swap`, `move`, `pair`)
`memory`	memory management (e.g., `unique_ptr`, `shared_ptr`, `addressof`)
`functional`	functors (e.g., `less`, `greater`)
`type_traits`	type traits (e.g., `is_integral`, `is_reference`)
`chrono`	clocks (e.g., `system_clock`, `steady_clock`, `high_resolution_clock`)

Strings Library

Header File	Description
`string`	C++ string classes (e.g., `string`)
`cstring`	C-style strings, similar to `string.h` from C (e.g., `strlen`)
`cctype`	character classification, similar to `ctype.h` from C (e.g., `isdigit`, `isalpha`)

Commonly-Used Header Files (Continued 2)

Containers, Iterators, and Algorithms Libraries

Header File	Description
`array`	`array` class
`vector`	`vector` class
`deque`	`deque` class
`list`	`list` class
`set`	set classes (i.e., `set`, `multiset`)
`map`	map classes (i.e., `map`, `multimap`)
`unordered_set`	unordered set classes (i.e., `unordered_set`, `unordered_multiset`)
`unordered_map`	unordered map classes (i.e., `unordered_map`, `unordered_multimap`)
`iterator`	iterators (e.g., `reverse_iterator`, `back_inserter`)
`algorithm`	algorithms (e.g., `min`, `max`, `sort`)
`forward_list`	`forward_list` class

Commonly-Used Header Files (Continued 3)

Numerics Library

Header File	Description
`cmath`	C math library, similar to `math.h` from C (e.g., `sin`, `cos`)
`complex`	complex numbers (e.g., `complex`)
`numeric`	generalized numeric operations (e.g., `gcd`, `lcm`, `inner_product`)
`random`	random number generation (e.g., `uniform_int_distribution`, `uniform_real_distribution`, `normal_distribution`)

Commonly-Used Header Files (Continued 4)

I/O Library

Header File	Description
iostream	iostream objects (e.g., cin, cout, cerr)
istream	input streams (e.g., istream)
ostream	output streams (e.g., ostream)
ios	base classes and other declarations for streams (e.g., ios_base, hex, fixed)
fstream	file streams (e.g., fstream)
sstream	string streams (e.g., stringstream)
iomanip	manipulators (e.g., setw, setprecision)

Regular-Expressions Library

Header File	Description
regexp	regular expressions (e.g., basic_regex)

Commonly-Used Header Files (Continued 5)

Atomic-Operations and Thread-Support Libraries

Header File	Description
`atomic`	atomics (e.g., `atomic`)
`thread`	threads (e.g., `thread`)
`mutex`	mutexes (e.g., `mutex`, `recursive_mutex`, `timed_mutex`)
`condition_variable`	condition variables (e.g., `condition_variable`)
`future`	futures (e.g., `future`, `shared_future`, `promise`)

Section 2.8.1

Containers, Iterators, and Algorithms

Standard Template Library (STL)

- large part of C++ standard library is collection of class/function templates known as standard template library (STL)
- STL comprised of three basic building blocks:
  1. containers
  2. iterators
  3. algorithms
- containers store elements for processing (e.g., vector)
- iterators allow access to elements for processing (which are often, but not necessarily, in containers)
- algorithms perform actual processing (e.g., search, sort)

Containers

- **container**: class that represents collection/sequence of elements
- usually container classes are template classes
- **sequence container**: collection in which every element has certain position that depends on time and place of insertion
- examples of sequence containers include:
  - `array` (fixed-size array)
  - `vector` (dynamic-size array)
  - `list` (doubly-linked list)
- **ordered/unordered associative container**: collection in which position of element in depends on its value or associated key and some predefined sorting/hashing criterion
- examples of associative containers include:
  - `set` (collection of unique keys, sorted by key)
  - `map` (collection of key-value pairs, sorted by key, keys are unique)

Sequence Containers and Container Adapters

Sequence Containers

Name	Description
`array`	fixed-size array
`vector`	dynamic-size array
`deque`	double-ended queue
`forward_list`	singly-linked list
`list`	doubly-linked list

Container Adapters

Name	Description
`stack`	stack
`queue`	FIFO queue
`priority_queue`	priority queue

Ordered Associative Containers

Name	Description
set	collection of unique keys, sorted by key
map	collection of key-value pairs, sorted by key, keys are unique
multiset	collection of keys, sorted by key, duplicate keys allowed
multimap	collection of key-value pairs, sorted by key, duplicate keys allowed

Unordered Associative Containers

Name	Description
unordered_set	collection of unique keys, hashed by key
unordered_map	collection of key-value pairs, hashed by key, keys are unique
unordered_multiset	collection of keys, hashed by key, duplicate keys allowed)
unordered_multimap	collection of key-value pairs, hashed by key, duplicate keys allowed

Typical Sequence Container Member Functions

- some member functions typically provided by sequence container classes listed below (where `T` denotes name of container class)

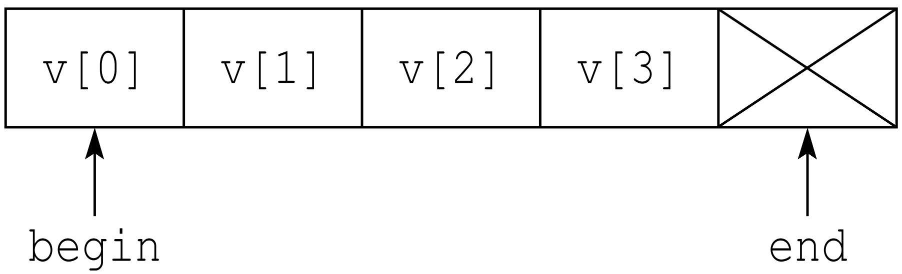
Function	Description
`T()`	create empty container (default constructor)
`T(const T&)`	copy container (copy constructor)
`T(T&&)`	move container (move constructor)
`~T`	destroy container (including its elements)
`empty`	test if container empty
`size`	get number of elements in container
`push_back`	insert element at end of container
`clear`	remove all elements from container
`operator=`	assign all elements of one container to other
`operator[]`	access element in container

Container Example

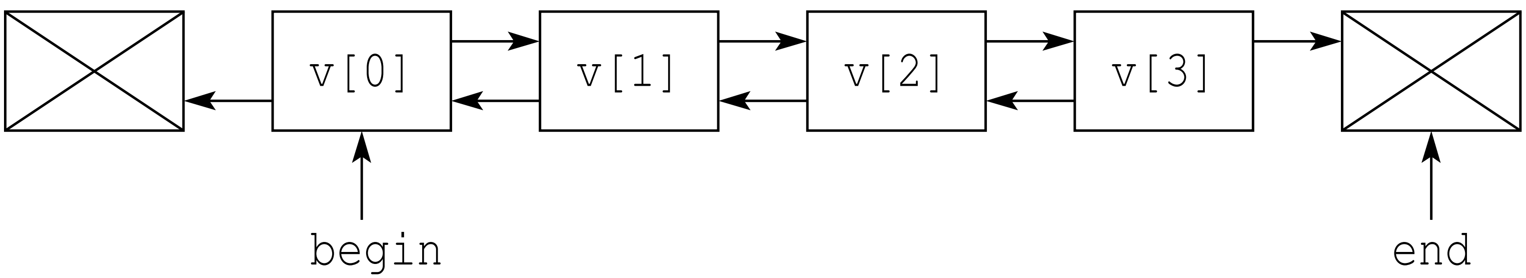
```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> values;

    // append elements with values 0 to 9
    for (int i = 0; i < 10; ++i) {
        values.push_back(i);
    }

    // print each element followed by space
    for (int i = 0; i < values.size(); ++i) {
        std::cout << values[i] << ' ';
    }
    std::cout << '\n';
}

/* This program produces the following output:
0 1 2 3 4 5 6 7 8 9
*/
```

Motivation for Iterators

- different containers organize elements (of container) differently in memory
- want uniform manner in which to access elements in any arbitrary container
- organization of elements in array/vector container:

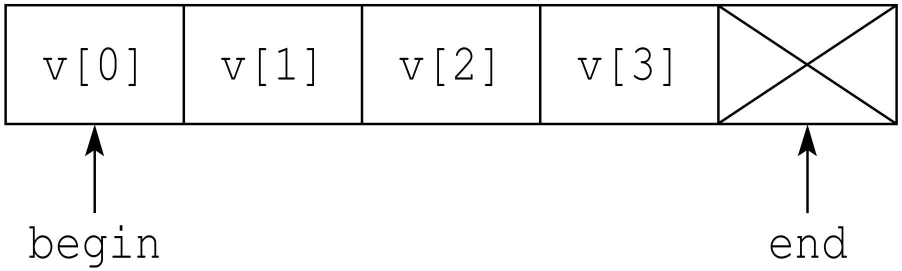


- organization of elements in doubly-linked list container:



Motivation for Iterators (Continued)

- consider array/vector container with **`int`** elements:



- suppose we want to set all elements in container to zero
- we could use code like:

```
// int* begin; int* end;
for (int* iter = begin; iter != end; ++iter)
    *iter = 0;
```

- could we make similar-looking code work for more complicated organization like doubly-linked list?
- yes, create user-defined type that provides all pointer operations used above (e.g., dereference, increment, comparison, assignment)
- this leads to notion of iterator

Iterators

- **iterator**: object that allows iteration over collection of elements, where elements are often (but not necessarily) in container
- iterators support many of same operations as pointers
- in some cases, iterator may actually be pointer; more frequently, iterator is user-defined type
- five different categories of iterators: 1) input, 2) output, 3) forward, 4) bidirectional, and 5) random access
- iterator has particular level of functionality, depending on category
- one of three possibilities of access order:
  1. forward (i.e., one direction only)
  2. forward and backward
  3. any order (i.e., random access)
- one of three possibilities in terms of read/write access:
  1. can only read referenced element (once or multiple times)
  2. can only write referenced element (once or multiple times)
  3. can read and write referenced element (once or multiple times)
- const and mutable (i.e., non-const) variants (i.e., read-only or read/write access, respectively)

Abilities of Iterator Categories

Category	Ability	Providers
Input	Reads (once only) forward	`istream` (`istream_iterator`)
Output	Writes (once only) forward	`ostream` (`ostream_iterator`), `inserter_iterator`
Forward	Reads and writes forward	`forward_list`, `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`
Bidirectional	Reads and writes forward and backward	`list`, `set`, `multiset`, `map`, `multimap`
Random access	Reads and writes with random access	(built-in) array, `array`, `vector`, `deque`, `string`

Input Iterators

Expression	Effect
`T(a)`	copies iterator (copy constructor)
`*a` `a->m`	dereference as rvalue (i.e., read only); cannot dereference at old position
`++a`	steps forward (returns new position)
`a++`	steps forward
`a == b`	test for equality
`a != b`	test for inequality

- not assignable (i.e., no assignment operator)

Output Iterators

Expression	Effect
`T(a)`	copies iterator (copy constructor)
`*a` `a->m`	dereference as lvalue (i.e., write only); can only be dereferenced once; cannot dereference at old position
`++a`	steps forward (returns new position)
`a++`	steps forward (returns old position)

- not assignable (i.e., no assignment operator)
- no comparison operators (i.e., **`operator`**`==`, **`operator`**`!=`)

Forward Iterators

Expression	Effect
`T()`	default constructor
`T(a)`	copy constructor
`a = b`	assignment
`*a` `a->m`	dereference
`++a`	steps forward (returns new position)
`a++`	steps forward (returns old position)
`a == b`	test for equality
`a != b`	test for inequality

- must ensure that valid to dereference iterator before doing so

Bidirectional Iterators

- bidirectional iterators are forward iterators that provide additional functionality of being able to iterate backward over elements
- bidirectional iterators have all functionality of forward iterators as well as those listed in table below

Expression	Effect
`--a`	steps backward (returns new position)
`a--`	steps backward (returns old position)

Random-Access Iterators

- random access iterators provide all functionality of bidirectional iterators as well as providing random access to elements
- random access iterators provide all functionality of bidirectional iterators as well as those listed in table below

Expression	Effect
`a[n]`	dereference element at index `n` (where `n` can be negative)
`a += n`	steps `n` elements forward (where `n` can be negative)
`a -= n`	steps `n` elements backward (where `n` can be negative)
`a + n`	iterator for `n`th next element
`n + a`	iterator for `n`th next element
`a - n`	iterator for `n`th previous element
`a - b`	distance from `a` to `b`
`a < b`	test if `a` before `b`
`a > b`	test if `a` after `b`
`a <= b`	test if `a` not after `b`
`a >= b`	test if `a` not before `b`

- pointers (built into language) are examples of random-access iterators

Iterator Example

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> values(10);

    std::cout << "number of elements: " <<
      (values.end() - values.begin()) << '\n';

    // initialize elements of vector to 0, 1, 2, ...
    for (std::vector<int>::iterator i = values.begin();
      i != values.end(); ++i) {
        *i = i - values.begin();
    }

    // print elements of vector
    for (std::vector<int>::const_iterator i =
      values.cbegin(); i != values.cend(); ++i) {
        std::cout << ' ' << *i;
    }
    std::cout << '\n';
}
```

Iterator Gotchas

- do not dereference iterator unless it is known to validly reference some object
- some operations on container can ***invalidate*** some or all iterators referencing elements in container
- critically important to know ***which operations invalidate*** iterators in order to avoid using iterator that has been invalidated
- incrementing iterator ***past end*** of container or decrementing iterator ***before beginning*** of container results in undefined behavior
- input and output iterators can only be dereferenced ***once*** at each position

Algorithms

- **algorithm**: sequence of computations applied to some generic type
- algorithms use iterators to access elements involved in computation
- often pair of iterators used to specify ***range*** of elements on which to perform some computation
- what follows only provides brief summary of algorithms
- for more details on algorithms, see:
  - `http://www.cplusplus.com/reference/algorithm`
  - `http://en.cppreference.com/w/cpp/algorithm`

Functions

Non-Modifying Sequence Operations

Name	Description
`all_of`	test if condition true for all elements in range
`any_of`	test if condition true for any element in range
`none_of`	test if condition true for no elements in range
`for_each`	apply function to range
`for_each_n`	apply function to first n elements in sequence
`find`	find values in range
`find_if`	find element in range
`find_if_not`	find element in range (negated)
`find_end`	find last subsequence in range
`find_first_of`	find element from set in range
`adjacent_find`	find equal adjacent elements in range
`count`	count appearances of value in range
`count_if`	count number of elements in range satisfying condition
`mismatch`	get first position where two ranges differ
`equal`	test whether elements in two ranges differ
`search`	find subsequence in range
`search_n`	find succession of equal values in range

Modifying Sequence Operations

Name	Description
copy	copy range of elements
copy_if	copy certain elements of range
copy_n	copy n elements
copy_backward	copy range of elements backwards
move	move range of elements
move_backward	move range of elements backwards
swap	exchange values of two objects (in utility header)
swap_ranges	exchange values of two ranges
iter_swap	exchange values of objects referenced by two iterators
transform	apply function to range
replace	replace value in range
replace_if	replace values in range
replace_copy	copy range replacing value
replace_copy_if	copy range replacing value
sample	selects n random elements from sequence

Modifying Sequence Operations (Continued)

Name	Description
`fill`	fill range with value
`fill_n`	fill sequence with value
`generate`	generate values for range with function
`generate_n`	generate values for sequence with function
`remove`	remove value from range (by shifting elements)
`remove_if`	remove elements from range (by shifting elements)
`remove_copy`	copy range removing value
`remove_copy_if`	copy range removing values
`unique`	remove consecutive duplicates in range
`unique_copy`	copy range removing duplicates
`reverse`	reverse range
`reverse_copy`	copy range reversed
`rotate`	rotate elements in range
`rotate_copy`	copies and rotates elements in range
`shuffle`	randomly permute elements in range

Partition Operations

Name	Description
is_partitioned	test if range is partitioned by predicate
partition	partition range in two
partition_copy	copies range partition in two
stable_partition	partition range in two (stable ordering)
partition_point	get partition point

Sorting

Name	Description
is_sorted	test if range is sorted
is_sorted_until	find first unsorted element in range
sort	sort elements in range
stable_sort	sort elements in range, preserving order of equivalents
partial_sort	partially sort elements in range
partial_sort_copy	copy and partially sort range
nth_element	sort element in range

Binary Search (operating on sorted ranges)

Name	Description
`lower_bound`	get iterator to lower bound
`upper_bound`	get iterator to upper bound
`equal_range`	get subrange of equal elements
`binary_search`	test if value exists in sorted range

Set Operations (on sorted ranges)

Name	Description
`merge`	merge sorted ranges
`inplace_merge`	merge consecutive sorted ranges
`includes`	test whether sorted range includes another sorted range
`set_union`	union of two sorted ranges
`set_intersection`	intersection of two sorted ranges
`set_difference`	difference of two sorted ranges
`set_symmetric_difference`	symmetric difference of two sorted ranges

Functions (Continued 5)

Heap Operations

Name	Description
`is_heap`	test if range is heap
`is_heap_until`	first first element not in heap order
`push_heap`	push element into heap range
`pop_heap`	pop element from heap range
`make_heap`	make heap from range
`sort_heap`	sort elements of heap

Functions (Continued 6)

Minimum/Maximum

Name	Description
`min`	get minimum of given values
`max`	get maximum of given values
`minmax`	get minimum and maximum of given values
`min_element`	get smallest element in range
`max_element`	get largest element in range
`minmax_element`	get smallest and largest elements in range
`clamp`	clamp value between pair of boundary values
`lexicographic_compare`	lexicographic less-than comparison
`is_permutation`	test if range permutation of another
`next_permutation`	transform range to next permutation
`prev_permutation`	transform range to previous permutation

Functions (Continued 7)

Numeric Operations

Name	Description
`iota`	fill range with successive values
`accumulate`	accumulate values in range
`adjacent_difference`	compute adjacent difference of range
`inner_product`	compute inner product of range
`partial_sum`	compute partial sums of range
`reduce`	similar to `accumulate` except out of order
`exclusive_scan`	similar to `partial_sum`, excludes ith input element from ith sum
`inclusive_scan`	similar to `partial_sum`, includes ith input element in ith sum
`transform_reduce`	applies functor, then reduces out of order
`transform_exclusive_scan`	applies functor then, calculates exclusive scan
`transform_inclusive_scan`	applies functor, then calculates inclusive scan

Operations on Uninitialized Memory

Name	Description
`uninitialized_copy`	copy range of objects to uninitialized area of memory
`uninitialized_copy_n`	copy number of objects to uninitialized area of memory
`uninitialized_fill`	copy object to uninitialized area of memory, defined by range
`uninitialized_fill_n`	copy object to uninitialized area of memory, defined by start and count
`uninitialized_move`	move range of objects to uninitialized area of memory
`uninitialized_move_n`	move number of objects to uninitialized area of memory

Functions (Continued 9)

Operations on Uninitialized Memory (Continued)

Name	Description
`uninitialized_default_construct`	construct objects by default initialization in uninitialized area of memory defined by range
`uninitialized_default_construct_n`	construct objects by default initialization in uninitialized area of memory defined by start and count
`uninitialized_value_construct`	construct objects by value initialization in uninitialized area of memory defined by range
`uninitialized_value_construct_n`	construct objects by value initialization in uninitialized area of memory defined by start and count
`destroy_at`	destroy object at given address
`destroy`	destroy range of objects
`destroy_n`	destroy number of objects in range

Other Numeric Algorithms

Name	Description
`gcd`	compute greatest common divisor of two integers
`lcm`	compute least common multiple of two integers

Algorithms Example

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>

int main() {
    std::vector<int> values;

    int x;
    while (std::cin >> x) {values.push_back(x);}

    std::cout << "zero count: " << std::count(
      values.begin(), values.end(), 0) << '\n';

    std::default_random_engine engine;
    std::shuffle(values.begin(), values.end(), engine);
    std::cout << "random order:";
    for (auto i : values) {std::cout << ' ' << i;}
    std::cout << '\n';

    std::sort(values.begin(), values.end());
    std::cout << "sorted order:";
    for (auto i : values) {std::cout << ' ' << i;}
    std::cout << '\n';
}
```

Prelude to Functor Example

- consider `std::transform` function template:

```
template <class InputIterator, class OutputIterator,
  class UnaryOperator>
  OutputIterator transform(InputIterator first,
  InputIterator last, OutputIterator result,
  UnaryOperator op);
```

- applies `op` to each element in range [`first`,`last`) and stores each returned value in range beginning at `result`
- `std::transform` might be written as:

```
template <class InputIterator, class OutputIterator,
  class UnaryOperator>
  OutputIterator transform(InputIterator first,
  InputIterator last, OutputIterator result,
  UnaryOperator op) {
    while (first != last) {
        *result = op(*first);
        ++first;
        ++result;
    }
    return result;
}
```

- `op` is entity that can be used with function call syntax (i.e., function or functor)

Functor Example

```
#include <iostream>
#include <vector>
#include <algorithm>

struct MultiplyBy { // Functor class
    MultiplyBy(double factor) : factor_(factor) {}
    double operator()(double x) const
      {return factor_ * x;}
private:
    double factor_; // multiplicative factor
};

int main() {
    MultiplyBy mb(2.0);
    std::vector v{1.0, 2.0, 3.0};
    // v contains 1 2 3
    std::transform(v.begin(), v.end(), v.begin(), mb);
    // v contains 2 4 6
    for (auto i : v) {std::cout << i << '\n';}
}
```

Section 2.8.2

The `std::array` Class Template

The `std::array` Class Template

- one-dimensional array type, where size of array is fixed at compile time
- `array` declared as:

```
template <class T, std::size_t N>
class array;
```

- `T`: type of elements in array
- `N`: number of elements in array
- what follows only intended to provide overview of `array`
- for additional details on `array`, see:
  - `http://en.cppreference.com/w/cpp/container/array`
  - `http://www.cplusplus.com/reference/stl/array`

Member Type	Description
value_type	T (i.e., element type)
size_type	type used for measuring size (i.e., std::size_t)
difference_type	type used to measure distance (i.e., std::ptrdiff_t)
reference	value_type&
const_reference	**const** value_type&
pointer	value_type*
const_pointer	**const** value_type*
iterator	***random-access*** iterator type
const_iterator	const ***random-access*** iterator type
reverse_iterator	reverse iterator type (i.e., reverse_iterator<iterator>)
const_reverse_iterator	const reverse iterator type (i.e., reverse_iterator<const_iterator>)

Member Functions

Construction, Destruction, and Assignment

Member Name	Description
constructor	initializes array
destructor	destroys each element of array
operator=	overwrites every element of array with corresponding element of another array

Iterators

Member Name	Description
`begin`	return iterator to beginning
`end`	return iterator to end
`cbegin`	return const iterator to beginning
`cend`	return const iterator to end
`rbegin`	return reverse iterator to beginning
`rend`	return reverse iterator to end
`crbegin`	return const reverse iterator to beginning
`crend`	return const reverse iterator to end

Member Functions (Continued 1)

Capacity

Member Name	Description
`empty`	test if array is empty
`size`	return size
`max_size`	return maximum size

Element Access

Member Name	Description
`operator``[]`	access element (no bounds checking)
`at`	access element (with bounds checking)
`front`	access first element
`back`	access last element
`data`	return pointer to start of element data

Modifiers

Member Name	Description
`fill`	fill container with specified value
`swap`	swap contents of two arrays

array Example

```
#include <array>
#include <iostream>
#include <algorithm>
#include <experimental/iterator>

int main() {
    std::array<int, 3> a1{3, 1, 2};
    std::array<int, 3> a2;
    a2.fill(42);
    for (auto i : a2) {
        std::cout << i << '\n';
    }
    a2 = a1;
    std::sort(a1.begin(), a1.end());
    std::copy(a1.begin(), a1.end(),
      std::experimental::make_ostream_joiner(std::cout, ", "));
    std::cout << '\n';
    for(auto i = a2.begin(); i != a2.end(); ++i) {
        std::cout << *i;
        if (i != a2.end() - 1) {std::cout << ", ";}
    }
    std::cout << '\n';
}
```

array Example

```cpp
#include <array>
#include <iostream>
#include <algorithm>

int main() {
    // Fixed-size array with 4 elements.
    std::array<int, 4> a{2, 4, 3, 1};

    // Print elements of array.
    for (auto i = a.cbegin(); i != a.cend(); ++i) {
        std::cout << ' ' << *i;
    }
    std::cout << '\n';

    // Sort elements of array.
    std::sort(a.begin(), a.end());

    // Print elements of array.
    for (auto i = a.cbegin(); i != a.cend(); ++i) {
        std::cout << ' ' << *i;
    }
    std::cout << '\n';
}
```

Section 2.8.3

The `std::vector` Class Template

The `std::vector` Class Template

- dynamically-sized one-dimensional array type, where type of array elements and storage allocator specified by template parameters
- `vector` declared as:

```
template <class T, class Allocator = allocator<T>>
class vector;
```

- `T`: type of elements in vector
- `Allocator`: type of object used to handle storage allocation (unless custom storage allocator needed, use default `allocator<T>`)
- what follows only intended to provide overview of `vector`
- for additional details on `vector`, see:
  - `http://www.cplusplus.com/reference/stl/vector`
  - `http://en.cppreference.com/w/cpp/container/vector`

Member Types

Member Type	Description
value_type	T (i.e., element type)
allocator_type	Allocator (i.e., allocator)
size_type	type used for measuring size (typically unsigned integral type)
difference_type	type used to measure distance (typically signed integral type)
reference	value_type&
const_reference	**const** value_type&
pointer	allocator_traits<Allocator>::pointer
const_pointer	allocator_traits<Allocator>::const_pointer
iterator	***random-access*** iterator type
const_iterator	const ***random-access*** iterator type
reverse_iterator	reverse iterator type (reverse_iterator<iterator>)
const_reverse_iterator	const reverse iterator type (reverse_iterator<const_iterator>)

Member Functions

Construction, Destruction, and Assignment

Member Name	Description
constructor	construct vector (overloaded)
destructor	destroy vector
`operator=`	assign vector

Iterators

Member Name	Description
`begin`	return iterator to beginning
`end`	return iterator to end
`cbegin`	return const iterator to beginning
`cend`	return const iterator to end
`rbegin`	return reverse iterator to beginning
`rend`	return reverse iterator to end
`crbegin`	return const reverse iterator to beginning
`crend`	return const reverse iterator to end

Member Functions (Continued 1)

Capacity

Member Name	Description
`empty`	test if vector is empty
`size`	return size
`max_size`	return maximum size
`capacity`	return allocated storage capacity
`reserve`	request change in capacity
`shrink_to_fit`	shrink to fit

Element Access

Member Name	Description
`operator``[]`	access element (no bounds checking)
`at`	access element (with bounds checking)
`front`	access first element
`back`	access last element
`data`	return pointer to start of element data

Member Functions (Continued 2)

Modifiers

Member Name	Description
`clear`	clear content
`assign`	assign vector content
`insert`	insert elements
`emplace`	insert element, constructing in place
`push_back`	add element at end
`emplace_back`	insert element at end, constructing in place
`erase`	erase elements
`pop_back`	delete last element
`resize`	change size
`swap`	swap content of two vectors

Allocator

Member Name	Description
`get_allocator`	get allocator used by vector

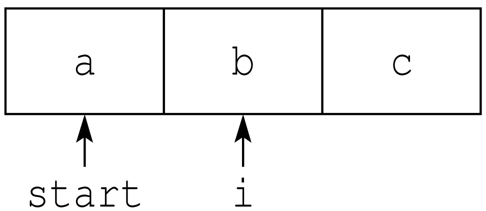
Invalidation of References, Iterators, and Pointers

- **capacity**: total number of elements that vector could hold without requiring reallocation of memory
- any operation that causes reallocation of memory used to hold elements of vector invalidates ***all*** iterators, references, and pointers referring to elements in vector
- any operation that changes capacity of vector causes reallocation of memory
- any operation that adds or deletes elements can invalidate references, iterators, and pointers
- operations that can potentially invalidate references, iterators, and pointers to elements in vector include:

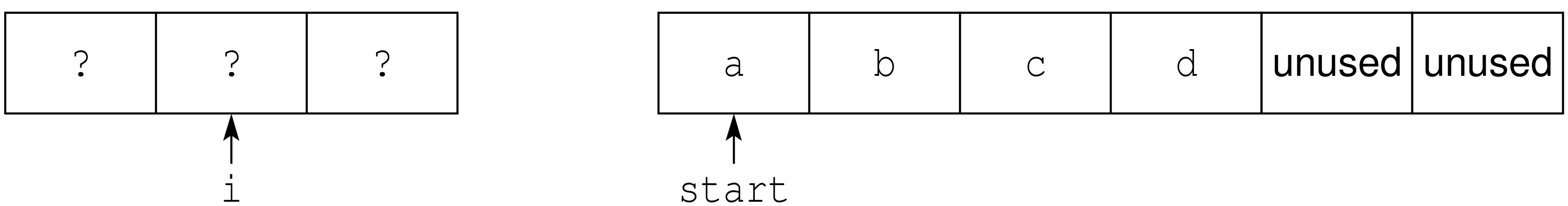
  `insert`, `erase`, `push_back`, `pop_back`, `emplace`, `emplace_back`, `resize`, `reserve`, **`operator`**`=`, `assign`, `clear`, `shrink_to_fit`, `swap` (past-the-end iterator only)

Iterator Invalidation Example

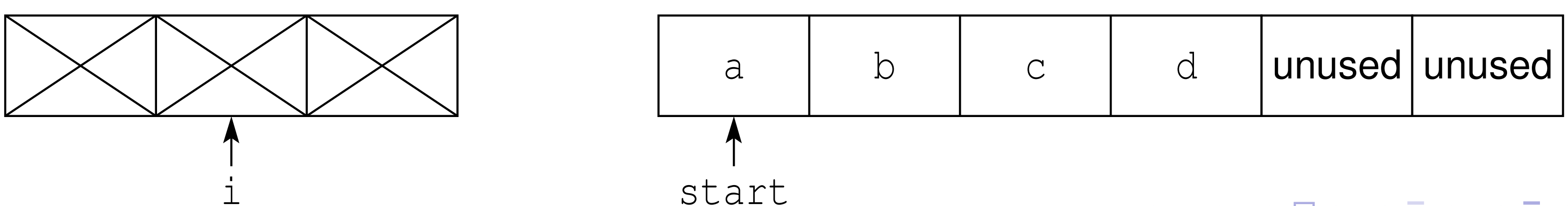
- `start` denotes pointer to first element in array holding elements of `vector`
- `i` is iterator for `vector` (e.g., `vector<T>::const_iterator` or `vector<T>::iterator`)
- initial vector has three elements and capacity of three



- `push_back(d)` invoked
- new larger array is allocated (say, twice size of original), contents of old array moved to new one, and then new element added



- elements in old array destroyed and memory for old array deallocated; iterator `i` is now ***invalid***:



vector Example: Constructors

```
std::vector<double> v0;
    // empty vector

std::vector<double> v1(10);
    // vector with 10 elements, default constructed
    // (which for double means uninitialized)

std::vector<double> v2(10, 5.0);
    // vector with 10 elements, each initialized to 5.0

std::vector<int> v3{1, 2, 3};
    // vector with 3 elements: 1, 2, 3
    // std::initializer_list (note brace brackets)
```

```
#include <iostream>
#include <vector>

int main() {
    std::vector v{0, 1, 2, 3};
    for (auto& i : v) {++i;}
    for (auto i : v) {
        std::cout << ' ' << i;
    }
    std::cout << '\n';
    for (auto i = v.begin(); i != v.end(); ++i) {
        --(*i);
    }
    for (auto i = v.cbegin(); i != v.cend(); ++i) {
        std::cout << ' ' << *i;
    }
    std::cout << '\n';
    for (auto i = v.crbegin(); i != v.crend(); ++i) {
        std::cout << ' ' << *i;
    }
    std::cout << '\n';
}
```

- program output:

```
 1 2 3 4
 0 1 2 3
 3 2 1 0
```

vector Example

```
#include <iostream>
#include <vector>

int main() {
    std::vector<double> values;
    // ...

    // Erase all elements and then read elements from
    // standard input.
    values.clear();
    double x;
    while (std::cin >> x) {
        values.push_back(x);
    }
    std::cout << "number of values read: " <<
      values.size() << '\n';

    // Loop over all elements and print the number of
    // negative elements found.
    int count = 0;
    for (auto i = values.cbegin(); i != values.cend(); ++i) {
        if (*i < 0.0) {
            ++count;
        }
    }
    std::cout << "number of negative values: " << count <<
      '\n';
}
```

```
#include <iostream>
#include <vector>

int main() {
    std::vector<std::vector<int>> v{{1, 2, 3}, {4, 5, 6}};
    v.emplace_back(10, 0);
    // The above use of emplace_back is more efficient than:
    // v.push_back(std::vector<int>(10, 0));
    for (const auto& i : v) {
        for (const auto& j : i) {
            std::cout << ' ' << j;
        }
        std::cout << '\n';
    }
}
```

- program output:

```
 1 2 3
 4 5 6
 0 0 0 0 0 0 0 0 0 0
```

Section 2.8.4

The `std::basic_string` Class Template

The std::basic_string Class Template

- character string type, parameterized on character type, character traits, and storage allocator
- basic_string declared as:

```
template <class CharT,
  class Traits = char_traits<CharT>,
  class Allocator = allocator<CharT>>
  class basic_string;
```

- CharT: type of characters in string
- Traits: class that describes certain properties of CharT (normally, use default)
- Allocator: type of object used to handle storage allocation (unless custom storage allocator needed, use default)
- string is simply abbreviation for basic_string<char>
- what follows is only intended to provide overview of basic_string template class (and string class)
- for more details on basic_string, see:
  - http://www.cplusplus.com/reference/string/basic_string
  - http://en.cppreference.com/w/cpp/string/basic_string

Member Types

Member Type	Description
traits_type	Traits (i.e., character traits)
value_type	Traits::char_type (i.e., character type)
allocator_type	Allocator
size_type	allocator_traits<Allocator>::size_type
difference_type	allocator_traits<Allocator>:: difference_type
reference	value_type&
const_reference	**const** value_type&
pointer	allocator_traits<Allocator>::pointer
const_pointer	allocator_traits<Allocator>:: const_pointer
iterator	***random-access*** iterator type
const_iterator	const ***random-access*** iterator type
reverse_iterator	reverse iterator type (reverse_iterator<iterator>)
const_reverse_iterator	const reverse iterator type (reverse_iterator<const_iterator>)

Construction, Destruction, and Assignment

Member Name	Description
constructor	construct
destructor	destroy
operator=	assign

Iterators

Member Name	Description
`begin`	return iterator to beginning
`end`	return iterator to end
`cbegin`	return const iterator to beginning
`cend`	return const iterator to end
`rbegin`	return reverse iterator to reverse beginning
`rend`	return reverse iterator to reverse end
`crbegin`	return const reverse iterator to reverse beginning
`crend`	return const reverse iterator to reverse end

Capacity

Member Name	Description
`empty`	test if string empty
`size`	get length of string
`length`	same as `size`
`max_size`	get maximum size of string
`capacity`	get size of allocated storage
`reserve`	change capacity
`shrink_to_fit`	shrink to fit

Element Access

Member Name	Description
`operator``[]`	access character in string (no bounds checking)
`at`	access character in string (with bounds checking)
`front`	access first character in string
`back`	access last character in string

Member Functions (Continued 2)

Operations

Member Name	Description
`clear`	clear string
`assign`	assign content to string
`insert`	insert into string
`push_back`	append character to string
`operator``+=`	append to string
`append`	append to string
`erase`	erase characters from string
`pop_back`	delete last character from string
`replace`	replace part of string
`resize`	resize string
`swap`	swap contents with another string

Operations (Continued)

Member Name	Description
`c_str`	get nonmodifiable C-string equivalent
`data`	obtain pointer to first character of string
`copy`	copy sequence of characters from string
`substr`	generate substring
`compare`	compare strings

Search

Member Name	Description
`find`	find first occurrence of content in string
`rfind`	find last occurrence of content in string
`find_first_of`	find first occurrence of characters in string
`find_first_not_of`	find first absence of characters in string
`find_last_of`	find last occurrence of characters in string
`find_last_not_of`	find last absence of characters in string

Member Functions (Continued 4)

Allocator

Member Name	Description
`get_allocator`	get allocator

Numeric Conversions

Name	Description
`stoi`	convert string to **`int`**
`stol`	convert string to **`long`**
`stoll`	convert string to **`long long`**
`stoul`	convert string to **`unsigned long`**
`stoull`	convert string to **`unsigned long long`**
`stof`	convert string to **`float`**
`stod`	convert string to **`double`**
`stold`	convert string to **`long double`**
`to_string`	convert integral or floating-point value to `string`
`to_wstring`	convert integral or floating-point value to `wstring`

```cpp
#include <iostream>
#include <string>

int main() {
    std::string s;
    if (!(std::cin >> s)) {
        s.clear();
    }
    std::cout << "string: " << s << '\n';
    std::cout << "length: " << s.size() << '\n';
    std::string b;
    for (auto i = s.crbegin(); i != s.crend(); ++i) {
        b.push_back(*i);
    }
    std::cout << "backwards: " << b << '\n';

    std::string msg = "Hello";
    msg += ", World!"; // append ", World!"
    std::cout << msg << '\n';

    const char* cstr = s.c_str();
    std::cout << "C-style string: " << cstr << '\n';
}
```

Numeric/String Conversion Example

```
#include <iostream>
#include <string>

int main() {
    double x = 42.24;
    // Convert double to string.
    std::string s = std::to_string(x);
    std::cout << s << '\n';

    s = "3.14";
    // Convert string to double.
    x = std::stod(s);
    std::cout << x << '\n';

}
```

Section 2.8.5

Other Container Classes

The `std::pair` Class Template

- collection of two heterogeneous objects
- `pair` declared as:

```
template <class T1, class T2>
struct pair;
```

- `T1`: type of first element in `pair`
- `T2`: type of second element in `pair`
- first and second elements accessible via data members `first` and `second`, respectively
- elements of `pair` can also be accessed with `std::get` function template
- `pair` can be created with `std::make_pair` function template
- `pair` is effectively equivalent to `std::tuple` (to be discussed shortly) with two elements

pair Example

```
#include <tuple>
#include <cassert>

int main() {
    std::pair p(true, 42);
    assert(p.first && p.second == 42);
    assert(p.first == std::get<0>(p) &&
      p.second == std::get<1>(p));
    std::pair q = std::make_pair(true, 42);
    assert(p == q);
    p = std::make_pair(false, 0);
    assert(p != q);
    p.swap(q);
    auto [b, i] = p;
    assert(b == true && i == 42);
    assert(std::get<bool>(p) && std::get<0>(p));
    assert(std::get<int>(p) == 42 &&
      std::get<1>(p) == 42);
}
```

The `std::tuple` Class Template

- fixed-size collection of heterogenous values
- `tuple` is generalization of `std::pair`
- `tuple` declared as:

```
template <class... Ts>
class tuple;
```

- `Ts`: types of elements that tuple holds (which may be empty)
- elements of `tuple` can be accessed with `std::get` function template
- `tuple` can be created with `std::make_tuple` function template

```cpp
#include <tuple>
#include <cassert>

int main() {
    std::tuple t(true, 42, 'Z');
    auto u = std::make_tuple(true, 42, 'Z');
    assert(t == u);
    assert(std::get<bool>(t) && std::get<0>(t));
    assert(std::get<char>(t) == 'Z' && std::get<2>(t) == 'Z');
    std::get<0>(t) = false;
    assert(t != u);
    std::tuple v(false, 0, '0');
    u = std::make_tuple(true, 1, '1');
    v.swap(u);
    assert(std::get<0>(v));
}
```

The `std::optional` Class Template

- simple container that manages optional value (i.e., value that may or may not be present)
- declaration:

```
template <class T> class optional;
```

- `T` is type of optional value
- `T` cannot be reference type
- at any given point in time, object either contains value or does not
- object can be given value by initialization or assignment
- common use case is return value of function that can fail
- object can be created via factory function `std::make_optional`
- `std::bad_optional_access` exception indicates checked access to `optional` object that does not contain value
- optional value is required to be stored directly in `optional` object itself

Construction, Destructon, and Assignment

Name	Description
constructor	constructs optional object
destructor	destroys optional object (and contained value)
operator=	assigns contents

Observers

Name	Description
operator->	accesses contained value
operator*	accesses contained value
operator bool	tests if object contains value
has_value	tests if object contains value
value	returns contained value
value_or	returns contained value if available and specified default value otherwise

Modifiers

Name	Description
`swap`	exchange contents
`reset`	clear any contained value
`emplace`	constructs contained value in place

optional Example

```
#include <optional>
#include <string>
#include <exception>
#include <cassert>
#include <iostream>

int main() {
    auto s = std::make_optional<std::string>("Hello!");
    assert(s && s.has_value());
    assert(s.value() == "Hello!");
    auto t = std::make_optional<std::string>("Goodbye!");
    s.swap(t);
    assert(*s == "Goodbye!" && *t == "Hello!");
    s.reset();
    assert(!s && !s.has_value());
    std::cout << s.value_or("Goodbye!") << '\n';
    try {std::cout << s.value() << '\n';}
    catch (const std::bad_optional_access&) {
        std::cout << "caught exception\n";
    }
    s.emplace("Salut!");
    std::cout << s.value() << '\n';

}
```

```cpp
#include <optional>
#include <string>
#include <fstream>
#include <iostream>

std::optional<std::string> read_file(const char* file_name) {
    std::ifstream in(file_name);
    std::optional<std::string> result;
    result.emplace(std::istreambuf_iterator<char>(in),
      std::istreambuf_iterator<char>());
    if (in.fail() && !in.eof()) {
        result.reset();
    }
    return result;
}

int main(int argc, char** argv) {
    if (argc <= 1) {return 1;}
    auto s = read_file(argv[1]);
    if (!s) {
        std::cerr << "unable to read file\n";
        return 1;
    }
    std::cout << *s;
}
```

The `std::variant` Class Template

- simple container that corresponds to type-safe union
- can hold single value of one of set of allowable types
- declaration:

```
template <class... Ts> class variant;
```

- `Ts` parameter pack containing all allowable types of value that can be stored in object
- container cannot hold references, arrays, or void
- can hold same type more than once and can hold differently cv-qualified versions of same type
- default initialized variant holds value of first alternative, which is default constructed
- `std::monostate` can be used as placeholder for empty type
- invalid accesses to value of `variant` object result in `std::bad_variant_access` exception being thrown

`variant` Member Functions

Construction, Destructon, and Assignment

Name	Description
constructor	constructs variant object
destructor	destroys variant object (and contained value)
operator=	assigns variant

Observers

Name	Description
`index`	returns zero-based index of alternative held by variant
`valueless_by_exception`	tests if variant in invalid state

Modifiers

Name	Description
`emplace`	constructs value in variant in place
`swap`	swaps value with another variant

variant Example

```
#include <variant>
#include <cassert>
#include <iostream>

int main() {
    std::variant<int, double> x;
    std::variant<int, double> y;
    x = 2;
    assert(std::get<int>(x) == std::get<0>(x));
    assert(!x.valueless_by_exception());
    y = 0.5;
    assert(std::get<double>(y) == std::get<1>(y));
    std::cout << std::get<int>(x) << '\n';
    std::cout << std::get<double>(y) << '\n';
    try {std::cout << std::get<double>(x) << '\n';}
    catch (const std::bad_variant_access&) {
        std::cout << "bad variant access\n";
    }
}
```

The `std::any` Class

- type-safe container for single value of any type
- container may also hold no value
- declaration:

```
class any;
```

- at any given time, object may or may not hold value
- non-member function `any_cast` provides type-safe access to contained object
- `std::bad_any_cast` exception thrown by value-returning forms of `any_cast` upon type mismatch

Construction, Destructon, and Assignment

Name	Description
constructor	constructs `any` object
destructor	destroys `any` object
`operator=`	assigns `any` object

Observers

Name	Description
`has_value`	tests if object holds value
`type`	returns typeid of contained value

Modifiers

Name	Description
`emplace`	change contained object by constructing new value in place
`reset`	clear any contained object
`swap`	swaps contents of two `any` objects

any Example

```
#include <any>
#include <cassert>
#include <string>
#include <iostream>

int main() {
    std::any x{std::string("Hello")};
    assert(x.has_value() && x.type() == typeid(std::string));
    std::any y;
    assert(!y.has_value());
    x.swap(y);
    assert(!x.has_value() && y.has_value());
    x = y;
    std::cout << std::any_cast<std::string>(x) << '\n';
    y.reset();
    assert(!y.has_value());
    try {std::any_cast<int>(x);}
    catch (const std::bad_any_cast&) {
        std::cout << "any_cast failed\n";
    }
}
```

Section 2.8.6

Time Measurement

Time Measurement

- time measurement capabilities provided by part of general utilities library (of standard library)
- header file `chrono`
- identifiers in namespace `std::chrono`
- **time point**: specific point in time (measured relative to epoch)
- **duration**: time interval
- **clock**: measures time in terms of time points
- several clocks provided for measuring time
- what follows only intended to provide overview of chrono part of library
- for additional information on chrono part of library, see:
  - `http://www.cplusplus.com/reference/chrono`
  - `http://en.cppreference.com/w/cpp/chrono`

std::chrono Types

Time Points and Intervals

Name	Description
`duration`	time interval
`time_point`	point in time

Clocks

Name	Description
`system_clock`	system clock (which may be adjusted)
`steady_clock`	monotonic clock that ticks at constant rate
`high_resolution_clock`	clock with shortest tick period available

std::chrono Example: Measuring Elapsed Time

```
#include <iostream>
#include <chrono>
#include <cmath>

double get_result() {
    double sum = 0.0;
    for (long i = 0L; i < 1000000L; ++i) {
        sum += std::sin(i) * std::cos(i);
    }
    return sum;
}

int main() {
    // Get the start time.
    auto start_time =
      std::chrono::high_resolution_clock::now();
    // Do some computation.
    double result = get_result();
    // Get the end time.
    auto end_time = std::chrono::high_resolution_clock::now();
    // Compute elapsed time in seconds.
    double elapsed_time = std::chrono::duration<double>(
      end_time - start_time).count();
    // Print result and elapsed time.
    std::cout << "result " << result << '\n';
    std::cout << "time (in seconds) " << elapsed_time << '\n';
}
```

std::chrono Example: Determining Clock Resolution

```
#include <iostream>
#include <chrono>

// Get the granularity of a clock in seconds.
template <class C>
double granularity() {
    return std::chrono::duration<double>(
      typename C::duration(1)).count();
}

int main() {
    std::cout << "system clock:\n" << "period "
      << granularity<std::chrono::system_clock>() << '\n'
      << "steady "
      << std::chrono::system_clock::is_steady << '\n';
    std::cout << "high resolution clock:\n" << "period "
      << granularity<std::chrono::high_resolution_clock>()
      << '\n' << "steady "
      << std::chrono::high_resolution_clock::is_steady << '\n';
    std::cout << "steady clock:\n" << "period "
      << granularity<std::chrono::steady_clock>() << '\n'
      << "steady "
      << std::chrono::steady_clock::is_steady << '\n';
}
```

Section 2.8.7

Miscellany

The std::basic_string_view Class Template

- std::basic_string_view class template represents constant contiguous sequence of **char**-like objects (i.e., read-only view of string)
- basic_string_view declared as:

```
template <class CharT,
  class Traits = char_traits<CharT>>
  class basic_string_view;
```

- CharT: type of characters in string
- Traits: class that describes certain properties of CharT (normally, use default)
- string_view is simply abbreviation for basic_string_view<**char**>
- for more details on basic_string_view, see:
  - http://en.cppreference.com/w/cpp/string/basic_string_view

std::basic_string_view Example

```
#include <string_view>
#include <string>
#include <iostream>
#include <cassert>

void output(std::string_view s) {
    std::cout << s << '\n';
}

int main() {
    std::string_view hello("hello");
    assert(!hello.empty());
    std::string_view he = hello.substr(0, 2);
    assert(he.size() == 2);
    assert(he[0] == 'h' && he[1] == 'e');
    assert(hello.find("ell") == 1);
    assert(hello.rfind("l") == 3);

    std::string goodbye("goodbye");
    std::string_view bye(goodbye);
    bye.remove_prefix(4);
    std::cout << bye << '\n';
    std::string_view good(goodbye);
    good.remove_suffix(3);
    std::cout << good << '\n';
    assert(goodbye.substr(4, 3) == bye);
    output(bye);
}
```

Section 2.9

Miscellany

Name Lookup

- Since C++ name lookup rules are quite complicated, we only present a simplified (and therefore not fully correct) description of them here.
- ***Qualified lookup.*** If the name `A` is preceded by the scope-resolution operator, as in `::A` or `X::A`, then use qualified name lookup.
  - In the first case, look in the global namespace for `A`. In the second case, look up `X`, and then look inside it for `A`.
  - If `X` is a class and `A` is not a direct member, look in all of the direct bases of `X` (and then each of their bases). If `A` is found in more than one base, fail.
- ***Argument-dependent lookup.*** Otherwise, if the name is used as a function call, such as `A(X)`, use argument-dependent lookup.
  - Look for `A` in the namespace in which the type of `X` was declared, in the friends of `X`, and if `X` is a template instantiation, similarly for each of the arguments involved.
- ***Unqualified lookup.*** Start with unqualified lookup if argument-dependent lookup does not apply.
  - Start at the current scope and work outwards until the name is found.

Argument-Dependent Lookup (ADL)

- argument-dependent lookup (ADL) applies to lookup of unqualified function name
- during ADL, other namespaces not considered during normal lookup may be searched
- in particular, namespace that declares each function argument type is included in search
- ADL also commonly referred to as Koenig lookup

ADL Example

```
#include <iostream>

namespace N {
    class C { /* ... */ };
    void f(C x) {std::cout << "N::f\n";}
    void g(int x) {std::cout << "N::g\n";}
    void h(C x) {std::cout << "N::h\n";}
}

struct D {
    struct E {};
    static void p(E e) {std::cout << "D::p\n";};
};

void h(N::C x) {std::cout << "::h\n";}

int main() {
    N::C x;
    f(x);    // OK: calls N::f via ADL
    N::f(x); // OK: calls N::f
    g(42);   // ERROR: g not found
    N::g(42); // OK: calls N::g
    h(x);    // ERROR: ambiguous function call due to ADL
    ::h(x);  // OK: calls ::h
    N::h(x); // OK: calls N::h
    D::E e;
    p(e);    // ERROR: ADL only considers namespaces
    D::p(e); // OK: calls D::p
}
```

ADL Example

```cpp
#include <iostream>

namespace N {
    struct W {};
    void f(W x) {std::cout << "N::f\n";}
}

struct C {
    void f(N::W x) {std::cout << "C::f\n";}
    void g() {
        N::W x;
        f(x); // calls C::f (not N::f)
    }
};

int main() {
    C c;
    c.g();
}
```

ADL Example

```
#include <iostream>
#include <string>

using namespace std::string_literals;

namespace N {
    struct C {};
    void f(int) {std::cout << "N::f\n";}
    void g(C x) {std::cout << "N::g\n";}
    void h(const std::string& x) {std::cout << "N::h\n";}
    namespace M {
        void f(int x) {std::cout << "N::M::f\n";}
          // hides N::f
        void g(int x) {std::cout << "N::M::g\n";}
          // hides N::g
        void h() {std::cout << "N::M::h\n";} // hides N::h
        void u() {
            N::C c;
            f(42); // calls N::M::f (ADL looks nowhere)
            g(c); // calls N::g via ADL (ADL looks in N)
            h("hi"s); // ERROR: lookup finds N::M::h
              // (ADL does not look in N)
        }
    }
}

int main() {N::M::u();}
```

Swapping Values and ADL

- Consider two objects `x` and `y` of class type `T` whose values are to be swapped.
- If the class `T` provides its own `swap` function for reasons of efficiency, one would normally want to use it.
- In the absence of such a function, one would normally want to fall back on the use of `std::swap`.
- The above behavior can be achieved using code like the following:

```
using std::swap;
swap(x, y);
```

- If the type `T` provides its own `swap` function, the name lookup on `swap` will yield this function through ADL.
- Otherwise, the name lookup will find `std::swap`.
- Thus, code like the above will result in a more efficient `swap` function being used if available, with the `std::swap` function used as a fallback.

Part 3

More C++

Section 3.1

Memory Management

- object said to have **dynamic storage duration** if its lifetime is independent of scope in which object created (i.e., lifetime of object does not end until explicitly ended)
- often need to use objects (or arrays of objects) with dynamic storage duration
- in what follows, we consider how such objects are managed
- new expressions used to create objects or arrays of objects with dynamic storage duration
- delete expressions used to destroy such objects
- in order to handle any necessary memory allocation and deallocation, new and delete expressions in turn use:
  - single-object operator new (i.e., `operator new`)
  - array operator new (i.e., `operator new[]`)
  - single-object operator delete (i.e., `operator delete`)
  - array operator delete (i.e., `operator delete[]`)

Potential Problems Arising in Memory Management

- **leaked object**: object created but not destroyed when no longer needed
- leaked objects are problematic because can cause program to waste memory or exhaust all available memory
- **premature deletion** (a.k.a. **dangling references**): object is deleted when one or more references to object still exist
- premature deletion is problematic because, if object accessed after deletion, results of doing so will be unpredictable (e.g., read garbage value or overwrite other variables in program)
- **double deletion**: object is deleted twice, invoking destructor twice
- double deletion is problematic invoking destructor on nonexistent object is unpredictable and furthermore double deletion can often corrupt data structures used by memory allocator

Section 3.1.1

New and Delete Expressions

Alignment

- `std::max_align_t` is type having maximum alignment supported by implementation in all contexts
- **extended alignment** is alignment exceeding `alignof(std::max_align_t)`
- in some contexts, may be possible to use extended alignment
- every alignment value must be nonnegative power of two

New Expressions

- new expression used to create object or array of objects with dynamic storage duration
- new expression has one of following forms:

  scope_prefix **`new`** *placement_args* *type* *initializer*

  scope_prefix **`new`** *placement_args* `(`*type*`)` *initializer*
- *scope_prefix*: optional unary `::` operator which controls lookup of allocation function
- *placement_args*: optional list of additional arguments for memory allocation function enclosed in parentheses
- *type*: type of object to be created which may be array type
- *initializer*: optional list of arguments used to initialize newly created object or array (e.g., constructor arguments for class type object)
- new expression where optional placement arguments provided referred to **placement new** expression
- new expression returns pointer to object created for non-array type or pointer to first element in array for array type

New Expressions (Continued)

- examples of new expressions:

```
int* ip1 = new int;
int* ip2 = new int(42);
std::vector<int>* vp1 = new std::vector<int>(100, 42);
int* aip1 = new int[256];
std::string* asp = new std::string[64];
int* aip2 = new (std::nothrow) int[10000];
alignas(std::string) char buf[sizeof(std::string)];
std::string* sp = new (static_cast<void*>(&buf))
  std::string("Hello");
```

- evaluating new expression performs following:
  1. invokes allocation function to obtain address in memory where new object or array of objects should be placed
  2. invokes constructors to create objects in storage obtained from allocation function
  3. if constructor fails (i.e., throws), any successfully constructed objects are destroyed (in reverse order from construction) and deallocation function called to free memory in which object or array was being constructed

New Expressions and Allocation

- for non-array types, allocation function is single-object operator new (i.e., **`operator new`**) (discussed later), which can be overloaded
- for array types, allocation function is array operator new (i.e., **`operator new`**`[]`) (discussed later), which can be overloaded
- allocation function need not allocate memory (since placement arguments of new expression may be used to specify address at which to place new object)
- if allocation function has non-throwing exception specification, new expression returns null pointer upon failure otherwise `std::bad_alloc` exception is thrown
- for array type, requested size of memory may exceed size of actual array data (i.e., overhead to store size of array for use at deletion time)
- if new expression begins with unary `::` operator, allocation function's name looked up in global scope; otherwise, looked up in class scope if applicable and then global scope

Allocation Function Overload Resolution

- overload resolution for (single-object and array) operator new performed using argument list consisting of:
  1. amount of space requested, which has type `std::size_t`
  2. if type has extended alignment, type's alignment, which has type `std::align_val_t`
  3. if placement new expression, placement arguments
- if no matching function found, alignment removed from argument list and overload resolution performed again
- expression "`new T`" results in one of following calls:

```
operator new(sizeof(T))
operator new(sizeof(T), std::align_val_t(alignof(T)))
```

- expression "`new(42, f) T`" results in one of following calls:

```
operator new(sizeof(T), 42, f)
operator new(sizeof(T), std::align_val_t(alignof(T),
  42, f)
```

Allocation Function Overload Resolution (Continued)

- expression "`new T[7]`" results in one of following calls:

  ```
  operator new[](sizeof(T) * 7 + x)
  operator new[](sizeof(T) * 7 + x, std::align_val_t(
    alignof(T)))
  ```

  where `x` is nonnegative implementation-dependent constant representing array allocation overhead

- expression "`new (42, f) T[7]`" results in one of following calls:

  ```
  operator new[](sizeof(T) * 7 + x, 42, f)
  operator new[](sizeof(T) * 7 + x, std::align_val_t(
    alignof(T)), 42, f)
  ```

  where `x` is nonnegative implementation-dependent constant representing array allocation overhead

New Expressions and Deallocation

- for non-array types, deallocation function is single-object operator delete (i.e., **`operator delete`**) (to be discussed shortly)
- for array types, deallocation function is array operator delete (i.e., **`operator delete`**`[]`) (to be discussed shortly)
- if new expression begins with unary `::` operator, deallocation function's name looked up in global scope; otherwise, looked up in class scope and then global scope

Delete Expressions

- delete expression used to destroy object or array of objects created by new expression and deallocate associated memory
- delete expression has one of two forms:

  scope_prefix **`delete`** *expr*
  scope_prefix **`delete`**`[]` *expr*

- *scope_prefix*: optional unary `::` operator which controls lookup of deallocation function
- *expr*: pointer to object or array previously created by new expression or null pointer
- first form (sometimes called **single-object delete expression**) is used to dispose of single object obtained from new expression
- second form (sometimes called **array delete expression**) is used to dispose of array of objects obtained from new expression
- delete expression has void type
- if *expr* is null pointer, evaluation of delete expression effectively does nothing (i.e., no destructors called and no deallocation function called)

Delete Expressions (Continued 1)

- single object created by new expression must be deleted with single-object delete expression
- array created by new expression must be deleted with array delete expression
- examples of delete expressions:

```
int *ip = new int(42);
delete ip;
std::vector<int> *vp = new std::vector<int>;
delete vp;
std::string* asp = new std::string[1024];
delete[] asp;
```

- examples of incorrect delete expressions:

```
std::string* sp = new std::string;
delete[] sp;
  // ERROR: must use single-object delete expression
std::string* asp = new std::string[1024];
delete asp;
  // ERROR: must use array delete expression
```

Delete Expressions (Continued 2)

- evaluating single-object delete expression performs following:
  1. if object of class type, invokes destructor
  2. invokes deallocation function for object
- evaluating array delete expression performs following:
  1. if array element of class type (with non-trivial destructor):
     1. determines size of array (which is typically stored just before or just after array element data)
     2. invokes destructor for each array element (in reverse order from construction, namely, backwards order)
  2. invokes deallocation function for array
- if delete expression prefixed by unary `::` operator, deallocation function's name looked up only at global scope; otherwise at class scope if applicable and then global scope

Operator New (i.e., `operator new`)

- operator new (i.e., **`operator new`**) is operator used to determine address at which to place new object to be created
- most frequently invoked indirectly via new expression, but can be called directly
- operator new may or may not allocate memory
- operator can be overloaded as global function or (implicitly static) member function
- operator has return type **`void`**`*` and returns address at which new object to be created should be placed
- first parameter to operator always of type `std::size_t` and specifies number of bytes of storage needed for new object to be created
- several overloads of global operator new provided by language and standard library
- `std::nothrow` is dummy variable of type **`const`** `std::nothrow_t` that can be used for overload disambiguation

Operator New Overloads

- `void* operator new(std::size_t size);`
  - allocates `size` bytes of storage that is suitably aligned for any object of this size not having extended alignment
  - throws `std::bad_alloc` exception upon failure
- `void* operator new(std::size_t size, std::align_val_t align);`
  - allocates `size` bytes of storage with guaranteed alignment of `align`
  - throws `std::bad_alloc` exception upon failure
- `void* operator new(std::size_t size, const std::nothrow_t& tag);`
  - allocates `size` bytes of storage suitably aligned for any object of this size not having extended alignment
  - returns ***null pointer*** upon failure
- `void* operator new(std::size_t size, std::align_val_t align, const std::nothrow_t& tag);`
  - allocates `size` bytes of storage with guaranteed alignment of `align`
  - returns ***null pointer*** upon failure

Operator New Overloads (Continued)

- `void* operator new(std::size_t size, void* ptr) noexcept;`
  - ***non-allocating***
  - simply returns `ptr`, assuming `ptr` points to storage of at least `size` bytes with appropriate alignment
  - cannot fail
  - not useful to invoke directly, since function effectively does nothing
  - intended to be invoked by non-allocating ***placement new*** expressions

Operator New Examples

```
#include <new>
#include <cassert>

void func_1() {
    // allocating operator new
    int* ip = static_cast<int*>(::operator new(sizeof(int)));
    assert(ip);
    // ... (dellocate memory)
}

void func_2() {
    // allocating and non-throwing operator new
    int* ip = static_cast<int*>(::operator new(sizeof(int),
      std::nothrow));
    // ip may be null
    // ... (deallocate memory)
}

void func_3() {
    int i;
    // non-allocating operator new
    int* ip = static_cast<int*>(::operator new(sizeof(int),
      static_cast<void*>(&i)));
    assert(ip == &i);
}
```

Operator Array New (i.e., `operator new[]`)

- operator array new (i.e., **`operator new`**`[]`) is operator used to determine address at which to place array of objects to be created
- operator array new may or may not allocate memory
- operator array new can be overloaded as global function or (implicitly static) member function
- operator has return type **`void`**`*` and returns address at which new array of objects to be created should be placed
- first parameter to operator always of type `std::size_t` and specifies number of bytes of storage needed for new array of objects to be created
- several overloads of global operator array new provided by language and standard library
- `std::nothrow` is dummy variable of type **`const`** `std::nothrow_t` that can be used for overload disambiguation

Operator Array New Overloads

- `void* operator new[](std::size_t size);`
  - allocates `size` bytes of storage that is suitably aligned for any object of this size not having extended alignment
  - throws `std::bad_alloc` exception upon failure
- `void* operator new[](std::size_t size, std::align_val_t align);`
  - allocates `size` bytes of storage with alignment of `align`
  - throws `std::bad_alloc` exception upon failure
- `void* operator new[]( std::size_t size, const std::nothrow_t& tag);`
  - allocates `size` bytes of storage suitably aligned for any object of this size not having extended alignment
  - returns ***null pointer*** upon failure
- `void* operator new[]( std::size_t size, std::align_val_t align, const std::nothrow_t& tag);`
  - allocates `size` bytes of storage with guaranteed alignment of `align`
  - returns ***null pointer*** upon failure

Operator Array New Overloads (Continued)

- **`void`**`* `**`operator new`**`[](std::size_t size, `**`void`**`* ptr)` **`noexcept`**`;`
  - ***non-allocating***
  - simply returns `ptr`, assuming `ptr` points to storage of at least `size` bytes with appropriate alignment
  - cannot fail
  - not useful to invoke directly, since function effectively does nothing
  - intended to be invoked by non-allocating (array) ***placement new*** expressions

Operator Array New Examples

```
#include <new>
#include <cassert>

void func_1() {
    // allocating operator array new
    int* ip = static_cast<int*>(::operator new[](1000 * sizeof(int)));
    assert(ip);
    // ... (deallocate)
}

void func_2() {
    int* ip = static_cast<int*>(::operator new[](1000 * sizeof(int),
      std::nothrow));
    // ip may be null
    // ... (deallocate)
}

void func_3() {
    static int a[1000];
    int* ip = static_cast<int*>(::operator new[](1000 * sizeof(int),
      static_cast<void*>(a)));
    assert(ip == a);
}
```

Operator Delete (i.e., `operator delete`)

- operator delete (i.e., **`operator delete`**) is operator used to deallocate memory for object allocated with operator new
- can be invoked through delete expression or through new expression if constructor throws exception
- always has return type of **`void`**
- first parameter always pointer of type **`void`**$*$
- standard library deallocation functions do nothing if pointer is null
- can be overloaded as global function or (implicitly static) member function

Operator Delete Overloads

- `void operator delete(void* ptr) noexcept;`
  `void operator delete( void* ptr, std::size_t size) noexcept;`
  `void operator delete ( void* ptr, std::align_val_t align) noexcept;`
  `void operator delete ( void* ptr, std::size_t size, std::align_val_t align) noexcept;`
  - deallocates storage associated with object at address `ptr`, which was allocated by operator new

Operator Delete Examples

```
#include <new>
#include <cassert>

void func_1() {
    // allocating operator new
    int* ip = static_cast<int*>(::operator new(sizeof(int)));
    assert(ip);
    ::operator delete(ip);
}

void func_2() {
    // allocating and non-throwing operator new
    int* ip = static_cast<int*>(::operator new(sizeof(int),
      std::nothrow));
    // ip may be null
    ::operator delete(ip);
}
```

Operator Array Delete (i.e., `operator delete[]`)

- operator array delete (i.e., **`operator delete[]`**) is operator used to deallocate memory for array of objects allocated with operator array new
- can be invoked through delete expression or through new expression if constructor throws exception
- always has return type of **`void`**
- first parameter always pointer of type **`void`***
- standard library deallocation functions do nothing if pointer is null
- can be overloaded as global function or (implicitly static) member function

Operator Array Delete Overloads

- `void operator delete[](void* ptr) noexcept;`
  `void operator delete[]( void* ptr, std::size_t size) noexcept;`
  `void operator delete[]( void* ptr, std::align_val_t align) noexcept;`
  `void operator delete[]( void* ptr, std::size_t size, std::align_val_t align) noexcept;`
  - deallocates storage associated with array of objects at address `ptr`, which was allocated by operator array new

Operator Array Delete Examples

```
#include <new>
#include <cassert>

void func_1() {
    // allocating operator array new
    int* ip = static_cast<int*>(::operator new[](1000 * sizeof(int)));
    assert(ip);
    ::operator delete[](ip);
}

void func_2() {
    int* ip = static_cast<int*>(::operator new[](1000 * sizeof(int),
      std::nothrow));
    // ip may be null
    ::operator delete[](ip);
}
```

Replacing Operator New and Operator Delete

- non-allocating global versions of single-object and array operator new and operator delete can be replaced
- to replace function, define in single translation unit
- undefined behavior if more than one replacement provided in program or if replacement defined with inline specifier

Example of Replacing Operator New and Operator Delete

```
#include <cstdio>
#include <cstdlib>
#include <new>

void* operator new(std::size_t size) {
    auto ptr = std::malloc(size);
    std::printf("operator new(%zu) returning %p\n", size, ptr);
    return ptr;
}

void operator delete(void* ptr) noexcept {
    std::printf("operator delete(%p)\n", ptr);
    std::free(ptr);
}

void* operator new[](std::size_t size) {
    auto ptr = std::malloc(size);
    std::printf("operator new[](%zu) returning %p\n", size, ptr);
    return ptr;
}

void operator delete[](void* ptr, std::size_t size) noexcept {
    std::printf("operator delete[](%p)\n", ptr);
    std::free(ptr);
}

int main() {
    int* ip = new int;
    delete ip;
    int* ap = new int[10];
    delete[] ap;
}
```

Motivation for Placement New

```
#include <cstdint>

// heap-allocated array of bounded size
template <class T>
class bvec {
public:
    // create empty vector that can hold max_size elements
    // why is this implementation extremely inefficient?
    bvec(std::size_t max_size) {
        start_ = new T[max_size];
        end_ = start_ + max_size;
        finish_ = start_; // mark array empty
    }
    // why is this implementation extremely inefficient?
    ~bvec() {
        delete[] start_;
    }
    // ...
private:
    T* start_; // start of storage for element data
    T* finish_; // one past end of element data
    T* end_; // end of storage for element data
};
```

Placement New

- placement new expression is new expression that specifies one or more (optional) placement arguments
- often, placement new used for purpose of constructing object ***at specific place*** in memory
- this is accomplished by forcing non-allocating overload of operator new to be used (via placement arguments of new expression)
- example:

```
alignas(std::string) char buffer[sizeof(std::string)];
std::string* sp =
  new (static_cast<void*>(buffer)) std::string("Hello");
assert(static_cast<void*>(sp) == buffer);
// ... (destroy)
```

- although, in principle, placement new can also be used with new expressions for arrays, not very practically useful (since objects in array can always be created using single-object placement new expressions)

Placement New Examples

```
#include <new>
#include <vector>
#include <cassert>
#include <utility>

void func_1() {
    int buffer;
    int* ip = new (static_cast<void*>(&buffer)) int(42);
    assert(ip == &buffer && buffer == 42);
}

void func_2() {
    alignas(int) char buffer[sizeof(int)];
    int* ip = new (static_cast<void*>(buffer)) int(42);
    assert(static_cast<void*>(ip) == buffer && *ip == 42);
}

template <class T, class... Args>
T* construct_at(void* ptr, Args&&... args) {
    return ::new (ptr) T(std::forward<Args>(args)...);
}

void func_3() {
    alignas(std::vector<int>) char buffer[sizeof(std::vector<int>)];
    std::vector<int>* vp = construct_at<std::vector<int>>(buffer, 1000, 42);
    assert(static_cast<void*>(vp) == buffer && vp->size() == 1000 &&
      (*vp)[0] == 42);
    // ... (destroy vector)
}
```

Direct Destructor Invocation

- can directly invoke destructor of class object
- only very special circumstances necessitate direct invocation of destructor
- used in situations where deallocation must be performed separately from destruction (in which case delete expression cannot be used as it performs both destruction and deallocation together)
- typical use case is for implementing container classes where destruction of object stored in container and deallocation of memory occupied by that object done at different points in time
- given pointer `p` to class object of type `T`, can directly invoke destructor through pointer using syntax:

```
p->~T()
```

- example:

```
alignas(std::vector<int>) char buf[
  sizeof(std::vector<int>)];
std::vector<int>* vp = new (static_cast<void*>(buf))
  std::vector<int>(1024);
vp->~vector();
```

Section 3.1.2

More on Memory Management

std::addressof Function Template

- for memory management purposes, often necessary to obtain address of object
- if class overloads address-of operator, obtaining address of object becomes more difficult
- for convenience, standard library provides `std::addressof` function template for querying address of object, which yields correct result even if class overloads address-of operator
- declaration:

```
template <class T>
  constexpr T* addressof(T& arg) noexcept;
template <class T>
  const T* addressof(const T&&) = delete;
```

- `addressof` function should be used any time address of object is required whose class may have overloaded address-of operator
- example:

```
template <class T> foo(const T& x) {
    const T* p = std::addressof(x);
    // ...
}
```

std::addressof Example

```
#include <iostream>
#include <cassert>
#include <memory>

// class that overloads address-of operator
class Foo {
public:
    Foo(int i) : i_(i) {}
    const Foo* operator&() const {return nullptr;}
    Foo* operator&() {return nullptr;}
    int get() const {return i_;}
    // ...
private:
    int i_;
};

int main() {
    Foo f(42);
    assert(&f == nullptr);
    assert(std::addressof(f) != nullptr &&
      std::addressof(f)->get() == 42);
    std::cout << std::addressof(f) << '\n';
}
```

The std::aligned_storage Class Template

- often need can arise for buffer of particular size and alignment
- for convenience, standard library provides `std::aligned_storage` class template for specifying such buffers
- declaration:

```
template <std::size_t Size, std::size_t Align =
  __default_alignment> struct aligned_storage;
```

- `Size` is size of storage buffer in bytes
- `Align` is alignment of storage buffer (which has implementation-dependent default)
- for additional convenience, `std::aligned_storage_t` alias template also provided
- declaration:

```
template <std::size_t Size, std::size_t Align =
  __default_alignment> using aligned_storage_t = typename
  aligned_storage<Len, Align>::type;
```

- example:

```
std::aligned_storage_t<sizeof(std::string),
  alignof(std::string)> buffer;
```

Optional Value Example

- consider container class called `optval` that can hold optional value
- class templated on type `T` of optional value
- container object in one of two states:
  1. holding value of type `T`
  2. not holding any value
- can query if container is holding value, and if so, access held value
- somewhat similar in spirit to `std::optional`
- want to store object of type `T` ***in*** `optval` ***object itself***
- no memory allocation required
- example demonstrates use of placement new (to construct object at particular place in memory) and direct invocation of destructor

Optional Value Example: optval.hpp

```cpp
#include <new>
#include <type_traits>

template <class T> class optval {
public:
    optval() : valid_(false) {}
    ~optval() {clear();}
    optval(const optval&) = delete; // for simplicity
    optval& operator=(const optval&) = delete; // for simplicity
    bool has_value() const noexcept {return valid_;}
    const T& get() const {return reinterpret_cast<const T&>(storage_);}
    void clear() noexcept {
        if (valid_) {
            reinterpret_cast<T*>(&storage_)->~T();
            valid_ = false;
        }
    }
    void set(const T& value) {
        clear();
        ::new (static_cast<void*>(&storage_)) T(value);
        valid_ = true;
    }
private:
    bool valid_; // is value valid?
    std::aligned_storage_t<sizeof(T), alignof(T)> storage_;
      // storage for value
      // or alternatively: alignas(T) char storage_[sizeof(T)];
};
```

```cpp
#include <cassert>
#include <string>
#include <iostream>
#include "optional_1_util.hpp"

int main() {
    optval<std::string> s;
    assert(!s.has_value());
    s.set("Hello, World");
    assert(s.has_value());
    std::cout << s.get() << '\n';
    s.clear();
    assert(!s.has_value());
}
```

Handling Uninitialized Storage

- sometimes need may arise to work with uninitialized storage
- may want to construct objects in uninitialized storage (by using placement new to invoke constructor) and later destroy objects
- may want to move or copy objects into uninitialized storage (by using placement new to invoke move or copy constructor)
- code required to perform above operations is not very long, but must be written with some care to ensure that exceptions handled correctly
- standard library provides functions that perform these operations for convenience
- these functions useful for code that manages memory without using standard-compliant allocators

Functions for Uninitialized Storage

Operations on Uninitialized Memory

Name	Description
`uninitialized_copy`	copy range of objects to uninitialized area of memory
`uninitialized_copy_n`	copy number of objects to uninitialized area of memory
`uninitialized_fill`	copy object to uninitialized area of memory, defined by range
`uninitialized_fill_n`	copy object to uninitialized area of memory, defined by start and count
`uninitialized_move`	move range of objects to uninitialized area of memory
`uninitialized_move_n`	move number of objects to uninitialized area of memory

Functions for Uninitialized Storage (Continued)

Operations on Uninitialized Memory (Continued)

Name	Description
`uninitialized_default_construct`	construct objects by default initialization in uninitialized area of memory defined by range
`uninitialized_default_construct_n`	construct objects by default initialization in uninitialized area of memory defined by start and count
`uninitialized_value_construct`	construct objects by value initialization in uninitialized area of memory defined by range
`uninitialized_value_construct_n`	construct objects by value initialization in uninitialized area of memory defined by start and count
`destroy_at`	destroy object at given address
`destroy`	destroy range of objects
`destroy_n`	destroy number of objects in range

Some Example Implementations

```
template<class InputIter, class ForwardIter>
ForwardIter uninitialized_copy(InputIter first, InputIter last,
  ForwardIter result) {
    using Value = typename std::iterator_traits<ForwardIter>::value_type;
    ForwardIter current = result;
    try {
        for (; first != last; ++first, (void) ++current) {
            ::new (static_cast<void*>(std::addressof(*current))) Value(*first);
        }
    } catch (...) {
        for (; result != current; ++result) {
            result->~Value();
        }
        throw;
    }
    return current;
}
```

```
template <class ForwardIter>
void destroy(ForwardIter first, ForwardIter last) {
    for (; first != last; ++first) {
        std::destroy_at(std::addressof(*first));
    }
}
```

```
template <class T>
void destroy_at(T* p) {p->~T();}
```

Bounded Array Example

- consider class `array` for bounded one-dimensional array whose maximum size is compile-time constant
- class templated on element type `T` and number `N` of elements in array
- array element data is stored ***in* `array` *object itself***
- no memory allocation required
- provide only basic container functionality in order to keep example to reasonable size for slides
- example demonstrates handling of uninitialized memory using standard library functions
- similar in spirit to `boost::static_vector`

Bounded Array Example: aligned_buffer.hpp

```
// type-aware aligned buffer class
// provides buffer suitably aligned for N elements of type T
template <class T, std::size_t N>
  class aligned_buffer {
public:
    const T* start() const noexcept
      {return reinterpret_cast<const T*>(storage_);}
    T* start() noexcept {return reinterpret_cast<T*>(storage_);}
    const T* end() const noexcept {return start() + N;}
    T* end() noexcept {return start() + N;}
private:
    alignas(T) char storage_[N * sizeof(T)]; // aligned buffer
};
```

Bounded Array Example: `array.hpp` (1)

```
#include <memory>
#include <algorithm>
#include <type_traits>
#include "aligned_buffer.hpp"

template <class T, std::size_t N> class array {
public:
    array() : finish_(buf_.start()) {}
    array(const array& other);
    array(array&& other);
    ~array() {clear();}
    array& operator=(const array& other);
    array& operator=(array&& other);
    explicit array(std::size_t size);
    array(std::size_t size, const T& value);
    constexpr std::size_t max_size() const noexcept {return N;}
    std::size_t size() const noexcept {return finish_ - buf_.start();}
    T& operator[](std::size_t i) {return buf_.start()[i];}
    const T& operator[](std::size_t i) const {return buf_.start()[i];}
    T& back() {return finish_[-1];}
    const T& back() const {return finish_[-1];}
    void push_back(const T& value);
    void pop_back();
    void clear() noexcept;
private:
    T* finish_; // one past last element in buffer
    aligned_buffer<T, N> buf_; // buffer for array elements
};
```

Bounded Array Example: array.hpp (2)

```
template <class T, std::size_t N>
array<T, N>::array(const array& other) {
    finish_ = std::uninitialized_copy(other.buf_.start(),
      const_cast<const T*>(other.finish_), buf_.start());
}

template <class T, std::size_t N>
array<T, N>::array(array&& other) {
    finish_ = std::uninitialized_move(other.buf_.start(), other.finish_,
      buf_.start());
}

template <class T, std::size_t N>
array<T, N>& array<T, N>::operator=(const array& other) {
    if (this != &other) {
        clear();
        finish_ = std::uninitialized_copy(other.buf_.start(),
          const_cast<const T*>(other.finish_), buf_.start());
    }
    return *this;
}

template <class T, std::size_t N>
array<T, N>& array<T, N>::operator=(array&& other) {
    if (this != &other) {
        clear();
        finish_ = std::uninitialized_move(other.buf_.start(), other.finish_,
          buf_.start());
    }
    return *this;
}
```

Bounded Array Example: `array.hpp` (3)

```
template <class T, std::size_t N>
array<T, N>::array(std::size_t size) {
    if (size > max_size()) {size = max_size();}
    std::uninitialized_default_construct(buf_.start(), buf_.start() + size);
    finish_ = buf_.start() + size;
}

template <class T, std::size_t N>
array<T, N>::array(std::size_t size, const T& value) {
    if (size > max_size()) {size = max_size();}
    finish_ = std::uninitialized_fill_n(buf_.start(), size, value);
}

template <class T, std::size_t N>
void array<T, N>::push_back(const T& value) {
    if (finish_ == buf_.end()) {return;}
    finish_ = std::uninitialized_fill_n(finish_, 1, value);
}

template <class T, std::size_t N>
void array<T, N>::pop_back() {
    --finish_;
    std::destroy_at(finish_);
}
```

Bounded Array Example: `array.hpp` (4)

```
template <class T, std::size_t N>
void array<T, N>::clear() noexcept {
    std::destroy(buf_.start(), finish_);
    finish_ = buf_.start();
}
```

Vector Example

- consider class `vec` that is one-dimensional dynamically-resizable array
- class templated on array element type `T`
- storage for element data allocated with ***operator new***
- similar in spirit to `std::vector` but much simplified:
  - cannot specify allocator to be used (i.e., always uses operator new and operator delete for memory allocation)
  - does not provide iterators

Vector Example: vec.hpp (1)

```
#include <new>
#include <algorithm>
#include <type_traits>
#include <memory>

template <class T> class vec {
public:
    vec() : start_(nullptr), finish_(nullptr), end_(nullptr) {}
    vec(const vec& other);
    vec(vec&& other) noexcept;
    ~vec();
    vec& operator=(const vec& other);
    vec& operator=(vec&& other) noexcept;
    explicit vec(std::size_t size);
    vec(std::size_t n, const T& value);
    std::size_t capacity() const noexcept {return end_ - start_;}
    std::size_t size() const noexcept {return finish_ - start_;}
    T& operator[](int i) {return start_[i];}
    const T& operator[](int i) const {return start_[i];}
    T& back() {return finish_[-1];}
    const T& back() const {return finish_[-1];}
    void push_back(const T& value);
    void pop_back();
    void clear() noexcept;
private:
    void grow(std::size_t n);
    T* start_; // start of element storage
    T* finish_; // one past last valid element
    T* end_; // end of element storage
};
```

Vector Example: vec.hpp (2)

```
template <class T>
vec<T>::vec(const vec& other) {
    start_ = static_cast<T*>(::operator new(other.size() * sizeof(T)));
    end_ = start_ + other.size();
    try {
        finish_ = std::uninitialized_copy(other.start_, other.finish_, start_);
    } catch (...) {
        ::operator delete(start_);
        throw;
    }
}

template <class T>
vec<T>::vec(vec&& other) noexcept {
    start_ = other.start_;
    other.start_ = nullptr;
    end_ = other.end_;
    other.end_ = nullptr;
    finish_ = other.finish_;
    other.finish_ = nullptr;
}

template <class T>
vec<T>::~vec() {
    clear();
    ::operator delete(start_);
}
```

```
template <class T>
vec<T>& vec<T>::operator=(const vec& other) {
    if (this != &other) {
        clear();
        if (other.size() > capacity()) {grow(other.size());}
        finish_ = std::uninitialized_copy(other.start_, other.finish_, start_);
    }
    return *this;
}

template <class T>
vec<T>& vec<T>::operator=(vec&& other) noexcept {
    if (this != &other) {
        clear();
        ::operator delete(start_);
        start_ = other.start_;
        other.start_ = nullptr;
        finish_ = other.finish_;
        other.finish_ = nullptr;
        end_ = other.end_;
        other.end_ = nullptr;
    }
    return *this;
}
```

Vector Example: vec.hpp (4)

```
template <class T>
vec<T>::vec(std::size_t n) {
    start_ = static_cast<T*>(::operator new(n * sizeof(T)));
    end_ = start_ + n;
    try {std::uninitialized_default_construct_n(start_, n);}
    catch (...) {
        ::operator delete(start_);
        throw;
    }
    finish_ = end_;
}

template <class T>
vec<T>::vec(std::size_t n, const T& value) {
    start_ = static_cast<T*>(::operator new(n * sizeof(T)));
    end_ = start_ + n;
    try {std::uninitialized_fill_n(start_, n, value);}
    catch (...) {
        ::operator delete(start_);
        throw;
    }
    finish_ = end_;
}
```

```
template <class T>
void vec<T>::push_back(const T& value) {
    if (finish_ == end_) {
        // might want to check for overflow here
        grow(2 * capacity());
    }
    finish_ = std::uninitialized_fill_n(finish_, 1, value);
}

template <class T>
void vec<T>::pop_back() {
    --finish_;
    std::destroy_at(finish_);
}

template <class T>
void vec<T>::clear() noexcept {
    if (size()) {
        std::destroy(start_, finish_);
        finish_ = start_;
    }
}
```

Vector Example: vec.hpp (6)

```
template <class T>
void vec<T>::grow(std::size_t n) {
    T* new_start = static_cast<T*>(::operator new(n * sizeof(T)));
    std::size_t old_size = size();
    try {
        std::uninitialized_move(start_, finish_, new_start);
    } catch (...) {
        ::operator delete(new_start);
        throw;
    }
    ::operator delete(start_);
    start_ = new_start;
    finish_ = new_start + old_size;
    end_ = new_start + n;
}
```

Section 3.1.3

Allocators

Allocators

- allocators provide uniform interface for allocating and deallocating memory for object of particular type
- interface that allocator must provide specified in C++ standard
- each allocator type embodies particular memory allocation policy
- perform allocation, construction, destruction, and deallocation
- allocation separate from construction
- destruction separate from deallocation
- encapsulate information about allocation strategy and addressing model
- hide memory management and addressing model details from containers
- allow reuse of code implementing particular allocation strategy with any allocator-aware container

Containers, Allocators, and the Default Allocator

- container class templates typically take allocator type as parameter
- this allows more than one memory allocation policy to be used with given container class template
- in case of standard library, many container class templates take allocator type as template parameter, including:
  - `vector`, `list`
  - `set`, `multiset`, `map`, `multimap`
  - `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`
- all container class templates in standard library that take allocator as parameter use default of `std::allocator<T>` where `T` must be type of element held by container
- `std::allocator` employs operator new and operator delete for memory allocation
- in many contexts, default allocator is quite adequate

Application Use of Allocator

```
#include <memory>
#include <vector>
#include <map>
#include <cassert>
#include <boost/pool/pool_alloc.hpp>

int main() {
    // use default allocator
    std::vector<int> u;
    u.push_back(42);

    // explicitly specify default allocator
    std::vector<int, std::allocator<int>> v;
    static_assert(std::is_same_v<decltype(u), decltype(v)>);
    assert(u.get_allocator() == v.get_allocator());
    v.push_back(42);

    // specify an allocator type from Boost
    std::vector<int, boost::pool_allocator<int>> w;
    w.push_back(42);

    // explicitly specify default allocator
    std::map<int, long, std::less<int>,
      std::allocator<std::pair<const int, long>>> x;
    x.insert({1, 2});
}
```

Why Not Just Always Use the Default Allocator?

- custom allocators used when greater control is needed over how memory is managed
- often this greater control is desired for:
  - improved efficiency (e.g., better locality and less contention)
  - debugging
  - performance analysis (e.g., collecting statistics on memory allocation)
  - testing (e.g., forcing allocation failures)
  - security (e.g., locking and clearing memory)
- since many allocation strategies are possible, one strategy cannot be best in all situations
- some allocation strategies include:
  - stack-based allocation
  - per-container allocation
  - per-thread allocation (which avoids synchronization issues)
  - pooled allocation
  - arena allocation
- may want to handle relocatable data (e.g., shared memory)
- may want to use memory mapped files

Examples of Allocators

- other examples of (standard-compliant) allocators include:
  - `std::pmr::polymorphic_allocator` (allocator whose behavior depends on memory resource with which it was constructed)
  - `boost::interprocess::allocator` (shared memory allocator)
  - `boost::pool_alloc` (pool allocator)
  - `boost::fast_pool_alloc` (pool allocator)

Allocators

- allocator handles memory allocation for objects of specific type (e.g., allocator for **`int`**s)
- allocator normally accessed by container type through interface of traits class called `std::allocator_traits`
- container class typically use allocator for managing memory associated with container element data
- four basic types of operations provided by allocator through traits class:
  - allocate memory
  - deallocate memory
  - construct object
  - destroy object
- two allocator instances deemed ***equal*** if memory allocated with each instance can be deallocated with other
- allocator objects may have state

Allocator Members

- allocator type for objects of (cv-unqualified) type `T`
- many members are optional, with `std::allocator_traits` class effectively providing defaults for omitted members
- `value_type`:
  - type `T` of object for which allocator manages (i.e., allocates and deallocates) memory
- `pointer`:
  - pointer type used to refer to storage obtained from allocator (not necessarily `T*`)
  - optional: default of `T*` provided by `allocator_traits`
- `const_pointer`:
  - const version of `pointer`
  - optional: default of `const T*` provided by `allocator_traits`
- `pointer allocate(size_type n)`:
  - allocate storage suitable for `n` objects of type `T`
- `void deallocate(pointer ptr, size_type n)`:
  - deallocates storage pointed to by `ptr`, where `ptr` must have been obtained by previous call to `allocate` and `n` must match value given in that call

Allocator Members (Continued)

- **void** `construct(value_type* ptr, Args&&... args)`:
  - constructs object of type `T` in storage pointed to by `ptr` using specified arguments `args`
  - optional: default behavior provided by `allocator_traits` is to use placement new expression
- **void** `destroy(value_type* ptr)`:
  - destroys object of type `T` in storage pointed to by `ptr`
  - optional: default behavior provided by `allocator_traits` is to directly invoke destructor

Remarks on Allocators

- `pointer` and `const_pointer` must satisfy requirements of random-access and contiguous iterators
- `pointer` and `const_pointer` can be fancy pointers (i.e., smart pointers)
- fancy pointers useful, for example, in allocating storage in shared memory region

```
#include <cstdlib>
#include <new>

template <class T>
struct mallocator {
    using value_type = T;
    mallocator() noexcept {};
    template <class U> mallocator(const mallocator<U>&) noexcept {}
    T* allocate(std::size_t n) const;
    void deallocate(T* p, std::size_t n) const noexcept;
    template <class U> bool operator==(const mallocator<U>&)
      const noexcept {return true;}
    template <class U> bool operator!=(const mallocator<U>&)
      const noexcept {return false;}
};

template <class T>
T* mallocator<T>::allocate(std::size_t n) const {
    if (!n) {return nullptr;}
    if (n > static_cast<std::size_t>(-1) / sizeof(T))
      {throw std::bad_array_new_length();}
    void* p = std::malloc(n * sizeof(T));
    if (!p) {throw std::bad_alloc();}
    return static_cast<T*>(p);
}

template <class T>
void mallocator<T>::deallocate(T* p, std::size_t) const noexcept
  {std::free(p);}
```

Malloc-Based Allocator: User Code

```
#include "mallocator.hpp"
#include <cassert>
#include <vector>
#include <type_traits>

int main() {
    std::vector<int, mallocator<int>> v;
      // uses mallocator<int> for memory allocation
    std::vector<int> w;
      // or equivalently, std::vector<int, std::allocator<int>>
      // uses std::allocator<int> for memory allocation
    static_assert(!std::is_same_v<decltype(v)::allocator_type,
      decltype(w)::allocator_type>);
    for (int i = 0; i < 128; ++i) {
        v.push_back(42);
        w.push_back(42);
    }
    std::vector<int, mallocator<int>> x;
    assert(v.get_allocator() == x.get_allocator());
}
```

Allocator Propagation

- in certain contexts, must consider if and how allocators should be propagated between container objects
- **lateral propagation** refers to propagation of allocator when copying, moving, and swapping containers:
  - when container copy/move constructed, what allocator does new container receive?
  - when container copy/move assigned, what allocator does copied-to/moved-to container receive?
  - when containers swapped, what allocator does each container receive?
- **deep propagation** refers to propagation of allocator from parent container to its descendents in hierarchy of nested containers:
  - if container contains types which themselves require allocators, how can contained elements be made aware of container's allocator so that compatible allocator can be used?
- each allocator has its own lateral propagation properties, which can be accessed via `std::allocator_traits`
- deep allocator propagation can be controlled via `std::scoped_allocator_adaptor`

```
#include <new>
#include <type_traits>

template <class T>
struct allocator {
    using value_type = T;
    using propagate_on_container_move_assignment = std::true_type;
    using is_always_equal = std::true_type;
    allocator() noexcept {};
    allocator(const allocator&) noexcept {};
    template <class U> allocator(const allocator<U>&) noexcept {}
    ~allocator() {}
    T* allocate(std::size_t n);
    void deallocate(T* p, std::size_t n) const noexcept
      {::operator delete(p);}
};

template <class T>
T* allocator<T>::allocate(std::size_t n) {
    if (n > static_cast<std::size_t>(-1) / sizeof(T))
      {throw std::bad_array_new_length();}
    return static_cast<T*>(::operator new(n * sizeof(T)));
}

template <class T, class U>
inline bool operator==(const allocator<T>&, const allocator<U>&) noexcept
  {return true;}

template <class T, class U>
inline bool operator!=(const allocator<T>&, const allocator<U>&) noexcept
  {return false;}
```

Fixed-Size Arena Allocator: Example

- consider example of simple allocator that allocates memory from fixed-size buffer
- arena class (called `arena`) provides memory allocation from fixed-size buffer with some prescribed minumum alignment
- allocator class (called `salloc`) provides interface to particular `arena` instance
- `salloc` object holds pointer to `arena` object (so allocator is stateful)
- `arena` object makes no attempt to deallocate memory (i.e., deallocate operation does nothing)
- allocator might be used for relatively small allocations from stack (where arena object would be local variable)
- allocator always propagated for copy, move, and swap (i.e., POCMA, POCCA, and POCS, as defined later, all true)
- two instances of allocator not necessarily equal

Fixed-Size Arena Allocator: Code (1)

```cpp
#include <memory>
#include <cstddef>
#include <new>

template <std::size_t N, std::size_t Align = alignof(std::max_align_t)>
class arena {
public:
    arena() : ptr_(buf_) {}
    arena(const arena&) = delete;
    arena& operator=(const arena&) = delete;
    ~arena() = default;
    constexpr std::size_t alignment() const {return Align;}
    constexpr std::size_t capacity() const {return N;}
    constexpr std::size_t used() const {return ptr_ - buf_;}
    constexpr std::size_t free() const {return N - used();}
    template <std::size_t ReqAlign> void* allocate(std::size_t n);
    void deallocate(void* ptr, std::size_t n) {}
    void clear() {ptr_ = buf_;}
private:
    template <std::size_t ReqAlign>
    static char* align(char* ptr, std::size_t n, std::size_t max);
    alignas(Align) char buf_[N]; // storage buffer
    char* ptr_; // pointer to first unused byte
};
```

Fixed-Size Arena Allocator: Code (2)

```
template <std::size_t N, std::size_t Align>
template <std::size_t ReqAlign>
char* arena<N, Align>::align(char* ptr, std::size_t n, std::size_t max) {
    void* p = ptr;
    return static_cast<char*>(std::align(ReqAlign, n, p, max));
}

template <std::size_t N, std::size_t Align>
template <std::size_t ReqAlign>
void* arena<N, Align>::allocate(std::size_t n) {
    char* ptr = this->align<std::max(Align, ReqAlign)>(ptr_, n, free());
    if (!ptr) {throw std::bad_alloc();}
    ptr_ = ptr + n;
    return ptr;
}
```

Fixed-Size Arena Allocator: Code (3)

```
template <class T, std::size_t N, std::size_t Align = alignof(T)>
class salloc {
public:
    using value_type = T;
    using propagate_on_container_move_assignment = std::true_type;
    using propagate_on_container_copy_assignment = std::true_type;
    using propagate_on_container_swap = std::true_type;
    using is_always_equal = std::false_type;
    using arena_type = arena<N, Align>;
    salloc select_on_container_copy_construction() const {return *this;}
    template <class U> struct rebind {using other = salloc<U, N, Align>;};
    template <class T2>
    salloc(const salloc<T2, N, Align>& other) : a_(other.a_) {}
    salloc(arena_type& a) : a_(&a) {}
    ~salloc() = default;
    salloc(const salloc&) = default;
    salloc(salloc&& other) = default;
    salloc& operator=(const salloc&) = default;
    salloc& operator=(salloc&& other) = default;
    T* allocate(std::size_t n) {
        if (n > static_cast<std::size_t>(-1) / sizeof(T))
          {throw std::bad_alloc();}
        return static_cast<T*>(a_->template allocate<alignof(T)>(
          n * sizeof(T)));
    }
    void deallocate(T* p, std::size_t n)
      {return a_->deallocate(p, n * sizeof(T));}
```

Fixed-Size Arena Allocator: Code (4)

```
private:
    template <class T1, std::size_t N1, std::size_t A1, class T2,
      std::size_t N2, std::size_t A2>
    friend bool operator==(const salloc<T1, N1, A1>&,
      const salloc<T2, N2, A2>&);
    template <class, std::size_t, std::size_t> friend class salloc;
    arena_type* a_; // arena from which to allocate storage
};

template <class T1, std::size_t N1, std::size_t A1, class T2, std::size_t N2,
  std::size_t A2>
inline bool operator==(const salloc<T1, N1, A1>& a,
  const salloc<T2, N2, A2>& b)
  {return N1 == N2 && A1 == A2 && a.a_ == b.a_;}

template <class T1, std::size_t N1, std::size_t A1, class T2, std::size_t N2,
  std::size_t A2>
inline bool operator!=(const salloc<T1, N1, A1>& a,
  const salloc<T2, N2, A2>& b)
  {return !(a == b);}
```

Fixed-Size Arena Allocator: User Code

```
#include <vector>
#include <list>
#include <iostream>
#include "salloc.hpp"

int main() {
    using alloc = salloc<int, 1024, sizeof(int)>;
    alloc::arena_type a;
    std::vector<int, alloc> v{{0, 1, 2, 3}, a};
    std::vector<int, alloc> w{{0, 2, 4, 6}, a};
    std::list<int, alloc> p{{1, 3, 5, 7}, a};
    std::cout << a.free() << '\n';
    v.push_back(42);
    for (auto&& i : v) {std::cout << i << '\n';}
    for (auto&& i : w) {std::cout << i << '\n';}
    for (auto&& i : p) {std::cout << i << '\n';}
    std::cout << a.free() << '\n';

    // std::vector<int, alloc> x(1024);
    // std::list<int, alloc> y;
    // ERROR: allocator cannot be default constructed
}
```

Allocator-Aware Containers

- container that uses allocator sometimes referred to as **allocator aware**
- typically much more difficult to develop allocator-aware container than container that does not use allocator
- type of pointer returned by allocator not necessarily same as pointer to element type, which sometimes complicates code somewhat
- much of complexity in implementing allocator-aware container, however, arises from issue of allocator propagation

The `std::allocator_traits` Class Template

- allocators intended to be used via allocator user (e.g., container) indirectly through traits class `std::allocator_traits`
- declaration:

```
template <class Alloc> struct allocator_traits;
```

- `allocator_traits` provides uniform interface to allocators used by containers
- some properties of allocator types are optional
- in cases where allocator type did not specify optional properties, `allocator_traits` provides default

Lateral Allocator Propagation

- properties of allocator in `std::allocator_traits` used to control lateral allocator propagation
- container copy constructor obtains allocator for new container by invoking `select_on_container_copy_construction` in `allocator_traits`
- container move constructor always propagates allocator by move
- container copy assignment replaces allocator (in copied-to container) only if `propagate_on_container_copy_assignment` (POCCA) in `allocator_traits` is true
- container move assignment replaces allocator (in moved-to container) only if `propagate_on_container_move_assignment` (POCMA) in `allocator_traits` is true
- container swap will swap allocators of two containers only if `propagate_on_container_swap` (POCS) in `allocator_traits` is true
- if POCS is false, swapping two standard-library containers with unequal allocators is undefined behavior (since swap must not invalidate iterators and iterators would have to be invalidated in this case)

Allocator-Traits Querying Example

```
#include <memory>
#include <type_traits>
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <iostream>

template <class T> void print(std::ostream& out = std::cout) {
    out << std::is_same_v<typename T::pointer, typename T::value_type*> << ' '
      << std::is_same_v<typename T::const_pointer,
      const typename T::value_type*> << ' '
      << T::is_always_equal::value << ' '
      << T::propagate_on_container_move_assignment::value << ' '
      << T::propagate_on_container_copy_assignment::value << ' '
      << T::propagate_on_container_swap::value << '\n';
}

int main() {
    namespace bi = boost::interprocess;
    print<std::allocator_traits<std::allocator<int>>>();
    print<std::allocator_traits<bi::allocator<int,
      bi::managed_shared_memory::segment_manager>>>();
}

/* Output:
1 1 1 1 0 0
0 0 0 0 0 0
*/
```

Optional Value Example

- consider container class template called `optval` that can hold optional value
- class templated on element type `T` and allocator type
- container object in one of two states:
  1. holding value of type `T`
  2. not holding any value
- can query if container is holding value, and if so, access held value
- want to store object of type `T` in memory obtained ***from allocator***
- example illustrates basic use of allocator in container class

Optional Value Example: Code (1)

```
#include <memory>
#include <type_traits>
#include <utility>

template <class T, class Alloc = std::allocator<T>>
class optval : private Alloc {
public:
    using value_type = T;
    using allocator_type = Alloc;
private:
    using traits = typename std::allocator_traits<Alloc>;
public:
    using pointer = typename traits::pointer;
    using const_pointer = typename traits::const_pointer;
    optval(std::allocator_arg_t, const allocator_type& alloc) :
      Alloc(alloc), value_(nullptr) {}
    optval() : optval(std::allocator_arg, allocator_type()) {}
    optval(std::allocator_arg_t, const allocator_type& alloc,
      const optval& other);
    optval(const optval& other);
    optval(std::allocator_arg_t, const allocator_type& alloc, optval&& other)
      noexcept;
    optval(optval&& other) noexcept;
    optval(std::allocator_arg_t, const allocator_type& alloc, const T& value);
    optval(const T& value);
    ~optval();
    optval& operator=(const optval& other);
    optval& operator=(optval&& other)
      noexcept(traits::propagate_on_container_move_assignment::value);
    void swap(optval& other) noexcept;
```

Optional Value Example: Code (2)

```
    allocator_type get_allocator() const {return alloc_();}
    bool has_value() const noexcept {return value_;}
    const T& get() const {return *value_;}
    void clear() noexcept;
    void set(const T& value);
private:
    pointer copy_(allocator_type a, const value_type& value);
    allocator_type& alloc_() {return *this;}
    const allocator_type& alloc_() const {return *this;}
    pointer value_; // pointer to optional value
};

template <class T, class Alloc>
optval<T, Alloc>::optval(const optval& other) : optval(std::allocator_arg,
  traits::select_on_container_copy_construction(other.alloc_()), other) {}

template <class T, class Alloc>
optval<T, Alloc>::optval(std::allocator_arg_t, const allocator_type& alloc,
  const optval& other) : Alloc(alloc), value_(nullptr) {
    if (other.value_) {value_ = copy_(alloc_(), *other.value_);}
}

template <class T, class Alloc>
optval<T, Alloc>::optval(optval&& other) noexcept : Alloc(std::move(other)) {
    value_ = other.value_;
    other.value_ = nullptr;
}
```

Optional Value Example: Code (3)

```
template <class T, class Alloc>
optval<T, Alloc>::optval(std::allocator_arg_t, const allocator_type& alloc,
  optval&& other) noexcept : Alloc(alloc) {
    value_ = other.value_;
    other.value_ = nullptr;
}

template <class T, class Alloc>
optval<T, Alloc>::optval(std::allocator_arg_t, const allocator_type& alloc,
  const T& value) : Alloc(alloc), value_(nullptr)
  {value_ = copy_(alloc_(), value);}

template <class T, class Alloc>
optval<T, Alloc>::optval(const T& value) : optval(std::allocator_arg,
  allocator_type(), value) {}

template <class T, class Alloc>
optval<T, Alloc>::~optval()
  {clear();}
```

Optional Value Example: Code (4)

```
template <class T, class Alloc>
auto optval<T, Alloc>::operator=(const optval& other) -> optval& {
    if (this != &other) {
        if constexpr(traits::propagate_on_container_copy_assignment::value) {
            allocator_type a = other.alloc_();
            pointer p = other.value_ ? copy_(a, *other.value_) : nullptr;
            clear();
            alloc_() = other.alloc_();
            value_ = p;
        } else {
            pointer p = other.value_ ? copy_(alloc_(), *other.value_) : nullptr;
            clear();
            value_ = p;
        }
    }
    return *this;
}
```

Optional Value Example: Code (5)

```
template <class T, class Alloc>
auto optval<T, Alloc>::operator=(optval&& other)
  noexcept(traits::propagate_on_container_move_assignment::value) -> optval& {
    if (this != &other) {
        if constexpr (traits::propagate_on_container_move_assignment::value) {
            clear();
            std::swap(alloc_(), other.alloc_());
            std::swap(value_, other.value_);
        } else if (alloc_() == other.alloc_()) {
            clear();
            std::swap(value_, other.value_);
        } else {
            pointer p = copy_(alloc_(), other.value_);
            std::swap(value_, other.value_);
            other.clear();
            value_ = p;
        }
    }
    return *this;
}

template <class T, class Alloc>
void optval<T, Alloc>::swap(optval& other) noexcept {
    // require POCS to be true or allocators equivalent
    assert(traits::propagate_on_container_swap::value ||
      alloc_() == other.alloc_());
    if constexpr (traits::propagate_on_container_swap::value)
      {std::swap(alloc_(), other.alloc_());}
    std::swap(value_, other.value_);
}
```

Optional Value Example: Code (6)

```
template <class T, class Alloc>
void optval<T, Alloc>::clear() noexcept {
    if (value_) {
        traits::destroy(alloc_(), std::addressof(*value_));
        traits::deallocate(alloc_(), value_, 1);
        value_ = nullptr;
    }
}

template <class T, class Alloc>
void optval<T, Alloc>::set(const T& value) {
    pointer p = copy_(alloc_(), value);
    clear();
    value_ = p;
}

template <class T, class Alloc>
auto optval<T, Alloc>::copy_(allocator_type a, const value_type& value) ->
  pointer {
    pointer p = traits::allocate(alloc_(), 1);
    try {traits::construct(a, std::addressof(*p), value);}
    catch (...) {
        traits::deallocate(a, p, 1);
        throw;
    }
    return p;
}
```

The `std::scoped_allocator_adaptor` Class Template

- when using stateful allocators with nested containers, often need to ensure that allocator state is propagated from parent container to its descendants
- `std::scoped_allocator_adaptor` can be used to address this type of allocator propagation problem (i.e., deep allocator propagation)
- declaration:

```
template <class OuterAlloc, class... InnerAllocs>
class scoped_allocator_adaptor : public OuterAlloc;
```

- `OuterAlloc`: allocator type for outermost container in nesting
- `InnerAllocs`: parameter pack with allocator types for each subsequent container in nesting
- if `InnerAllocs` has too few allocator types for number of nesting levels, last allocator type repeated as necessary
- `scoped_allocator_adaptor` useful when all containers in nesting must use same stateful allocator, such as typically case when using shared-memory-segment allocator

scoped_allocator_adaptor Example

```
#include <scoped_allocator>
#include <vector>
#include <list>
#include <iostream>
#include "salloc.hpp"

int main() {
    constexpr std::size_t align = alignof(std::max_align_t);
    using inner_alloc = salloc<int, 1024, align>;
    using inner = inner_alloc::value_type;
    using outer_alloc = salloc<std::list<int, inner_alloc>, 1024,
      align>;
    using outer = outer_alloc::value_type;
    using alloc = std::scoped_allocator_adaptor<outer_alloc,
      inner_alloc>;

    using container = std::vector<outer, alloc>;
    alloc::arena_type a;
    container v(container::allocator_type(a, a));
    v.reserve(4);
    std::list<inner, inner_alloc> p({1, 2, 3}, a);
    v.push_back(p);
    for (auto&& y : v) {
        for (auto&& x : y) {
            std::cout << x << '\n';
        }
    }
}
```

```
#include <vector>
#include <scoped_allocator>
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/adaptive_pool.hpp>

namespace bi = boost::interprocess;

template <class T>
using alloc = typename bi::adaptive_pool<T, typename
  bi::managed_shared_memory::segment_manager>;

int main () {
    using row = std::vector<int, alloc<int>>;
    using matrix = std::vector<row,
      std::scoped_allocator_adaptor<alloc<row>>>;
    bi::managed_shared_memory s(bi::create_only, "data", 8192);
    matrix v(s.get_segment_manager());
    v.resize(4);
    for (int i = 0; i < 4; ++i) {v[i].push_back(0);}
    bi::shared_memory_object::remove("data");
}
```

Section 3.1.4

References

References I

1. T. Koppe, A Visitor's Guide to C++ Allocators, https://rawgit.com/google/cxx-std-draft/allocator-paper/allocator_user_guide.html.

Talks I

1. Alisdair Meredith, Making Allocators Work, CppCon, Sept. 10, 2014. Available online at http://youtu.be/YkiYOP3d64E and http://youtu.be/Q5kyiFevMJQ. (This talk is in two parts.)
2. Alisdair Meredith, Allocators in C++11, C++Now, Aspen, CO, USA, May 2013. Available online at https://youtu.be/v7B_8IbHjxA.
3. Andrei Alexandrescu, std::allocator is to Allocation What std::vector is to Vexation, CppCon, Bellevue, WA, USA, Sept. 24, 2015. Available online at https://youtu.be/LIb3L4vKZ7U.
4. Alisdair Meredith, An allocator model for std2, CppCon, Bellevue, WA, USA, Sept. 25, 2017. Available online at https://youtu.be/oCi_QZ6K_qk.
   This talk explains how allocators evolved from C++98 to C++17 and briefly how they might be further evolved in future versions of the C++ standard.

Talks II

5. Bob Steagall, How to Write a Custom Allocator, CppCon, Bellevue, WA, USA, Sept. 28, 2017. Available online at `https://youtu.be/kSWfushlvB8`.

   This talk discusses how to write allocators for C++14/C++17 and how to use such allocators in containers.

6. Bob Steagall, Testing the Limits of Allocator Awareness, C++Now, Aspen, CO, USA, May 18, 2017. Available online at `https://youtu.be/fmJfKm9ano8`.

   This talk briefly introduces allocators and then describes a test suite for allocators and presents some results obtained with this test suite.

7. Pablo Halpern, Modern Allocators: The Good Parts, CppCon, Bellevue, WA, USA, Sept. 29, 2017. Available online at `https://youtu.be/v3dz-AKOVL8`.

   This talk introduces polymorphic allocators and considers a simple example of a polymorphic allocator and a container that uses a polymorphic allocator.

Talks III

8. Sergey Zubkov, From Security to Performance to GPU Programming: Exploring Modern Allocators, CppCon, Bellevue, WA, USA, Sept. 25, 2017. Available online at `https://youtu.be/HdQ4aOZyuHw`.
9. Stephan Lavavej, STL Features and Implementation Techniques, CppCon, Bellevue, WA, USA, 2014. Available online at `https://youtu.be/dTeKf5Oek2c`.

   This talk briefly discusses allocators in C++11 at 26:26–31:32.

Section 3.2

Smart Pointers

Section 3.2.1

Introduction

Memory Management, Ownership, and Raw Pointers

- responsibility of owner of chunk of dynamically-allocated memory to deallocate that memory when no longer needed
- so managing dynamically-allocated memory essentially reduces to problem of ownership management
- raw pointer does not have any ownership relationship with memory to which pointer refers
- consequently, raw pointer does not itself directly participate in memory management (e.g., deallocation)
- raw pointers often problematic in presence of exceptions, since such pointers do not know how to free their pointed-to memory
- raw pointers should only be used in situations where no ownership responsibility for pointees is needed (e.g., to simply observe object without managing its associated memory)

Smart Pointers

- **smart pointer** is object that has interface similar to raw pointer (e.g., provides operations such as indirection/dereferencing and assignment) but offers some additional functionality
- smart pointers provide RAII mechanism for managing memory resource (i.e., pointed-to memory)
- unlike raw pointer, smart pointer owns its pointed-to memory
- consequently, smart pointer must provide mechanism for deallocating pointed-to memory when no longer needed
- some smart-pointer types allow only exclusive ownership, while others allow shared ownership
- destructor for smart pointer releases memory to which pointer refers if no longer needed (i.e., no other owners remain)
- smart pointers play crucial role in writing exception-safe code
- smart pointers should always be used (instead of raw pointers) when ownership of piece of memory needs to be tracked (e.g., so that it can be deallocated when no longer needed)

Section 3.2.2

The `std::unique_ptr` Class Template

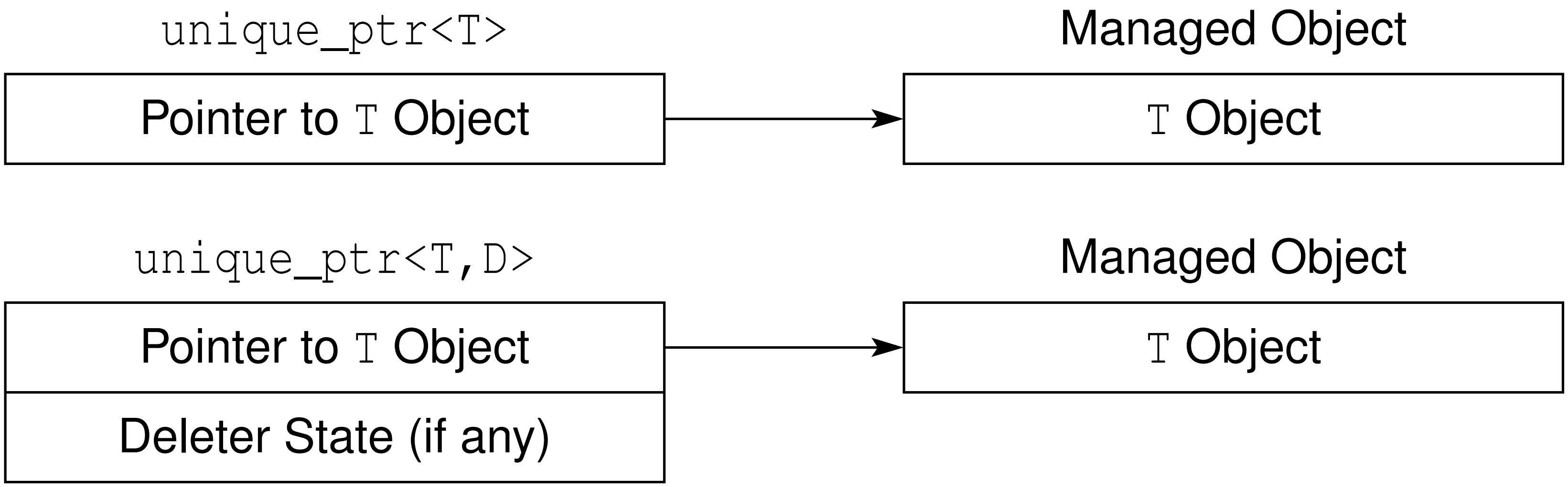
The `std::unique_ptr` Template Class

- `std::unique_ptr` is ***smart pointer*** that retains ***exclusive*** ownership of object through pointer
- declaration:

```
template <class T, class Deleter = std::default_delete<T>>
  class unique_ptr;
```

- `T` is type of object to be managed (i.e., owned object)
- `Deleter` is callable entity used to delete owned object
- also correctly handles array types via partial specialization (e.g., `T` could be array of **`char`**)
- owned object destroyed when `unique_ptr` object goes out of scope
- no two `unique_ptr` objects can own same object
- `unique_ptr` object is ***movable***; move operation transfers ownership
- `unique_ptr` object is ***not copyable***, as copying would create additional owners
- `std::make_unique` template function often used to create `unique_ptr` objects (for exception-safety reasons)

The `std::unique_ptr` Template Class (Continued)



- reasonable implementation would have zero memory cost for deleter state in case of:
  - default deleter
  - deleter of functor/closure type with no state
- if no memory cost for deleter state, `unique_ptr` has same memory cost as raw pointer

`std::unique_ptr` Member Functions

Construction, Destruction, and Assignment

Member Name	Description
constructor	constructs new `unique_ptr`
destructor	destroys managed object (if any)
`operator=`	assigns `unique_ptr`

Modifiers

Member Name	Description
`release`	returns pointer to managed object and releases ownership
`reset`	replaces managed object
`swap`	swaps managed objects

Observers

Member Name	Description
`get`	returns pointer to managed object
`get_deleter`	returns deleter used for destruction of managed object
`operator bool`	check if there is associated managed object

`std::unique_ptr` Member Functions (Continued)

Dereferencing/Subscripting

Member Name	Description
`operator*`	dereferences pointer to managed object
`operator->`	dereferences pointer to managed object
`operator[]`	provides indexed access to managed array

std::unique_ptr Example

```cpp
#include <memory>
#include <cassert>

void func() {
    auto p1(std::make_unique<int>(42));
    assert(*p1 == 42);

    // std::unique_ptr<int> p3(p1); // ERROR: not copyable
    // p3 = p1; // ERROR: not copyable

    std::unique_ptr<int> p2(std::move(p1)); // OK: movable
      // Transfers ownership from p1 to p2, invalidating p1.
    assert(p1.get() == nullptr && *p2 == 42);

    p1 = std::move(p2); // OK: movable
      // Transfers ownership from p2 to p1, invalidating p2.
    assert(p2.get() == nullptr && *p1 == 42);

    p1.reset();
      // Invalidates p1.
    assert(p1.get() == nullptr);
}
```

std::unique_ptr Example

```
#include <memory>
#include <cassert>

int main() {
    auto p0 = std::make_unique<int>(0);
    assert(*p0 == 0);
    int* r0 = p0.get();
    auto p1 = std::make_unique<int>(1);
    assert(*p1 == 1);
    auto r1 = p1.get();
    p0.swap(p1);
    assert(p0.get() == r1 && p1.get() == r0);
    p1.swap(p0);
    assert(p0.get() == r0 && p1.get() == r1);
    p1.reset();
    assert(p1.get() == nullptr);
    assert(!p1);
    int* ip = p1.release();
    assert(!p1);
    // ... Do not throw exceptions here.
    delete ip;
    p1.reset(new int(42));
    assert(*p1 == 42);
}
```

Example: std::unique_ptr with Custom Deleter

```
#include <memory>
#include <iostream>
#include <cstring>
#include <cstdlib>

using up = std::unique_ptr<char[], void(*)(char*)>;

char *allocate(std::size_t n) {
    return static_cast<char*>(std::malloc(n));
}

void deallocate(char* p) {
    std::cout << "deallocate called\n";
    std::free(p);
}

up string_duplicate(const char *s) {
    std::size_t len = std::strlen(s);
    up result(allocate(len + 1), deallocate);
    std::strcpy(result.get(), s);
    return result;
}

int main() {
    auto p = string_duplicate("Hello, World!");
    std::cout << p.get() << '\n';
}
```

Section 3.2.3

The `std::shared_ptr` Class Template

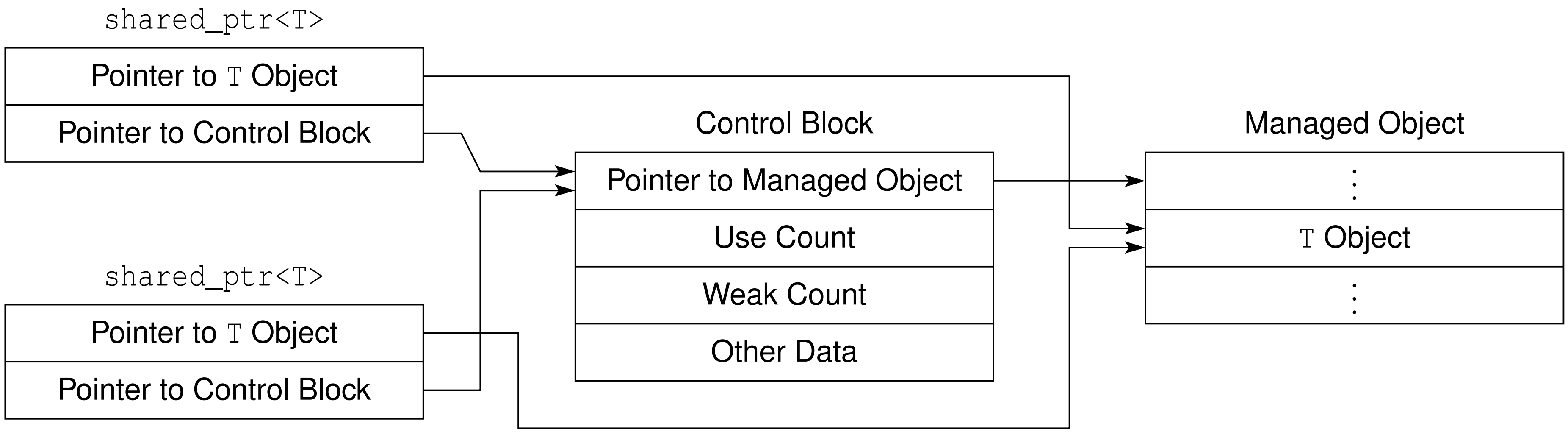
The `std::shared_ptr` Template Class

- `std::shared_ptr` is ***smart pointer*** that retains ***shared*** ownership of object through pointer
- declaration:

```
template <class T> class shared_ptr;
```

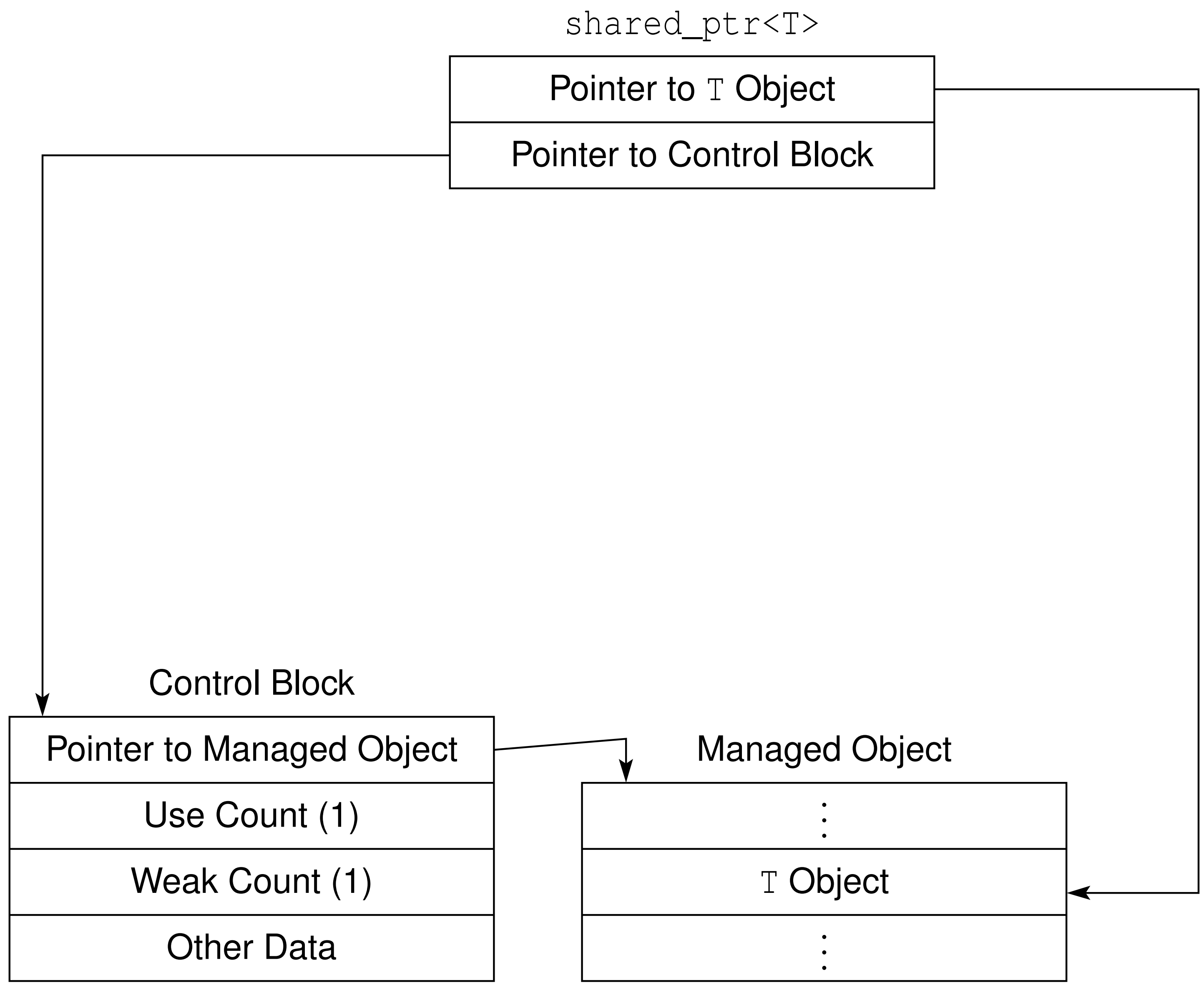
- `T` is type of object to be managed (i.e., owned object)
- multiple `shared_ptr` objects may own same object
- owned object is deleted when last remaining owning `shared_ptr` object is destroyed, assigned another pointer via assignment, or reset via `reset`
- `shared_ptr` object is ***movable***, where move transfers ownership
- `shared_ptr` object is ***copyable***, where copy creates additional owner
- thread safety guaranteed for `shared_ptr` object itself but not owned object
- `std::make_shared` (and `std::allocate_shared`) often used to create `shared_ptr` objects (for both efficiency and exception-safety reasons)
- `shared_ptr` has more overhead than `unique_ptr` so `unique_ptr` should be preferred unless shared ownership required

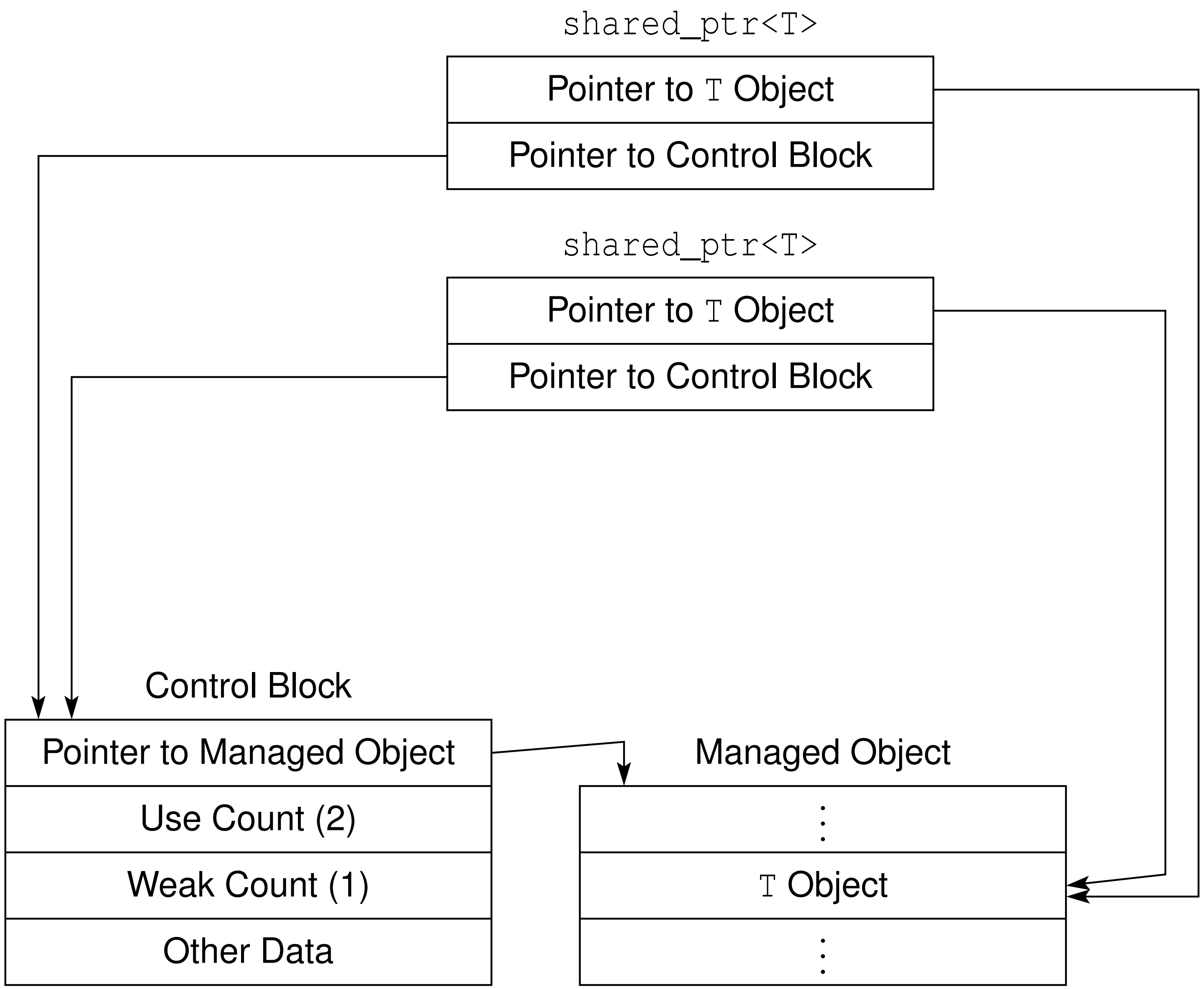
The `std::shared_ptr` Template Class (Continued)

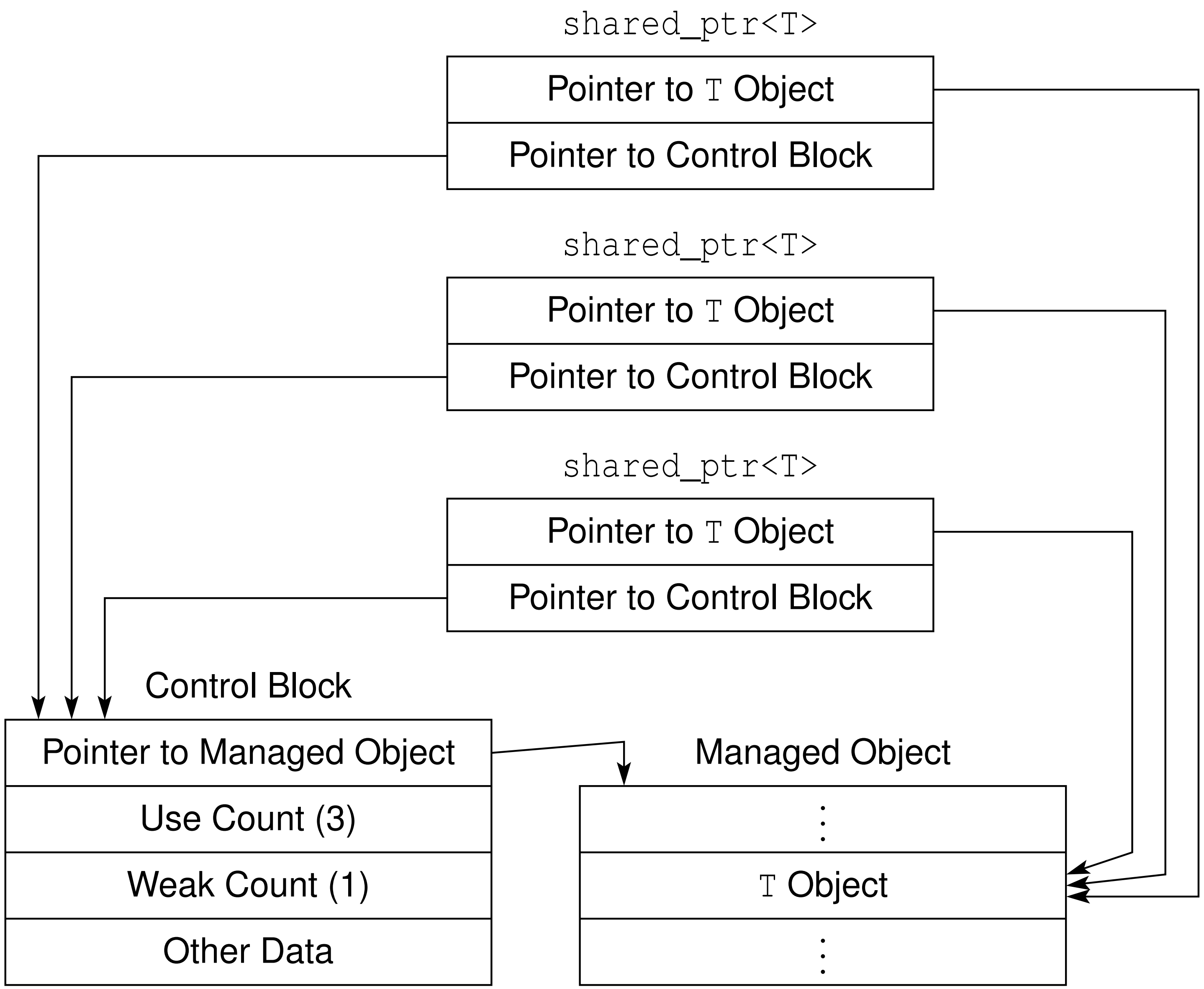


- each `shared_ptr<T>` object contains:
  - pointer to object of type `T` (i.e., managed object or subobject thereof)
  - pointer to control block
- control block contains:
  - pointer to managed object (for deletion)
  - use count: number of `shared_ptr` instances pointing to object
  - weak count: to be discussed later
  - other data (i.e., deleter and allocator)
- managed object is deleted when use count reaches zero
- `make_shared` can allow memory for control block and managed object to be allocated together in single memory allocation

`std::shared_ptr` Reference Counting Example




shared_ptr<T>
Pointer to T Object
Pointer to Control Block
shared_ptr<T>
Pointer to T Object
Pointer to Control Block
Control Block
Pointer to Managed Object
Use Count (2)
Weak Count (1)
Other Data
Managed Object
⋮
T Object
⋮


shared_ptr<T>
Pointer to T Object
Pointer to Control Block
shared_ptr<T>
Pointer to T Object
Pointer to Control Block
shared_ptr<T>
Pointer to T Object
Pointer to Control Block
Control Block
Pointer to Managed Object
Use Count (3)
Weak Count (1)
Other Data
Managed Object
⋮
T Object
⋮

`std::shared_ptr` Member Functions

Construction, Destruction, and Assignment

Member Name	Description
constructor	constructs new `shared_ptr`
destructor	destroys managed object if no other references to it remain
`operator=`	assign `shared_ptr`

Modifiers

Member Name	Description
`reset`	replaced managed object
`swap`	swaps managed objects

std::shared_ptr Member Functions (Continued)

Observers

Member Name	Description
get	returns pointer to managed object
use_count	returns number of shared_ptr objects referring to same managed object
unique	checks if managed object is managed only by current shared_ptr instance
operator bool	checks if there is associated managed object
owner_before	provide owner-based ordering of shared pointers

Dereferencing/Subscripting

Member Name	Description
operator*	dereference pointer to managed object
operator->	dereference pointer to managed object
operator[]	provide indexed access to managed array

std::shared_ptr Example

```
#include <memory>
#include <cassert>

int main() {
    auto p1(std::make_shared<int>(0));
    assert(*p1 == 0 && p1.use_count() == 1 && p1.unique());

    std::shared_ptr<int> p2(p1);
    assert(*p2 == 0 && p2.use_count() == 2 && !p2.unique());

    *p2 = 42;
    assert(*p1 == 42);

    p2.reset();
    assert(!p2);
    assert(*p1 == 42 && p1.use_count() == 1 && p1.unique());

    int* ip = p1.get();
    assert(*ip == 42);;

    ip = p2.get();
    assert(ip == nullptr);
}
```

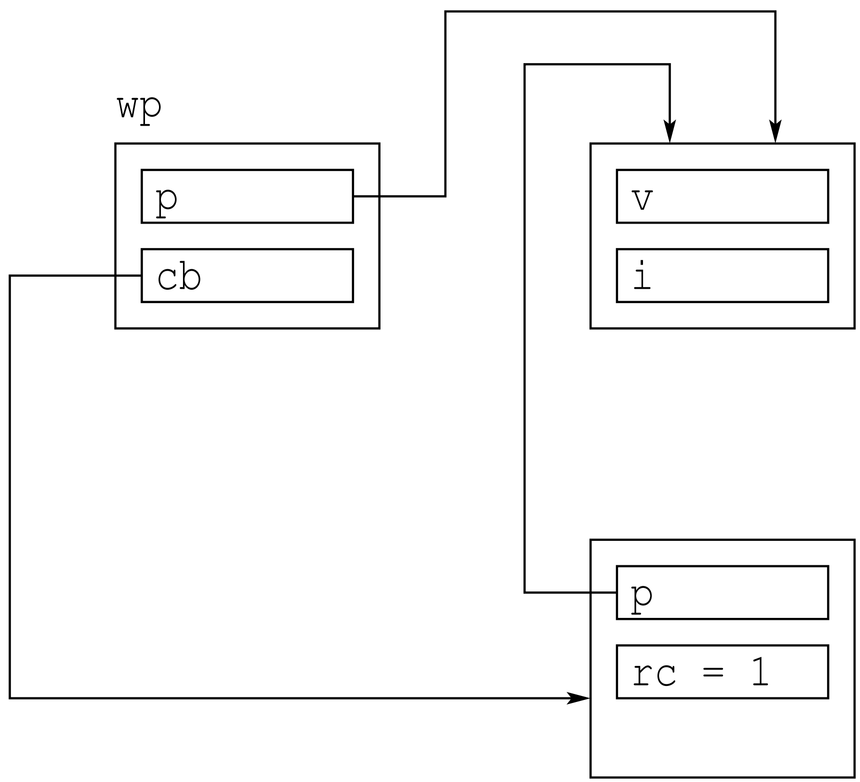
std::shared_ptr and const

```
#include <memory>
#include <iostream>
#include <string>

int main() {
    std::shared_ptr<std::string> s =
      std::make_shared<std::string>("hello");

    std::shared_ptr<const std::string> cs = s;

    *s = "goodbye";

    // *cs = "bonjour"; // ERROR: const

    std::cout << *cs.get() << '\n';
}
```

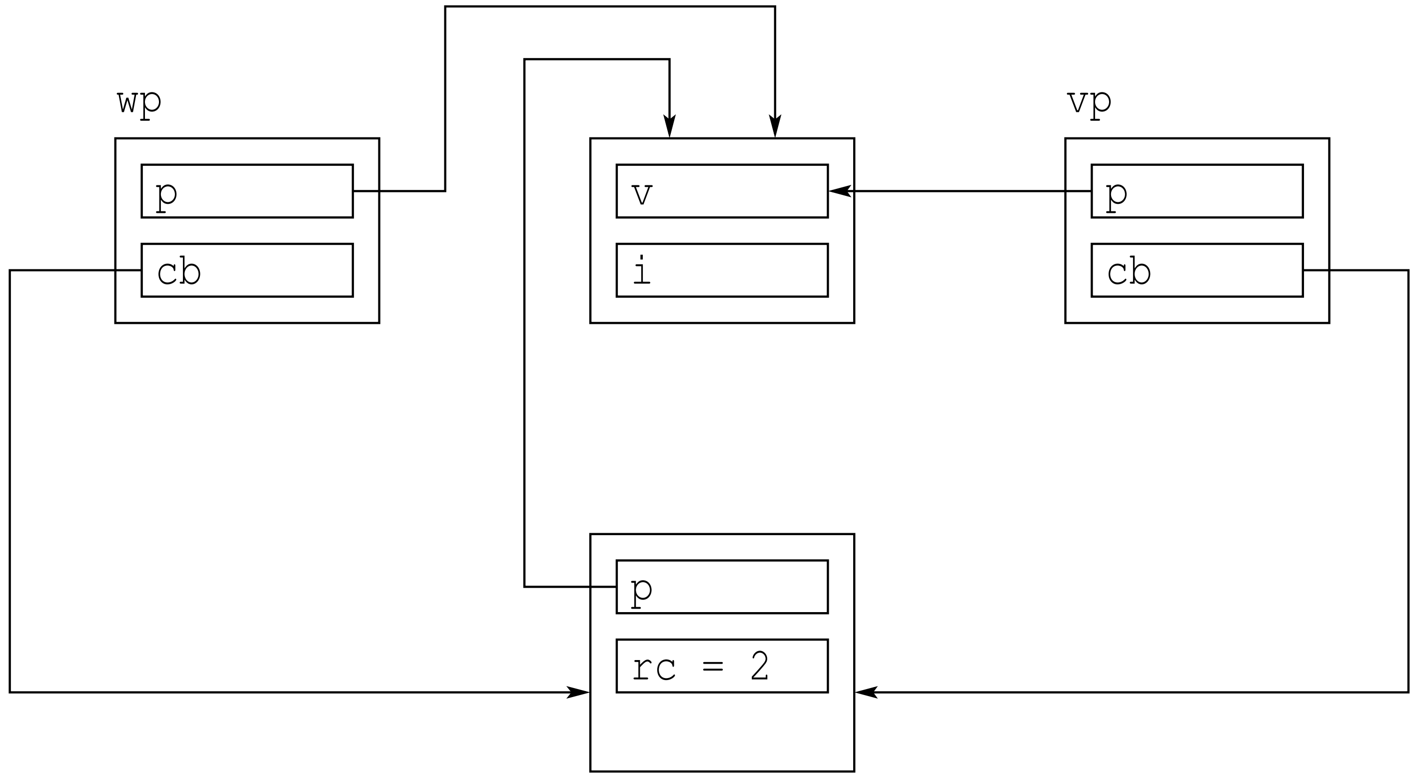
Example: Shared Pointer to Subobject of Managed Object

```
#include <memory>
#include <vector>
#include <cassert>
#include <iostream>

struct Widget {
    Widget(int i_, const std::vector<int>& v_) :
      i(i_), v(v_) {}
    ~Widget() {std::cout << "destructor called\n";}
    int i;
    std::vector<int> v;
};

int main() {
    auto wp(std::make_shared<Widget>(42,
      std::vector<int>{1, 2, 3}));
    assert(wp.use_count() == 1);
    assert(wp->i == 42 && wp->v.size() == 3);
    std::shared_ptr<std::vector<int>> vp(wp, &wp->v);
    assert(wp.use_count() == 2 && vp.use_count() == 2);
    assert(vp->size() == 3);
    wp = nullptr; // equivalently: wp.reset();
      // managed Widget object not destroyed
    assert(vp.use_count() == 1 && vp->size() == 3);
    vp = nullptr; // equivalently: vp.reset();
      // managed Widget object destroyed
    // ...
}
```

Example: Shared Pointer to Subobject of Managed Object (Continued 1)



Example: Shared Pointer to Subobject of Managed Object (Continued 2)



The std::enable_shared_from_this Class Template

- may want class object to be able to generate additional shared_ptr instances referring to itself
- requires object to have access to information in its associated shared_ptr control block
- access to such information obtained through use of std::enable_shared_from_this class template
- declaration:

```
template <class T> class enable_shared_from_this;
```

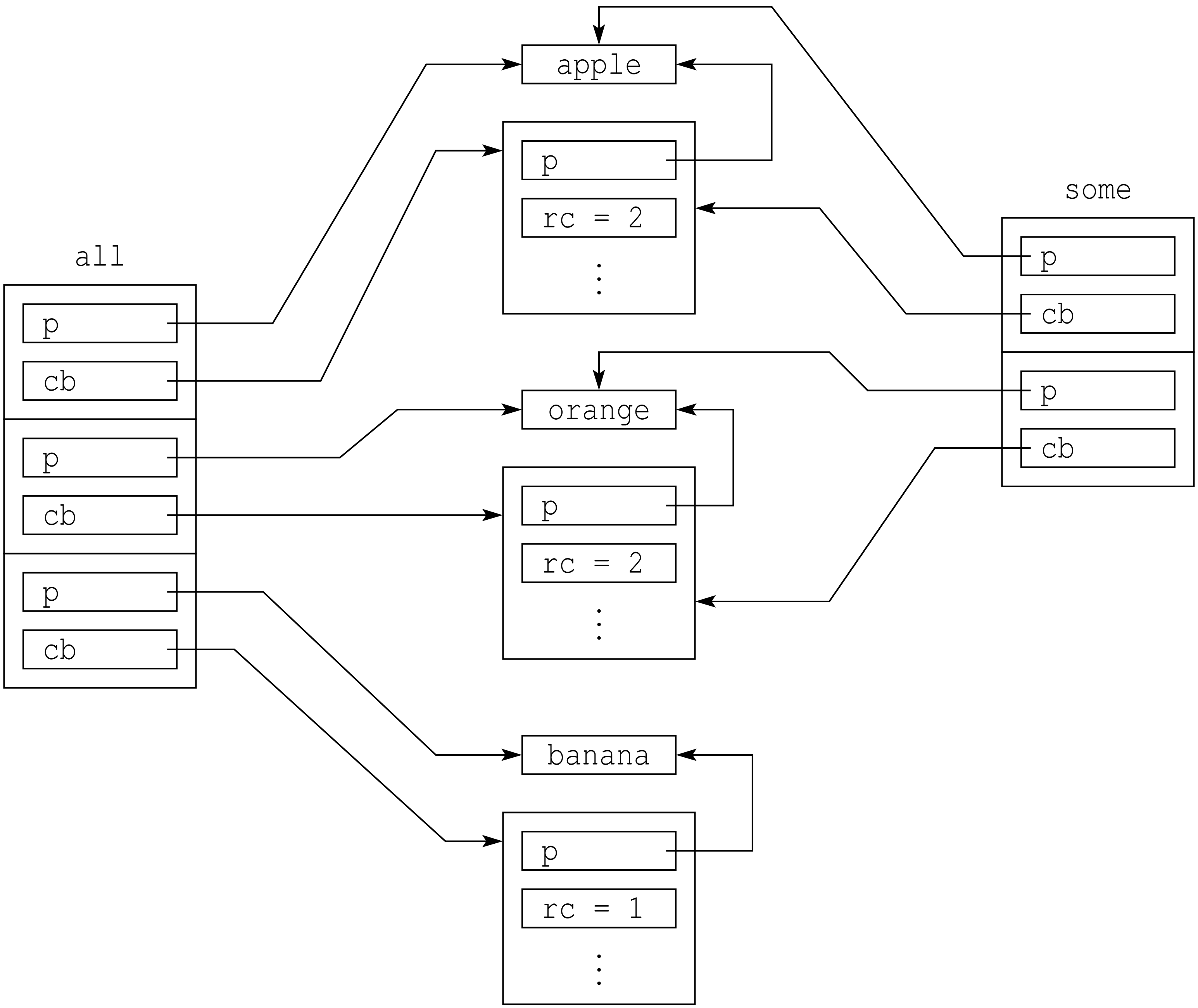
- T is type of object being managed by shared_ptr
- class can inherit from enable_shared_from_this to inherit shared_from_this member functions that can be used to obtain shared_ptr instance pointing to *this
- shared_from_this is overloaded to provide both const and non-const versions

```
#include <memory>
#include <cassert>

// Aside: This is an example of the CRTP.
class Widget : public std::enable_shared_from_this<Widget>
{
public:
    std::shared_ptr<Widget> getSharedPtr() {
        return shared_from_this();
    }
    std::shared_ptr<const Widget> getSharedPtr() const {
        return shared_from_this();
    }
    // ...
};

int main() {
    std::shared_ptr<Widget> a(new Widget);
    std::shared_ptr<Widget> b = a->getSharedPtr();
    assert(b == a);
    std::shared_ptr<const Widget> c = a->getSharedPtr();
    assert(c == a);
}
```

Example: std::shared_ptr

```
#include <memory>
#include <array>
#include <string>
#include <iostream>

using namespace std::literals;

int main() {
    std::array<std::shared_ptr<const std::string>, 3> all = {
        std::make_shared<const std::string>("apple"s),
        std::make_shared<const std::string>("orange"s),
        std::make_shared<const std::string>("banana"s)
    };
    std::array<std::shared_ptr<const std::string>, 2> some =
      {all[0], all[1]};

    for (auto& x : all) {
        std::cout << *x << ' ' << x.use_count() << '\n';
    }
}

/* output:
apple 2
orange 2
banana 1
*/
```

Example: `std::shared_ptr` (Continued)



Section 3.2.4

The `std::weak_ptr` Class Template

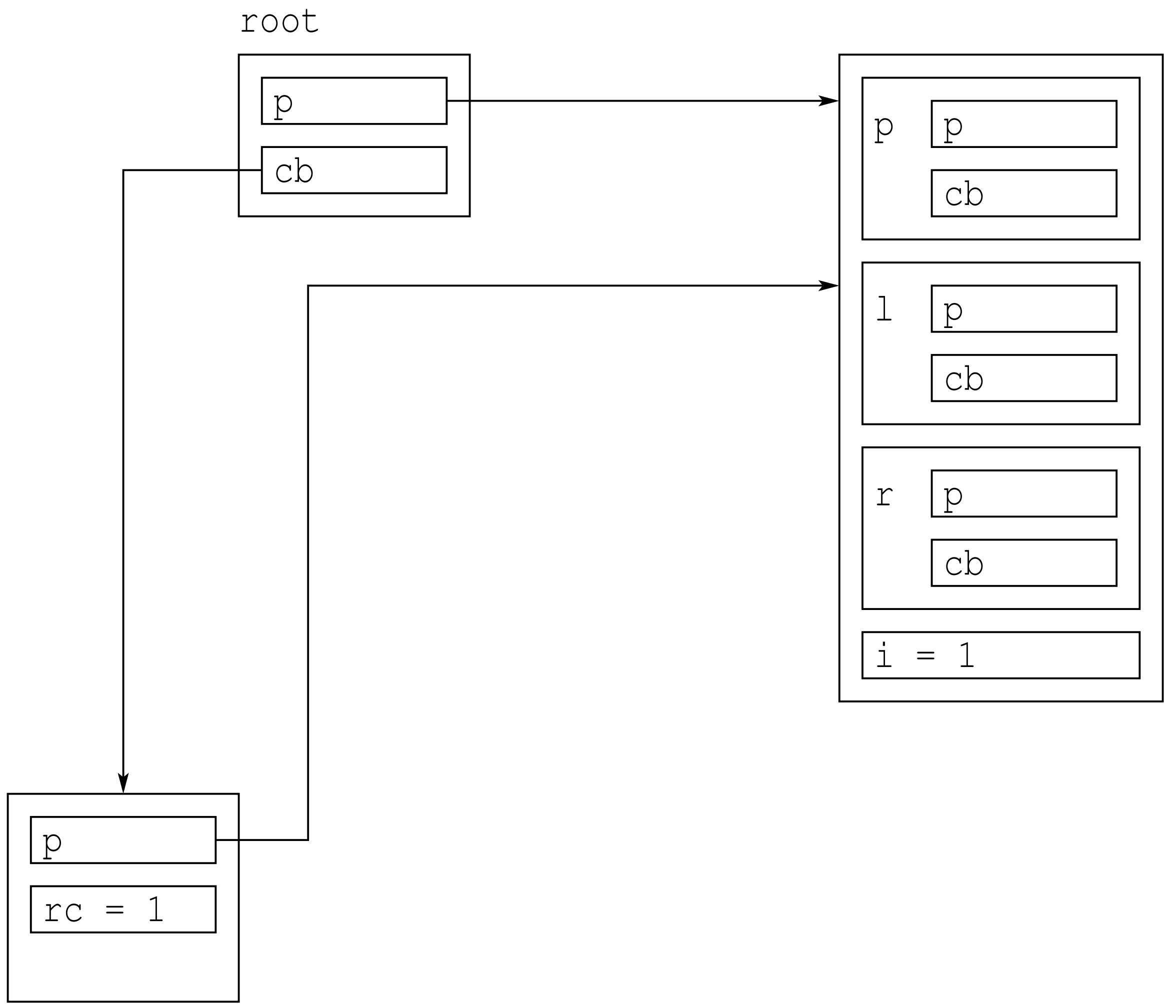
std::shared_ptr and Circular References

- reference counting nature of std::shared_ptr causes it to leak memory in case of circular references
- such cycles should be broken with std::weak_ptr (to be discussed shortly)

Circular Reference Example

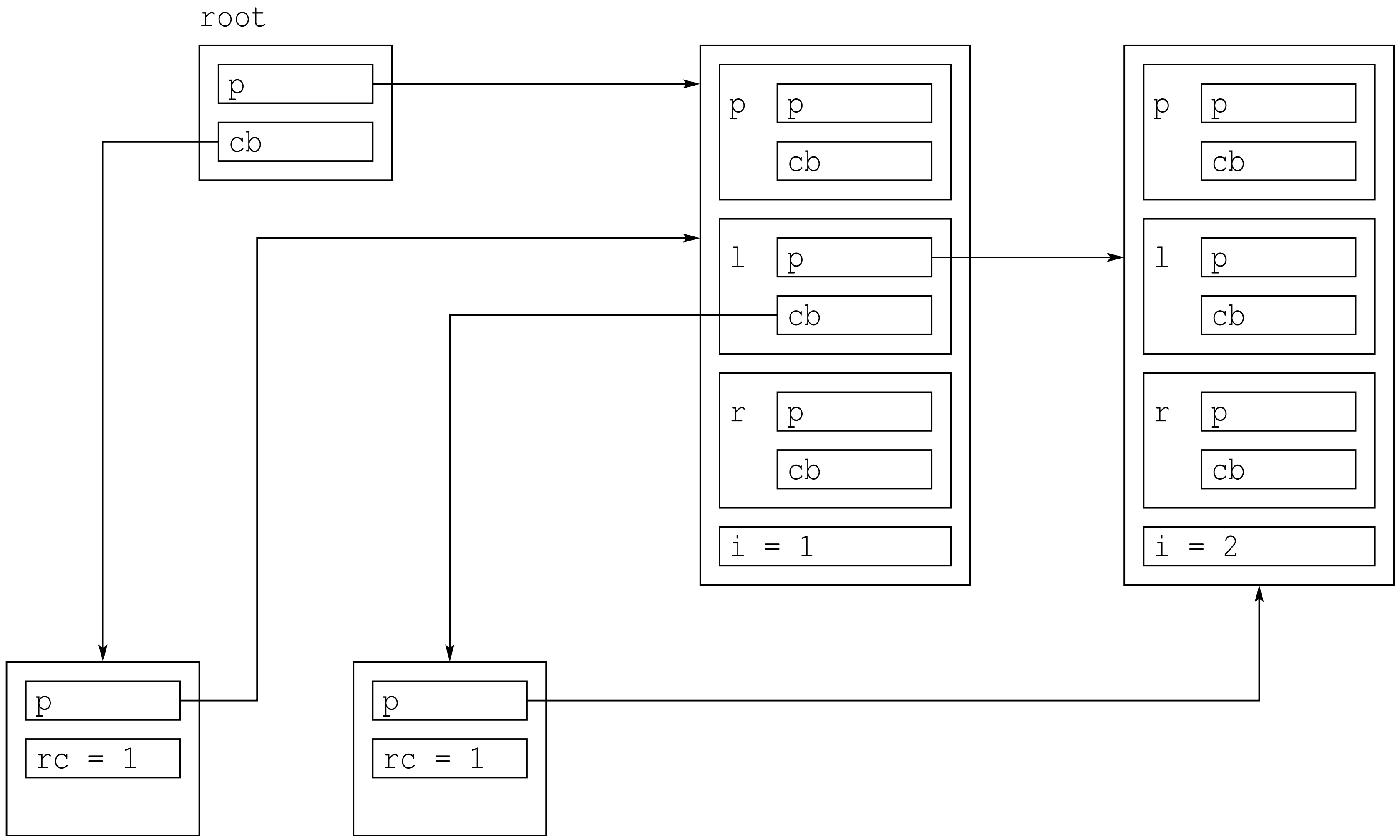
```
#include <memory>
#include <iostream>
#include <cassert>

struct Node {
    Node(int id_) : id(id_) {}
    ~Node() {std::cout << "destroying node " << id << '\n';}
    std::shared_ptr<Node> parent;
    std::shared_ptr<Node> left;
    std::shared_ptr<Node> right;
    int id;
};

void func() {
    std::shared_ptr<Node> root(std::make_shared<Node>(1));
    assert(root.use_count() == 1);
    root->left = std::make_shared<Node>(2);
    assert(root.use_count() == 1 &&
      root->left.use_count() == 1);
    root->left->parent = root;
    assert(root.use_count() == 2 &&
      root->left.use_count() == 1);
    // When root is destroyed, the reference count for each
    // of the managed Node objects does not reach zero, and
    // no Node object is destroyed.
    // Node::~Node is not called here
}
```

Circular Reference Example (Continued 1)



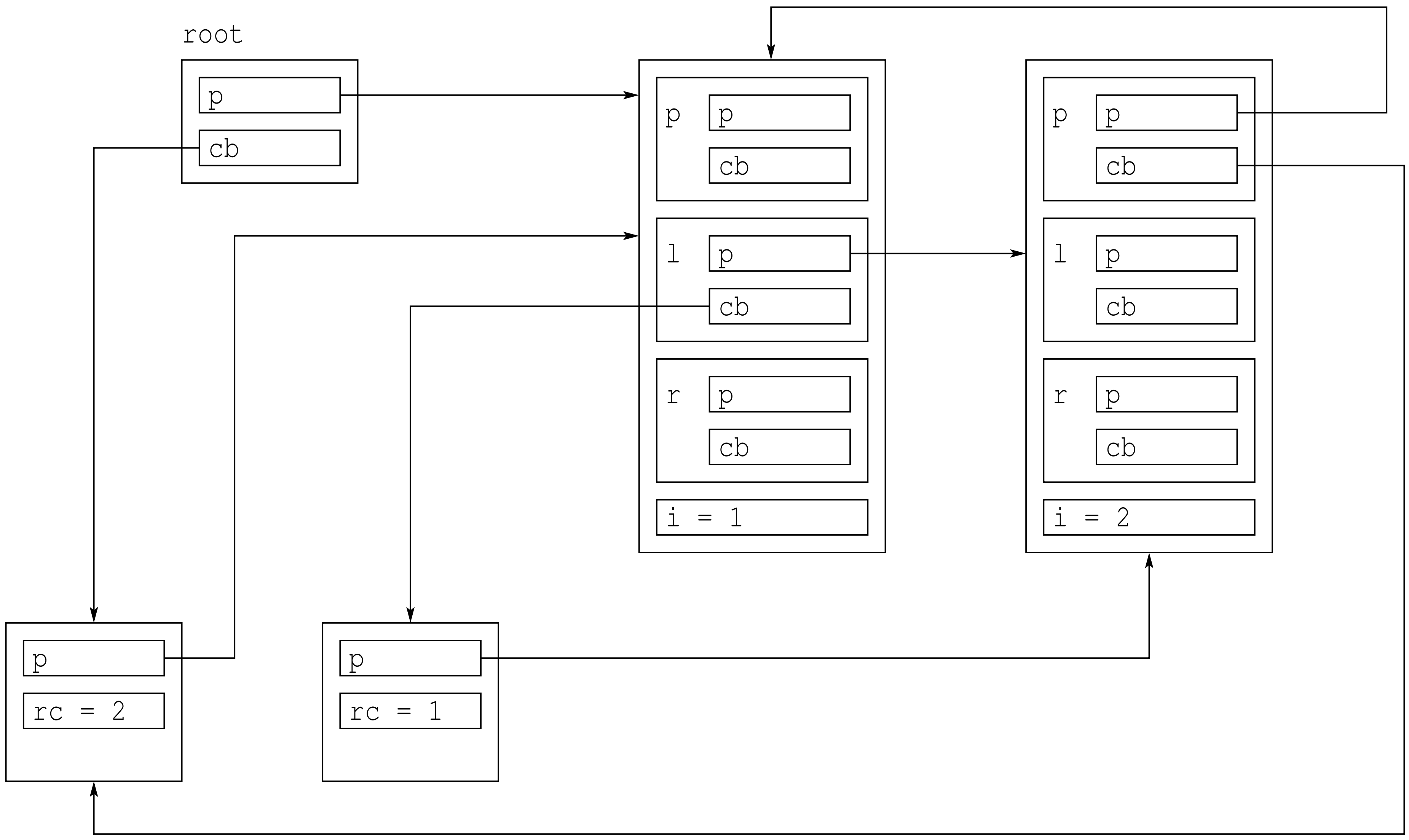
- create new node, referenced by `root`

Circular Reference Example (Continued 2)



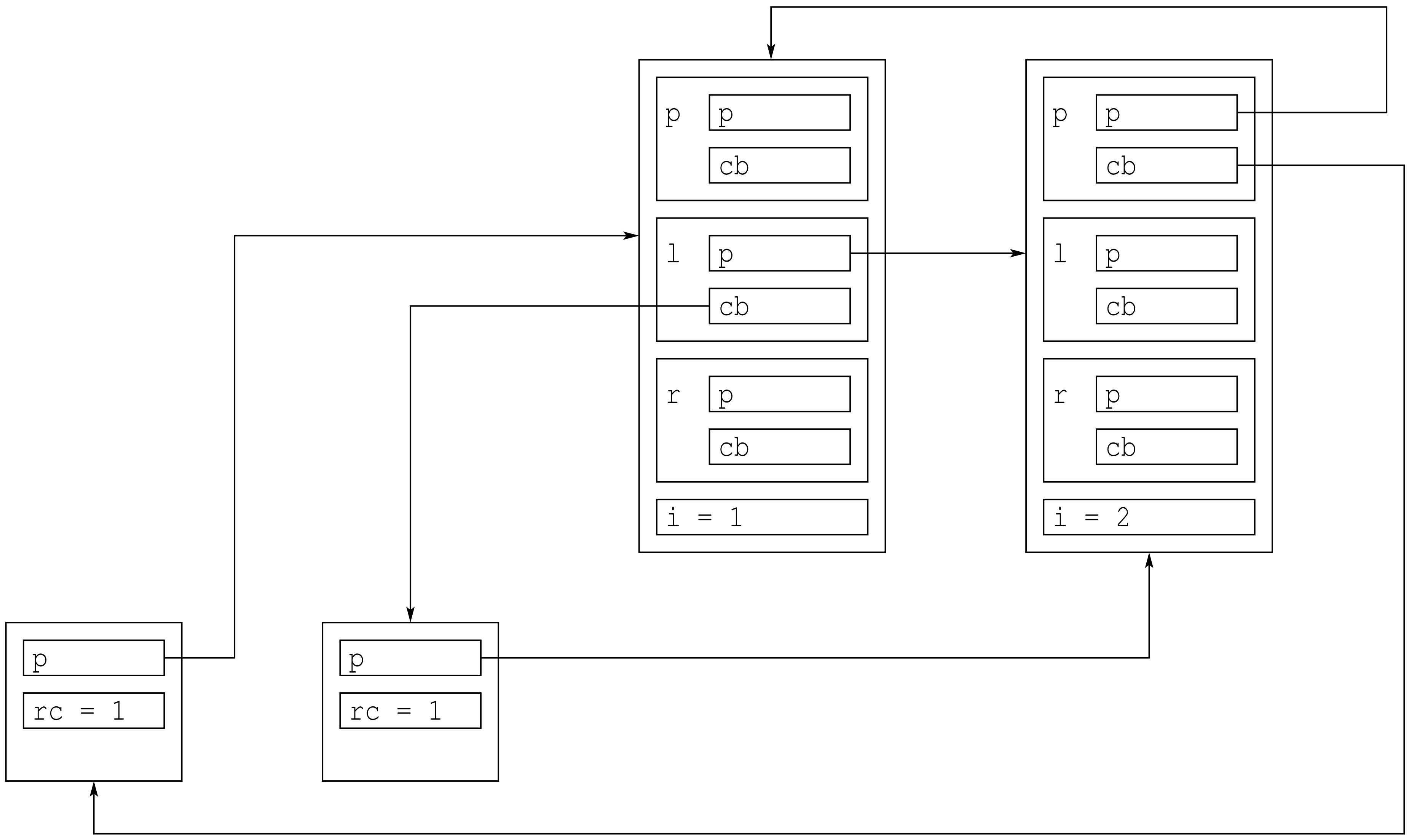
- create new node, making it left child of root node (parent link not set)

Circular Reference Example (Continued 3)



- set parent link for left child of root node

Circular Reference Example (Continued 4)



- after destroying `root`, neither node destroyed

The `std::weak_ptr` Template Class

- `std::weak_ptr` is ***smart pointer*** that holds ***non-owning*** (i.e., “weak”) reference to object managed by `std::shared_ptr`
- `weak_ptr` must be converted to `std::shared_ptr` in order to access referenced object
- declaration:

```
template <class T> class weak_ptr;
```

- `T` is type of referenced object
- `weak_ptr` object is ***movable*** and ***copyable***
- `std::weak_ptr` is used to break circular references with `std::shared_ptr`

`std::weak_ptr` Member Functions

Construction, Destruction, and Assignment

Member Name	Description
constructor	constructs new `weak_ptr`
destructor	destroys `weak_ptr`
`operator=`	assign `weak_ptr`

Modifiers

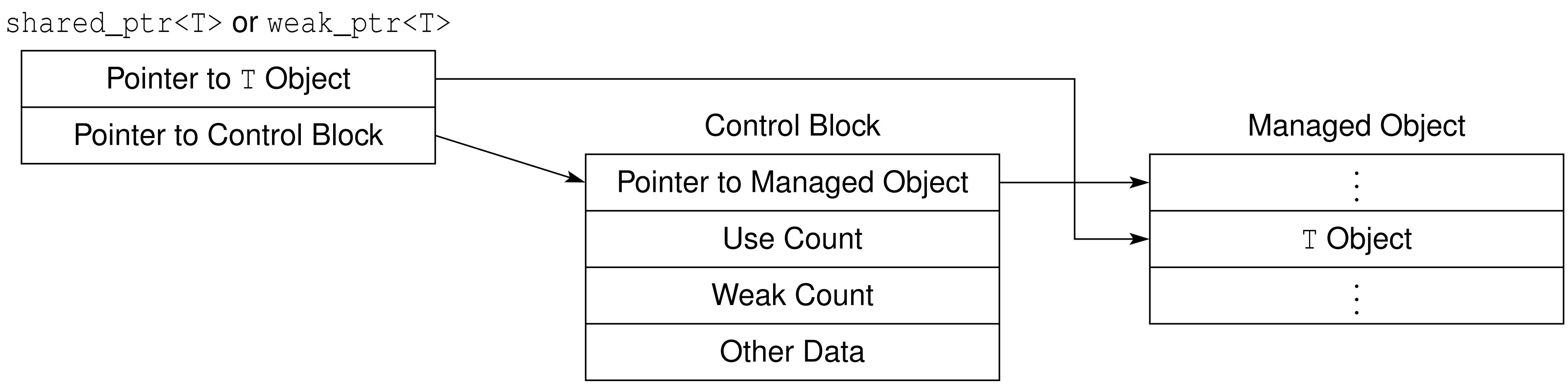
Member Name	Description
`reset`	replaced managed object
`swap`	swaps managed objects

std::weak_ptr Member Functions (Continued)

Observers

Member Name	Description
use_count	returns number of shared_ptr objects referring to same managed object
expired	checks if referenced object was already deleted
lock	creates shared_ptr that manages referenced object
owner_before	provide owner-based ordering of weak pointers

Typical shared_ptr/weak_ptr Implementation



- each shared_ptr<T> and weak_ptr<T> object contains:
  - pointer to object of type T (i.e., managed object or subobject thereof)
  - pointer to control block
- control block contains:
  - pointer to managed object (for deletion)
  - use count: number of shared_ptr instances pointing to object
  - weak count: number of weak_ptr instances pointing to object, plus one if use count is nonzero
  - other data (i.e., deleter and allocator)
- managed object is deleted when use count reaches zero
- control block is deleted when weak count reaches zero (which implies use count is also zero)

Typical shared_ptr/weak_ptr Implementation (Continued)

- shared_ptr destructor pseudocode:

```
decrement use count and if it reaches zero {
    delete managed object
    decrement weak count and if it reaches zero {
        delete control block
    }
}
```

- weak_ptr destructor pseudocode:

```
decrement weak count and if it reaches zero {
    delete control block
}
```

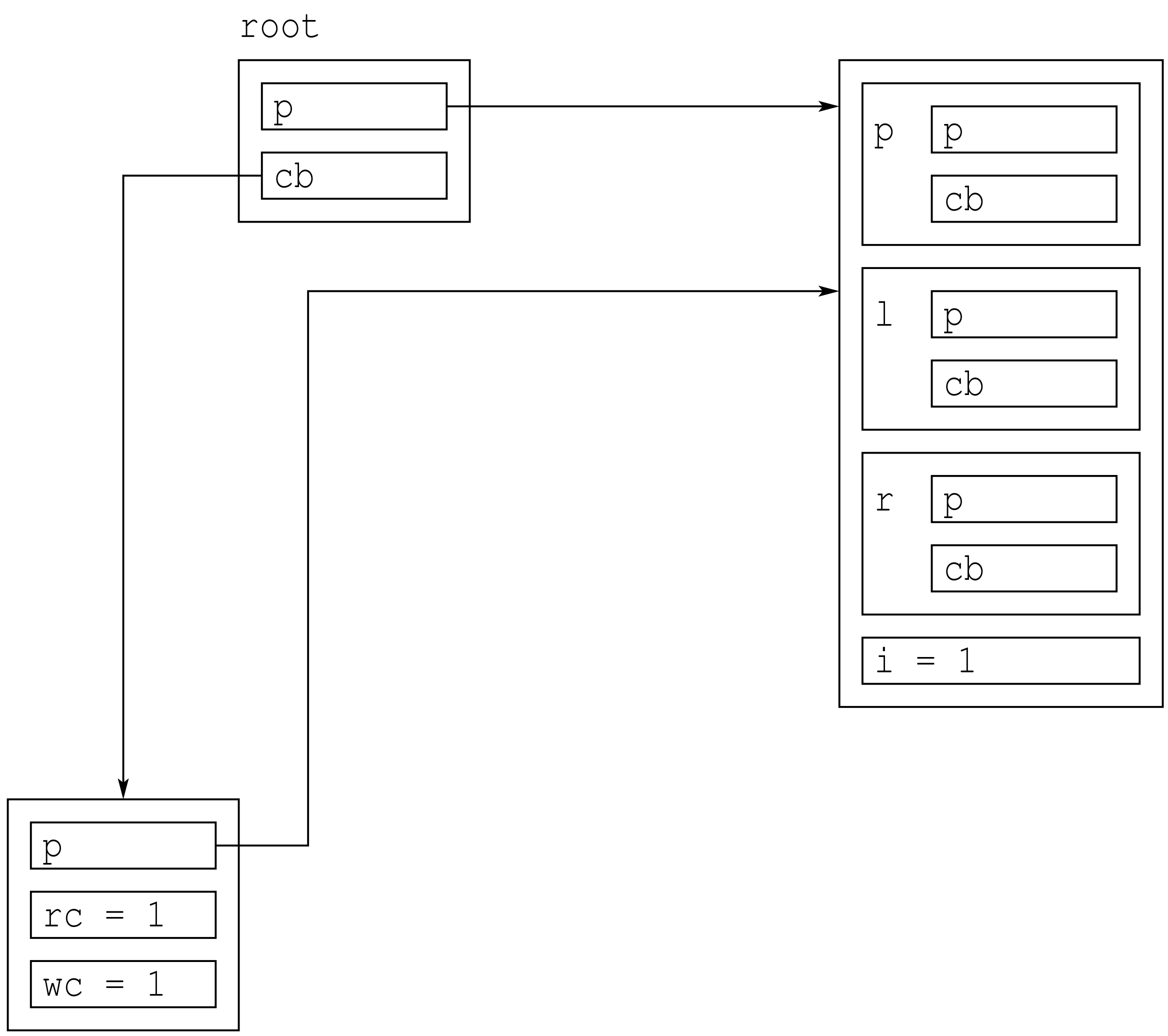
- must be thread safe

std::weak_ptr Example

```cpp
#include <memory>
#include <iostream>
#include <cassert>

void func(std::weak_ptr<int> wp) {
    auto sp = wp.lock();
    if (sp) {
        std::cout << *sp << '\n';
    } else {
        std::cout << "expired\n";
    }
}

int main() {
    std::weak_ptr<int> wp;
    {
        auto sp = std::make_shared<int>(42);
        wp = sp;
        assert(wp.use_count() == 1 && wp.expired() == false);
        func(wp);
        // When sp destroyed, wp becomes expired.
    }
    assert(wp.use_count() == 0 && wp.expired() == true);
    func(wp);
}
```

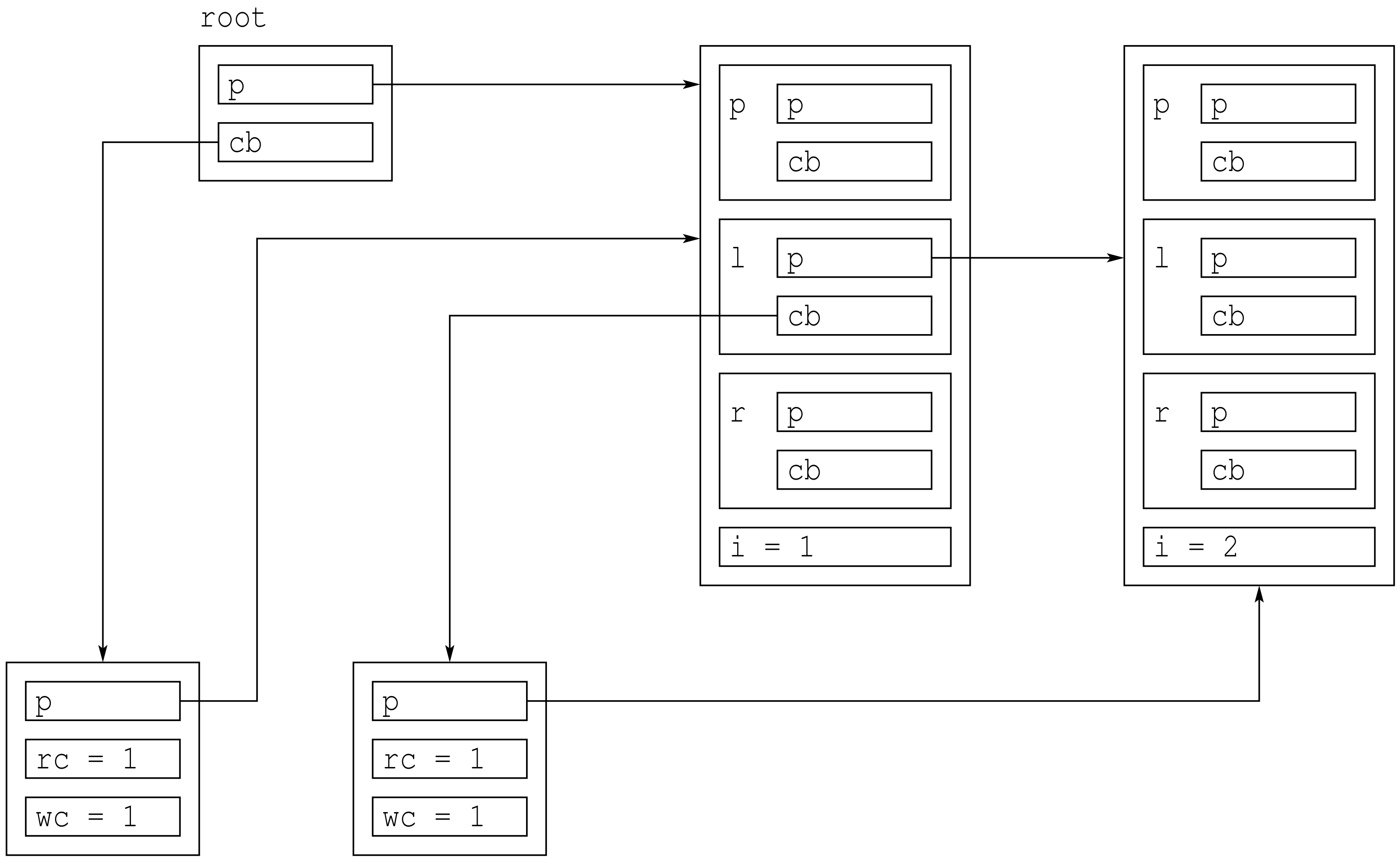
Avoiding Circular References With std::weak_ptr

```
#include <memory>
#include <iostream>
#include <cassert>

struct Node {
    Node(int id_) : id(id_) {}
    ~Node() {std::cout << "destroying node " << id << '\n';}
    std::weak_ptr<Node> parent;
    std::shared_ptr<Node> left;
    std::shared_ptr<Node> right;
    int id;
};

void func() {
    std::shared_ptr<Node> root(std::make_shared<Node>(1));
    assert(root.use_count() == 1);
    root->left = std::make_shared<Node>(2);
    assert(root.use_count() == 1 &&
      root->left.use_count() == 1);
    root->left->parent = root;
    assert(root.use_count() == 1 &&
      root->left.use_count() == 1);
    // The reference count for each of the managed Node
    // objects reaches zero, and these objects are
    // destroyed.
    // Node::~Node is called twice here
}
```

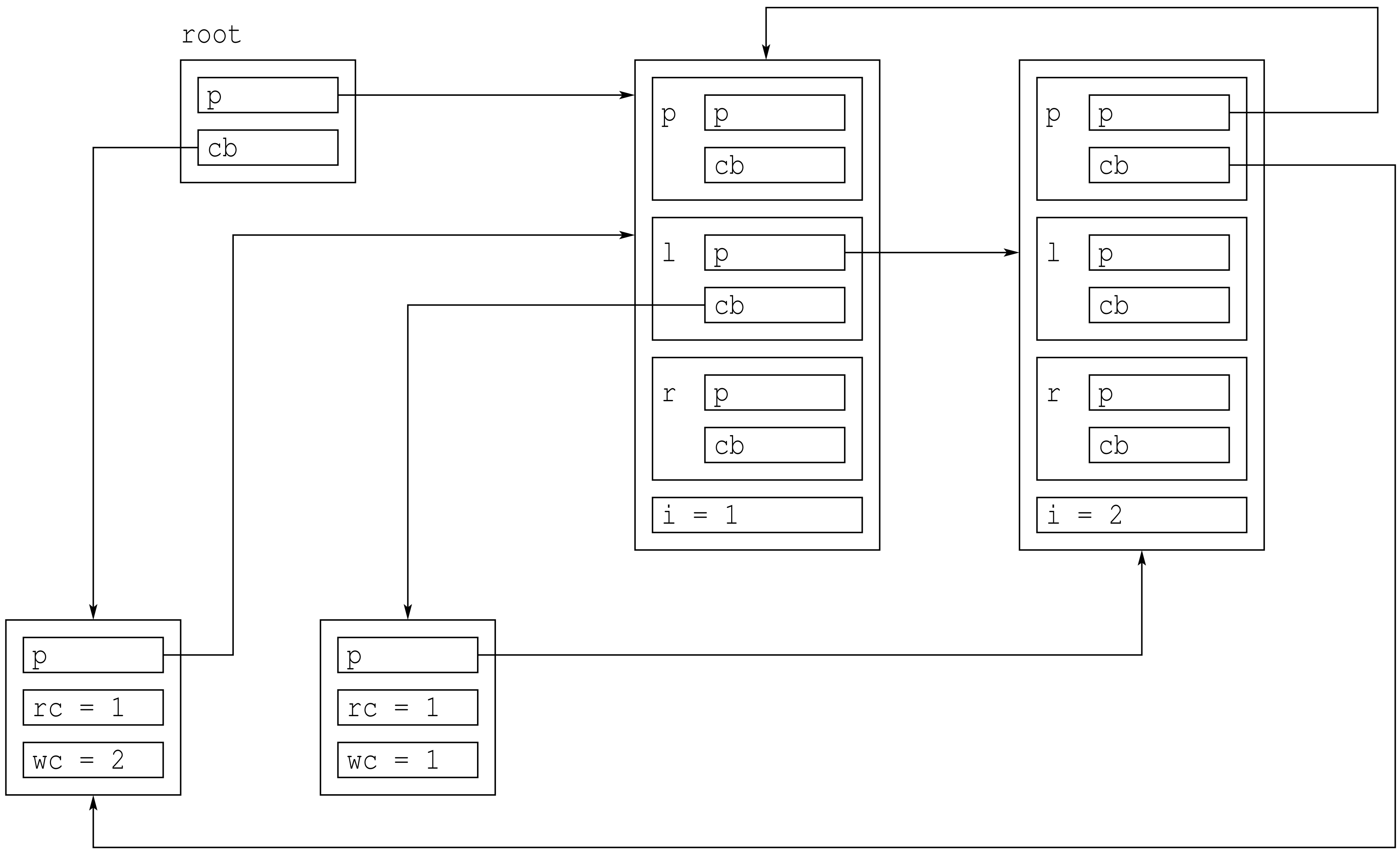


- created new node, referenced by `root`

Avoiding Circular References Example (Continued 2)

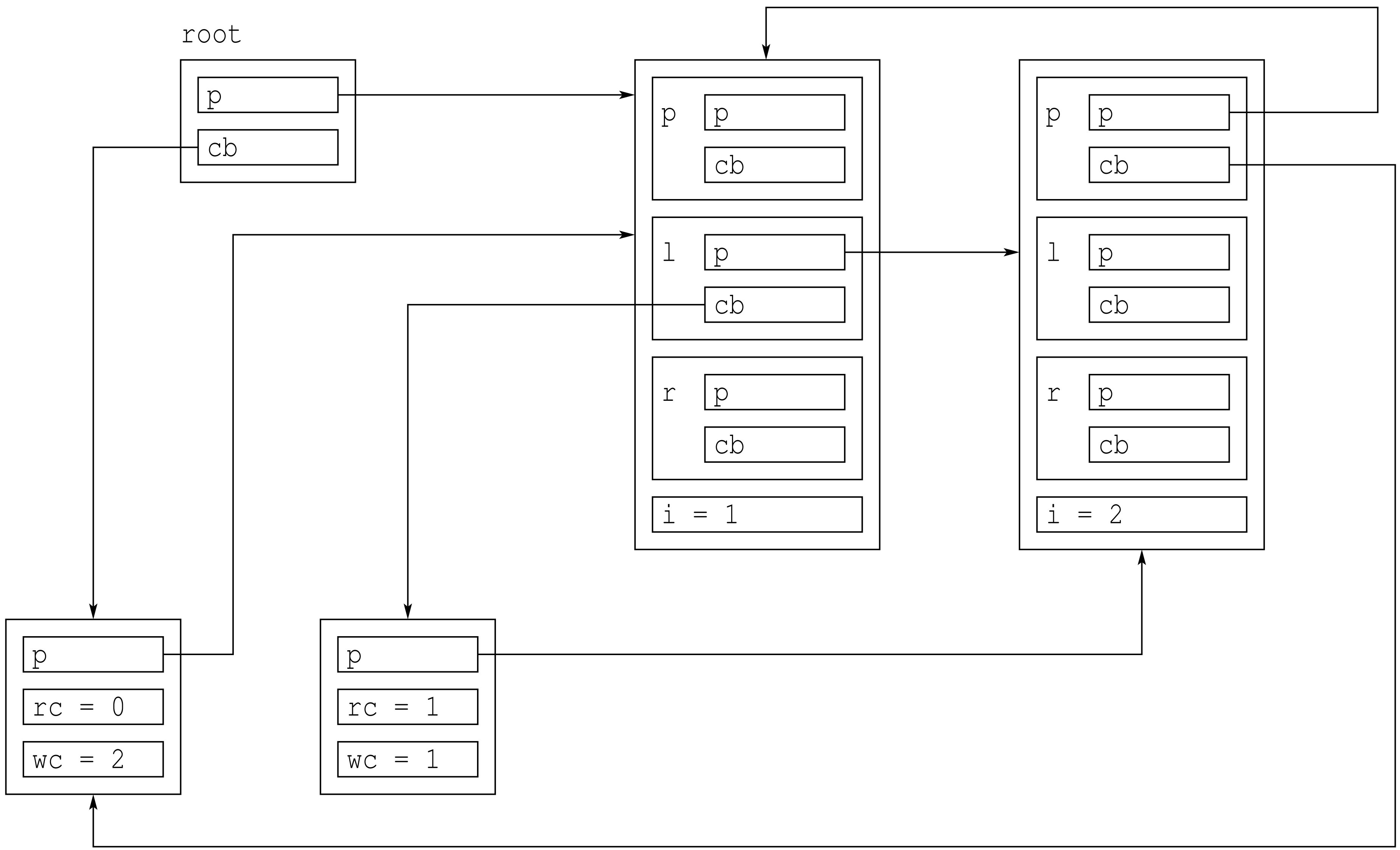


- created new node, making it left child of root node (parent link not set)



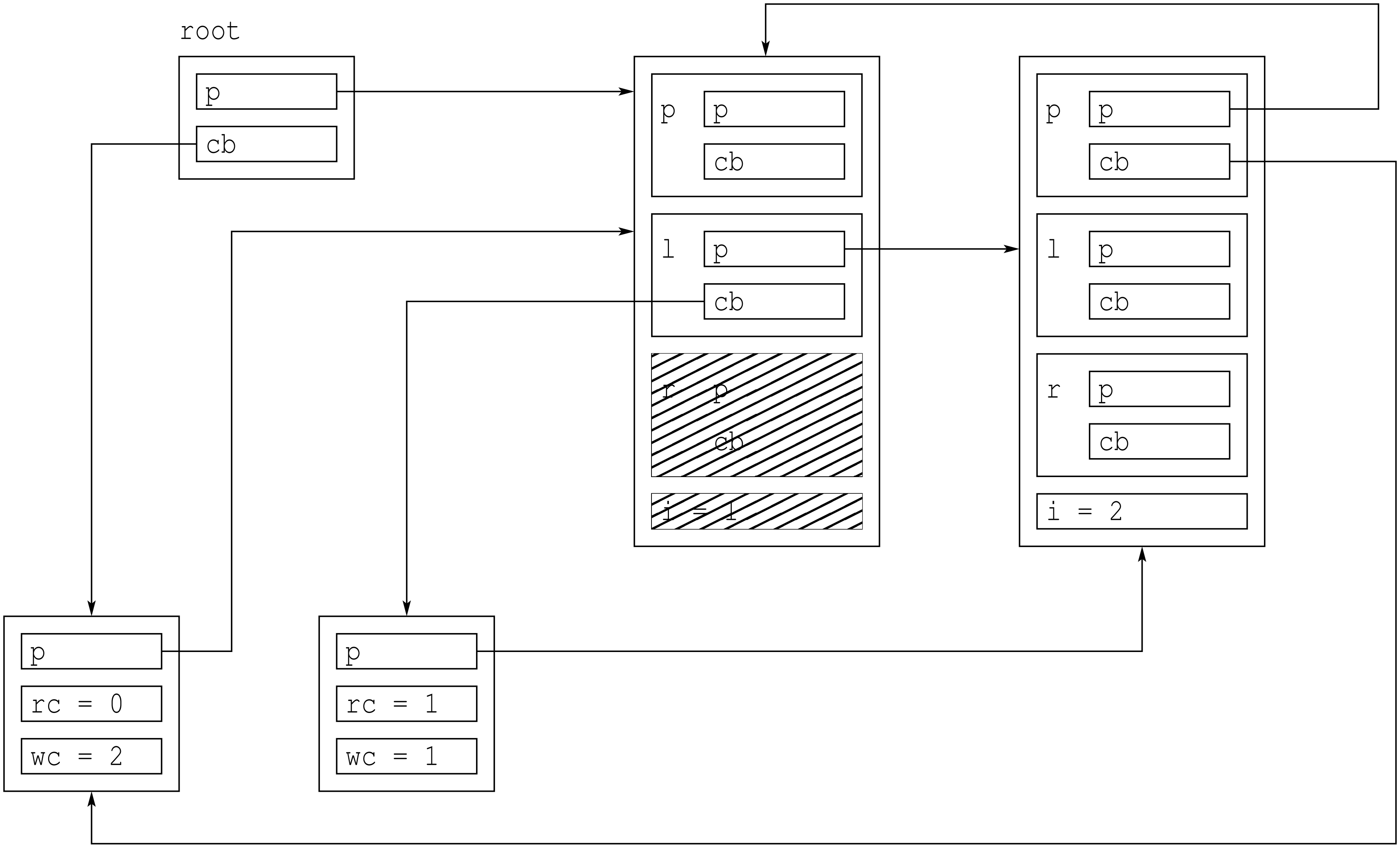
- set parent link (which is `weak_ptr`) for left child of root node

Avoiding Circular References Example (Continued 4)



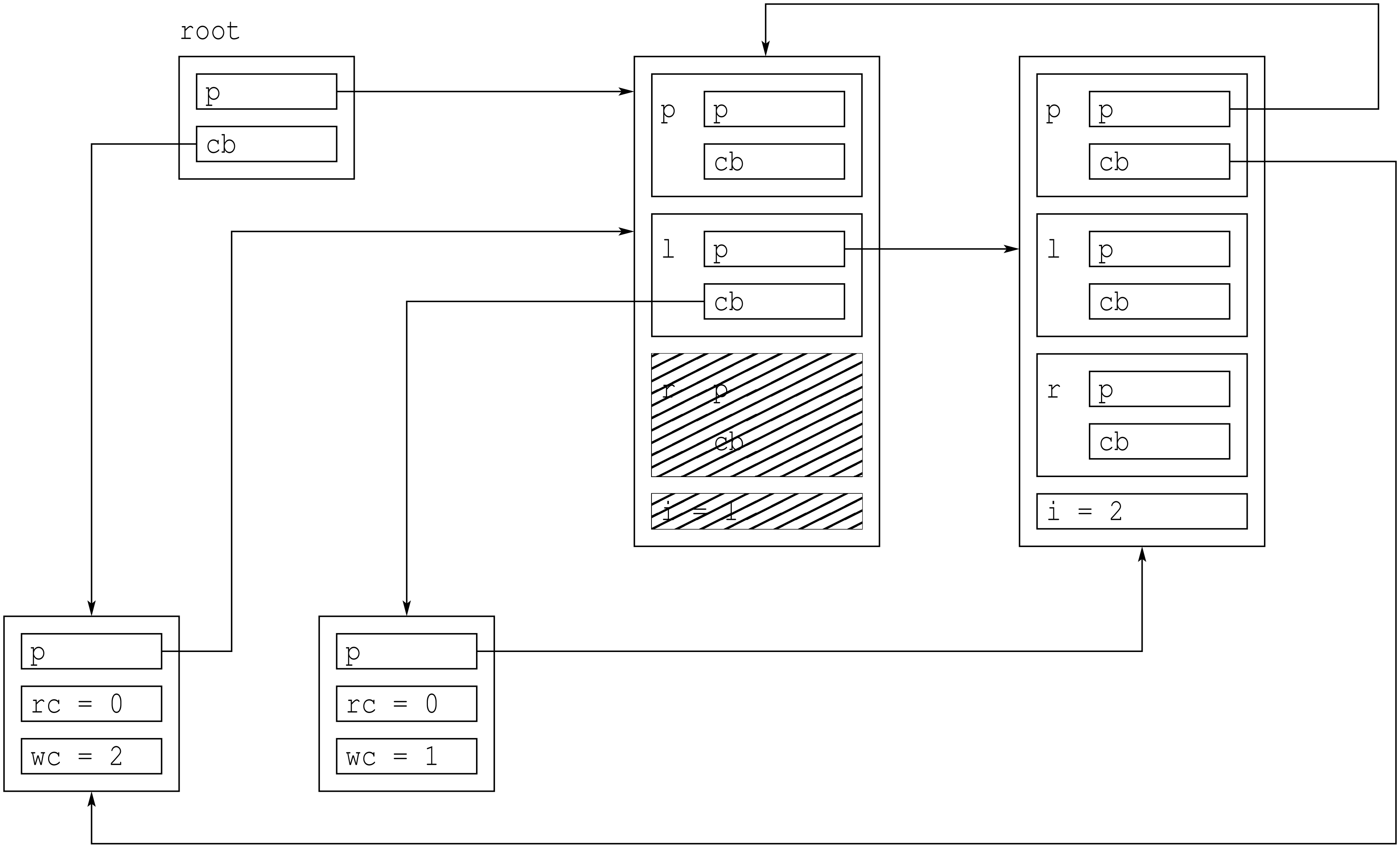
- started to destroy `root`; decremented use count, which reaches zero

Avoiding Circular References Example (Continued 5)



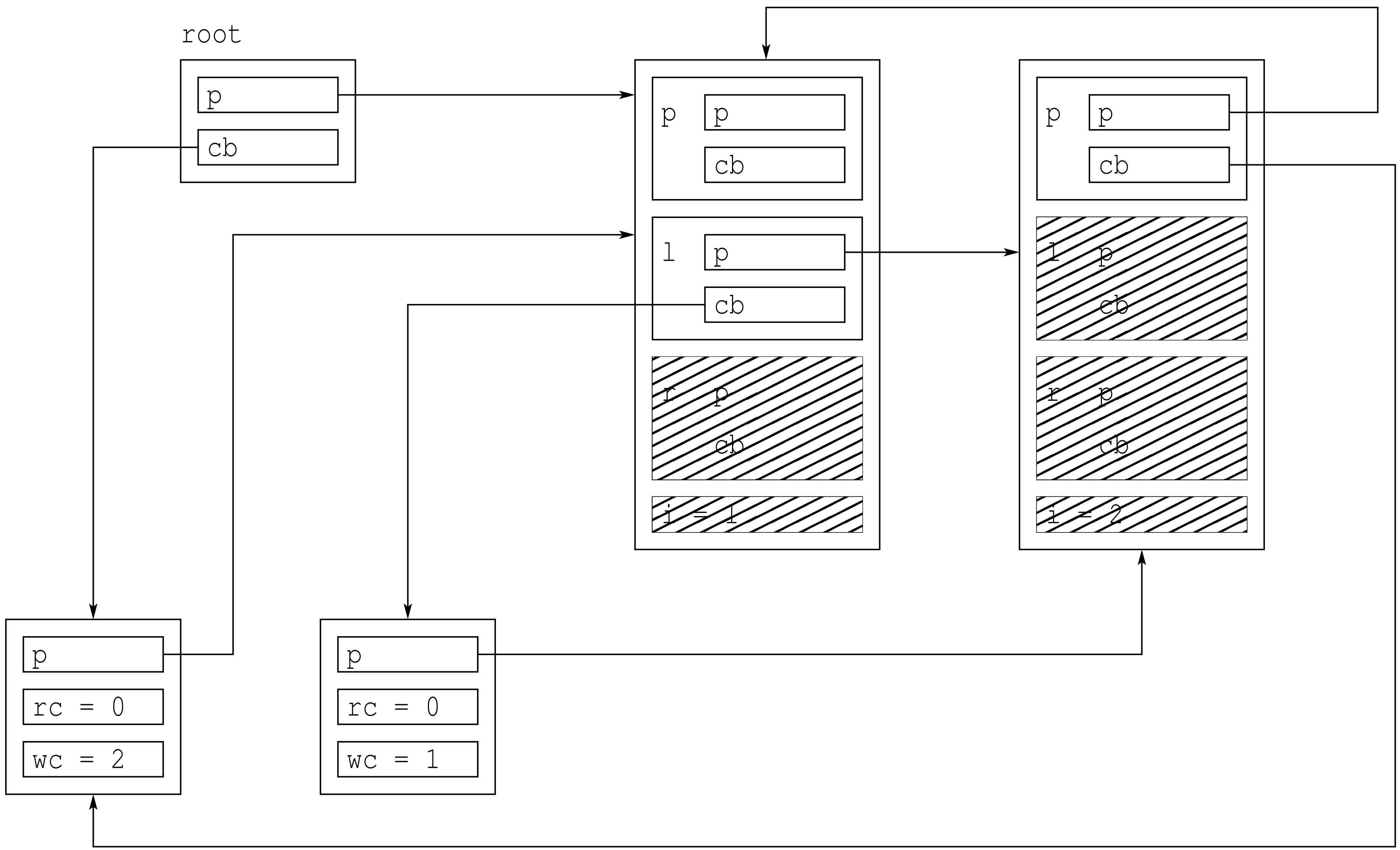
- started to destroy root node; `r` has been destroyed; about to destroy `l`

Avoiding Circular References Example (Continued 6)

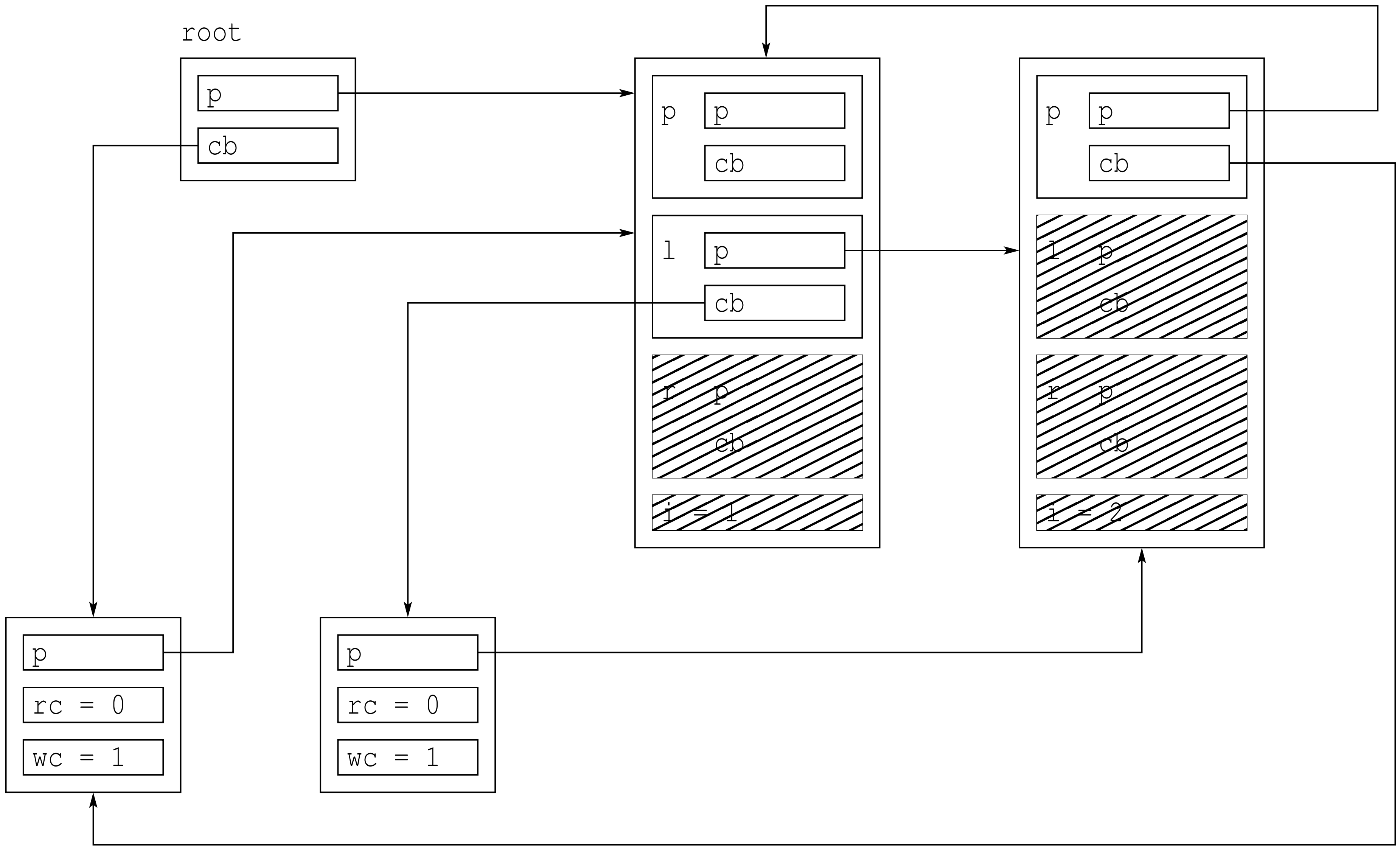


- started to destroy `l` (in root node); decremented use count, which reaches zero

Avoiding Circular References Example (Continued 7)

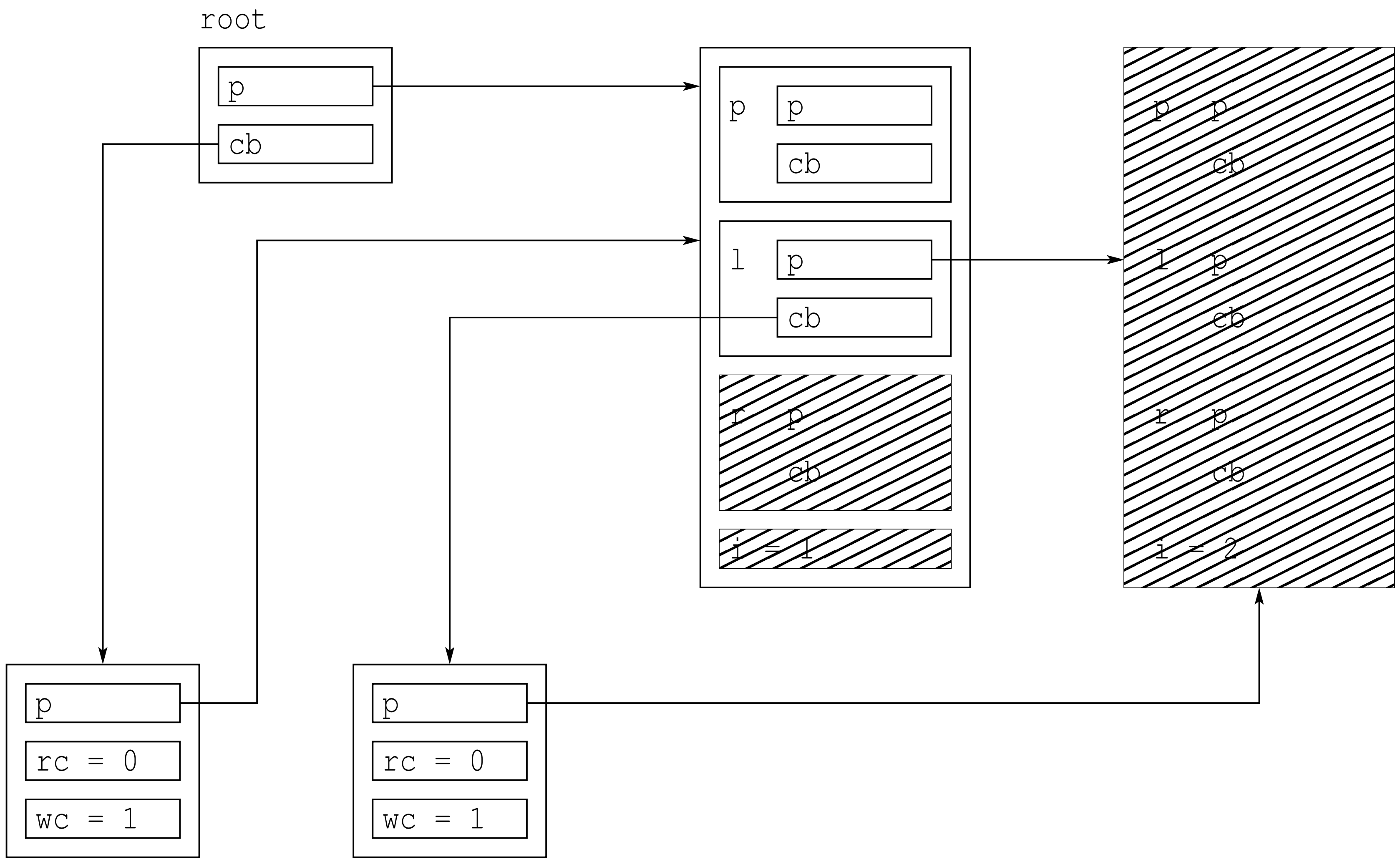


- started to destroy left node



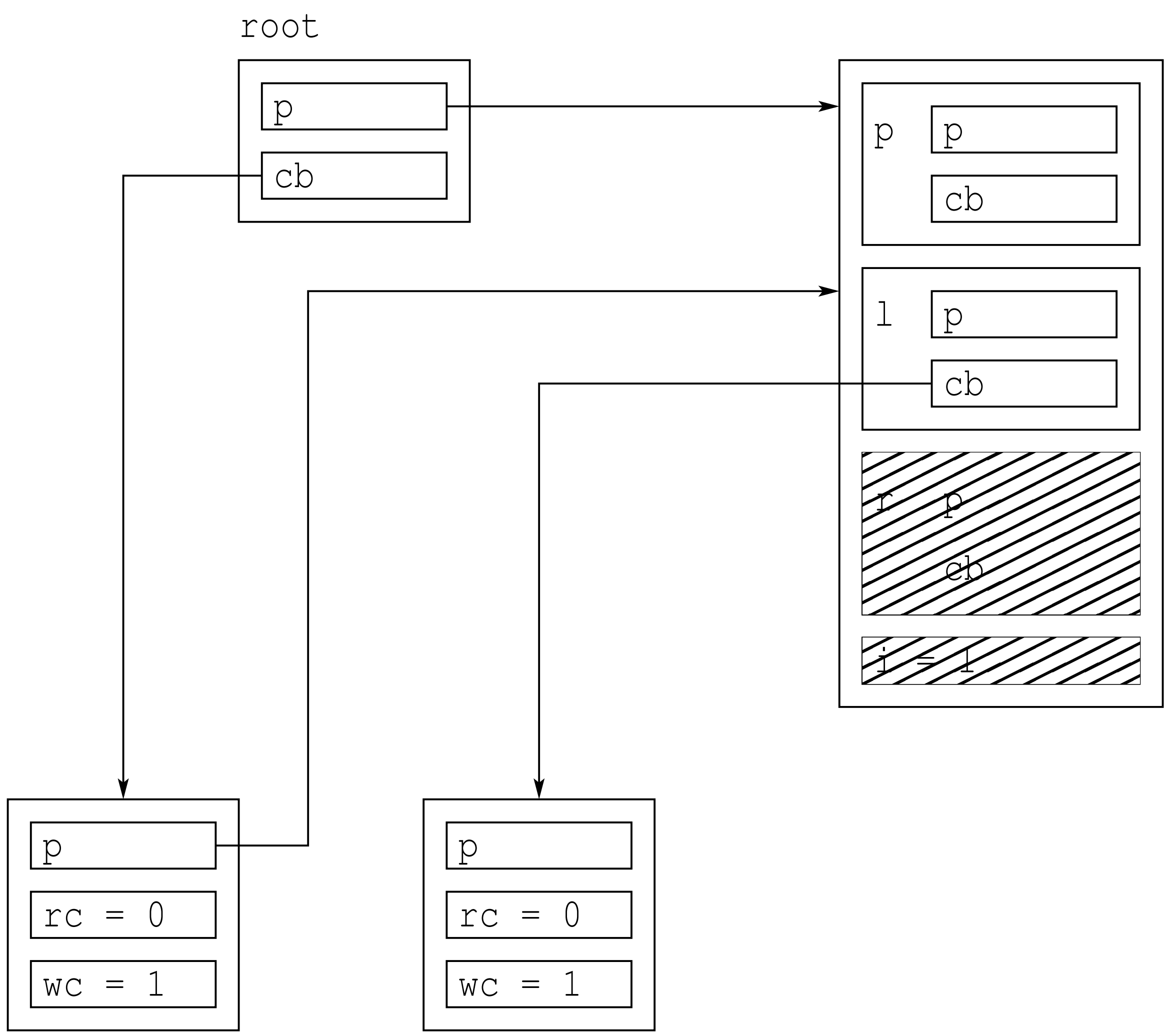
- started to destroy `p` in left node; decremented weak count (which is not yet zero)

Avoiding Circular References Example (Continued 9)

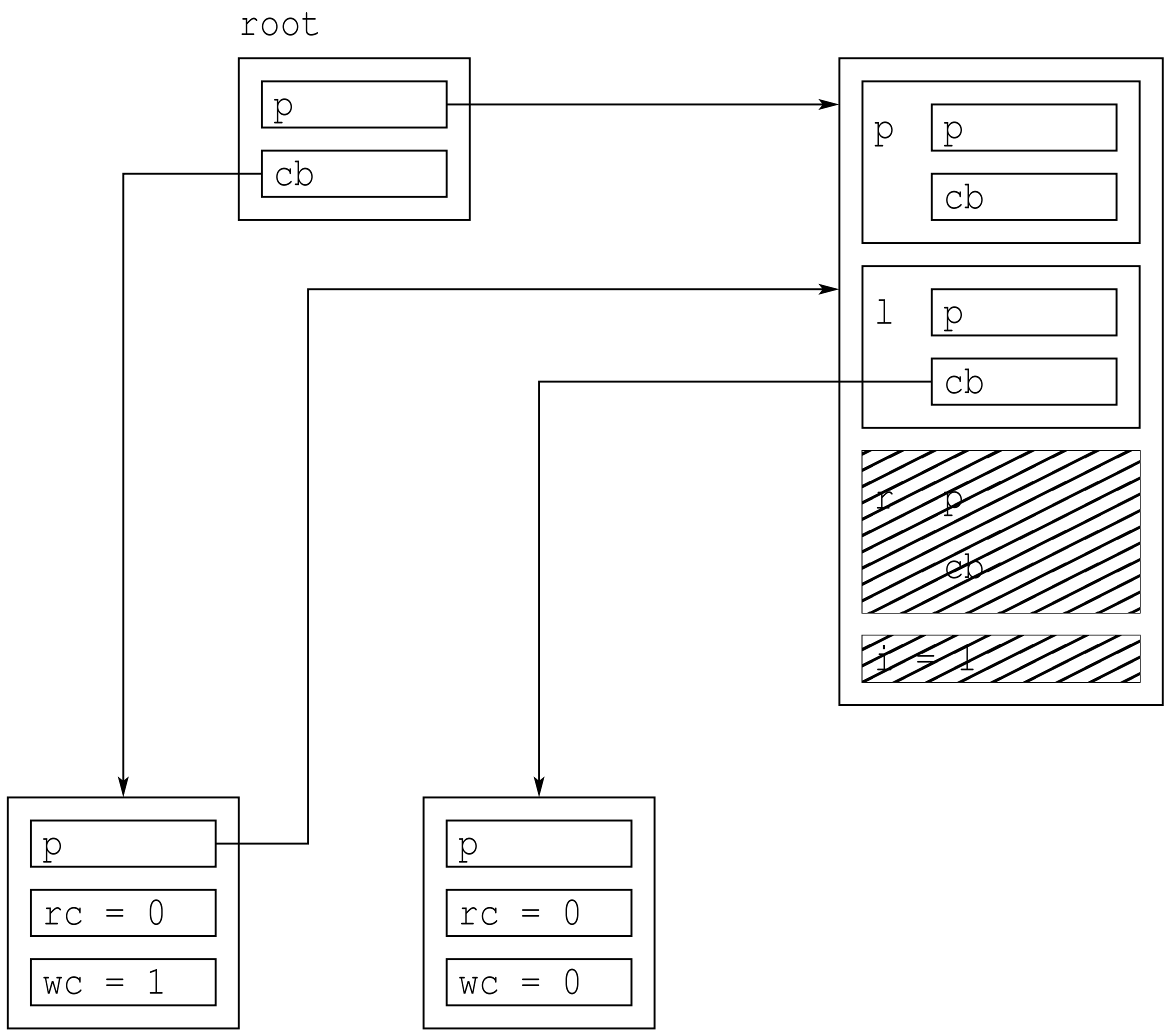


- destroyed `p` in left node, and completed destruction of left node

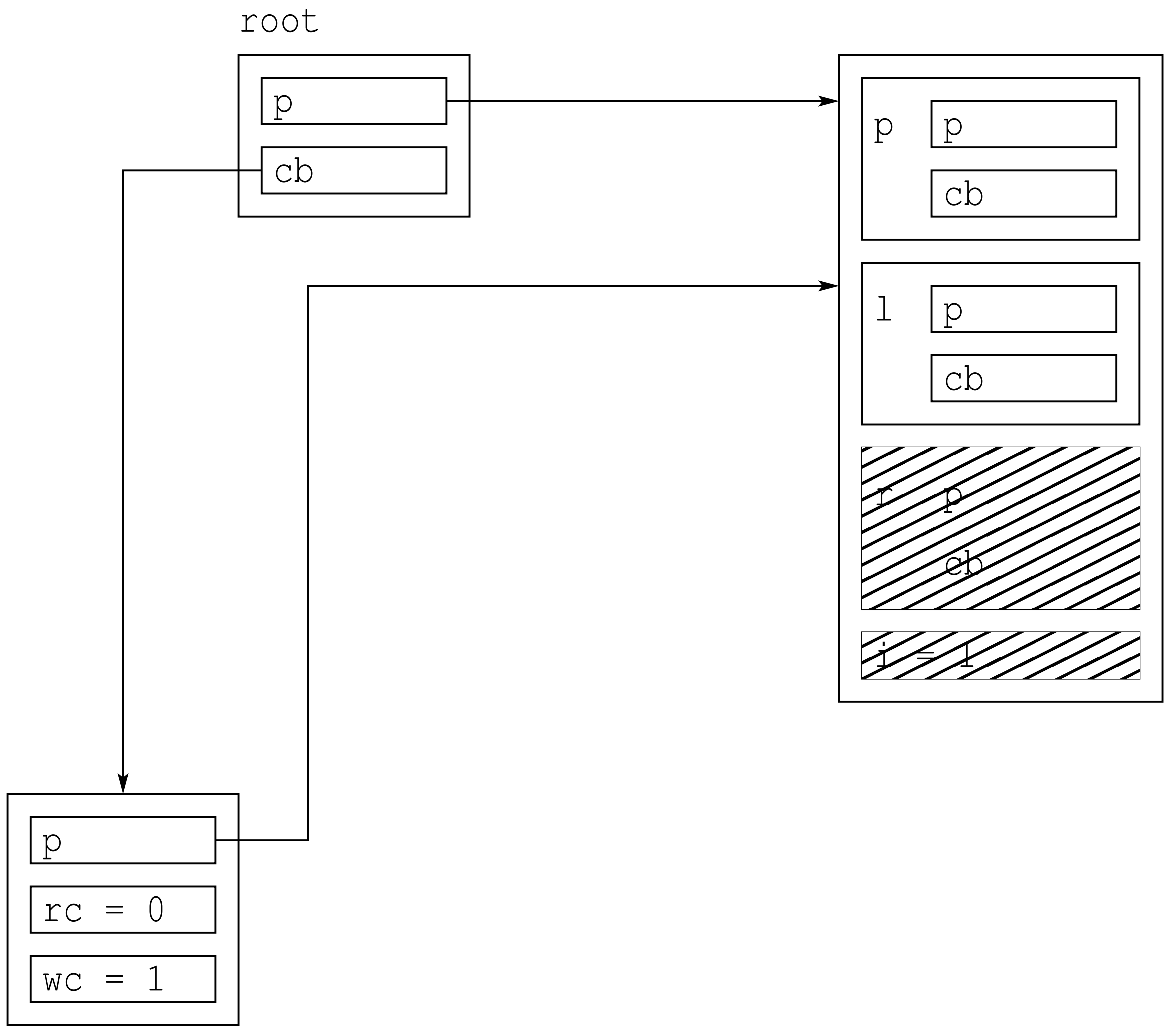
Avoiding Circular References Example (Continued 10)



- left node has been destroyed

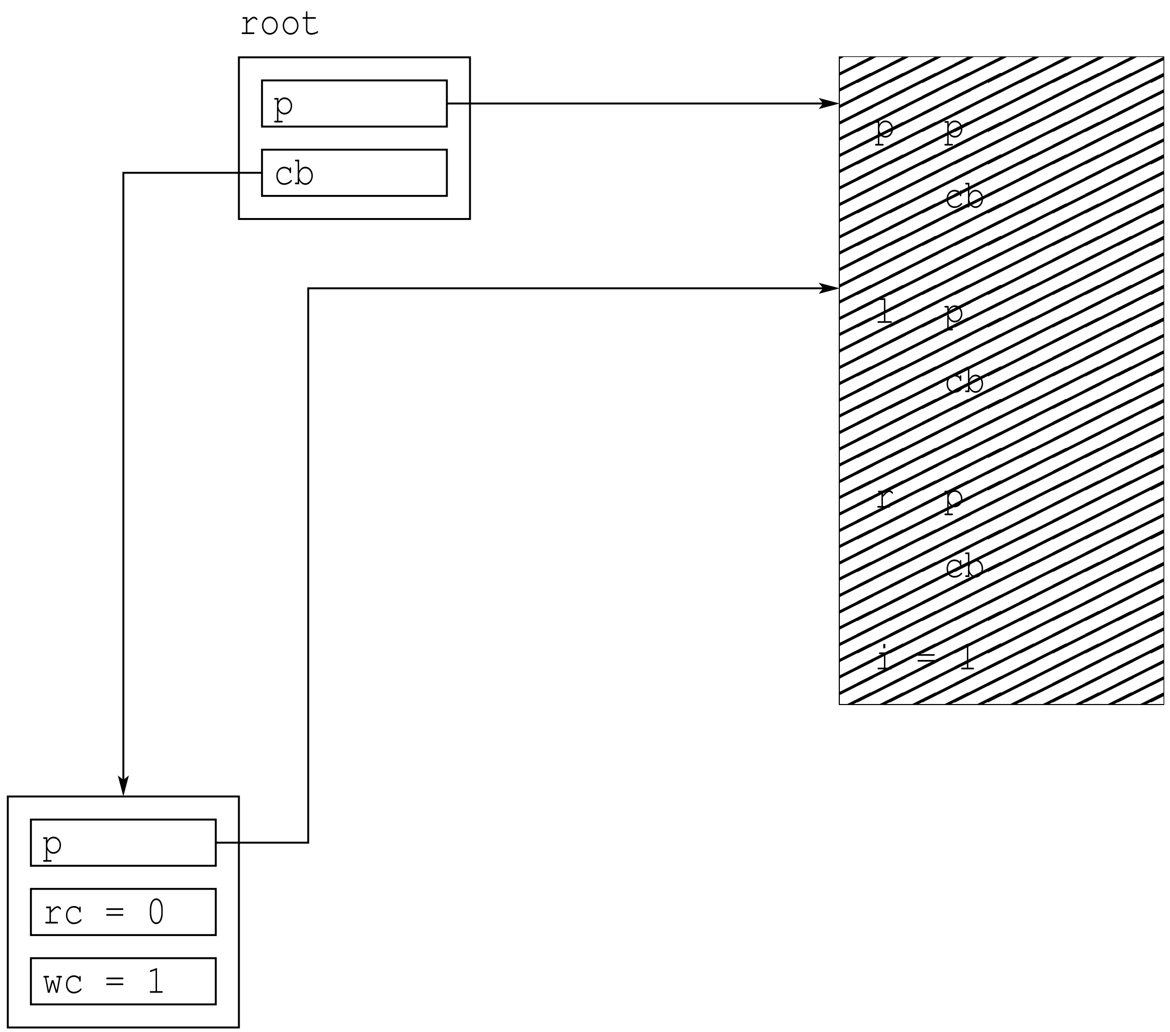


- continue destruction of `l` in root node; decrement weak count, which reaches zero



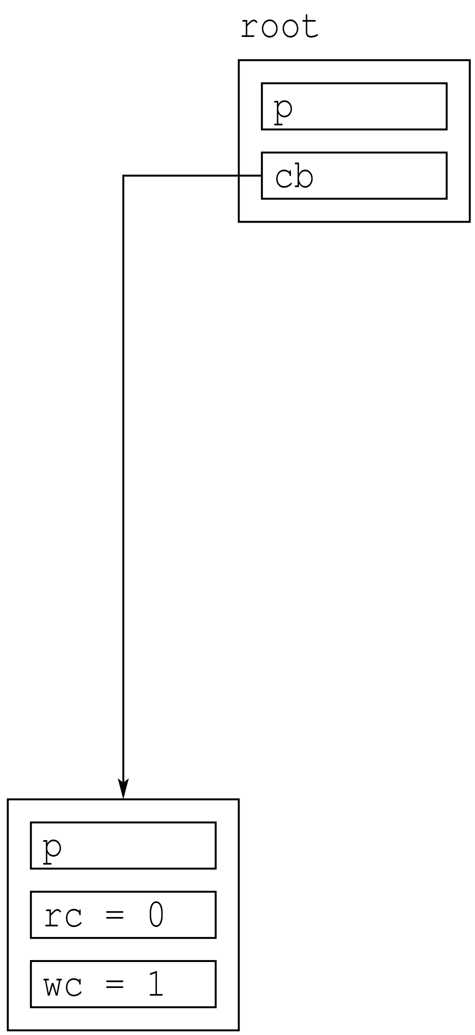
- destroyed control block for (previously destroyed) left child of root node

Avoiding Circular References Example (Continued 13)

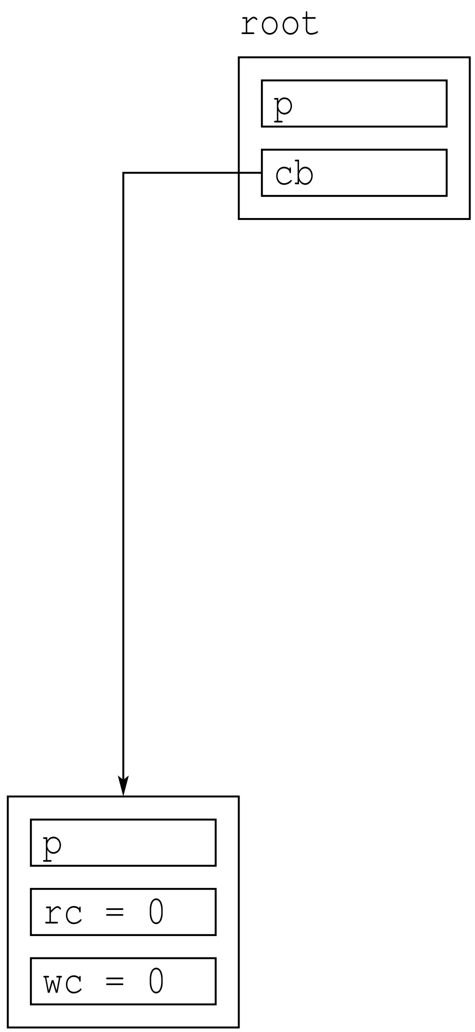


- finished destroying `l` in root node; destroyed `p` in root node; and completed destruction of root node

Avoiding Circular References Example (Continued 14)

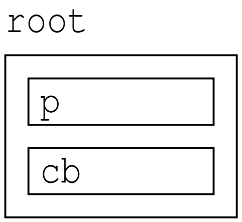


- root node has been destroyed



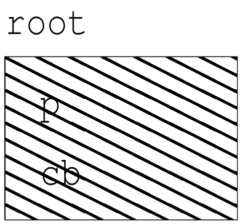
- continuing with destruction of `root`; decremented weak count, which reaches zero

Avoiding Circular References Example (Continued 16)



- destroyed control block

Avoiding Circular References Example (Continued 17)



- `root` has been destroyed

Section 3.2.5

The `boost::intrusive_ptr` Class Template

The boost::intrusive_ptr Class Template

- boost::intrusive_ptr provides intrusive shared pointer type
- aside from being intrusive, similar to boost::shared_ptr (which is similar to std::shared_ptr)
- declaration:

```
template <class T> class intrusive_ptr;
```

- T is type of referenced object
- new reference is added by calling function (provided by user) with signature:

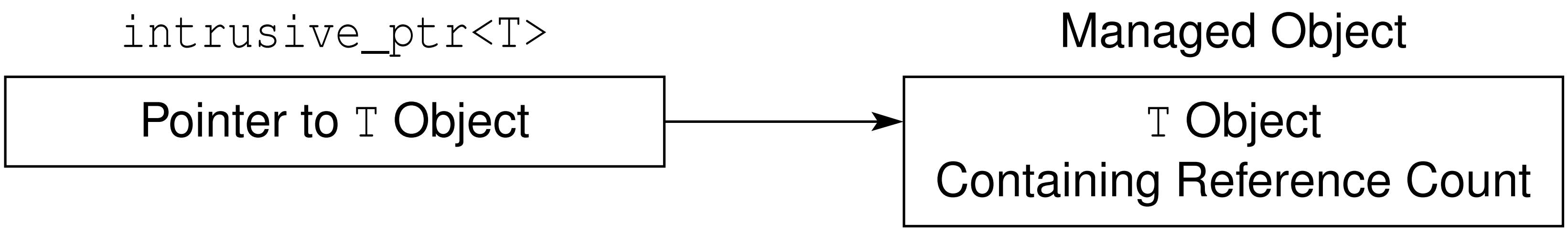
```
void intrusive_ptr_add_ref(T*)
```

- reference is eliminated by calling function (provided by user) with signature:

```
void intrusive_ptr_release(T*)
```

- intrusive_ptr_release responsible for destroying underlying object when reference count reaches zero

The `boost::intrusive_ptr` Class Template (Continued 1)



- `intrusive_ptr` itself has same memory cost as raw pointer
- managed object (of type `T`) must provide means for reference counting, which is accessed through user-provided functions `intrusive_ptr_add_ref` and `intrusive_ptr_release`

intrusive_ptr Example

```
#include <boost/intrusive_ptr.hpp>
#include <iostream>
#include <string>
#include <cassert>

class Person {
public:
    Person(const std::string& name) : name_(name),
      refCount_(0) {}
    void hold() {++refCount_;}
    void release() {if (--refCount_ == 0) {delete this;}}
    unsigned refCount() const {return refCount_;}
private:
    ~Person() {std::cout << "dtor called\n";}
    std::string name_;
    unsigned refCount_; // reference count
};

void intrusive_ptr_add_ref(Person* p) {p->hold();}
void intrusive_ptr_release(Person* p) {p->release();}

int main() {
    boost::intrusive_ptr<Person> a(new Person("Bjarne"));
    {
        boost::intrusive_ptr<Person> b = a;
        assert(a->refCount() == 2);
    }
    assert(a->refCount() == 1);
}
```

Section 3.2.6

Smart-Pointer Usage Examples

Temporary Heap-Allocated Objects

- create heap-allocated object for temporary use inside function/block
- object will be automatically deallocated upon leaving function/block

---

```
#include <memory>

void func() {
    // ...
    int size = /* ... */;
    auto buffer(std::make_unique<char[]>(size));
    // ... (use buffer)
    // when buffer destroyed, pointee automatically
    // freed
}
```

Decoupled Has-A Relationship

- instead of making object member of class, store object outside class and make pointer to object member of class
- might want to do this for object that:
  - is optional (e.g., is not always used or is lazily initialized)
  - has one of several base/derived types
- pointer in class object owns decoupled object

```
#include <memory>

class Widget {
    // ...
private:
    // ...
    std::unique_ptr<Type> item_;
      // decoupled object has type Type
};
```

Decoupled Fixed-But-Dynamically-Sized Array

- array stored outside class object, where array size fixed but determined at run time
- class object has pointer that owns decoupled array

```
#include <memory>

class Widget {
public:
    using Element = int;
    Widget(std::size_t size) :
      array_(std::make_unique(Element[]>(size),
      size_(size) {}
    // ...
private:
    // ...
    const std::unique_ptr<Element[]> array_;
    std::size_t size_;
};
```

Pimpl Idiom

- interface and implementation split across two classes: (i.e., handle class and body class)
- known as pointer to implementation (pimpl) idiom
- can be used, for example, to reduce compile-time dependencies (to facilitate faster compiles)
- class object has pointer that owns implementation object

```
#include <memory>

class Widget {
    // ...
private:
    class Impl; // defined elsewhere
    const std::unique_ptr<Impl> impl_;
      // incomplete type Impl is allowed
    // ...
};
```

Tree

- tree, where tree owns root node and each node owns its children
- recursive destruction of nodes may cause stack-overflow problems, especially for unbalanced trees (but such problems can be avoided by dismantling tree from bottom upwards)

```
#include <memory>
#include <array>

class Tree {
public:
    class Node {
        // ...
    private:
        std::array<std::unique_ptr<Node>, 2> children_;
          // owning pointers (parent owns children)
        Node* parent_; // non-owning pointer
        // ...
    };
    // ...
private:
    std::unique_ptr<Node> root_;
    // ...
};
```

Doubly-Linked List

- doubly-linked list, where list owns first list node and each list node owns its successor
- recursive destruction of nodes can cause stack-overflow problems, for sufficiently large lists (but deep recursions can be avoided with extra work)

```
#include <memory>

class List {
public:
    class Node {
        // ...
    private:
        std::unique_ptr<Node> next_;
          // owning pointer (node owns successor)
        Node* prev_; // non-owning pointer
    };
    // ...
private:
    // ...
    std::unique_ptr<Node> head_;
};
```

Tree That Provides Strong References

- tree that provides strong references to data in nodes
- tree owns root node and each node owns its children
- accessor for node data returns object having pointer that keeps node alive

---

```
#include <memory>
#include <array>

class Tree {
public:
    using Data = /* ... */;
    class Node {
        // ...
    private:
        std::array<std::shared_ptr<Node>, 2> children_;
        std::weak_ptr<Node> parent_;
        Data data_;
    };
    std::shared_ptr<Data> find(/* ... */) {
        std::shared_ptr<Node> sp;
        // ...
        return {sp, &(sp->data)};
          // use shared_ptr aliasing constructor
    }
private:
    std::shared_ptr<Node> root_;
};
```

Directed Acyclic Graph

- encapsulated directed acyclic graph (DAG), where graph owns root nodes and each node owns its children
- pointers in graph object own root nodes
- pointers in each node object owns children

```
#include <memory>
#include <vector>

class Dag {
public:
    class Node {
        // ...
    private:
        std::vector<std::shared_ptr<Node>> children_;
          // owning pointers
        std::vector<Node*> parents_;
          // non-owning pointers
        // ...
    };
private:
    std::vector<std::shared_ptr<Node>> roots_;
      // owning pointers
};
```

- factory function that returns object on heap
- factory function should use `std::unique_ptr` if object will not be shared
- factory function should use `std::shared_ptr` if object will be shared
- provide factory functions using each of `std::unique_ptr` and `std::shared_ptr` if both sharing and non-sharing cases are common

---

```
#include <memory>

std::unique_ptr<Widget> makeWidget() {
    return std::make_unique<Widget>();
}

std::shared_ptr<Gadget> makeGadget() {
    return std::make_shared<Gadget>();
}
```

Factory Function With Cache

- cache of objects on heap
- object in cache should only continue to live while it has external user
- object returned to user is owning pointer
- cache entries have non-owning pointers to corresponding objects

---

```
#include <memory>

std::shared_ptr<Widget> makeWidget(int id) {
    static std::map<int, std::weak_ptr<Widget>> cache;
    static std::mutex mut;
    std::lock_guard<std::mutex> lock(mut);
    auto sp = cache[id].lock();
    if (!sp) {
        sp = std::make_shared<Widget>(id);
        cache[id] = sp;
    }
    return sp;
}
```

Section 3.2.7

References

Talks I

1. Michael VanLoon, Lightning Talk: Anatomy of a Smart Pointer, CppCon, Bellevue, WA, USA, Sept. 9, 2014. Available online at `https://youtu.be/bxaj_0o4XAI`.
2. Herb Sutter, Leak-Freedom in C++... By Default, CppCon, Bellevue, WA, USA, Sept. 23, 2016. Available online at `https://youtu.be/JfmTagWcqoE`.

Section 3.3

Exceptions

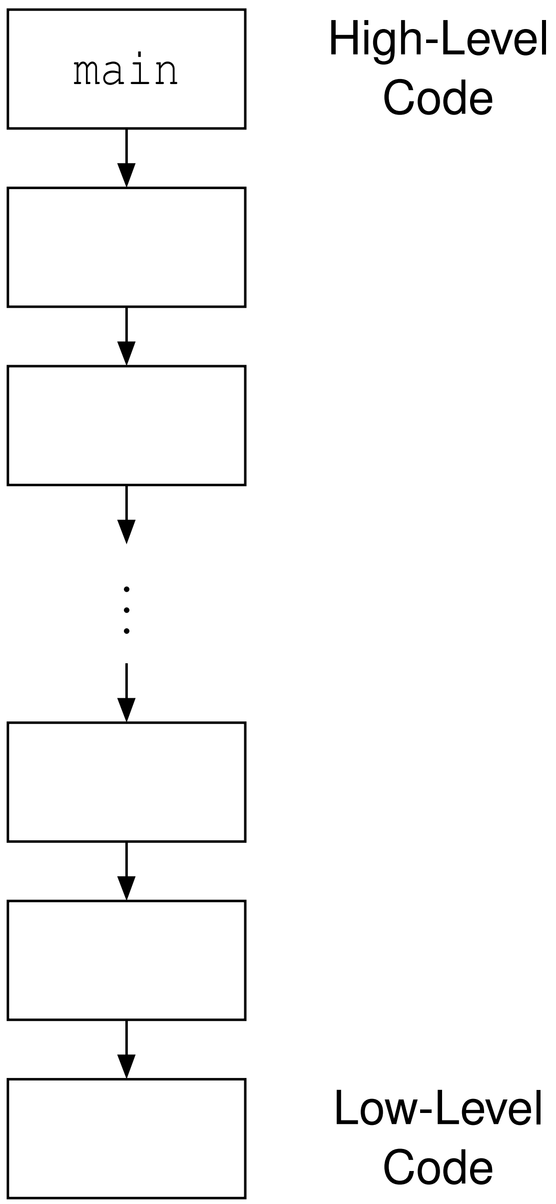
Section 3.3.1

Introduction

Exceptions

- exceptions are language mechanism for handling exceptional (i.e., abnormal) situations
- exceptional situation perhaps best thought of as case when code could not do what it was asked to do and usually (but not always) corresponds to error condition
- exceptions often employed for error handling
- exceptions propagate information from point where error ***detected*** to point where error ***handled***
- code that encounters error that it is unable to handle throws exception
- code that wants to handle error catches exception and performs processing necessary to handle error
- exceptions provide convenient way in which to ***separate error detection from error handling***

The Problem



- error detected in low-level code
- want to handle error in high-level code
- must propagate error information up call chain

Traditional Error Handling

- if any error occurs, terminate program
  - overly draconian
- pass error code back from function (via return value, reference parameter, or global object) and have caller check error code
  - errors are ignored by default (i.e., explicit action required to check for error condition)
  - caller may forget to check error code allowing error to go undetected
  - code can become cluttered with many checks of error codes, which can adversely affect code readability and maintainability
- call error handler if error detected
  - may not be possible or practical for handler to recover from particular error (e.g., handler may not have access to all information required to recover from error)

Example: Traditional Error Handling

```cpp
#include <iostream>

bool func3() {
    bool success = false;
    // ...
    return success;
}

bool func2() {
    if (!func3()) {return false;}
    // ...
    return true;
}

bool func1() {
    if (!func2()) {return false;}
    // ...
    return true;
}

int main() {
    if (!func1()) {
        std::cout << "failed\n";
        return 1;
    }
    // ...
}
```

Error Handling With Exceptions

- when error condition detected, signalled by throwing exception (with **`throw`** statement)
- exception is object that describes error condition
- thrown exception caught by handler (in **`catch`** clause of **`try`** statement), which takes appropriate action to handle error condition associated with exception
- handler can be in different function from where exception thrown
- error-free code path tends to be relatively simple, since no need to explicitly check for error conditions
- error condition less likely to go undetected, since uncaught exception terminates program

Example: Exceptions

```cpp
#include <iostream>
#include <stdexcept>

void func3() {
    bool success = false;
    // ...
    if (!success) {throw std::runtime_error("Yikes!");}
}

void func2() {
    func3();
    // ...
}

void func1() {
    func2();
    // ...
}

int main() {
    try {func1();}
    catch (...) {
        std::cout << "failed\n";
        return 1;
    }
    // ...
}
```

safe_divide Example: Traditional Error Handling

```
#include <iostream>
#include <vector>
#include <utility>

std::pair<bool, int> safe_divide(int x, int y) {
    if (!y) {
        return std::make_pair(false, 0);
    }
    return std::make_pair(true, x / y);
}

int main() {
    std::vector<std::pair<int, int>> v = {{10, 2}, {10, 0}};
    for (auto p : v) {
        auto result = safe_divide(p.first, p.second);
        if (result.first) {
            int quotient = result.second;
            std::cout << quotient << '\n';
        } else {
            std::cerr << "division by zero\n";
        }
    }
}
```

safe_divide Example: Exceptions

```
#include <iostream>
#include <vector>
#include <utility>

class divide_by_zero {};

int safe_divide(int x, int y) {
    if (!y) {
        throw divide_by_zero();
    }
    return x / y;
}

int main() {
    std::vector<std::pair<int, int>> v = {{10, 2}, {10, 0}};
    for (auto p : v) {
        try {
            std::cout << safe_divide(p.first, p.second) <<
              '\n';
        }
        catch(const divide_by_zero& e) {
            std::cerr << "division by zero\n";
        }
    }
}
```

Exceptions Versus Traditional Error Handling

- advantages of exceptions:
  - exceptions allow for error handling code to be easily separated from code that detects error
  - exceptions can easily pass error information many levels up call chain
  - passing of error information up call chain managed by language (no explicit code required)
- disadvantages of exceptions:
  - writing code that always behaves correctly in presence of exceptions requires great care (as we shall see)
  - although possible to have no execution-time cost when exceptions not thrown, still have memory cost (to store information needed for stack unwinding for case when exception is thrown)

Section 3.3.2

Exceptions

Exceptions

- exceptions are objects
- type of object used to indicate kind of error
- value of object used to provide details about particular occurrence of error
- exception object can have any type (built-in or class type)
- for convenience, standard library provides some basic exception types
- all exception classes in standard library derived (directly or indirectly) from `std::exception` class
- exception object is propagated from one part of code to another by throwing and catching
- exception processing disrupts normal control flow

Exception Classes Derived from `exception` Class

Type	Description
`logic_error`	faulty logic in program
`runtime_error`	error caused by circumstances beyond scope of program
`bad_typeid`	invalid operand for **`typeid`** operator
`bad_cast`	invalid expression for **`dynamic_cast`**
`bad_weak_ptr`	bad `weak_ptr` given
`bad_function_call`	`function` has no target
`bad_alloc`	storage allocation failure
`bad_exception`	use of invalid exception type in certain contexts
`bad_variant_access`	`variant` accessed in invalid way

Standard Exception Classes (Continued 1)

Exception Classes Derived from `bad_cast` Class

Type	Description
`bad_any_cast`	invalid cast for `any`

Exception Classes Derived from `logic_error` Class

Type	Description
`domain_error`	domain error (e.g., square root of negative number)
`invalid_argument`	invalid argument
`length_error`	length too great (e.g., resize `vector` beyond `max_size`)
`out_of_range`	out of range argument (e.g., subscripting error in `vector::at`)
`future_error`	invalid operations on `future` objects
`bad_optional_access`	`optional` accessed in invalid way

Exception Classes Derived from `runtime_error` Class

Type	Description
`range_error`	range error
`overflow_error`	arithmetic overflow error
`underflow_error`	arithmetic underflow error
`regex_error`	error in regular expressions library
`system_error`	operating-system or other low-level error

Exception Classes Derived from `runtime_error::system_error` Class

Type	Description
`ios_base::failure`	I/O failure

Section 3.3.3

Throwing and Catching Exceptions

Throwing Exceptions

- throwing exception accomplished by **`throw`** statement
- throwing exception transfers control to handler
- object is passed
- type of object determines which handlers can catch it
- handlers specified with **`catch`** clause of **`try`** block
- for example

```
throw "OMG!";
```

can be caught by handler of **`const char`**`*` type, as in:

```
try {
    // ...
}
catch (const char* p) {
    // handle character string exceptions here
}
```

Throwing Exceptions (Continued)

- throw statement initializes temporary object called **exception object**
- type of exception object determined by ***static*** type of operand of `throw` (so slicing can occur)
- if thrown object is class object, copy/move constructor and destructor must be accessible
- temporary may be moved/copied several times before caught
- advisable for type of exception object to be user defined to reduce likelihood of different parts of code using type in conflicting ways

Catching Exceptions

- exception can be caught by **`catch`** clause of **`try`**-**`catch`** block
- code that might throw exception placed in **`try`** block
- code to handle exception placed in **`catch`** block
- **`try`**-**`catch`** block can have multiple **`catch`** clauses
- **`catch`** clauses checked for match in order specified and only first match used
- **`catch`** `(...)` can be used to catch any exception
- example:

```
try {
    // code that might throw exception
}
catch (const std::logic_error& e) {
    // handle logic_error exception
}
catch (const std::runtime_error& e) {
    // handle runtime_error exception
}
catch (...) {
    // handle other exception types
}
```

Catching Exceptions (Continued)

- catch exceptions by reference in order to:
  - avoid copying, which might throw
  - allow exception object to be modified and then rethrown
  - avoid slicing

Exception During Exception: Catching By Value

```
#include <iostream>
#include <stdexcept>

class Error {
public:
    Error(int value) : value_(value) {}
    Error(Error&& e) : value_(e.value_) {}
    Error(const Error&) {throw std::runtime_error("copy");}
    int get() const {return value_;}
private:
    int value_; // error code
};

void func2() {throw Error(42);} // might move

void func1() {
    try {func2();}
    // catch by value (copy throws)
    catch (Error e) {
        std::cerr << "yikes\n";
    }
}

int main() {
    try {func1();}
    catch (...) {std::cerr << "exception\n";}
}
```

Throwing Polymorphically: Failed Attempt

```
#include <iostream>

class Base {};
class Derived : public Base {};

void func(Base& x) {
    throw x; // always throws Base
}

int main() {
    Derived d;
    try {func(d);}
    catch (Derived& e) {
        std::cout << "Derived\n";
    }
    catch (...) {
        std::cout << "not Derived\n";
    }
}
```

- type of exception object determined from ***static*** type of throw expression

Throwing Polymorphically

```
#include <iostream>

class Base {
public:
    virtual void raise() {throw *this;}
};
class Derived : public Base {
public:
    virtual void raise() {throw *this;}
};

void func(Base& x) {
    x.raise();
}

int main() {
    Derived d;
    try {func(d);}
    catch (Derived& e) {
        std::cout << "Derived\n";
    }
    catch (...) {
        std::cout << "not Derived\n";
    }
}
```

Rethrowing Exceptions

- caught exception can be rethrown by **throw** statement with no operand
- example:

```
try {
    // code that may throw exception
}
catch (...) {
    throw; // rethrow caught exception
}
```

Rethrowing Example: Exception Dispatcher Idiom

```
void handle_exception() {
    try {throw;}
    catch (const exception_1& e) {
        log_error("exception_1 occurred");
        // ...
    }
    catch (const exception_2& e) {
        log_error("exception_2 occurred");
        // ...
    }
    // ...
}

void func() {
    try {operation();}
    catch (...) {handle_exception();}
    // ...
    try {another_operation();}
    catch (...) {handle_exception();}
}
```

- allows reuse of exception handling code

Transfer of Control from Throw Site to Handler

- when exception is thrown, control is transferred to nearest handler (in catch clause) with matching type, where “nearest” means handler for try block most recently entered (by thread) and not yet exited
- if no matching handler found, `std::terminate()` is called
- as control passes from throw expression to handler, destructors are invoked for all automatic objects constructed since try block entered, where automatic objects destroyed in reverse order of construction
- process of calling destructors for automatic objects constructed on path from try block to throw expression called **stack unwinding**
- object not deemed to be constructed if constructor exits due to exception (in which case destructor will not be invoked)
- do not throw exception in destructor since destructors called during exception processing and throwing exception during exception processing will terminate program

Stack Unwinding Example

```
void func1() {
    std::string dave("dave");
    try {
        std::string bye("bye");
        func2();
    }
    catch (const std::runtime_error& e) { // Handler
        std::cerr << e.what() << '\n';
    }
}

void func2() {
    std::string world("world");
    func3(0);
}

void func3(int x) {
    std::string hello("hello");
    if (x == 0) {
        std::string first("first");
        std::string second("second");
        throw std::runtime_error("yikes"); // Throw site
    }
}
```

- calling `func1` will result in exception being thrown in `func3`
- during stack unwinding, destructors called in order for `second`, `first`, `hello`, `world`, and `bye` (i.e., reverse order of construction); `dave` unaffected

Function Try Blocks

- function try blocks allow entire function to be wrapped in try block
- function returns when control flow reaches end of catch block (return statement needed for non-void function)
- example:

```
#include <iostream>
#include <stdexcept>

int main()
try {
    throw std::runtime_error("yikes");
}
catch (const std::runtime_error& e) {
    std::cerr << "runtime error " << e.what() << '\n';
}
```

- although function try blocks can be used for any function, most important use cases are for constructors and destructors
- function try block only way to catch exceptions thrown during construction of data members or base objects (which happens before constructor body is entered) or during destruction of data members or base objects (which happens after destructor body exited)

Exceptions and Construction/Destruction

- order of construction:
  1. base class objects as listed in type definition left to right
  2. data members as listed in type definition top to bottom
  3. constructor body
- order of destruction is exact reverse of order of construction, namely:
  1. destructor body
  2. data members as listed in type definition bottom to top
  3. base class objects as listed in type definition right to left
- lifetime of object begins when constructor completes
- constructor might throw in:
  - constructor of base class object
  - constructor of data member
  - constructor body
- need to perform cleanup for constructor body
- will assume destructors do not throw (since very bad idea to throw in destructor)
- any exception caught in function try block of constructor or destructor rethrown implicitly (at end of catch block)

Construction/Destruction Example

```
#include <string>
#include <iostream>

struct Base {
    Base() {}
    ~Base() {};
};

class Widget : public Base {
public:
    Widget() {}
    ~Widget() {}
    // ...
private:
    std::string s_;
    std::string t_;
};

int main() {
    Widget w;
    // ...
}
```

Function Try Block Example

```
#include <iostream>
#include <stdexcept>

class Gadget {
public:
    Gadget() {throw std::runtime_error("ctor");}
    ~Gadget() {}
};

class Widget {
public:
    // constructor uses function try block
    Widget()
    try {std::cerr << "ctor body\n";}
    catch (...) {std::cerr << "exception in ctor\n";}
    ~Widget() {std::cerr << "dtor body\n";}
private:
    Gadget g_;
};

int main()
try {Widget w;}
catch (...) {
    std::cerr << "terminating due to exception\n";
    return 1;
}
```

Section 3.3.4

Exception Specifications

The `noexcept` Specifier

- **`noexcept`** specifier in function declaration indicates whether or not function can throw exceptions
- **`noexcept`** specifier with **`bool`** constant expression argument indicates function does not throw exceptions if expression **`true`** (otherwise, may throw)
- **`noexcept`** without argument equivalent to **`noexcept`**`(`**`true`**`)`
- except for destructors, not providing **`noexcept`** specifier equivalent to **`noexcept`**`(`**`false`**`)`
- if **`noexcept`** specifier not provided for destructor, specifier identical to that of implicit declaration (which is, in practice, usually **`noexcept`**)
- example:

```
void func1(); // may throw anything
void func2() noexcept(false); // may throw anything
void func3() noexcept(true); // does not throw
void func4() noexcept; // does not throw
template <class T>
void func5(T) noexcept(sizeof(T) <= 4);
  // does not throw if sizeof(T) <= 4
```

The `noexcept` Specifier (Continued 1)

- exception specification for function is part of function's type
- example:
```
void f() noexcept;
auto g = f; // g is noexcept
```
- exception specification for function is not part of function's signature
- consequently, cannot overload on noexcept specifier
- example:
```
void f();
void f() noexcept;
  // ERROR: both functions have same signature
```

The **noexcept** Specifier (Continued 2)

- nontrivial **bool** expression for **noexcept** specifier often useful in templates
- example (swap function):

```
#include <type_traits>
#include <utility>

// swap two values
template <class T>
void exchange(T& a, T& b) noexcept(
  std::is_nothrow_move_constructible<T>::value &&
  std::is_nothrow_move_assignable<T>::value) {
    T tmp(std::move(a)); // move construction
    a = std::move(b);  // move assignment
    b = std::move(tmp); // move assignment
}
```

The `noexcept` Specifier (Continued 3)

- if function with `noexcept(true)` specifier throws exception, `std::terminate` is called immediately
- example:

```
// This function will terminate the program.
void die_die_die() noexcept {
    throw 0;
}
```

- advisable not to use `noexcept(true)` specifier unless clear that no reasonable usage of function can throw (in current or ***any future*** version of code)
- in practice, can often be difficult to guarantee that function will never throw exception (especially when considering ***all future*** versions of code)

Exceptions and Function Calls

- for some (nonreference) class type `T` and some constant **`bool`** expression `expr`, consider code such as:

```
T func(T) noexcept(expr);
T x;
T y = func(x); // function call
```

- function call can throw exception as result of:
  1. parameter passing (if pass by value)
  2. function execution ***including return statement***
- in parameter passing, construction and destruction of each parameter happens in context of ***calling*** function
- consequently, invocation of noexcept function can still result in exception being thrown due to parameter passing
- in case of return by value, construction of temporary (if not elided) to hold return value happens in context of ***called*** function
- consequently, must exercise care not to violate noexcept contract if noexcept function returns by value

Avoiding Exceptions Due to Function Calls

- if exception due to parameter passing must be avoided:
  - pass by reference; or
  - ensure **`noexcept`** move and/or copy constructor as appropriate; or
  - ensure function invoked in manner such that copy elision is guaranteed
- if exception due to return by value must be avoided:
  - ensure **`noexcept`** move or copy constructor as appropriate; or
  - ensure that function invoked in manner such that copy elision is guaranteed

noexcept Operator

- **noexcept** operator takes expression and returns **bool** indicating if expression can throw exception
- does not actually evaluate expression
- in determining result, only considers **noexcept** specifications for functions involved
- example:

```
#include <cstdlib>
#include <cassert>
#include <utility>

void increment(int&) noexcept;
char* memAlloc(std::size_t);

// does not throw exception, but not declared noexcept
void doesNotThrow() {};

int main() {
    assert(noexcept(1 + 1) == true);
    assert(noexcept(memAlloc(0)) == false);
    // Note: does not evaluate expression
    assert(noexcept(increment(*((int*)0))) == true);
    assert(noexcept(increment(std::declval<int&>())) ==
      true);
    // Note: only uses noexcept specifiers
    assert(noexcept(doesNotThrow()) == false);
}
```

noexcept Operator (Continued)

- **noexcept** operator particularly useful for templates
- example:

```
#include <iostream>

class Int256 { /* ... */ }; // 256-bit integer
class BigInt { /* ... */ }; // arbitrary-precision integer

// function will not throw exception
Int256 operator+(const Int256& x, const Int256& y)
  noexcept;

// function may throw exception
BigInt operator+(const BigInt& x, const BigInt& y);

// whether function may throw exception depends on T
template <class T>
T add(const T& x, const T& y) noexcept(noexcept(x + y) &&
  std::is_nothrow_move_constructible<T>::value)
{return x + y;}

int main() {
    Int256 i1, i2;
    BigInt b1, b2;
    std::cout << "int " << noexcept(add(1, 1)) << '\n'
      << "Int256 " << noexcept(add(i1, i2)) << '\n'
      << "BigInt " << noexcept(add(b1, b2)) << '\n';
}
```

Dynamic Exception Specifications

- language offers another mechanism for stating exception specifications known as dynamic exception specifications
- dynamic exception specifications are ***deprecated*** and ***should not be used***
- provide exception specification for function using **`throw`** specifier
- used to specify list of all types of exceptions that can be thrown
- in practice, such a list more of hindrance than help
- if list of all allowable exceptions specified, must check if thrown exception of expected type, which is unnecessary cost
- in terms of compiler optimization, what matters most is whether any exception (regardless of type) can be thrown at all

Section 3.3.5

Storing and Retrieving Exceptions

Storing and Retrieving Exceptions

- might want to store exception and then later retrieve and rethrow it
- exception can be stored using `std::exception_ptr` type
- current exception can be retrieved with `std::current_exception`
- rethrow exception stored in `exception_ptr` object using `std::rethrow_exception`
- provides mechanism for moving exceptions between threads:
  - store exception on one thread
  - then retrieve and rethrow stored exception on another thread
- `std::make_exception_ptr` can be used to make `exception_ptr` object

Example: Storing and Retrieving Exceptions

```
#include <exception>
#include <stdexcept>

void yikes() {
    throw std::runtime_error("Yikes!");
}

std::exception_ptr getException() {
    try {
        yikes();
    }
    catch (...) {
        return std::current_exception();
    }
    return nullptr;
}

int main() {
    std::exception_ptr e = getException();
    std::rethrow_exception(e);
}
```

Section 3.3.6

Exception Safety

Resource Management

- **resource**: physical or virtual component of limited availability within computer system
- examples of resources include: memory, files, devices, network connections, processes, threads, and locks
- essential that acquired resource properly released when no longer needed
- when resource not properly released when no longer needed, **resource leak** said to occur
- exceptions have important implications in terms of resource management
- must be careful to avoid resource leaks

Resource Leak Example

```
void useBuffer(char* buf) { /* ... */ }

void doWork() {
    char* buf = new char[1024];
    useBuffer(buf);
    delete[] buf;
}
```

- if `useBuffer` throws exception, code that deletes `buf` is never reached

Cleanup

- cleanup operations should always be performed in destructors
- following structure for code is ***fundamentally flawed***:

```
void func()
{
    initialize();
    do_work();
    cleanup();
}
```

- code with preceding structure ***not exception safe***
- if `do_work` throws exception, `cleanup` never called and cleanup operation not performed
- in best case, not performing cleanup will probably cause resource leak

Exception Safety and Exception Guarantees

- in order for exception mechanism to be useful, must know what can be assumed about state of program when exception thrown
- operation said to be **exception safe** if it leaves program in valid state when operation is terminated by exception
- several levels of exception safety: basic, strong, nothrow
- **basic guarantee**: all invariants preserved and no resources leaked
- with basic guarantee, partial execution of failed operation may cause side effects
- **strong guarantee**: in addition to basic guarantee, failed operation guaranteed to have no side effects (i.e., commit semantics)
- with strong guarantee, operation can still fail causing exception to be thrown
- **nothrow guarantee**: in addition to basic guarantee, promises not to emit exception (i.e., operation guaranteed to succeed even in presence of exceptional circumstances)

Exception Guarantees

- assume all functions throw if not known otherwise
- code must always provide basic guarantee
- nothrow guarantee should always be provided by destructors
- whenever possible, nothrow guarantee should be provided by:
  - move operations (i.e., move constructors and move assignment operators)
  - swap operations
- provide strong guarantee when natural to do so and not more costly than basic guarantee
- examples of strong guarantee:
  - `push_back` for container, subject to certain container-dependent conditions being satisfied (e.g., for `std::vector`, element type has nonthrowing move or is copyable)
  - `insert` on `std::list`
- examples of nothrow guarantee:
  - `swap` of two containers
  - `pop_back` for container

Resource Acquisition Is Initialization (RAII)

- resource acquisition is initialization (RAII) is programming idiom used to ***avoid resource leaks*** and ***provide exception safety***
- associate resource with owning object (i.e., RAII object)
- period of time over which resource held is tied to lifetime of RAII object
- resource acquired during creation of RAII object
- resource released during destruction of RAII object
- provided RAII object properly destroyed, resource leak cannot occur

Resource Leak Example Revisited

- implementation 1 (not exception safe; has memory leak):

```
void useBuffer(char* buf) { /* ... */ }

void doWork() {
    char* buf = new char[1024];
    useBuffer(buf);
    delete[] buf;
}
```

- implementation 2 (exception safe):

```
template <class T>
class SmartPtr {
public:
    SmartPtr(int size) : ptr_(new T[size]) {}
    ~SmartPtr() {delete[] ptr_;}
    operator T*() {return ptr_;}
    // ...
private:
    T* ptr_;
};

void useBuffer(char* buf) { /* ... */ }

void doWork() {
    SmartPtr<char> buf(1024);
    useBuffer(buf);
}
```

Section 3.3.7

Exceptions: Implementation, Cost, and Usage

Implementation of Exception Handling

- standard does not specify how exception handling is to be implemented; only specifies behavior of exception handling
- consider typical implementation here
- potentially significant memory overhead for storing exception object and information required for stack unwinding
- possible to have zero time overhead if no exception thrown
- time overhead significant when exception thrown
- not practical to create exception object on stack, since object frequently needs to be propagated numerous levels up call chain
- exception objects tend to be small
- exception object can be stored in small fixed-size buffer falling back on heap if buffer not big enough

Implementation of Exception Handling (Continued)

- memory required to maintain sufficient information to unwind stack when exception thrown
- two common strategies for maintaining information for stack unwinding: stack-based and table-based strategies
- stack-based strategy:
  - information for stack unwinding is saved on call stack, including list of destructors to execute and exception handlers that might catch exception
  - when exception is thrown, walk stack executing destructors until matching catch found
- table-based strategy:
  - store information to assist in stack unwinding in static tables outside stack
  - call stack used to determine which scopes entered but not exited
  - use look-up operation on static tables to determine where thrown exception will be handled and which destructors to execute
- table-based strategy uses less space on stack but potentially requires considerable storage for tables

Appropriateness of Using Exceptions

- use of exceptions not appropriate in all circumstances
- in practice, exceptions can sometimes (depending on C++ implementation) have prohibitive memory cost for systems with ***very limited memory*** (e.g., some embedded systems)
- since throwing exception has significant time overhead only use for ***infrequently occurring*** situations (not common case)
- in code where exceptions can occur, often much more difficult to bound how long code path will take to execute
- since difficult to predict response time of code in presence of exceptions, exceptions often cannot be used in ***time critical*** component of real-time system (where operation must be guaranteed to complete in specific maximum time)
- considerable amount of code in existence that is ***not exception safe***, especially legacy code
- cannot use exceptions in manner that would allow exceptions to propagate into code that is not exception safe

Enforcing Invariants: Exceptions Versus Assertions

- whether invariants should be enforced by exceptions or assertions somewhat controversial
- would recommend only using exceptions for errors from which recovery is likely to be possible
- if error condition detected is indicative of serious programming error, program state may already be sufficiently invalid (e.g., stack trampled, heap corrupted) that exception handling will not work correctly anyhow
- tendency amongst novice programmers is to use exceptions in places where their use is either highly questionable or clearly inappropriate

Section 3.3.8

Smart Pointers and Other RAII Classes

TwoBufs Example With Resource Leak

```
#include <cstddef>
#include <limits>

class TwoBufs {
public:
    TwoBufs(std::size_t aSize, std::size_t bSize) :
      a_(nullptr), b_(nullptr) {
        a_ = new char[aSize];
        // If new throws, a_ will be leaked.
        b_ = new char[bSize];
    }
    ~TwoBufs() {
        delete[] a_;
        delete[] b_;
    }
    // ...
private:
    char* a_;
    char* b_;
};

void doWork() {
    // This may leak memory.
    TwoBufs x(1000000,
      std::numeric_limits<std::size_t>::max());
    // ...
}
```

TwoBufs Example Corrected With unique_ptr

```
#include <cstddef>
#include <limits>
#include <memory>

class TwoBufs {
public:
    TwoBufs(std::size_t aSize, std::size_t bSize) :
      a_(std::make_unique<char[]>(aSize)),
      b_(std::make_unique<char[]>(bSize)) {}
    ~TwoBufs() {}
    // ...
private:
    std::unique_ptr<char[]> a_;
    std::unique_ptr<char[]> b_;
};

void doWork() {
    // This will not leak memory.
    TwoBufs x(1000000,
      std::numeric_limits<std::size_t>::max());
}
```

RAII Example: Stream Formatting Flags

```cpp
#include <iostream>
#include <ios>
#include <boost/io/ios_state.hpp>

// not exception safe
void unsafeOutput(std::ostream& out, unsigned int x) {
    auto flags = out.flags();
    // if exception thrown during output of x, old
    // formatting flags will not be restored
    out << std::hex << std::showbase << x << '\n';
    out.flags(flags);
}

// exception safe
void safeOutput(std::ostream& out, unsigned int x) {
    boost::io::ios_flags_saver ifs(out);
    out << std::hex << std::showbase << x << '\n';
}
```

- RAII objects can be used to save and restore state

Other RAII Examples

- `std::lock_guard`, `std::unique_lock`, and `std::shared_lock` can be used to manage mutexes (lock is released in destructor)
- `std::ifstream`, `std::ofstream`, and `std::fstream` can be used to manage files (file is closed in destructor)
- `std::string` can be used to manage strings (string buffer freed in destructor)
- `std::vector` can be used to manage dynamic arrays (array data freed in destructor)

Section 3.3.9

Exception Gotchas

shared_ptr Example: Not Exception Safe (Prior to C++17)

```
#include <memory>

class T1 { /* ... */ };
class T2 { /* ... */ };

void func(std::shared_ptr<T1> p, std::shared_ptr<T2> q)
{ /* ... */ }

void doWork() {
    // potential memory leak
    func(std::shared_ptr<T1>(new T1),
      std::shared_ptr<T2>(new T2));
    // ...
}
```

- one problematic order:
  1. allocate memory for T1
  2. construct T1
  3. allocate memory for T2
  4. construct T2
  5. construct shared_ptr<T1>
  6. construct shared_ptr<T2>
  7. call func
- if step 3 or 4 throws, memory leaked

- another problematic order:
  1. allocate memory for T1
  2. allocate memory for T2
  3. construct T1
  4. construct T2
  5. construct shared_ptr<T1>
  6. construct shared_ptr<T2>
  7. call func
- if step 3 or 4 throws, memory leaked

shared_ptr Example: Exception Safe (Prior to C++17)

```
#include <memory>

class T1 { /* ... */ };
class T2 { /* ... */ };

void func(std::shared_ptr<T1> p, std::shared_ptr<T2> q)
{ /* ... */ }

void doWork() {
    func(std::make_shared<T1>(), std::make_shared<T2>());
    // ...
}
```

- previously problematic line of code now does following:
  1. perform following operations in any order:
     - construct `shared_ptr<T1>` via `make_shared<T1>`
     - construct `shared_ptr<T2>` via `make_shared<T2>`
  2. call `func`
- each of `T1` and `T2` objects managed by `shared_ptr` at all times so no memory leak possible if exception thrown
- similar issue arises in context of `std::unique_ptr` and can be resolved by using `std::make_unique` in similar way as above

Stack Example

- stack class template parameterized on element type T

```
template <class T>
class Stack
{
public:
    // ...
    // Pop the top element from the stack.
    T pop() {
        // If the stack is empty...
        if (top_ == start_)
            throw "stack is empty";
        // Remove the last element and return it.
        return *(--top_);
    }
private:
    T* start_; // start of array of stack elements
    T* end_;  // one past end of array
    T* top_;  // one past current top element
};
```

- what is potentially problematic about this code with respect to exceptions?

Section 3.3.10

Miscellany

safe_add Example: Traditional Error Handling

```
#include <limits>
#include <vector>
#include <iostream>

std::pair<bool, int> safe_add(int x, int y) {
    return ((y > 0 && x > std::numeric_limits<int>::max() - y)
      || (y < 0 && x < std::numeric_limits<int>::min() - y)) ?
      std::make_pair(false, 0) : std::make_pair(true, x + y);
}

int main() {
    constexpr int int_min = std::numeric_limits<int>::min();
    constexpr int int_max = std::numeric_limits<int>::max();
    std::vector<std::pair<int, int>> v{
        {int_max, int_max}, {1, 2}, {int_min, int_min},
        {int_max, int_min}, {int_min, int_max}
    };
    for (auto x : v) {
        auto result = safe_add(x.first, x.second);
        if (result.first) {
            std::cout << result.second << '\n';
        } else {
            std::cout << "overflow\n";
        }
    }
}
```

safe_add Example: Exceptions

```
#include <limits>
#include <vector>
#include <iostream>
#include <stdexcept>

int safe_add(int x, int y) {
    return ((y > 0 && x > std::numeric_limits<int>::max() - y)
      || (y < 0 && x < std::numeric_limits<int>::min() - y)) ?
      throw std::overflow_error("addition") : x + y;
}

int main() {
    constexpr int int_min = std::numeric_limits<int>::min();
    constexpr int int_max = std::numeric_limits<int>::max();
    std::vector<std::pair<int, int>> v{
        {int_max, int_max}, {1, 2}, {int_min, int_min},
        {int_max, int_min}, {int_min, int_max}
    };
    for (auto x : v) {
        try {
            int result = safe_add(x.first, x.second);
            std::cout << result << '\n';
        }
        catch (const std::overflow_error&) {
            std::cout << "overflow\n";
        }
    }
}
```

Section 3.3.11

References

1. D. Abrahams. Exception-safety in generic components.
In *Lecture Notes in Computer Science*, volume 1766, pages 69–79. Springer, 2000.
A good tutorial on exception safety by an expert on the subject.

2. T. Cargill. Exception handling: A false sense of security.
C++ Report, 6(9), Nov. 1994.
Available online at http://ptgmedia.pearsoncmg.com/images/020163371x/supplements/Exception_Handling_Article.html.
An early paper that first drew attention to some of the difficulties in writing exception-safe code.

3. Exception-Safe Coding in C++, http://exceptionsafecode.com, 2014.

4. V. Kochhar, How a C++ Compiler Implements Exception Handling, http://www.codeproject.com/Articles/2126/How-a-C-compiler-implements-exception-handling, 2002.

References II

5. C++ FAQ — Exceptions and Error Handling, https://isocpp.org/wiki/faq/exceptions, 2016.

Talks I

1. Jon Kalb. Exception-Safe Code, CppCon, Bellevue, WA, USA, Sep 7–12, 2014. Available online at https://youtu.be/W7fIy_54y-w, https://youtu.be/b9xMIKb1jMk, and https://youtu.be/MiKxfdkMJW8. (This talk is in three parts.)
2. Jon Kalb. Exception-Safe Coding, C++Now, Aspen, CO, USA, May 13–18, 2012. Available online at https://youtu.be/N9bR0ztmmEQ and https://youtu.be/UiZfODgB-Oc. (This talk is in two parts.)

Section 3.4

Rvalue References

Section 3.4.1

Introduction

Motivation Behind Rvalue References

- Rvalue references were added to the language in C++11 in order to provide support for:
  1. move operations; and
  2. perfect forwarding.
- A move operation is used to propagate the value from one object to another, much like a copy operation, except that a move operation makes fewer guarantees, allowing for greater efficiency and flexibility in many situations.
- Perfect forwarding relates to being able to pass function arguments from a template function through to another function (called by the template function) while preserving certain properties of those arguments.

Terminology: Named and Cv-Qualified

- A type that includes one or both of the qualifiers **`const`** and **`volatile`** is called a **cv-qualified type**.
- A type that is not cv-qualified is called **cv-unqualified**.
- Example:
  The types **`const int`** and **`volatile char`** are cv-qualified.
  The types **`int`** and **`char`** are cv-unqualified.
- An object or function that is named by an identifier is said to be **named**.
- An object or function that cannot be referred to by name is said to be **unnamed**.
- Example:

```
std::vector<int> v = {1, 2, 3, 4};
std::vector<int> w;
w = v; // w and v are named
w = std::vector<int>(2, 0);
  // w is named
  // std::vector<int>(2, 0) is unnamed
```

Section 3.4.2

Copying and Moving

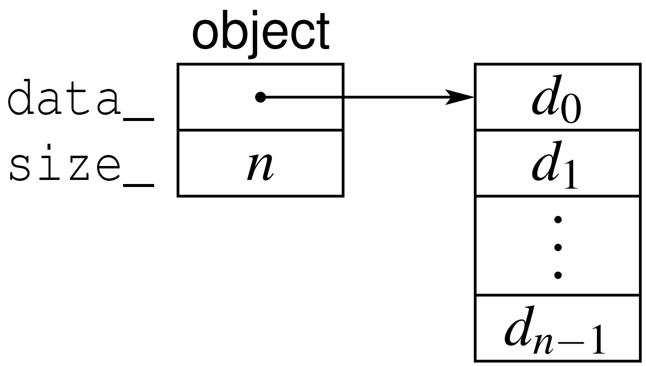
Propagating Values: Copying and Moving

- Suppose that we have two objects of the same type and we want to propagate the value of one object (i.e., the source) to the other object (i.e., the destination).
- This can be accomplished in one of two ways:
  1. copying; or
  2. moving.
- **Copying** propagates the value of the source object to the destination object ***without modifying the source object***.
- **Moving** propagates the value of the source object to the destination object and is ***permitted to modify the source object***.
- Moving is always at least as efficient as copying, and for many types, moving is ***more efficient*** than copying.
- For some types, ***copying does not make sense***, while moving does (e.g., `std::ostream`, `std::istream`).

- Consider a class that represents a one-dimensional array.

```
template <class T>
class Vector {
public:
    // ...
private:
    T* data_; // pointer to element data
              // (allocated with new)
    unsigned int size_; // number of elements
};
```

- Pictorially, the data structure looks like the following:



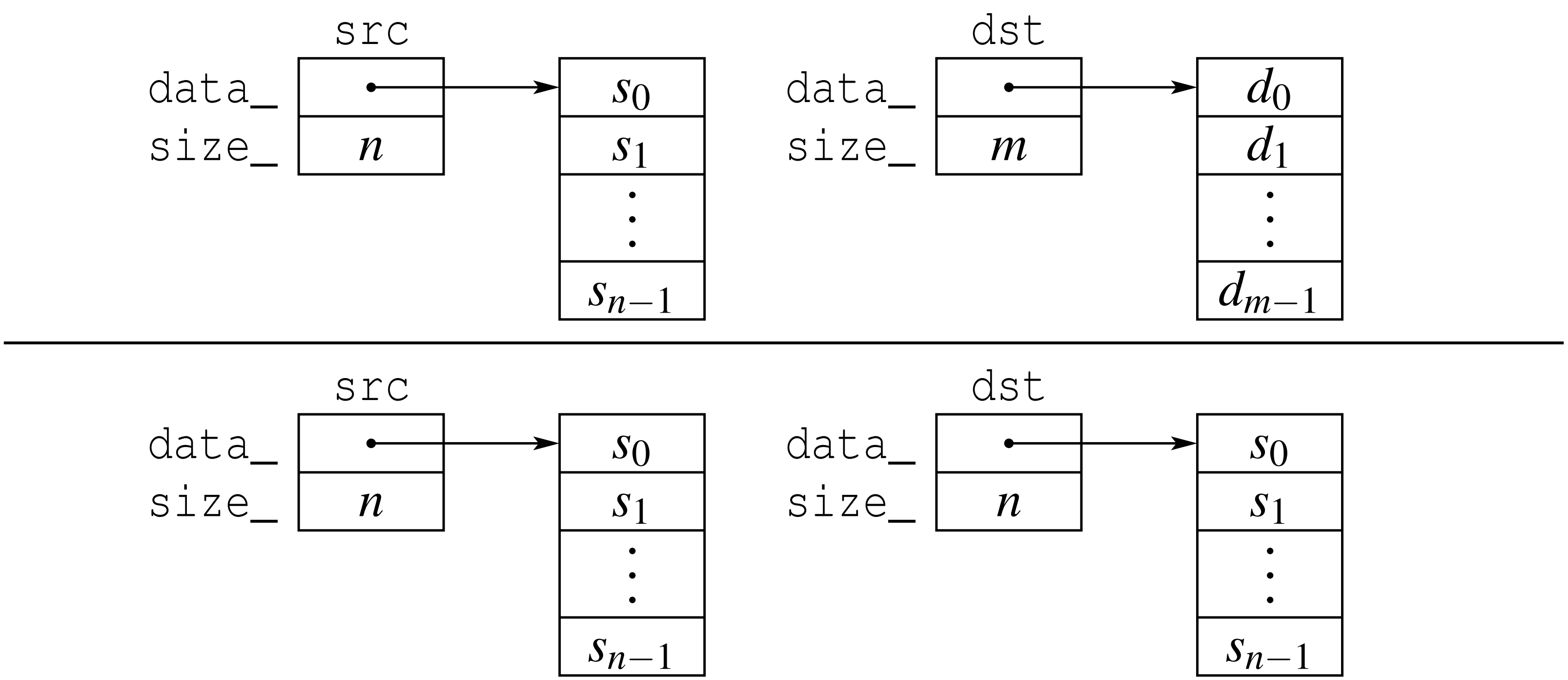
- How would copying be implemented?
- How would moving be implemented?

Vector Example: Copying

- code for copying from source `src` to destination `dst` (not self assignment):

```
delete [] dst.data_;
dst.data_ = new T[src.size_];
dst.size_ = src.size_;
std::copy_n(src.data_, src.size_, dst.data_);
```

- copying requires: one array delete (destruction, memory deallocation), one array new (memory allocation, construction), copying of element data (copy assignment, etc.), and updating `data_` and `size_` data members
- copying proceeds as follows:

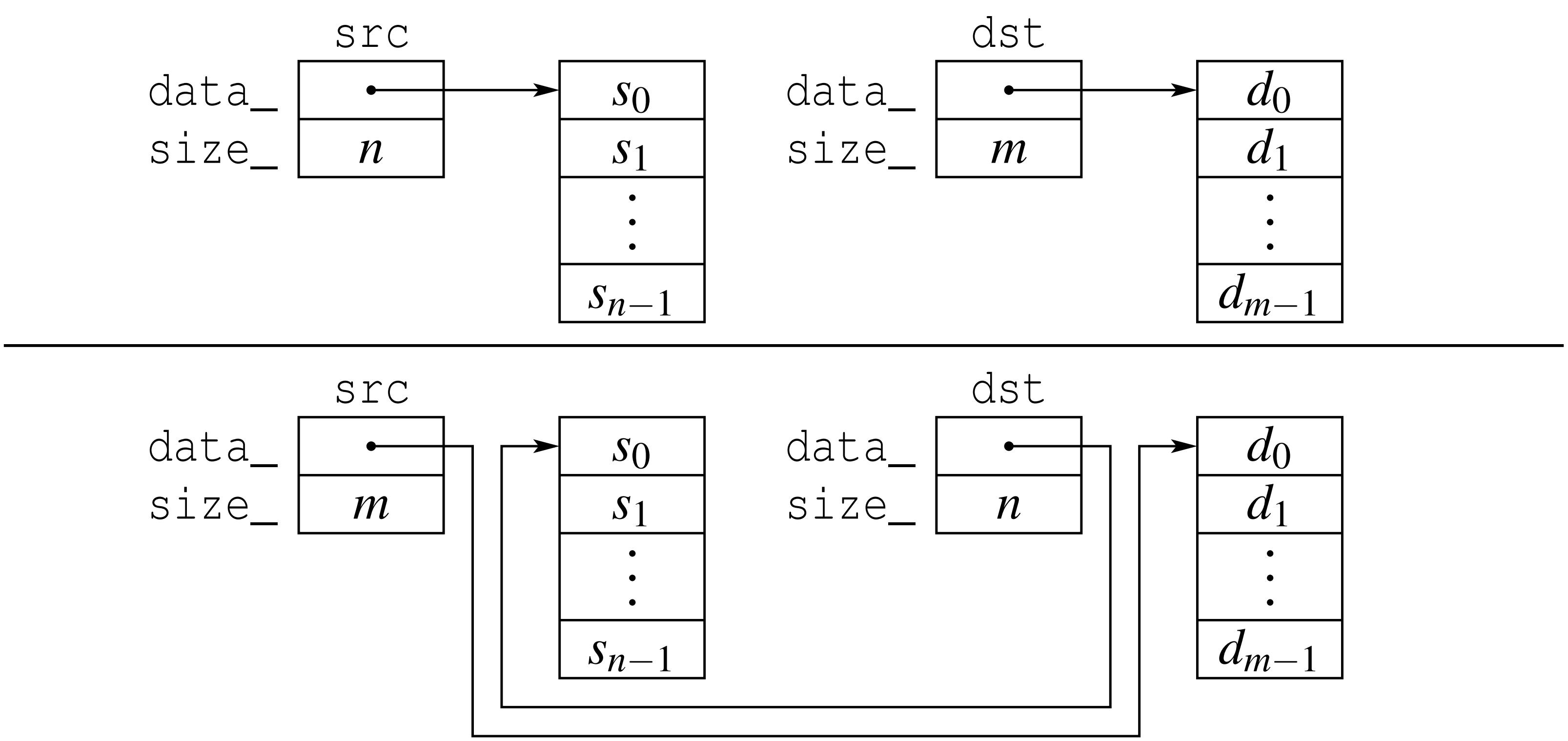


Vector Example: Moving

- code for moving from source `src` to destination `dst`:

```
std::swap(src.data_, dst.data_);
std::swap(src.size_, dst.size_);
```

- moving only requires updating `data_` and `size_` data members
- although not considered here, could also free data array associated with `src` if desirable to release memory as soon as possible
- moving proceeds as follows:



Moving Versus Copying

- Moving is usually more efficient than copying, often by very large margin.
- So, we should prefer moving to copying.
- We can safely replace a copy by a move when subsequent code does not depend on the value of source object.
- It would be convenient if the language could provide a mechanism for automatically using a move (instead of a copy) in situations where doing so is always guaranteed to be safe.
- For reasons of efficiency, it would also be desirable for the language to provide a mechanism whereby the programmer can override the normal behavior and force a move (instead of a copy) in situations where such a transformation is known to be safe only due to some special additional knowledge about program behavior.
- Rvalue references provide the above mechanisms.

Section 3.4.3

References and Expressions

References

- A **reference** is an alias (i.e., nickname) for an already existing object.
- The language has two kinds of references:
  1. lvalue references
  2. rvalue references
- An **lvalue reference** is denoted by `&` (often read as “ref”).

```
int i = 5;
int& j = i; // j is lvalue reference to int
const int& k = i; k is lvalue reference to const int
```

- An **rvalue reference** is denoted by `&&` (often read as “ref ref”).

```
int&& i = 5; // i is rvalue reference to int
const int&& j = 17; // j is rvalue reference to const int
```

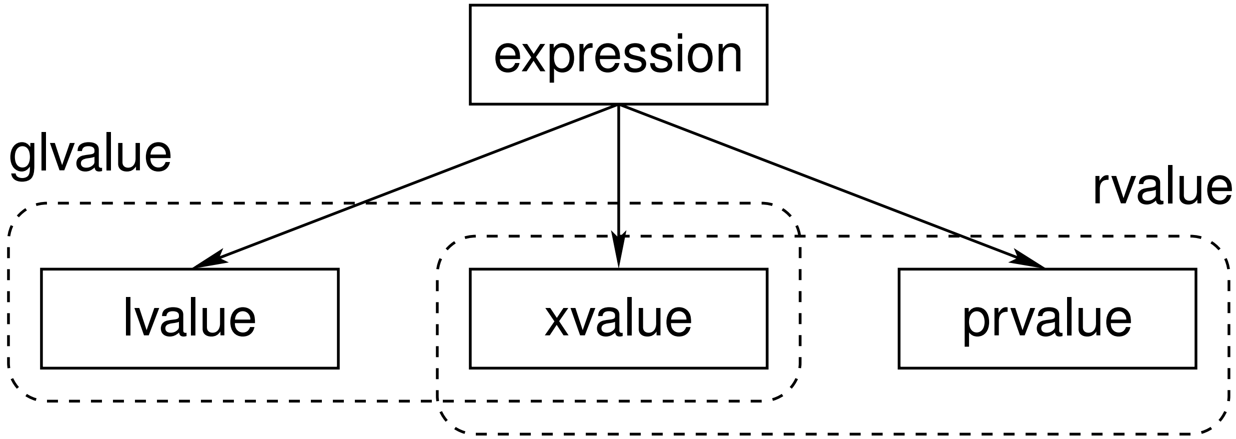
- The act of initializing a reference is known as **reference binding**.
- Lvalue and rvalues references differ only in their properties relating to:
  - reference binding; and
  - overload resolution.

- An **expression** is a sequence of operators and operands that specifies a computation.
- An expression has a type and, if the type is not void, a value.
- Example:

```
int x = 0;
int y = 0;
int* p = &x;
double d = 0.0;
// Evaluate some
// expressions here.
```

Expression	Type	Value
`x`	`int`	`0`
`y = x`	`int&`	reference to `y`
`x + 1`	`int`	`1`
`x * x + 2 * x`	`int`	`0`
`y = x * x`	`int&`	reference to `y`
`x == 42`	`bool`	`false`
`*p`	`int&`	reference to `x`
`p == &x`	`bool`	`true`
`x > 2 * y`	`bool`	`false`
`std::sin(d)`	`double`	`0.0`

Categories of Expressions



- Every expression can be classified into ***exactly one*** of the three following categories:
  1. lvalue
  2. prvalue (pure rvalue)
  3. xvalue (expiring value)
- An expression that is an lvalue or xvalue is called a **glvalue** (generalized lvalue).
- An expression that is a prvalue or an xvalue is called an **rvalue**.
- Every expression is either an lvalue or an rvalue (but not both).
- Whether or not it is safe to move (instead of copy) depends on whether an lvalue or rvalue is involved.

Lvalues

- An **lvalue** is an expression that:
  - designates a function or object ; and
  - has an identity (i.e., occupies some *identifiable* location in memory and therefore, in principle, can have its address taken).
- *Named objects* and *named functions* are lvalues. Example:

```
int getValue();
int i = 0;
const int j = 1;
i = j + 1; // i and j are lvalues
getValue(); // getValue is lvalue [Note: not getValue()]
```

- *Dereferenced pointer.* If `e` is an expression of pointer type, then `*e` is an lvalue. Example:

```
char buffer[] = "Hello";
char* s = buffer;
*s = 'a'; // *s is lvalue
*(s + 1) = 'b'; // *(s + 1) is lvalue
```

Lvalues (Continued)

- The result of calling a function whose ***return type is an lvalue reference type*** is an lvalue. Example:

```
std::vector<int> v = {{1, 2, 3}};
// int& std::vector<int>::operator[](int);
int i = v[0]; // v[0] is lvalue
```

- A ***string literal*** is an lvalue. Example: `"Hello World"`
- ***Named rvalue references*** are lvalues. Example:

```
int&& i = 1 + 3;
int j = i; // i is lvalue
```

- Rvalue references to functions (both named and unnamed) are lvalues.

Moving and Lvalues

- Using a move (instead of a copy) is ***not guaranteed to be safe*** when the source is an lvalue (since other code can access the associated object by name or through a pointer or reference).
- Example:

```
Vector<int> x;
Vector<int> y(x);
  // can we construct by moving (instead of copying)?
  // source x is lvalue
  // not safe to move x to y since value of x
  // might be used below
y = x;
  // can we assign by moving (instead of copying)?
  // source x is lvalue
  // not safe to move x to y since value of x
  // might be used below
```

Prvalues

- A **prvalue** (pure rvalue) is an expression that:
  - is a temporary object or subobject thereof, or a value that is not associated with an object; and
  - does not have an identity.
- A prvalue is a kind of rvalue.
- ***Temporary objects*** are prvalues. Example:

```
std::vector<int> v;
v = std::vector<int>(10, 2);
    // std::vector<int>(10, 2) is prvalue
std::complex<double> u;
u = std::complex<double>(1, 2);
    // std::complex<double>(1, 2) is prvalue
```

- A function call whose ***return type is not a reference type*** is a prvalue. Example:

```
int func();
int i = func(); // func() is prvalue
```

Prvalues (Continued)

- All ***literals other than string literals*** are prvalues. Examples:

```
double pi = 3.1415; // 3.1415 is prvalue
int i = 42; // 42 is prvalue
i = 2 * i + 1; // 2 and 1 are prvalues
char c = 'A'; // 'A' is prvalue
```

- The result yielded by certain built-in operators (e.g., +, -) is a prvalue. Example:

```
int i, j;
i = 3 + 5; // 3 + 5 is prvalue
j = i * i; // i * i is prvalue
```

- The **`this`** keyword is a prvalue expression.
- Prvalues need not have any storage associated with them.
- Not requiring prvalue expressions to have storage gives the compiler more freedom in generating code for such expressions.

```
int i = 2;
  // 2 is prvalue and need not ever be stored in memory
```

Moving and Prvalues

- Using a move (instead of a copy) is ***always safe*** when the source is a prvalue (since the prvalue cannot correspond to an object with an identity).
- Example (move from temporary object):

```
Vector<int> getVector();
Vector<int> x;
Vector<int> y(getVector());
  // can we construct by moving (instead of copying)?
  // source getVector() is prvalue
  // safe to move since temporary object could not be
  // used below
x = getVector();
  // can we assign by moving (instead of copying)?
  // source getVector() is prvalue
  // safe to move since temporary object could not be
  // used below
```

Xvalues

- An **xvalue** (expiring value) is an expression that:
  - refers to an object (usually near the end of its lifetime);
  - has an identity; and
  - is ***deemed to be safe*** to use as the source for a move.
- An xvalue is a kind of rvalue.
- An xvalue is the result of certain kinds of expressions involving rvalue references.
- The result of calling a function whose ***return type is an rvalue reference type*** is an xvalue. Example:

```
std::string s("Hello");
std::string t = std::move(s); // std::move(s) is xvalue
```

- In the above example, the template function `std::move` converts its argument to an xvalue (since it returns an rvalue reference type).
- ***Unnamed rvalue references to objects*** are xvalues.

```
std::string s("Hello");
std::string t;
t = static_cast<std::string&&>(s);
  // static_cast<std::string&&>(s) is xvalue
```

Moving and Xvalues

- Using a move (instead of a copy) is ***deemed to be safe*** when the source is an xvalue.
- Example (forced move):

```
Vector<int> v(100, 5);
Vector<int> u(200, -1);
for (auto i : v) std::cout << i << '\n';
for (auto i : u) std::cout << i << '\n';
v = std::move(u);
    // std::move(u) is xvalue
    // safe to force move since later code does
    // not to use value of u
for (auto i : v) std::cout << i << '\n';
// later code known not to use value of u
```

- The function `std::move` only allows for an object to be treated as if it were safe to use as source of a move, but does not perform a move.

- With regard to propagating the value from one object to another, we can summarize the preceding results as given below.
- If the source is an ***rvalue*** (i.e., prvalue or xvalue), using a move instead of a copy is ***always safe***.
- If the source is an ***lvalue***, using a move instead of a copy is ***not guaranteed to be safe***.
- It would be highly desirable if the language would provide a mechanism that would automatically allow a move to be used in the rvalue case and a copy to be employed otherwise.
- In fact, this is exactly what the language does.

More on Lvalues and Rvalues

- Lvalues and rvalues can be either ***modifiable or nonmodifiable***. Example:

```
int i = 0;
const int j = 2;
i = j + 3;
  // i is modifiable lvalue
  // j is nonmodifiable lvalue
  // j + 3 is modifiable rvalue
const std::string getString();
std::string s = getString();
  // getString() is nonmodifiable rvalue
```

- Class rvalues can have cv-qualified types, while non-class rvalues ***always have cv-unqualified types***. Example:

```
const int getConstInt(); // const is ignored
const std::string getConstString();
int i = getConstInt();
  // getConstInt() is modifiable rvalue of type int
  // (not const int)
std::string s = getConstString();
  // getConstString() is nonmodifiable rvalue
```

Exercise: Expressions

```cpp
#include <iostream>
#include <string>
#include <utility>

std::string&& func1(std::string& x) {
    return std::move(x);
    // x? std::move(x)?
}

int main() {
    const std::string hello("Hello");
    std::string a;
    std::string b;

    a = hello + "!";
    // hello? hello + "!"? a = hello + "!"?
    std::cout << a << '\n';
    // std::cout? std::cout << a?

    a = std::string("");
    // std::string("")? a = std::string("")?
    ((a += hello) += "!");
    // a += hello?
    b = func1(a);
    // func1(a)? b = func1(a)?
    std::cout << b << '\n';
}
```

Built-In Operators, Rvalues, and Lvalues

- Aside from the exceptions noted below, all of the built-in operators ***require operands that are rvalues***.
- The operand of each of the following built-in operators must be an lvalue:
  - address of (i.e., unary `&`),
  - prefix and postfix increment (i.e., `++`),
  - prefix and postfix decrement (i.e., `--`)
- The left operand of the following built-in operators must be an lvalue:
  - assignment (i.e., `=`)
  - compound assignment (e.g., `+=`, `-=`, `*=`, `/=`, etc.)
- Aside from the exceptions noted below, all of the built-in operators ***yield a result that is an rvalue***.
- The following operators yield a result that is an lvalue:
  - subscript (i.e., `[]`)
  - dereference (i.e., unary `*`)
  - assignment (i.e., `=`) and compound assignment (e.g., `+=`, `-=`, etc.)
  - prefix increment (i.e., `++`) and prefix decrement (i.e., `--`)
  - function call (i.e., `()`) invoking a function that returns a reference type
  - cast to reference type

Operators, Lvalues, and Rvalues

- Whether an operator for a ***class type*** requires operands that are lvalues or rvalues or yield lvalues or rvalues is determined by the parameter types and return type of the operator function.
- The member selection operator may yield an lvalue or rvalue, depending on the particular manner in which the operator is used. (The behavior is fairly intuitive.)
- The lvalue/rvalue-ness and type of the result produced by the ternary conditional operator depends on the particular manner in which the operator is employed.

Implicit Lvalue-to-Rvalue Conversion

- An implicit conversion from lvalues to rvalues is provided, which can be used in most (but not all) circumstances.
- Example:

```
int i = 1;
int j = 2;
int k = i + j;
  // operands of + must be rvalues
  // i and j converted to rvalues
```

Section 3.4.4

Reference Binding and Overload Resolution

References: Binding and Overload Resolution

- The kinds of expressions, to which lvalue and rvalue references can ***bind***, differ.
- For a nonreference type `T` (such as **`int`** or **`const int`**), what kinds of expressions can validly be placed in each of the boxes in the example below?

```
T& r = [          ] ;
T&& r = [          ] ;
```

- Lvalue and rvalue references also behave differently with respect to ***overload resolution***.
- Let `T` be a cv-unqualified nonreference type. Which overloads of `func` will be called in the example below?

```
T operator+(const T&, const T&);
void func(const T&);
void func(T&&);
T x;
func(x); // calls which version of func?
func(x + x); // calls which version of func?
```

Reference Binding

- Implicit lvalue-to-rvalue conversion is disabled when binding to references.
- An lvalue reference can bind to an lvalue as long as doing so would not result in the *loss* of any cv qualifiers.

```
const int i = 0;
int& r1 = i; // ERROR: drops const
const int& r2 = i; // OK
const volatile int& r3 = i; // OK
```

- The loss of cv qualifiers must be avoided for ***const and volatile correctness***.
- Similarly, an rvalue reference can bind to an rvalue as long as doing so would not result in the *loss* of any cv qualifiers.

```
const std::string getValue();
std::string&& r1 = getValue(); // ERROR: drops const
const std::string&& r2 = getValue(); // OK
```

- Again, the loss of cv qualifiers must be avoided for ***const and volatile correctness***.

Reference Binding (Continued)

- An lvalue reference can be bound to an rvalue only if doing so would not result in the ***loss*** of any cv qualifier and the lvalue reference is ***const***.

```
const std::string getConstValue();
std::string& r1 = getConstValue(); // ERROR: drops const
const std::string& r2 = getValue(); // OK
int& ri1 = 42; // ERROR: not const reference
const int& ri2 = 42; // OK
```

- The requirement that the lvalue reference be const is to prevent temporary objects from being modified in a very uncontrolled manner, which can lead to subtle bugs.
- An rvalue reference can ***never*** be bound to an lvalue.

```
int i = 0;
int&& r1 = i; // ERROR: cannot bind to lvalue
int&& r2 = 42; // OK
```

- Allowing rvalue reference to bind to lvalues would violate the principle of type-safe overloading , which can lead to subtle bugs.

Why Rvalue References Cannot Bind to Lvalues

- In effect, rvalue references were introduced into the language to allow a function to know if one of its reference parameters is bound to an object whose value is safe to change without impacting other code, namely, an rvalue (i.e., a temporary object or xvalue).
- Since an rvalue reference can only bind to an rvalue, any rvalue reference parameter to a function is ***guaranteed*** to be bound to a temporary object or xvalue.
- Example:

```
class Thing {
public:
    // Move constructor
    // parameter x known to be safe to use as source for move
    Thing(Thing&& x);
    // Move assignment operator
    // parameter x known to be safe to use as source for move
    Thing& operator=(Thing&& x);
    // ...
};
// parameter x known to be safe to modify
void func(Thing&& x);
```

- If rvalue references could bind to lvalues, the above guarantee could not be made, as an rvalue reference could then refer to an object whose value cannot be changed safely, namely, an lvalue.

Why Non-Const Lvalue References Cannot Bind to Rvalues

- If non-const lvalue references could bind to rvalues, temporary objects could be modified in many undesirable circumstances.

```
void func(int& x) {
    // ...
}

int main() {
    int i = 1;
    int j = 2;
    func(i + j);
      // ERROR: cannot bind non-const lvalue
      // reference to rvalue
      // What would be consequence if allowed?
}
```

Reference Binding Summary

	Rvalue				Lvalue			
	`T`	`const T`	`volatile T`	`const volatile T`	`T`	`const T`	`volatile T`	`const volatile T`
`T&&`	✓	C	V	C,V	✗	✗	✗	✗
`const T&&`	✓	✓	V	V	✗	✗	✗	✗
`volatile T&&`	✓	✗	✓	C	✗	✗	✗	✗
`const volatile T&&`	✓	✓	✓	✓	✗	✗	✗	✗
`T&`	✗	✗	✗	✗	✓	C	V	C,V
`const T&`	✓	✓	V	V	✓	✓	V	V
`volatile T&`	✗	✗	✗	✗	✓	C	✓	C
`const volatile T&`	✗	✗	✗	✗	✓	✓	✓	✓

✓: allowed   C: strips **`const`**   V: strips **`volatile`**   ✗: other

Reference Binding Example

```
#include <string>
using std::string;

string value() {
    return string("Hello");
}

const string constValue() {
    return string("World");
}

int main() {
    string i("mutable");
    const string j("const");

    string& r01 = i;
    string& r02 = j; // ERROR: drops const
    string& r03 = value(); // ERROR: non-const lvalue reference from rvalue
    string& r04 = constValue(); // ERROR: non-const lvalue reference from rvalue

    const string& r05 = i;
    const string& r06 = j;
    const string& r07 = value();
    const string& r08 = constValue();

    string&& r09 = i; // ERROR: rvalue reference from lvalue
    string&& r10 = j; // ERROR: rvalue reference from lvalue
    string&& r11 = value();
    string&& r12 = constValue(); // ERROR: drops const

    const string&& r13 = i; // ERROR: rvalue reference from lvalue
    const string&& r14 = j; // ERROR: rvalue reference from lvalue
    const string&& r15 = value();
    const string&& r16 = constValue();
}
```

Overload Resolution

- Lvalues strongly prefer binding to lvalue references.
- Rvalues strongly prefer binding to rvalue references.
- Modifiable lvalues and rvalues weakly prefer binding to non-const references.

Overload Resolution Summary

	Priority							
	Rvalue				Lvalue			
	`T`	`const T`	`volatile T`	`const volatile T`	`T`	`const T`	`volatile T`	`const volatile T`
`T&&`	1							
`const T&&`	2	1						
`volatile T&&`	2		1					
`const volatile T&&`	3	2	2	1				
`T&`					1			
`const T&`	4	3			2	1		
`volatile T&`					2		1	
`const volatile T&`					3	2	2	1

Overloading Example 1

```
#include <iostream>
#include <string>

void func(std::string& x) {
    std::cout << "func(std::string&) called\n";
}

void func(const std::string& x) {
    std::cout << "func(const std::string&) called\n";
}

void func(std::string&& x) {
    std::cout << "func(std::string&&) called\n";
}

void func(const std::string&& x) {
    std::cout << "func(const std::string&&) called\n";
}

const std::string&& constValue(const std::string&& x) {
    return static_cast<const std::string&&>(x);
}

int main() {
    const std::string cs("hello");
    std::string s("world");
    func(s);
    func(cs);
    func(cs + s);
    func(constValue(cs + s));
}

/* Output:
func(std::string&) called
func(const std::string&) called
func(std::string&&) called
func(const std::string&&) called
*/
```

Overloading Example 2

```cpp
#include <iostream>
#include <string>

void func(const std::string& x) {
    std::cout << "func(const std::string&) called\n";
}

void func(std::string&& x) {
    std::cout << "func(std::string&&) called\n";
}

const std::string&& constValue(const std::string&& x) {
    return static_cast<const std::string&&>(x);
}

int main() {
    const std::string cs("hello");
    std::string s("world");
    func(s);
    func(cs);
    func(cs + s);
    func(constValue(cs + s));
}

/* Output:
func(const std::string&) called
func(const std::string&) called
func(std::string&&) called
func(const std::string&) called
*/
```

- If an rvalue reference could bind to an lvalue, this would violate the principle of type-safe overloading.

```
#include <iostream>
#include <string>

template <class T>
class Container {
public:
    // ...
    // Forget to provide the following function:
    // void push_back(const T& value); // Copy semantics
    void push_back(T&& value); // Move semantics
private:
    // ...
};

int main() {
    std::string s("Hello");
    Container<std::string> c;
    // What would happen here if lvalues
    // could bind to rvalue references?
    c.push_back(s);
    std::cout << s << '\n';
}
```

Section 3.4.5

Moving

Move Constructors

- A non-template constructor for class `T` is a **move constructor** if it can be called with one parameter that is of type `T&&`, `const T&&`, `volatile T&&`, or `const volatile T&&`.
- Example (assuming no optimization):

```
struct T {
    T();
    T(const T&); // copy constructor
    T(T&&); // move constructor
};
T func(int);

T a(func(1)); // calls T::T(T&&)
T b = a; // calls T::T(const T&)
```

Move Assignment Operators

- A **move assignment operator** `T::operator=` is a non-static non-template member function of class `T` with exactly one parameter of type `T&&`, `const T&&`, `volatile T&&`, or `const volatile T&&`.
- Example (assuming no optimization):

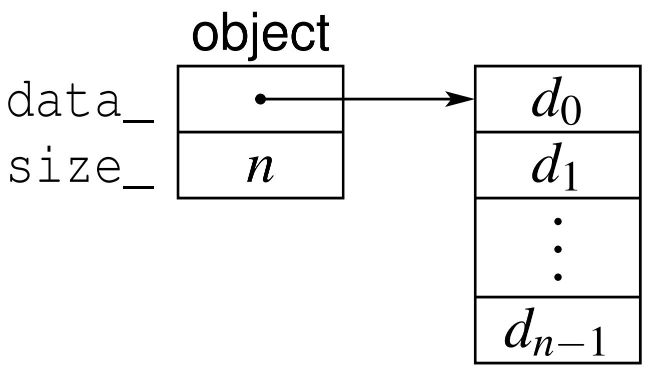
```
class T {
public:
    T();
    T(const T&); // copy constructor
    T(T&&); // move constructor
    T& operator=(const T&); // copy assignment operator
    T& operator=(T&&); // move assignment operator
    // ...
};
T func(int);

T a;
T b;
a = func(1); // calls T::operator=(T&&)
b = a; // calls T::operator=(const T&)
```

Vector Example Revisited

- Recall the class from earlier that represents a one-dimensional array.

```
template <class T>
class Vector {
public:
    // ...
private:
    T* data_; // pointer to element data
              // (allocated with new)
    unsigned int size_; // number of elements
};
```

- Pictorially, the data structure looks like the following:



Example Without Move Construction/Assignment

```
#include <algorithm>
#include <complex>

template <class T>
class Vector {
public:
    Vector(unsigned int size, T value = 0) : data_(new T[size]), size_(size)
      {std::fill_n(data_, size, value);}
    Vector(const Vector& a) : data_(new T[a.size_]), size_(a.size_)
      {std::copy_n(a.data_, a.size_, data_);}
    Vector& operator=(const Vector& a) {
        if (this != &a) {
            delete[] data_; size_ = a.size_; data_ = new T[a.size_];
            std::copy_n(a.data_, a.size_, data_);
        }
        return *this;
    }
    ~Vector() {delete[] data_;}
    // ...
private:
    T* data_; // pointer to element data
    unsigned int size_; // number of elements
};
using Vec = Vector<std::complex<double>>;
Vec getVector() {return Vec(1000, {0.0, 1.0});}

int main() {
    Vec v(0);
    Vec w = getVector(); // construct from temporary object
    v = Vec(2000, {1.0, 2.0}); // assign from temporary object
}
```

Example With Move Construction/Assignment

```
#include <algorithm>
#include <complex>

template <class T>
class Vector {
public:
    Vector(unsigned int size, T value = 0) : data_(new T[size]), size_(size)
      {std::fill_n(data_, size, value);}
    Vector(const Vector& a) : data_(new T[a.size_]), size_(a.size_)
      {std::copy_n(a.data_, a.size_, data_);}
    Vector& operator=(const Vector& a) {
        if (this != &a) {
            delete[] data_; size_ = a.size_; data_ = new T[a.size_];
            std::copy_n(a.data_, a.size_, data_);
        }
        return *this;
    }
    // Move constructor
    Vector(Vector&& a) : data_(a.data_), size_(a.size_)
      {a.size_ = 0; a.data_ = nullptr;}
    // Move assignment operator
    Vector& operator=(Vector&& a) {
        std::swap(size_, a.size_); std::swap(data_, a.data_);
        return *this;
    }
    ~Vector() {delete[] data_;}
    // ...
private:
    T* data_; // pointer to element data
    unsigned int size_; // number of elements
};
using Vec = Vector<std::complex<double>>;
Vec getVector() {return Vec(1000, {0.0, 1.0});}

int main() {
    Vec v(0);
    Vec w = getVector(); // construct from temporary object
    v = Vec(2000, {1.0, 2.0}); // assign from temporary object
}
```

Allowing Move Semantics in Other Contexts via `std::move`

- As we have seen, a reference parameter of a function that is bound to modifiable rvalue can be modified safely (i.e., no observable change in behavior outside of function).
- Sometimes may want to allow a move to be used instead of a copy, when this would not normally be permitted.
- We can allow moves by casting to a non-const rvalue reference.
- This casting can be accomplished by `std::move`, which is declared (in the header file `utility`) as:

```
template <class T>
constexpr typename std::remove_reference<T>::type&&
  move(T&&) noexcept;
```

- For an object `x` of type `T`, `std::move(x)` is similar to `static_cast<T&&>(x)` but saves typing and still works correctly when `T` is a reference type (a technicality yet to be discussed).

Old-Style Swap

- Prior to C++11, a swap function (such as `std::swap`) would typically look like this:

```
template <class T>
void swap(T& x, T& y) {
    T tmp(x); // copy x to tmp
    x = y;   // copy y to a
    y = tmp; // copy tmp to y
}
```

- In the above code, a swap requires three ***copy*** operations (namely, one copy constructor call and two copy assignment operator calls).
- For many types `T`, this use of copying is ***very inefficient***.
- Furthermore, the above code requires that `T` ***must be copyable*** (i.e., `T` has a copy constructor and copy assignment operator).
- In C++11, we can write a much better swap function.

Improved Swap

- As of C++11, a swap function would typically look like this:

```
template <class T>
void swap(T& x, T& y) {
    T tmp(std::move(x)); // move x to tmp
    x = std::move(y);  // move y to x
    y = std::move(tmp); // move tmp to y
}
```

- The function `std::move` casts its argument to an rvalue reference.
- Assuming that `T` provides a move constructor and move assignment operator, a swap requires three ***move*** operations (i.e., one move constructor call and two move assignment operator calls) and ***no copying***.
- The use of `std::move` above is essential in order for copying to be avoided.

Moving Versus Copying Example

```cpp
#include <iostream>
#include <utility>

class Widget {
public:
    Widget() {}
    ~Widget() {}
    Widget(Widget&&) {std::cout << "move construct\n";}
    Widget(const Widget&) {std::cout << "copy construct\n";}
    Widget& operator=(Widget&&)
      {std::cout << "move assign\n"; return *this;}
    Widget& operator=(const Widget&)
      {std::cout << "copy assign\n"; return *this;}
    // ...
};

Widget make_widget_1() {
    return Widget(); // NOTE: Returns temporary.
}

Widget make_widget_2() {
    Widget w;
    return w; // NOTE: Returns named object.
}

int main() {
    Widget a;
    Widget b(a); // copy construct
    Widget c(std::move(b)); // move construct
    Widget d(make_widget_1()); // guaranteed copy/move elision
    Widget e(make_widget_2()); // move construct if no NRVO
    c = a; // copy assign
    b = std::move(c); // move assign
    a = make_widget_1(); // move assign
    b = make_widget_2(); // move construct if no NRVO; move assign
}
```

Implication of Rvalue-Reference Type Function Parameters

- Due to the properties of rvalue references, a function parameter of rvalue-reference type may be regarded as being bound to an object whose value will not be relied upon in the caller.
- Therefore, an object associated with a function parameter of rvalue-reference type can always be safely modified (i.e., without fear of adversely affecting the caller).
- This fact can often be exploited in order to obtain more efficient code.
- Consider the code for a function with the following declaration:

```
void func(std::vector<double>&& x);
```

- Since `x` is of rvalue-reference type, we are guaranteed that the caller will not rely upon the value of the object referenced by `x`.
- If obliterating the value of `x` would allow us to more efficiently implement `func`, we can safely do so.
- For example, we could safely modify `x` in place or move from it, without fear of adversely affecting the caller.

Reference-Qualified Member Functions

- every nonstatic member function has implicit parameter `*this`
- possible to provide reference qualifiers for implicit parameter
- allows overloading member functions on lvalueness/rvalueness of `*this`
- cannot mix reference qualifiers and non-reference qualifiers in single overload set
- provides mechanism for treating lvalue and rvalue cases differently
- useful for facilitating move semantics or preventing operations not appropriate for lvalues or rvalues

Reference-Qualified Member Functions Example

```cpp
#include <iostream>

class Widget {
public:
    void func() const &
      {std::cout << "const lvalue\n";}
    void func() &
      {std::cout << "non-const lvalue\n";}
    void func() const &&
      {std::cout << "const rvalue\n";}
    void func() &&
      {std::cout << "non-const rvalue\n";}
};

const Widget getConstWidget() {return Widget();}

int main(){
    Widget w;
    const Widget cw;
    w.func(); // non-const lvalue
    cw.func(); // const lvalue
    Widget().func(); // non-const rvalue
    getConstWidget().func(); // const rvalue
}
```

```
class Int {
public:
    Int(int x = 0) : value_(x) {}
    // only allow prefix increment for lvalues
    Int& operator++() & {++value_; return *this;}
    // The following allows prefix increment for rvalues:
    // Int& operator++() {++value_; return *this;}
    // ...
private:
    int value_;
};

int one() {return 1;}

int main() {
    int i = 0;
    int j = ++i; // OK
    // int k = ++one(); // ERROR (not lvalue)
    Int x(0);
    Int y = ++x; // OK
    // Int z = ++Int(1); // ERROR (not lvalue)
}
```

Move Semantics and the `*this` Parameter

```
#include <iostream>
#include <vector>
#include <utility>

class Buffer {
public:
    Buffer(char value = 0) : data_(1024, value) {}
    void data(std::vector<char>& x) const &
      {x = data_;}
    void data(std::vector<char>& x) &&
      {x = std::move(data_);}
    // ...
private:
    std::vector<char> data_;
};

Buffer getBuffer() {return Buffer(42);}

int main() {
    std::vector<char> d;
    Buffer buffer;
    buffer.data(d); // copy into d
    getBuffer().data(d); // move into d
}
```

Section 3.4.6

Reference Collapsing and Forwarding References

References to References

- A reference to a reference is not allowed, since such a construct clearly makes no sense.

```
int i = 0;
int& & j = i; // ILLEGAL: reference to reference
```

- Although one cannot directly create a reference to a reference, a reference to a reference can arise indirectly in several contexts.
- Typedef name:

```
typedef int& RefToInt;
typedef RefToInt& T; // reference to reference
```

- Template function parameters:

```
template <class T> T func(const T& x) {return x;}
int x = 1;
func<int&>(x); // reference to reference
```

- Decltype specifier:

```
int i = 1;
decltype((i))& j = i; // reference to reference
```

References to References (Continued)

- Auto specifier:

```
int i = 0;
auto&& j = i; // reference to reference
```

- Class templates:

```
template <class T>
struct Thing {
    void func(T&&) {} // reference to reference
                      // if T is reference type
};
Thing<int&> x;
```

- If, during type analysis, a reference to a reference type is obtained, the reference to reference is converted to a simple reference via a process called **reference collapsing**.

Reference Collapsing Rules

- Let `TR` denote a type that is a reference to type `T` (where `T` may be cv qualified).
- The effect of reference collapsing is summarized below. .

Before Collapse	After Collapse
`TR&`	`T&`
`const TR&`	`T&`
`volatile TR&`	`T&`
`const volatile TR&`	`T&`
`TR&&`	`TR`
`const TR&&`	`TR`
`volatile TR&&`	`TR`
`const volatile TR&&`	`TR`

- In other words:
  - An lvalue reference to any reference yields an lvalue reference.
  - An rvalue reference to an lvalue reference yields an lvalue reference.
  - An rvalue reference to an rvalue reference yields rvalue reference.
  - Any cv qualifiers applied to a reference type are discarded (since cv qualifiers cannot be applied to a reference).

Reference Collapsing Examples

- Due to reference collapsing, `T&&` syntax may not always be an rvalue reference. Example:

```
using IntRef = int&;
int i = 0;
IntRef&& r = i; // r is int& (i.e., lvalue reference)
```

- Example:

```
using IntRef = int&;
using IntRefRef = int&&;
using ConstIntRefRef = const int&&;
using ConstIntRef = const int&;
using T1 = const IntRef&;     // T1 is int&
using T2 = const IntRefRef&; // T2 is int&
using T3 = IntRefRef&&;       // T3 is int&&
using T4 = ConstIntRef&&;     // T4 is const int&
using T5 = ConstIntRefRef&&; // T5 is const int&&
```

- Example:

```
int i = 0;
int& j = i;
auto&& k = j;
  // j cannot be inferred to have type int
  // since rvalue reference cannot be bound to lvalue
  // j inferred to have type int&
  // reference collapsing of int& && yields int&
```

Forwarding References

- A ***cv-unqualified*** rvalue reference that appears in a type-deducing context for template parameters is called a **forwarding reference**.
- Type deduction for template parameters of template functions is defined in such a way as to facilitate perfect forwarding.
- Consider the following template-parameter type-deduction scenario:

```
template<class T>
void f(T&& p);

f(expr); // invoke f
```

- Let `E` denote the type of the expression *expr*. The type `T` is then deduced as follows:
  1. If *expr* is an ***lvalue***, `T` is deduced as `E&`, in which case the type of `p` yielded by reference collapsing is `E&`.
  2. If *expr* is an ***rvalue***, `T` is deduced as `E`, in which case `p` will have the type `E&&`.
- Thus, the type `T&&` will be an lvalue reference type if *expr* is an lvalue, and an rvalue reference type if *expr* is an rvalue.
- Therefore, the lvalue/rvalue-ness of *expr* can be determined ***inside*** `f` based on whether `T&&` is an lvalue reference type or rvalue reference type.

Forwarding References Example

```
#include <utility>

template <class T> void f(T&& p);
int main() {
    int i = 42;
    const int ci = i;
    const int& rci = i;
    f(i);
      // i is lvalue with type int
      // T is int&
      // p has type int&
    f(ci);
      // ci is lvalue with type const int
      // T is const int&
      // p has type const int&
    f(rci);
      // rci is lvalue with type const int&
      // T is const int&
      // p has type const int&
    f(2);
      // 2 is rvalue with type int
      // T is int
      // p has type int&&
    f(std::move(i));
      // std::move(i) is rvalue with type int&&
      // T is int
      // p has type int&&
}
```

Section 3.4.7

Perfect Forwarding

Perfect Forwarding

- **Perfect forwarding** is the act of passing a template function's arguments to another function:
  - without rejecting any arguments that can be passed to that other function
  - without losing any information about the arguments' cv-qualifications or lvalue/rvalue-ness; and
  - without requiring overloading.
- In C++03, for example, the best approximations of perfect forwarding turn all rvalues into lvalues and require at least two (and often more) overloads.

Perfect-Forwarding Example

- Consider a ***template*** function `wrapper` and another function `func`, each of which takes one argument.
- Suppose that we want to perfectly forward the argument of `wrapper` to `func`.
- The function `wrapper` is to do nothing other than simply call `func`.
- In doing so, `wrapper` must pass its actual argument through to `func`.
- This must be done in such a way that the argument to `wrapper` and argument to `func` have ***identical properties*** (i.e., match in terms of cv-qualifiers and lvalue/rvalue-ness).
- In other words, the following two function calls must have ***identical behavior***, where *expr* denotes an arbitrary expression:

```
wrapper(expr);
func(expr);
```

- The solution to a perfect-forwarding problem, such as this one, turns out to be more difficult than it might first seem.

Perfect-Forwarding Example: First Failed Attempt

- For our first attempt, we propose the following code for the (template) function `wrapper`:

```
template <class T>
void wrapper(T p) {
    func(p);
}
```

- If `func` takes its parameter by reference, calls to `wrapper` and `func` (with the same argument) can have different behaviors.
- Suppose, for example, that we have the following declarations:

```
void func(int&); // uses pass by reference
int i;
```

- Then, the following two function calls are ***not equivalent***:

```
wrapper(i);
  // T is deduced as int
  // copy of i passed to func
  // wrapper cannot change i

func(i);
  // i passed by reference
  // func can change i
```

- Problem: The original and forwarded arguments are ***distinct objects***.

Perfect-Forwarding Example: Second Failed Attempt

- For our second attempt, we propose the following code for the (template) function `wrapper`:

```
template <class T>
void wrapper(T& p) {
    func(p);
}
```

- If, for example, the function argument is an rvalue (such as a non-string literal or temporary object), calls to `wrapper` and `func` (with the same argument) can have different behaviors.
- Suppose, for example, that we have the following declaration:

```
void func(int); // uses pass by value
```

- Then, the following two function calls are ***not equivalent***:

```
wrapper(42);
  // T is deduced as int
  // ERROR: cannot bind rvalue to
  //  nonconst lvalue reference

func(42);
  // OK
```

- Problem: The original and forwarded arguments do not match in terms of ***lvalue/rvalue-ness***.

Perfect-Forwarding Example: Third Failed Attempt

- For our third attempt, we propose the following code for the (template) function `wrapper`:

```
template <class T>
void wrapper(const T& p) {
    func(p);
}
```

- If, for example, the function argument is a non-const object, calls to `wrapper` and `func` (with the same argument) will have different behaviors.
- Suppose, for example, that we have the following declaration:

```
void func(int&);
int i;
```

- Then, the following two function calls are ***not equivalent***:

```
wrapper(i);
  // ERROR: wrapper cannot call func, as this
  // would discard const qualifier

func(i);
  // OK
```

- Problem: The original and forwarded arguments do not match in terms of ***cv-qualifiers***.

Perfect-Forwarding Example: Solution

- Finally, we propose the following code for the (template) function `wrapper`:

```
template <class T>
void wrapper(T&& p) {
    func(static_cast<T&&>(p));
}
```

- Consider now, for example, the following scenario:

```
int i = 42;
const int ci = i;
int& ri = i;
const int& rci = i;
wrapper(expr); // invoke wrapper
```

- The parameter `p` is an alias for the object yielded by the expression *expr*.
- The argument *expr* and argument to `func` match in terms of cv-qualifiers and lvalue/rvalue-ness.

	expr			argument to `func`	
expr	Type	Category	`T`	Type (`T&&`)	Category
`i`	`int`	lvalue	`int&`	`int&`	lvalue
`ci`	`const int`	lvalue	`const int&`	`const int&`	lvalue
`ri`	`int&`	lvalue	`int&`	`int&`	lvalue
`rci`	`const int&`	lvalue	`const int&`	`const int&`	lvalue
`42`	`int`	rvalue	`int`	`int&&`	rvalue

Perfect-Forwarding Example: Solution (Continued)

- Although we only considered one specific scenario on the previous slide, the solution works in general.
- That is, the `wrapper` function from the previous slide will perfectly forward its single argument, regardless of what the argument happens to be (or which overload of `func` is involved).
- Thus, we have a general solution to the perfect-forwarding problem in the single-argument case.
- This solution is easily extended to an arbitrary number of arguments.

The std::forward Template Function

- To avoid the need for an explicit type-cast operation when forwarding an argument, the standard library provides the std::forward function specifically for performing such a type conversion.
- The template function forward is defined as:

```
template<class T>
T&& forward(typename std::remove_reference<T>::type& x)
  noexcept {
    return static_cast<T&&>(x);
}
```

- A typical usage of forward might look something like:

```
template <class T1, class T2>
void wrapper(T1&& x1, T2&& x2) {
    func(std::forward<T1>(x1), std::forward<T2>(x2));
}
```

- The expression forward<T>(a) is an lvalue if T is an lvalue reference type and an rvalue otherwise.
- The use of std::forward instead of an explicit type cast improves code readability by making the programmer's intent clear.

Perfect-Forwarding Example Revisited

- We now revisit the perfect-forwarding example from earlier.
- In the earlier example, perfect forwarding was performed by the following function:

```
template <class T>
void wrapper(T&& e) {
    func(static_cast<T&&>(e));
}
```

- The above code can be made more readable, however, by rewriting it to make use of `std::forward` as follows:

```
template <class T>
void wrapper(T&& e) {
    func(std::forward<T>(e));
}
```

Forwarding Example

```
#include <iostream>
#include <string>
#include <utility>

void func(std::string& s) {
    std::cout << "func(std::string&) called\n";
}

void func(std::string&& s) {
    std::cout << "func(std::string&&) called\n";
}

template <class T>
void wrapper(T&& x) {
    func(std::forward<T>(x));
}

template <class T>
void buggy_wrapper(T x) {func(x);}

int main() {
    using namespace std::literals;
    std::string s("Hi"s);
    wrapper(s);          // which overload of func called?
    buggy_wrapper(s);    // which overload of func called?
    wrapper("Hi"s);      // which overload of func called?
    buggy_wrapper("Hi"s); // which overload of func called?
}
```

Perfect-Forwarding Use Case: Wrapper Functions

- A **wrapper function** is simply a function used to invoke another function, possibly with some additional processing.
- Example:

```
#include <iostream>
#include <utility>
#include <string>

std::string emphasize(const std::string& s)
  {return s + "!";}

std::string emphasize(std::string&& s)
  {return s + "!!!!";}

template <class A>
auto wrapper(A&& arg) {
    std::cout << "Calling with argument " << arg << '\n';
    auto result = emphasize(std::forward<A>(arg));
    std::cout << "Return value " << result << '\n';
    return result;
}

int main() {
    std::string s("Bonjour");
    wrapper(s);
    wrapper(std::string("Hello"));
}
```

Perfect-Forwarding Use Case: Factory Functions

- A **factory function** is simply a function used to create objects.
- Often, perfect forwarding is used by factory functions in order to pass arguments through to a constructor, which performs the actual object creation.
- Example:

```
#include <iostream>
#include <string>
#include <complex>
#include <utility>
#include <memory>

// Make an object of type T.
template<typename T, typename Arg>
std::shared_ptr<T> factory(Arg&& arg) {
    return std::shared_ptr<T>(
      new T(std::forward<Arg>(arg)));
}

int main() {
    using namespace std::literals;
    auto s(factory<std::string>("Hello"s));
    auto z(factory<std::complex<double>>(1.0i));
    std::cout << *s << ' ' << *z << '\n';
}
```

Perfect-Forwarding Use Case: Emplace Operations

- Many container classes provide an operation that creates a new element directly inside the container, often referred to as an **emplace operation**.
- Some or all of the arguments to a member function performing an emplace operation correspond to arguments for a constructor invocation.
- Thus, an emplace operation typically employs perfect forwarding.
- The member function performing the emplace operation forwards some or all of its arguments to the constructor responsible for actually creating the new object.
- Some examples of emplace operations in the standard library include:
  - `std::list` class: `emplace`, `emplace_back`, `emplace_front`
  - `std::vector` class: `emplace`, `emplace_back`
  - `std::set` class: `emplace`, `emplace_hint`
  - `std::forward_list` class: `emplace_front`, `emplace_after`

Other Perfect-Forwarding Examples

- `std::thread` constructor uses forwarding to pass through arguments to thread function
- `std::packaged_task` function-call operator uses forwarding to pass through arguments to associated function
- `std::async` uses forwarding to pass through arguments to specified callable entity
- `std::make_unique` forwards arguments to `std::unique_ptr` constructor
- `std::make_shared` forwards arguments to `std::shared_ptr` constructor
- `std::make_pair` forwards arguments to `std::pair` constructor
- `std::make_tuple` forwards arguments to `std::tuple` constructor

Section 3.4.8

References

References I

1. S. Meyers. Universal references in C++11. *Overload*, 111:8–12, Oct. 2012.
2. T. Becker, C++ Rvalue References Explained, 2013, http://thbecker.net/articles/rvalue_references/section_01.html
3. E. Bendersky, Understanding Lvalues and Rvalues in C and C++, 2011, http://eli.thegreenplace.net/2011/12/15/understanding-lvalues-and-rvalues-in-c-and-c
4. E. Bendersky, Perfect Forwarding and Universal References in C++, 2014, http://eli.thegreenplace.net/2014/perfect-forwarding-and-universal-references-in-c/
5. M. Kilpelainen. Lvalues and rvalues. *Overload*, 61:12–13, June 2004.

6. H. E. Hinnant, Forward, ISO/IEC JTC1/SC22/WG21/N2951, Sept. 27, 2009, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2951.html
7. H. Hinnant and D. Krugler, Proposed Wording for US 90, ISO/IEC JTC1/SC22/WG21/N3143, Oct. 15, 2010, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3143.html

Talks I

1. S. Meyers, Universal References in C++11, C++ and Beyond, Asheville, NC, USA, Aug. 5–8, 2012. Available online at `https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Scott-Meyers-Universal-References-in-Cpp11`.

   This talk discusses rvalue/forwarding references.

2. S. Meyers, Adventures in Perfect Forwarding, Facebook C++ Conference, Menlo Park, CA, USA, June 2, 2012. Available online at `https://www.facebook.com/Engineering/videos/10151094464083109`.

   This talk introduces perfect forwarding and discusses matters such as how to specialize forwarding templates and how to address interactions between forwarding and the pimpl idiom.

3. H. Hinnant, Everything You Ever Wanted to Know About Move Semantics, Inside Bloomberg, July 25, 2016. Available online at `https://youtu.be/vLinb2fgkHk`.

   This talk discusses various aspects of move semantics.

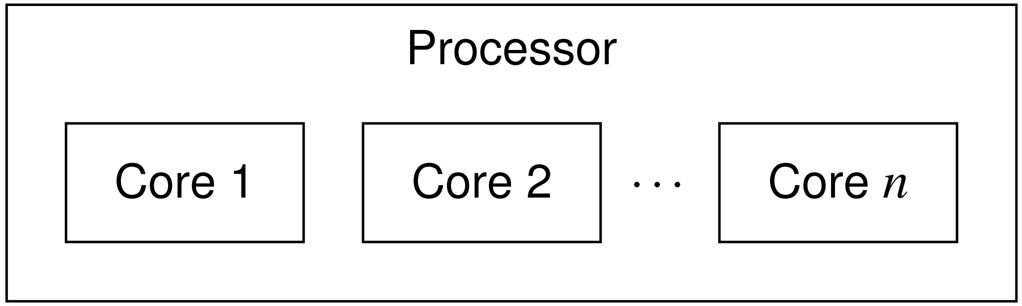
Section 3.5

Concurrency

Section 3.5.1

Preliminaries

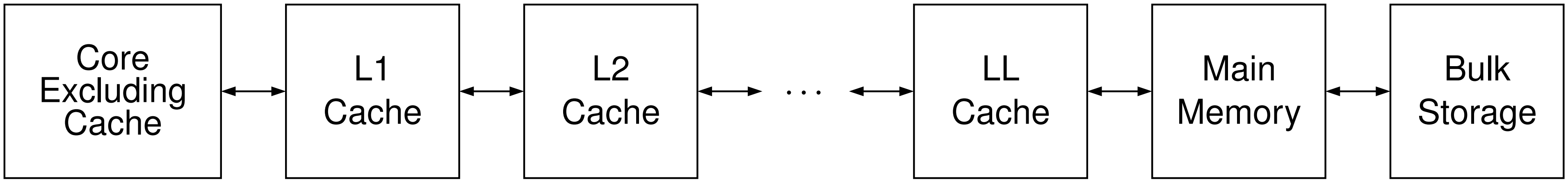
Processors



- A **core** is an independent processing unit that reads and executes program instructions, and consists of registers, an arithmetic logic unit (ALU), a control unit, and usually a cache.
- A **processor** is a computing element that consists of one or more cores, an external bus interface, and possibly a shared cache.
- A **thread** is a sequence of instructions (which can be executed by a core).
- At any given time, a core can execute one thread or, if the core supports simultaneous multithreading (such as hyperthreading), multiple threads.
- In the simultaneous multithreading case, the threads share the resources of the core.
- A processor with more than one core is said to be **multicore**.
- Most modern processors are multicore.
- Multicore processors can ***simultaneously*** execute ***multiple*** threads.

- A multicore processor said to be **homogeneous** if all of its cores are identical.
- A multicore processor said to be **heterogeneous** if its has more than one type of core.
- Different types of cores might be used in order to:
  - provide different types of functionality (e.g., CPU and GPU)
  - provide different levels of performance (e.g., high-performance CPU and energy-efficient CPU)

Memory Hierarchy



- The component of a system that stores program instructions and data is called **main memory**.
- A **cache** is fast memory used to store copies of instructions and/or data from main memory.
- Main memory is ***very slow*** compared to the speed of a processor core.
- Due to the latency of main memory, caches are ***essential*** for good performance.
- Instruction and data caches may be ***separate*** or ***unified*** (i.e., combined).
- A cache may be ***local*** to single core or ***shared*** between two or more cores.
- The lowest-level (i.e., L1) cache is usually on the core and local to the core.
- The higher-level (i.e., L2, L3,. . . , LL [last level]) caches are usually shared between some or all of the cores.

Examples of Multicore Processors

- Intel Core i7-3820QM Processor (Q2 2012)
  - used in Lenovo W530 notebook
  - 64 bit, 2.7 GHz
  - 128/128 KB L1 cache, 1 MB L2 cache, 8 MB L3 cache
  - *4 cores*
  - *8 threads* (2 threads/core)
- Intel Core i7-5960X Processor Extreme Edition (Q3 2014)
  - targets desktops/notebooks
  - 64 bit, 3 GHz
  - 256/256 KB L1 cache, 2 MB L2 cache, 20 MB L3 cache
  - *8 cores*
  - *16 threads* (2 threads/core)
- Intel Xeon Processor E7-8890 v2 (Q1 2014)
  - targets servers
  - 64 bit, 2.8 GHz
  - 480/480 KB L1 cache, 3.5 MB L2 cache, 37.5 MB L3 cache
  - *15 cores*
  - *30 threads* (2 threads/core)

Examples of Multicore SoCs

- Qualcomm Snapdragon 805 SoC (Q1 2014)
  - used in *Google Nexus 6*
  - 32-bit 2.7 GHz *quad-core* Qualcomm Krait 450 (ARMv7-A)
  - 16/16 KB L1 cache (per core), 2 MB L2 cache (shared)
  - 600 MHz Qualcomm Adreno 420 GPU
- Samsung Exynos 5 Octa 5433 SoC
  - used in *Samsung Galaxy Note 4*
  - high-performance 1.9 GHz *quad-core* ARM Cortex-A57 paired with energy-efficient 1.3 GHz *quad-core* ARM Cortex-A53 (big.LITTLE); both 32-bit (64-bit capable but disabled) (ARMv8-A)
  - Cortex-A57: 48/32 KB L1 cache, 512 KB to 2 MB L2 cache?
  - 700 MHz Mali-T760MP6 GPU
- Apple A8 SoC (2014)
  - used in *Apple iPhone 6*, *Apple iPhone 6 Plus*
  - 64-bit 1.4 GHz *dual-core* CPU (ARMv8-A)
  - 64/64 KB L1 cache (per core), 1 MB L2 cache (shared), 4 MB L3 cache
  - PowerVR Series 6XT GX6450 (quad-core) GPU

Why Multicore Processors?

- in past, greater processing power obtained through ***higher clock rates***
- ***clock rates have stopped rising***, topping out at about 5 GHz (little change since about 2005)
- power consumption is linear in clock frequency and quadratic in voltage, but higher frequency typically requires higher voltage; so, considering effect of frequency and voltage together, power consumption grows approximately with ***cube*** of frequency
- greater power consumption translates into ***increased heat production***
- higher clock rates would result in processors ***overheating***
- transistor counts ***still increasing*** (Moore's law: since 1960s, transistor count has doubled approximately every 18 months)
- instead of increasing processing power by raising clock rate of processor core, simply ***add more processor cores***
- n cores running at clock rate f use significantly less power and generate less heat than single core at clock rate nf
- going multicore allows for ***greater processing power*** with ***lower power consumption*** and ***less heat production***

Section 3.5.2

Multithreaded Programming

Concurrency

- A **thread** is a sequence of instructions that can be independently managed by the operating-system scheduler.
- A **process** provides the resources that program needs to execute (e.g., address space, files, and devices) and at least one thread of execution.
- All threads of a process share the ***same*** address space.
- **Concurrency** is the situation where multiple threads execute over time periods (i.e., from start of execution to end) that ***overlap*** (but no threads are required to run simultaneously).
- **Parallelism** refers to the situation where multiple threads execute ***simultaneously***.
- Concurrency can be achieved with:
  1. multiple single-threaded processes; or
  2. a single multithreaded process.
- A single multithreaded process is usually preferable, since this approach is typically much less resource intensive and data can often be shared much more easily between threads in a single process (due to the threads having a common address space).

Why Multithreading?

- Keep all of the processor cores busy (i.e., ***fully utilize*** all cores).
  - Most modern systems have multiple processor cores, due to having either multiple processors or a single processor that is multicore.
  - A single thread cannot fully utilize the computational resources available in such systems.
- Keep processes ***responsive***.
  - In graphics applications, keep the GUI responsive while the application is performing slow operations such as I/O.
  - In network server applications, keep the server responsive to new connections while handling already established ones.
- ***Simplify*** the coding of cooperating tasks.
  - Some programs consist of several logically distinct tasks.
  - Instead of having the program manage when the computation associated with different tasks is performed, each task can be placed in a separate thread and the operating system can perform scheduling.
  - For certain types of applications, multithreading can significantly reduce the conceptual complexity of the program.

Section 3.5.3

Multithreaded Programming Models

Memory Model

- A **memory model** (also known as a **memory-consistency model**) is a formal specification of the effect of read and write operations on the memory system, which in effect describes how memory appears to programs.
- A memory model is essential in order for the semantics of a multithreaded program to be well defined.
- The memory model must address issues such as:
  - ordering
  - atomicity
- The memory model affects:
  - programmability (i.e., ease of programming)
  - performance
  - portability

Sequential Consistency (SC)

- The environment in which a multithreaded program is run is said to have **sequential consistency (SC)** if the result of any execution of the program is the same as if the operations of all threads are executed in ***some sequential order***, and the operations of each individual thread appear in this sequence in ***the order specified by the program***.
- In other words, in a sequentially-consistent execution of a multithreaded program, threads behave as if their operations were simply ***interleaved***.
- Consider the multithreaded program (with two threads) shown below, where `x`, `y`, `a`, and `b` are all integer variables and initially zero.

Thread 1 Code	Thread 2 Code
`x = 1;` `a = y;`	`y = 1;` `b = x;`

- Some sequentially-consistent executions of this program include:
  - `x = 1; y = 1; b = x; a = y;`
  - `y = 1; x = 1; a = y; b = x;`
  - `x = 1; a = y; y = 1; b = x;`
  - `y = 1; b = x; x = 1; a = y;`

Sequential-Consistency (SC) Memory Model

- Since SC implies that memory must behave in a particular manner, SC implicitly defines a memory model, known as the **SC memory model**.
- In particular, SC implies that each write operation is ***atomic*** and becomes visible to all threads ***simultaneously***.
- Thus, with the SC model, ***all*** threads see write operations on memory occur ***atomically*** in the ***same*** order, leading to all threads having a ***consistent view*** of memory.
- The SC model precludes (or makes extremely difficult) many hardware optimizations, such as:
  - store buffers
  - caches
  - out-of-order instruction execution
- The SC model also precludes many compiler optimizations, including:
  - reordering of loads and stores
- Although the SC model very is ***intuitive***, it comes at a ***very high cost*** in terms of performance.

Load/Store Reordering Example: Single Thread

- Consider the program with the code below, where `x` and `y` are integer variables, all initially zero.

Original Thread 1 Code

```
x = 1;
y = 1;
// ...
```

- Suppose that, during optimization, the compiler transforms the preceding code to that shown below, effectively ***reordering two stores***.

Optimized Thread 1 Code

```
y = 1;
x = 1;
// ...
```

- The execution of the optimized code is ***indistinguishable*** from a sequentially-consistent execution of the original code.
- The optimized program runs ***as if*** it were the original program.
- In a ***single-threaded*** program, loads and stores can be reordered without invalidating the SC model (if data dependencies are correctly considered).

Load/Store Reordering Example: Multiple Threads

- Consider the addition of a second thread to the program to yield the code below.

Original Thread 1 Code

```
x = 1;
y = 1;
// ...
```

Thread 2 Code

```
if (y == 1) {
    assert(x == 1);
}
```

- Suppose that the compiler makes the same optimization to the code for thread 1 as on the previous slide, yielding the code below.

Optimized Thread 1 Code

```
y = 1;
x = 1;
// ...
```

(Unchanged) Thread 2 Code

```
if (y == 1) {
    assert(x == 1);
}
```

- Thread 2 can observe `x` and `y` being modified in the wrong order (i.e., an order that is inconsistent with SC execution).
- The assertion in thread 2 can never fail in the original program, but can sometimes fail in the optimized program.
- In a ***multithreaded*** program, the reordering of loads and stores must be avoided ***if SC is to be maintained***.

Store-Buffer Example: Without Store Buffer

- Consider the program below, where `x`, `y`, `a`, and `b` are integer variables, all initially zero.

Thread 1 Code

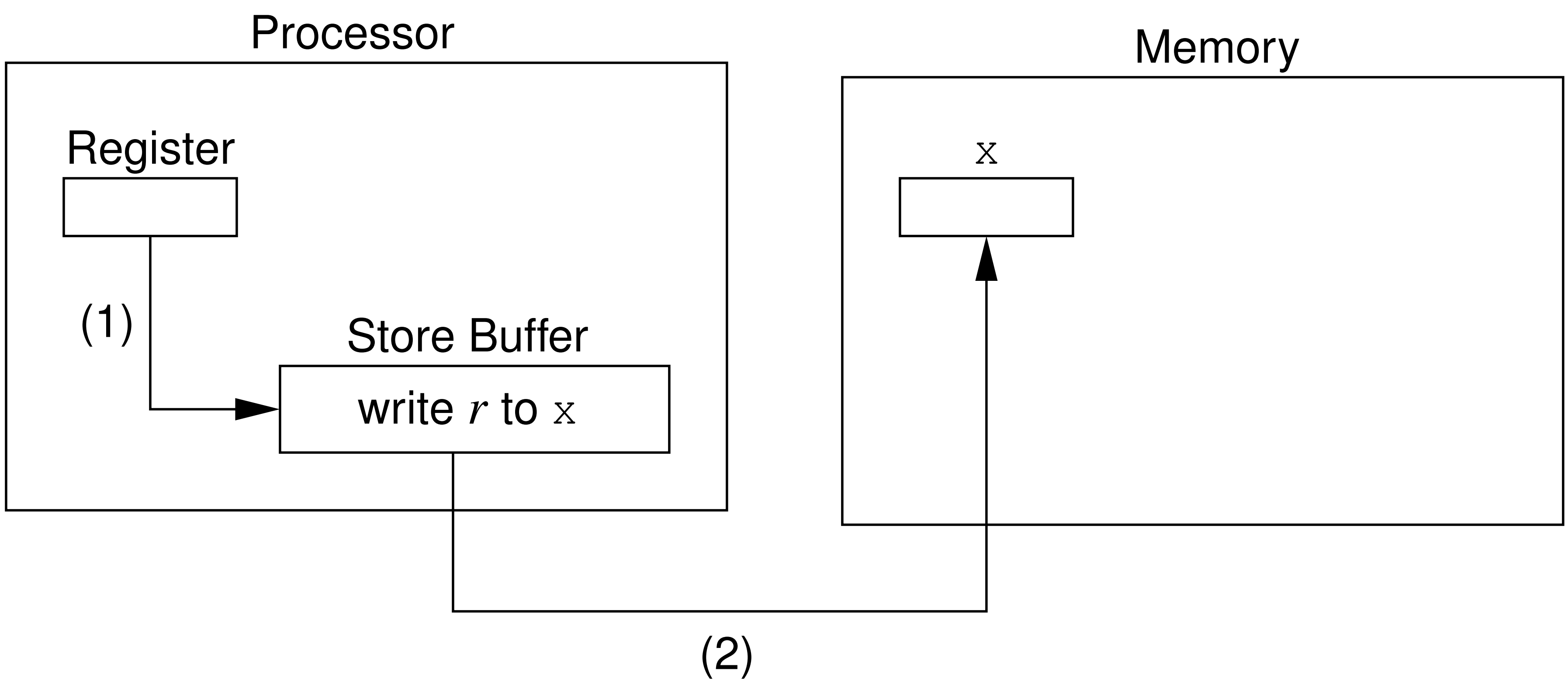
```
x = 1;
a = y;
```

Thread 2 Code

```
y = 1;
b = x;
```

- Some possible sequentially-consistent executions of the program include:
  - `x = 1; y = 1; b = x; a = y;` (`a` is 1, `b` is 1)
  - `y = 1; x = 1; a = y; b = x;` (`a` is 1, `b` is 1)
  - `x = 1; a = y; y = 1; b = x;` (`a` is 0, `b` is 1)
  - `y = 1; b = x; x = 1; a = y;` (`a` is 1, `b` is 0)
- In every sequentially-consistent execution of the program, one of "`x = 1;`" or "`y = 1;`" must execute first.
- If "`x = 1;`" executes first, then `b` cannot be assigned `0`.
- If "`y = 1;`" executes first, then `a` cannot be assigned `0`.
- No sequentially-consistent execution can result in `a` and `b` ***both*** being `0`.

Store-Buffer Example: Store Buffer



(1) transfer data from register to store buffer
(2) flush store buffer to memory

Store-Buffer Example: With Store Buffer (Not SC)

Core 1		Core 2		Memory	
Code	Store Buffer	Code	Store Buffer	x	y
`x = 1;`	write 1 to `x` pending			0	0
	no change	`y = 1;`	write 1 to `y` pending	0	0
`a = y;` `//a = 0;`	no change		no change	0	0
	no change	`b = x;` `//b = 0;`	no change	0	0
	write 1 to `x` completed		no change	1	0
			write 1 to `y` completed	1	1

- The execution of the program results in `a` and `b` ***both*** being 0, which ***violates SC***.
- The program behaves as if the lines of code in each thread were ***reordered*** (i.e., reversed), yielding: `a = y; b = x; x = 1; y = 1;`.
- A store buffer (or cache) must be avoided, ***if SC is to be maintained***.

Atomicity of Memory Operations

- A fundamental property of SC is that all memory operations are ***atomic***.
- Atomic memory operations require ***synchronization*** between processor cores.
- This synchronization ***greatly increases the time required to access memory***, as a result of the time needed by processor cores to communicate and coordinate access to memory.
- Therefore, requiring all memory operations to be atomic is not desirable.
- Allowing non-atomic memory operations, however, would be ***inconsistent with a fundamental property of SC***.

Data Races

- If memory operations are ***not all atomic***, the possibility exists for something known as a data race.
- Two memory operations are said to **conflict** if they access the ***same*** memory location and ***at least one*** of the operations is a write.
- Two conflicting memory operations form a **data race** if they are from different threads and can be executed ***at the same time***.
- A program with data races usually has unpredictable behavior (e.g., due to torn reads, torn writes, or worse).
- Example (data race):
  - Consider the multithreaded program listed below, where `x`, `y`, and `z` are (nonatomic) integer variables shared between threads and are initially zero.

Thread 1 Code	Thread 2 Code
`x = 1;` `a = y + z;`	`y = 1;` `b = x + z;`

  - The program has data races on both `x` and `y`.
  - Since `z` is not modified by any thread, `z` cannot participate in a data race.

Torn Reads

- A **torn read** is a read operation that (due to lack of atomicity) has only partially read its value when another (concurrent) write operation on the same location is performed.
- Consider a two-byte unsigned (big-endian) integer variable `x`, which is initially 1234 (hexadecimal).
- Suppose that the following (nonatomic) memory operations overlap in time:
  - thread 1 reads `x`; and
  - thread 2 writes 5678 (hexadecimal) to `x`.
- Initially, `x` is 1234:

Byte 0	Byte 1
12	34

- Thread 1 reads 12 from the first byte of `x`.
- Thread 2 writes 56 and 78 to the first and seconds bytes of `x`, respectively, yielding:

Byte 0	Byte 1
56	78

- Thread 1 reads the second byte of `x` to obtain the value 78.
- The value read by thread 1 (i.e., 1278) is neither the value of `x` prior to the write by thread 2 (i.e., 1234) nor the value of `x` after the write by thread 2 (i.e., 5678).

Torn Writes

- A **torn write** is a write operation that (due to lack of atomicity) has only partially written its value when another (concurrent) read or write operation on the same location is performed.
- Consider a two-byte unsigned (big-endian) integer variable `x`, which is initially 0.
- Suppose that the following (nonatomic) memory operations overlap in time:
  - thread 1 writes 1234 (hexadecimal) to `x`; and
  - thread 2 writes 5678 (hexadecimal) to `x`.
- Initially, `x` is 0:

Byte 0	Byte 1
00	00

- Thread 1 writes 12 to the first byte of `x`, yielding:

Byte 0	Byte 1
12	00

- Thread 2 writes 56 and 78 to the first and second bytes of `x`, respectively, yielding:

Byte 0	Byte 1
56	78

- Thread 1 writes 34 to the second byte of `x`, yielding:

Byte 0	Byte 1
56	34

- The resulting value in `x` (i.e., 5634) is neither the value written by thread 1 (i.e., 1234) nor the value written by thread 2 (i.e., 5678).

SC Data-Race Free (SC-DRF) Memory Model

- From a programmability standpoint, SC is extremely desirable, as it allows one to reason easily about the behavior of a multithreaded program.
- Unfortunately, as we saw earlier, SC precludes almost all useful compiler optimizations and hardware optimizations.
- As it turns out, if we drop the requirement that all memory operations be atomic and then restrict programs to be data-race free, SC can be provided while still allowing most compiler and hardware optimizations.
- This observation is the motivation behind the so called SC-DRF memory model.
- The **sequential-consistency for data-race free programs (SC-DRF) model** provides SC ***only for programs that are data-race free***.
- The data-race free constraint is not overly burdensome, since data races will likely result in bugs anyhow.
- Several programming languages have used SC-DRF as the basis for their memory model, including C++, C, and Java.

C++ Memory Model

- The C++ programming language employs, at its default memory model, the ***SC-DRF*** model.
- Again, with the SC-DRF model, a program behaves as if its execution is sequentially consistent, provided that the program is data-race free.
- Support is also provided for other (more relaxed) memory models.
- For certain memory accesses, it is possible to override the default (i.e., SC-DRF) memory model, if desired.
- The execution of a program that is not data-race free results in ***undefined behavior***.

Section 3.5.4

Thread Management

The `std::thread` Class

- `std::thread` class provides means to create new thread of execution, wait for thread to complete, and perform other operations to manage and query state of thread
- `thread` object may or may not be associated with thread (of execution)
- `thread` object that is associated with thread said to be **joinable**
- default constructor creates `thread` object that is ***unjoinable***
- can also construct `thread` object by providing callable entity (e.g., function or functor) and arguments (if any), resulting in new thread invoking callable entity
- thread function provided with ***copies*** of arguments so must use reference wrapper class like `std::reference_wrapper` for reference semantics
- `thread` class is movable but ***not copyable***
- each `thread` object has ID
- IDs of joinable `thread` objects are ***unique***
- all unjoinable `thread` objects have ***same*** ID, distinct from ID of every joinable `thread` object

The `std::thread` Class (Continued)

- **join operation** waits for `thread` object's thread to complete execution and results in object becoming unjoinable
- **detach operation** dissociates thread from `thread` object (allowing thread to continue to execute independently) and results in object becoming unjoinable
- using `thread` object as source for move operation results in object becoming unjoinable
- if `thread` object joinable when destructor called, exception is thrown
- `hardware_concurrency` member function returns number of hardware threads that can run simultaneously (or zero if not well defined)
- thread creation and join operations establish synchronizes-with relationship (to be discussed later)

`std::thread` Members

Member Types

Member Name	Description
`id`	thread ID type
`native_handle_type`	system-dependent handle type for underlying thread entity

Construction, Destruction, and Assignment

Member Name	Description
constructor	construct thread (overloaded)
destructor	destroy thread
`operator=`	move assign thread

`std::thread` Members (Continued)

Member Functions

Member Name	Description
`joinable`	check if thread joinable
`get_id`	get ID of thread
`native_handle`	get native handle for thread
`hardware_concurrency` (static)	get number of concurrent threads supported by hardware
`join`	wait for thread to finish executing
`detach`	permit thread to execute independently
`swap`	swap threads

Example: Hello World With Threads

```cpp
#include <iostream>
#include <thread>

void hello()
{
    std::cout << "Hello World!\n";
}

int main()
{
    std::thread t(hello);
    t.join();
}
```

```cpp
#include <iostream>
#include <thread>

int main()
{
    std::thread t([](){
        std::cout << "Hello World!\n";
    });
    t.join();
}
```

Example: Thread-Function Argument Passing (Copy Semantics)

```
#include <iostream>
#include <thread>

void doWork(int i, int j)
{
    std::cout << i << ' ' << j << '\n';
}

int main()
{
    int i = 42;
    std::thread t1(doWork, i, 1);
    t1.join();
}
```

```
#include <iostream>
#include <vector>
#include <functional>
#include <thread>

void doWork(const std::vector<int>& v)
{
    for (auto i : v) {
        std::cout << i << '\n';
    }
}

int main()
{
    std::vector v{1, 2, 3, 4};

    // copy semantics
    std::thread t1(doWork, v);
    t1.join();

    // reference semantics
    std::thread t2(doWork, std::ref(v));
    t2.join();
}
```

Example: Thread-Function Argument Passing (Move Semantics)

```
#include <iostream>
#include <vector>
#include <utility>
#include <thread>

void doWork(std::vector<int>&& v)
{
    for (auto i : v) {
        std::cout << i << '\n';
    }
}

int main()
{
    std::vector v{1, 2, 3, 4};

    // move semantics
    std::thread t1(doWork, std::move(v));
    t1.join();
}
```

Example: Moving Threads

```cpp
#include <thread>
#include <iostream>
#include <utility>

// Return a thread that prints a greeting message.
std::thread makeThread() {
    return std::thread([](){
        std::cout << "Hello World!\n";
    });
}

// Return the same thread that was passed as an argument.
std::thread identity(std::thread t) {
    return t;
}

int main() {
    std::thread t1(makeThread());
    std::thread t2(std::move(t1));
    t1 = std::move(t2);
    t1 = identity(std::move(t1));
    t1.join();
}
```

Example: Lifetime Bug

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <thread>
#include <numeric>

void threadFunc(const std::vector<int>* v) {
    std::cout << std::accumulate(v->begin(), v->end(), 0)
      << '\n';
}

void startThread() {
    std::vector<int> v(1000000, 1);
    std::thread t(threadFunc, &v);
    t.detach();
    // v is destroyed here but detached thread
    // may still be using v
}

int main() {
    startThread();
    // Give the thread started by startThread
    // sufficient time to complete its work.
    std::this_thread::sleep_for(std::chrono::seconds(5));
}
```

The `std::this_thread` Namespace

Name	Description
`get_id`	get ID of current thread
`yield`	suggest rescheduling current thread so as to allow other threads to run
`sleep_for`	blocks execution of current thread for at least specified duration
`sleep_until`	blocks execution of current thread until specified time reached

Example: Identifying Threads

```
#include <thread>
#include <iostream>

// main thread ID
std::thread::id mainThread;

void func() {
    if (std::this_thread::get_id() == mainThread) {
        std::cout << "called by main thread\n";
    } else {
        std::cout << "called by secondary thread\n";
    }
}

int main() {
    mainThread = std::this_thread::get_id();
    std::thread t([](){
        // call func from secondary thread
        func();
    });
    // call func from main thread
    func();
    t.join();
}
```

Thread Local Storage

- **thread storage duration**: object initialized before first use in thread and, if constructed, destroyed on thread exit
- each thread has its own instance of object
- only objects declared **thread_local** have this storage duration
- **thread_local** implies **static** for variable of block scope
- **thread_local** can appear together with **static** or **extern** to adjust linkage
- example:

```
thread_local int counter = 0;
static thread_local int x = 0;
thread_local int y;

void func() {
    thread_local int counter = 0;
    // equivalent to:
    // static thread_local int counter = 0;
}
```

Example: Thread Local Storage

```cpp
#include <iostream>
#include <vector>
#include <thread>

thread_local int counter = 0;

void doWork(int id) {
    static const char letters[] = "abcd";
    for (int i = 0; i < 10; ++i) {
        std::cout << letters[id] << counter << '\n';
        ++counter;
    }
}

int main() {
    std::vector<std::thread> workers;
    for (int i = 1; i <= 3; ++i) {
        // invoke doWork in new thread
        workers.emplace_back(doWork, i);
    }
    // invoke doWork in main thread
    doWork(0);
    for (auto& t : workers) {t.join();}
}
```

The `std::thread` Class and Exception Safety

- The astute reader will notice that most code examples on these lecture slides (both earlier and later) that directly employ `std::thread` are not exception safe.
- Some of the exception safety problems in these examples could be eliminated by using a RAII class to wrap `std::thread` objects.
- Unfortunately, the standard library does not provide such a RAII class.
- At a very basic level, one could provide a thread wrapper class that has similar functionality to `std::thread`, except that its destructor automatically joins with the underlying thread if the thread is still joinable at destruction time. (See next slide.)
- Although such an approach will work in some situations (such as in the case of many of the simple code examples on these lecture slides), it can potentially lead to deadlocks and other problems in more complex code.
- A more general solution would be to provide a class that allows arbitrary code to be executed just prior to thread destruction, in order to perform the appropriate (application-dependent) "clean-up" action. (For example, see `boost::scoped_thread` in the Boost Threads library.)

The `std::thread` Class and Exception Safety (Continued)

```
#include <thread>

// A minimalist inheritance-based replacement for std::thread
// that joins automatically in the destructor.
// (One must be careful not to use this type polymorphically
// since the destructor is not virtual.)
class scoped_thread : public std::thread {
public:
    using std::thread::thread;
    scoped_thread(scoped_thread&&) = default;
    scoped_thread& operator=(scoped_thread&&) = default;
    scoped_thread(const scoped_thread&) = delete;
    scoped_thread& operator=(const scoped_thread&) = delete;
    ~scoped_thread() {
        if (joinable()) {
            join();
        }
    }
};
```

Section 3.5.5

Sharing Data Between Threads

Shared Data

- In multithreaded programs, it is often necessary to ***share resources*** between threads.
- Shared resources might include such things as variables, memory, files, devices, and so on.
- The sharing of resources, however, can lead to various problems when multiple threads want access to the same resource simultaneously.
- The most commonly shared resource is ***variables***.
- When variables are shared between threads, the possibility exists that one thread may attempt to access a variable while another thread is modifying the same variable.
- Such ***conflicting accesses*** to variables can lead to data corruption and other problems.
- More generally, when any resource is shared, the potential for problems exists.
- Therefore, mechanisms are needed for ensuring that shared resources can be accessed safely.

Race Conditions

- A **race condition** is a behavior where the outcome depends on the relative ordering of the execution of operations on two or more threads.
- Sometimes, a race condition may be benign (i.e., does not cause any problem).
- Usually, the term "race condition" used to refer to a race condition that is not benign (i.e., breaks invariants or results in undefined behavior).
- A data race is a particularly evil type of race condition.
- A **deadlock** is a situation in which two or more threads are unable to make progress due to being ***blocked*** waiting for resources held by each other.
- A **livelock** is a situation in which two or more threads are ***not blocked*** but are unable to make progress due to needing resources held by each other.
- Often, race conditions can lead to deadlocks, livelocks, crashes, and other unpredictable behavior.

- A **critical section** is a piece of code that accesses a shared resource (e.g., data structure) that must not be simultaneously accessed by more than one thread.
- A synchronization mechanism is needed at the entry to and exit from a critical section.
- The mechanism needs to provide ***mutual exclusion*** (i.e., prevent critical sections in multiple threads from executing simultaneously).
- Example (FIFO queue):
  - One thread is adding an element to a queue while another thread is removing an element from the same queue.
  - Since both threads modify the queue at the same time, they could corrupt the queue data structure.
  - Synchronization must be employed so that the execution of the parts of the code that add and remove elements are executed in a ***mutually exclusive*** manner (i.e., cannot run at the same time).

Data-Race Example

Shared (Global) Data

```
double balance = 100.00; // bank account balance
double credit = 50.00; // amount to deposit
double debit = 10.00; // amount to withdraw
```

Thread 1 Code

```
// double tmp = balance;
// tmp = tmp + credit;
// balance = tmp;
balance += credit;
```

Thread 2 Code

```
// double tmp = balance;
// tmp = tmp - debit;
// balance = tmp;
balance -= debit;
```

- above code has data race on `balance` object (i.e., more than one thread may access `balance` at same time with at least one thread writing)

Example: Data Race (Counter)

```cpp
#include <iostream>
#include <thread>

unsigned long long counter = 0;

void func() {
    for (int i = 0; i < 1000000; ++i) {
        ++counter;
    }
}

int main() {
    std::thread t1(func);
    std::thread t2(func);
    t1.join();
    t2.join();
    std::cout << counter << '\n';
}
```

Example: Data Race and/or Race Condition (IntSet)

```
#include <thread>
#include <iostream>
#include <set>

class IntSet {
public:
    bool contains(int i) const
      {return s_.find(i) != s_.end();}
    void add(int i)
      {s_.insert(i);}
private:
    std::set<int> s_;
};

IntSet s;

int main() {
    std::thread t1([](){
        for (int i = 0; i < 1000; ++i) s.add(2 * i);
    });
    std::thread t2([](){
        for (int i = 0; i < 1000; ++i) s.add(2 * i + 1);
    });
    t1.join(); t2.join();
    std::cout << s.contains(1000) << '\n';
}
```

Section 3.5.6

Mutexes

Mutexes

- A **mutex** is a locking mechanism used to synchronize access to a shared resource by providing ***mutual exclusion***.
- A mutex has two basic operations:
  - **acquire**: lock (i.e., hold) the mutex
  - **release**: unlock (i.e., relinquish) the mutex
- A mutex can be held ***by only one thread*** at any given time.
- If a thread attempts to acquire a mutex that is already held by another thread, the operation will either block until the mutex can be acquired or fail with an error.
- A thread holding a (nonrecursive) mutex ***cannot relock*** the mutex.
- A thread acquires the mutex before accessing the shared resource and releases the mutex when finished accessing the resource.
- Since only one thread can hold a mutex at any given time and the shared resource is only accessed by the thread holding the mutex, mutually-exclusive access is guaranteed.

The `std::mutex` Class

- `std::mutex` class provides mutex functionality
- ***not movable*** and ***not copyable***
- `lock` member function acquires mutex (blocking as necessary)
- `unlock` member function releases mutex
- thread that owns mutex should not attempt to lock mutex again
- all prior `unlock` operations on given mutex ***synchronize with*** `lock` operation (on ***same*** mutex) (synchronizes-with relationship to be discussed later)

std::mutex Members

Member Types

Name	Description
native_handle_type	system-dependent handle type for underlying mutex entity

Construction, Destruction, and Assignment

Name	Description
constructor	construct mutex
destructor	destroy mutex

Other Member Functions

Name	Description
lock	acquire mutex, blocking if not available
try_lock	try to lock mutex without blocking
unlock	release mutex
native_handle	get handle for underlying mutex entity

Example: Avoiding Data Race Using Mutex (Counter) (mutex)

```cpp
#include <iostream>
#include <thread>
#include <mutex>

std::mutex m;
unsigned long long counter = 0;

void func() {
    for (int i = 0; i < 1000000; ++i) {
        m.lock(); // acquire mutex
        ++counter;
        m.unlock(); // release mutex
    }
}

int main() {
    std::thread t1(func);
    std::thread t2(func);
    t1.join();
    t2.join();
    std::cout << counter << '\n';
}
```

The std::lock_guard Template Class

- std::lock_guard is RAII class for mutexes
- declaration:

```
template <class T> class lock_guard;
```

- template parameter T specifies type of mutex (e.g., std::mutex, std::recursive_mutex)
- avoids problem of inadvertently forgetting to release mutex (e.g., due to exception or forgetting unlock call)
- constructor takes mutex as argument
- ***not movable*** and ***not copyable***
- acquires mutex in constructor
- releases mutex in destructor
- since language ensures that all objects destroyed at end of lifetime, release of mutex guaranteed (even if some code skipped due to thrown exception)
- advisable to use lock_guard instead of calling lock and unlock explicitly

`std::lock_guard` Members

Member Types

Name	Description
`mutex_type`	underlying mutex type

Construction, Destruction, and Assignment

Name	Description
constructor	construct lock guard
destructor	destroy lock guard

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex m;
unsigned long long counter = 0;

void func() {
    for (int i = 0; i < 1000000; ++i) {
        // lock_guard constructor acquires mutex
        std::lock_guard lock(m);
        ++counter;
        // lock_guard destructor releases mutex
    }
}

int main() {
    std::thread t1(func);
    std::thread t2(func);
    t1.join();
    t2.join();
    std::cout << counter << '\n';
}
```

```cpp
#include <thread>
#include <iostream>
#include <set>
#include <mutex>

class IntSet {
public:
    bool contains(int i) const {
        std::lock_guard lg(m_);
        return s_.find(i) != s_.end();
    }
    void add(int i) {
        std::lock_guard lg(m_);
        s_.insert(i);
    }
private:
    std::set<int> s_;
    mutable std::mutex m_;
};

IntSet s;

int main() {
    std::thread t1([](){
        for (int i = 0; i < 1000; ++i) s.add(2 * i);
    });
    std::thread t2([](){
        for (int i = 0; i < 1000; ++i) s.add(2 * i + 1);
    });
    t1.join(); t2.join();
    std::cout << s.contains(1000) << '\n';
}
```

The `std::scoped_lock` Template Class

- `std::scoped_lock` is RAII class for mutexes
- declaration:

```
template <class... Ts> class scoped_lock;
```

- parameter pack `Ts` specifies types of mutexes to be locked
- can be used with any mutex types providing necessary locking interface (e.g., `std::mutex` and `std::recursive_mutex`)
- constructor takes one or more mutexes as arguments
- mutexes acquired in constructor and released in destructor
- `scoped_lock` objects are ***not movable*** and ***not copyable***
- using `scoped_lock` avoids problem of inadvertently failing to release mutexes (e.g., due to exception or forgetting `unlock` calls)
- in multiple mutex case, employs deadlock avoidance algorithm from `std::lock` (discussed later) when acquiring mutexes
- advisable to use `scoped_lock` instead of calling `lock` and `unlock` explicitly
- `scoped_lock` effectively replaces (and extends) `lock_guard`

Example: Avoiding Data Race Using Mutex (IntSet) (scoped_lock)

```
#include <thread>
#include <iostream>
#include <unordered_set>
#include <mutex>

class IntSet {
public:
    bool contains(int i) const {
        std::scoped_lock lock(m_);
        return s_.find(i) != s_.end();
    }
    void add(int i) {
        std::scoped_lock lock(m_);
        s_.insert(i);
    }
private:
    std::unordered_set<int> s_;
    mutable std::mutex m_;
};

IntSet s;

int main() {
    std::thread t1([](){
        for (int i = 0; i < 10'000; ++i) {s.add(2 * i);}
    });
    std::thread t2([](){
        for (int i = 0; i < 10'000; ++i) {s.add(2 * i + 1);}
    });
    t1.join(); t2.join();
    std::cout << s.contains(1000) << '\n';
}
```

The `std::unique_lock` Template Class

- `std::unique_lock` is another RAII class for mutexes
- declaration:

  ```
  template <class T> class unique_lock;
  ```

- template parameter `T` specifies type of mutex (e.g., `std::mutex`, `std::recursive_mutex`)
- unlike case of `std::lock_guard`, in case of `unique_lock` do not have to hold mutex over entire lifetime of RAII object
- have choice of whether to acquire mutex upon construction
- also can acquire and release mutex many times throughout lifetime of `unique_lock` object
- upon destruction, if mutex is held, it is released
- since mutex is always guaranteed to be released by destructor, cannot forget to release mutex
- `unique_lock` is used in situations where want to be able to transfer ownership of lock (e.g., return from function) or RAII object needed for mutex but do not want to hold mutex over entire lifetime of RAII object
- ***movable*** but ***not copyable***

std::unique_lock Members

Member Types

Name	Description
mutex_type	underlying mutex type

Construction, Destruction, and Assignment

Name	Description
constructor	construct unique lock
destructor	destroy unique lock
operator=	move assign

Locking Functions

Name	Description
lock	acquire mutex, blocking if not available
try_lock	try to lock mutex without blocking
try_lock_for	try to lock mutex without blocking
try_lock_until	try to lock mutex without blocking
unlock	release mutex

Observer Functions

Name	Description
`owns_lock`	tests if lock owns associated mutex
`operator bool`	tests if lock owns associated mutex

Example: Avoiding Data Race Using Mutex (Counter) (unique_lock)

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex m;
unsigned long long counter = 0;

void func() {
    for (int i = 0; i < 1000000; ++i) {
        // Create a lock object without locking the mutex.
        std::unique_lock lock(m, std::defer_lock);
        // ...
        // Lock the mutex.
        lock.lock();
        ++counter;
        // The unique_lock destructor releases the mutex.
    }
}

int main() {
    std::thread t1(func);
    std::thread t2(func);
    t1.join();
    t2.join();
    std::cout << counter << '\n';
}
```

The `std::lock` Template Function

- `std::lock` variadic template function that can acquire multiple locks simultaneously without risk of deadlock (assuming the only locks involved are ones passed to `lock`)
- declaration:

```
template <class T1, class T2, class... TN>
void lock(T1&, T2&, TN& ...);
```

- takes as arguments one or more locks to be acquired

Example: Acquiring Two Locks for Swap (Incorrect)

```
#include <thread>
#include <vector>
#include <mutex>

class BigBuf // A Big Buffer
{
public:
    static constexpr int size() {return 16 * 1024 * 1024;}
    BigBuf() : data_(size()) {}
    BigBuf& operator=(const BigBuf&) = delete;
    BigBuf& operator=(BigBuf&&) = delete;
    void swap(BigBuf& other) {
        if (this == &other)
            return;
        // acquiring the two mutexes in this way can result in deadlock
        std::lock_guard lock1(m_);
        std::lock_guard lock2(other.m_);
        std::swap(data_, other.data_);
    }
    // ...
private:
    std::vector<char> data_;
    mutable std::mutex m_;
};

BigBuf a;
BigBuf b;

int main()
{
    std::thread t1([](){
        for (int i = 0; i < 100000; ++i) a.swap(b);
    });
    std::thread t2([](){
        for (int i = 0; i < 100000; ++i) b.swap(a);
    });
    t1.join(); t2.join();
}
```

Example: Acquiring Two Locks for Swap [unique_lock and lock]

```
#include <mutex>
#include <thread>
#include <utility>
#include <vector>

class BigBuf // A Big Buffer
{
public:
    static constexpr int size() {return 16 * 1024 * 1024;}
    BigBuf() : data_(size()) {}
    BigBuf& operator=(const BigBuf&) = delete;
    BigBuf& operator=(BigBuf&&) = delete;
    void swap(BigBuf& other) {
        if (this == &other)
            return;
        std::unique_lock lock1(m_, std::defer_lock);
        std::unique_lock lock2(other.m_, std::defer_lock);
        std::lock(lock1, lock2);
        std::swap(data_, other.data_);
    }
    // ...
private:
    std::vector<char> data_;
    mutable std::mutex m_;
};

BigBuf a;
BigBuf b;

int main() {
    std::thread t1([](){
        for (int i = 0; i < 100000; ++i) a.swap(b);
    });
    std::thread t2([](){
        for (int i = 0; i < 100000; ++i) b.swap(a);
    });
    t1.join(); t2.join();
}
```

Example: Acquiring Two Locks for Swap [scoped_lock]

```
#include <mutex>
#include <thread>
#include <utility>
#include <vector>

class BigBuf // A Big Buffer
{
public:
    static constexpr int size() {return 16 * 1024 * 1024;}
    BigBuf() : data_(size()) {}
    BigBuf& operator=(const BigBuf&) = delete;
    BigBuf& operator=(BigBuf&&) = delete;
    void swap(BigBuf& other) {
        if (this == &other)
            return;
        std::scoped_lock sl(m_, other.m_);
        std::swap(data_, other.data_);
    }
    // ...
private:
    std::vector<char> data_;
    mutable std::mutex m_;
};

BigBuf a;
BigBuf b;

int main() {
    std::thread t1([](){
        for (int i = 0; i < 100000; ++i) a.swap(b);
    });
    std::thread t2([](){
        for (int i = 0; i < 100000; ++i) b.swap(a);
    });
    t1.join(); t2.join();
}
```

The std::timed_mutex Class

- std::timed_mutex class provides mutex that allows timeout to be specified when acquiring mutex
- if mutex cannot be acquired in time specified, acquire operation fails (i.e., does not lock mutex) and error returned
- adds try_lock_for and try_lock_until member functions to try to lock mutex with timeout

Example: Acquiring Mutex With Timeout (std::timed_mutex)

```
#include <vector>
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

std::timed_mutex m;

void doWork() {
    for (int i = 0; i < 10000; ++i) {
        std::unique_lock lock(m, std::defer_lock);
        int count = 0;
        while (!lock.try_lock_for(
          std::chrono::microseconds(1))) {++count;}
        std::cout << count << '\n';
    }
}

int main() {
    std::vector<std::thread> workers;
    for (int i = 0; i < 16; ++i) {
        workers.emplace_back(doWork);
    }
    for (auto& t : workers) {t.join();}
}
```

Recursive Mutexes

- A **recursive mutex** is a mutex for which a thread may own ***multiple*** locks ***at the same time***.
- After a mutex is first locked by thread A, thread A can acquire additional locks on the mutex (without releasing the lock already held).
- The mutex is not available to other threads until thread A releases all of its locks on the mutex.
- A recursive mutex is typically used when code that locks a mutex must call other code that locks the same mutex (in order to avoid deadlock).
- For example, a function that acquires a mutex and recursively calls itself (resulting in the mutex being relocked) would need to employ a recursive mutex.
- A recursive mutex has ***more overhead*** than a nonrecursive mutex.
- Code that uses recursive mutexes can often be ***more difficult to understand*** and therefore ***more prone to bugs***.
- Consequently, the use of recursive mutexes should be ***avoided if possible***.

Recursive Mutex Classes

- recursive mutexes provided by classes `std::recursive_mutex` and `std::recursive_timed_mutex`
- `recursive_mutex` class similar to `std::mutex` class except allows relocking
- `recursive_timed_mutex` class similar to `std::timed_mutex` class except allows relocking
- implementation-defined limit to number of levels of locking allowed by recursive mutex

Shared Mutexes

- A **shared mutex** (also known as a **multiple-reader/single-writer mutex**) is a mutex that allows both ***shared and exclusive*** access.
- A shared mutex has ***two types of locks***: shared and exclusive.
- **Exclusive lock**:
  - ***Only one*** thread can hold an ***exclusive*** lock on a mutex.
  - While a thread holds an exclusive lock on a mutex, no other thread can hold any type of lock on the mutex.
- **Shared lock**:
  - ***Any number*** of threads (within implementation limits) can take a ***shared*** lock on a mutex.
  - While any thread holds a shared lock on a mutex, no thread may take an exclusive lock on the mutex.
- A shared mutex would typically be used to protect shared data that is seldom updated but cannot be safely updated if any thread is reading it.
- A thread takes a shared lock for reading, thus allowing ***multiple readers***.
- A thread takes an exclusive lock for writing, thus allowing ***only one writer with no readers***.
- A shared mutex need not be fair in its granting of locks (e.g., readers could starve writers).

The `std::shared_mutex` Class

- `std::shared_mutex` class provides shared mutex functionality
- ***not movable*** and ***not copyable***
- `lock` member function acquires exclusive ownership of mutex (blocking as necessary)
- `unlock` member function releases exclusive ownership
- `lock_shared` member function acquires shared ownership of mutex (blocking as necessary)
- `unlock_shared` member function releases shared ownership

std::shared_mutex Members

Construction, Destruction, and Assignment

Name	Description
constructor	construct mutex
destructor	destroy mutex
operator= [deleted]	not movable or copyable

Exclusive Locking Functions

Name	Description
`lock`	acquire exclusive ownership of mutex, blocking if not available
`try_lock`	try to acquire exclusive ownership of mutex without blocking
`unlock`	release exclusive ownership of mutex

std::shared_mutex Members (Continued)

Shared Locking Functions

Name	Description
lock_shared	acquire shared ownership of mutex, blocking if not available
try_lock_shared	try to acquire shared ownership of mutex without blocking
unlock_shared	release shared ownership of mutex

Other Functions

Name	Description
native_handle	get handle for underlying mutex entity

The std::shared_lock Template Class

- std::shared_lock is RAII class for shared mutexes
- declaration:

```
template <class T> class shared_lock;
```

- template parameter T specifies type of mutex (e.g., std::shared_mutex or std::shared_timed_mutex)
- similar interface as std::unique_lock but uses shared locking
- constructor may optionally acquire mutex
- may acquire and release mutex many times throughout lifetime of object
- destructor releases mutex if held
- all operations mapped onto shared locking primitives (e.g., lock mapped to lock_shared, unlock mapped to unlock_shared)
- for exclusive locking with shared mutexes, std::unique_lock and std::lock_guard can be used

Example: std::shared_mutex

```
#include <thread>
#include <mutex>
#include <iostream>
#include <vector>
#include <shared_mutex>

std::mutex coutMutex;
int counter = 0;
std::shared_mutex counterMutex;

void writer() {
    for (int i = 0; i < 10; ++i) {
        {
            std::lock_guard lock(counterMutex);
            ++counter;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

void reader() {
    for (int i = 0; i < 100; ++i) {
        int c;
        {
            std::shared_lock lock(counterMutex);
            c = counter;
        }
        {
            std::lock_guard lock(coutMutex);
            std::cout << std::this_thread::get_id() << ' ' << c << '\n';
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}

int main() {
    std::vector<std::thread> threads;
    threads.emplace_back(writer);
    for (int i = 0; i < 16; ++i) threads.emplace_back(reader);
    for (auto& t : threads) t.join();
}
```

The `std::shared_timed_mutex` Class

- `std::shared_timed_mutex` class provides shared mutex
- `shared_timed_mutex` interface similar to that of `shared_mutex` but allows timeout for acquiring mutex
- adds `try_lock_for` and `try_lock_until` member functions to try to acquire exclusive ownership of mutex with timeout
- adds `try_lock_shared_for` and `try_lock_shared_until` member functions to try to acquire shared ownership of mutex with timeout

Example: std::shared_timed_mutex

```
#include <thread>
#include <mutex>
#include <iostream>
#include <vector>
#include <shared_mutex>

std::mutex coutMutex;
int counter = 0;
std::shared_timed_mutex counterMutex;

void writer() {
    for (int i = 0; i < 10; ++i) {
        {
            std::lock_guard lock(counterMutex);
            ++counter;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

void reader() {
    for (int i = 0; i < 100; ++i) {
        int c;
        {
            std::shared_lock lock(counterMutex);
            c = counter;
        }
        {
            std::lock_guard lock(coutMutex);
            std::cout << std::this_thread::get_id() << ' ' << c << '\n';
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}

int main() {
    std::vector<std::thread> threads;
    threads.emplace_back(writer);
    for (int i = 0; i < 16; ++i) threads.emplace_back(reader);
    for (auto& t : threads) t.join();
}
```

- sometimes may want to perform action only once in code executed in multiple threads
- can be achieved through use of std::once_flag type in conjunction with std::call_once template function
- std::once_flag class represents flag used to track if action performed
- declaration of std::call_once:

```
template <class Callable, class... Args>
void call_once(std::once_flag& flag, Callable&& f,
  Args&&... args);
```

- std::call_once invokes f only once based on value of flag object
- first invocation of f is guaranteed to complete before any threads return from call_once
- useful for one-time initialization of dynamically generated objects

Example: One-Time Action

```
#include <iostream>
#include <vector>
#include <thread>
#include <mutex>

std::once_flag flag;

void worker(int id) {
    std::call_once(flag, [id](){
        // This code will be invoked only once.
        std::cout << "first: " << id << '\n';
    });
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 16; ++i) {
        threads.emplace_back(worker, i);
    }
    for (auto& t : threads) {
        t.join();
    }
}
```

Example: One-Time Initialization

```
#include <vector>
#include <thread>
#include <mutex>
#include <cassert>
#include <memory>

std::unique_ptr<int> value;
std::once_flag initFlag;

void initValue() {value = std::make_unique<int>(42);}

const int& getValue() {
    std::call_once(initFlag, initValue);
    return *value.get();
}

void doWork() {
    const int& v = getValue();
    assert(v == 42);
    // ...
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 4; ++i)
      {threads.emplace_back(doWork);}
    for (auto& t : threads) {t.join();}
}
```

Static Local Variable Initialization and Thread Safety

- initialization of static local object is thread safe
- object is initialized first time control passes through its declaration
- object deemed initialized upon completion of initialization
- if control enters declaration concurrently while object being initialized, concurrent execution waits for completion of initialization
- code like following is thread safe:

```
const std::string& meaningOfLife() {
    static const std::string x("42");
    return x;
}
```

Section 3.5.7

Condition Variables

Condition Variables

- In concurrent programs, the need often arises for a thread to ***wait until a particular event occurs*** (e.g., I/O has completed or data is available).
- Having a thread ***repeatedly check*** for the occurrence of an event can be ***inefficient*** (i.e., can waste processor resources).
- It is often better to have the thread block and then only resume execution after the event of interest has occurred.
- A **condition variable** is a synchronization primitive that allows threads to ***wait (by blocking)*** until a particular condition occurs.
- A condition variable corresponds to some event of interest.
- A thread that wants to wait for an event, performs a ***wait operation*** on the condition variable.
- A thread that wants to notify one or more waiting threads of an event performs a ***signal operation*** on the condition variable.
- When a signalled thread resumes, however, the signalled condition is not guaranteed to be true (and must be rechecked), since another thread may have caused condition to change.

The `std::condition_variable` Class

- `std::condition_variable` class provides condition variable
- ***not movable*** and ***not copyable***
- `wait`, `wait_for`, and `wait_until` member functions used to wait for condition
- `notify_one` and `notify_all` used to signal waiting thread(s) of condition
- must re-check condition when awaking from wait since:
  - spurious awakenings are permitted
  - between time thread is signalled and time it awakens and locks mutex, another thread could cause condition to change
- concurrent invocation is allowed for `notify_one`, `notify_all`, `wait`, `wait_for`, `wait_until`
- each of `wait`, `wait_for`, and `wait_until` atomically releases mutex and blocks
- `notify_one` and `notify_all` are atomic

std::condition_variable Members

Member Types

Name	Description
`native_handle_type`	system-dependent handle type for underlying condition variable entity

Construction, Destruction, and Assignment

Name	Description
constructor	construct object
destructor	destroy object
operator= [deleted]	not movable or copyable

std::condition_variable Members (Continued)

Notification and Waiting Member Functions

Name	Description
notify_one	notify one waiting thread
notify_all	notify all waiting threads
wait	blocks current thread until notified
wait_for	blocks current thread until notified or specified duration passed
wait_until	blocks current thread until notified or specified time point reached

Native Handle Member Functions

Name	Description
native_handle	get native handle associated with condition variable

Example: Condition Variable (IntStack)

```
#include <iostream>
#include <vector>
#include <thread>
#include <mutex>
#include <condition_variable>

class IntStack {
public:
    IntStack() {};
    IntStack(const IntStack&) = delete;
    IntStack& operator=(const IntStack&) = delete;
    int pop() {
        std::unique_lock lock(m_);
        c_.wait(lock, [this](){return !v_.empty();});
        int x = v_.back();
        v_.pop_back();
        return x;
    }
    void push(int x) {
        std::lock_guard lock(m_);
        v_.push_back(x);
        c_.notify_one();
    }
private:
    std::vector<int> v_;
    mutable std::mutex m_;
    mutable std::condition_variable c_; // not empty
};

constexpr int numIters = 1000;
IntStack s;

int main() {
    std::thread t1([](){
        for (int i = 0; i < numIters; ++i) s.push(2 * i + 1);
    });
    std::thread t2([](){
        for (int i = 0; i < numIters; ++i) std::cout << s.pop() << '\n';
    });
    t1.join(); t2.join();
}
```

The `std::condition_variable_any` Class

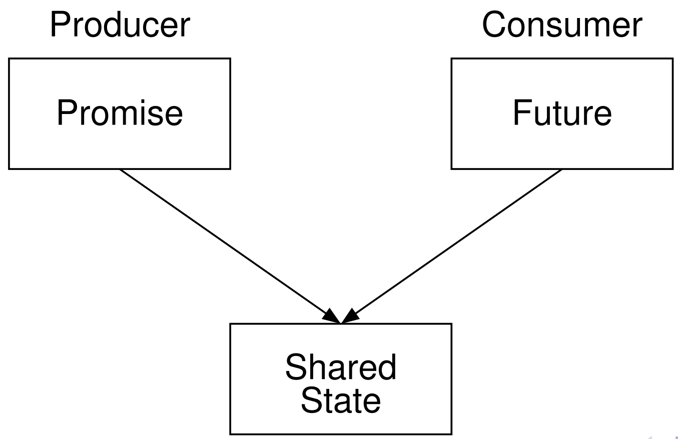
- with `std::condition_variable` class, `std::unique_lock<std::mutex>` class must be used for wait operation
- `std::condition_variable_any` class allows any mutex type (meeting certain basic requirements) to be used
- interface of `std::condition_variable_any` class similar to that of `std::condition_variable` class
- prefer `condition_variable` to `condition_variable_any` since former may be more efficient

Section 3.5.8

Promises and Futures

Promises and Futures

- promise and future together form ***one-time*** communication channel for passing result (i.e., value or exception) of computation from one thread to same or another thread
- **promise**: object associated with promised result (i.e., value or exception) to be produced
- **future**: object through which promised result later made available
- **shared state**: holds promised result for access through future object (shared by promise object and corresponding future object)
- producer of result uses promise object to store result in shared state
- consumer uses future object (corresponding to promise) to retrieve result from shared state



Promises and Futures (Continued)

- promises and futures useful in both single-threaded and multithreaded programs
- in single-threaded programs, might be used to propagate exception to another part of program
- in multithreaded program, often need arises to do some computation asynchronously and then later get result when ready
- requires synchronization between threads producing and consuming result
- thread consuming result must ***wait until result is available***
- must ***avoid data races*** when accessing result shared between threads
- this type of synchronization can be accomplished via promise and future

The `std::promise` Template Class

- `std::promise` provides access to promise-future shared state for writing result
- declaration:

```
template <class T> class promise;
```

- `T` is type of result associated with promise (which can be **`void`**)
- movable but ***not copyable***
- `set_value` member function sets result to particular value
- `set_exception` member function sets result to exception
- can set result ***only once***
- `get_future` member function retrieves future associated with promise
- `get_future` may be called ***only once***
- if `promise` object is destroyed before its associated result is set, `std::future_error` exception will be thrown if attempt made to retrieve result from corresponding `future` object

Construction, Destruction, and Assignment

Name	Description
constructor	construct object
destructor	destroy object
`operator=`	move assignment

Other Functions

Name	Description
`swap`	swap two promise objects
`get_future`	get future associated with promised result
`set_value`	set result to specified value
`set_value_at_thread_exit`	set result to specified value while delivering notification only at thread exit
`set_exception`	set result to specified exception
`set_exception_at_thread_exit`	set result to specified exception while delivering notification only at thread exit

The std::future Template Class

- std::future provides access to promise-future shared state for reading result
- declaration:

```
template <class T> class future;
```

- T is type of result associated with future (which can be **void**)
- movable but ***not copyable***
- get member function retrieves result, blocking if result not yet available
- get may be called ***only once***
- wait member function waits for result to become available without actually retrieving result

Construction, Destruction, and Assignment

Name	Description
constructor	construct object
destructor	destroy object
`operator``=`	move assignment

Other Functions

Name	Description
`share`	transfer shared state to `shared_future` object
`get`	get result
`valid`	check if `future` object refers to shared state
`wait`	wait for result to become available
`wait_for`	wait for result to become available or time duration to expire
`wait_until`	wait for result to become available or time point to be reached

Example: Promises and Futures (Without std::async)

```
#include <future>
#include <thread>
#include <iostream>
#include <utility>

double computeValue() {
    return 42.0;
}

void produce(std::promise<double> p) {
    // write result to promise
    p.set_value(computeValue());
}

int main() {
    std::promise<double> p;
    auto f = p.get_future(); // save future before move
    std::thread producer(produce, std::move(p));
    std::cout << f.get() << '\n';
    producer.join();
}
```

The std::shared_future Template Class

- std::shared_future similar to future except object can be copied
- shared_future object can be obtained by using share member function of future class to transfer contents of future object into shared_future object
- shared_future is ***copyable*** (unlike future)
- allows multiple threads to wait for same result (associated with shared_future object)
- get member can be called multiple times

```
#include <iostream>
#include <vector>
#include <thread>
#include <future>

void consume(std::shared_future<int> f) {
    std::cout << f.get() << '\n';
}

int main() {
    std::promise<int> p;
    std::shared_future f = p.get_future().share();
    std::vector<std::thread> consumers;
    for (int i = 0; i < 16; ++i) {
        consumers.emplace_back(consume, f);
    }
    p.set_value(42);
    for (auto& i : consumers) {
        i.join();
    }
}
```

The `std::async` Template Function

- `std::async` template function used to launch callable entity (e.g., function or functor) asynchronously
- declaration (uses default launch policy):

```
template <class Func, class... Args>
future<typename result_of<typename decay<Func>::type(
  typename decay<Args>::type...)>::type>
  async(Func&& f, Args&&... args);
```

- declaration (with launch policy parameter):

```
template <class Func, class... Args>
future<typename result_of<typename decay<Func>::type(
  typename decay<Args>::type...)>::type>
  async(launch policy, Func&& f, Args&&... args);
```

- numerous launch policies supported via bitmask `std::launch`
- if `async` bit set, execute on new thread
- if `deferred` bit set, execute on calling thread when result needed
- if multiple bits set, implementation free to choose between them
- in asynchronous execution case, essentially creates promise to hold result and returns associated future; launches thread to execute function/functor and sets promise when function/functor returns

The `std::async` Template Function (Continued)

- future (i.e., `future` and `shared_future`) objects created by `async` function have slightly different behavior than future objects created in other ways
- in case of future object created by `async` function: if future object is ***last*** future object referencing its shared state, destructor for future object will ***block*** until result associated with future object becomes ready

Example: Promises and Futures (With std::async)

```cpp
#include <future>
#include <iostream>

double computeValue() {
    return 42.0;
}

int main() {
    // invoke computeValue function asynchronously in
    // separate thread
    auto f = std::async(std::launch::async, computeValue);
    std::cout << f.get() << '\n';
}
```

Example: Futures and Exceptions

```
#include <iostream>
#include <vector>
#include <cmath>
#include <future>
#include <stdexcept>

double squareRoot(double x) {
    if (x < 0.0) {
        throw std::domain_error(
          "square root of negative number");
    }
    return std::sqrt(x);
}

int main() {
    std::vector values{1.0, 2.0, -1.0};
    std::vector<std::future<double>> results;
    for (auto x : values) {
        results.push_back(std::async(squareRoot, x));
    }
    for (auto& x : results) {
        try {
            std::cout << x.get() << '\n';
        } catch (const std::domain_error&) {
            std::cout << "error\n";
        }
    }
}
```

The `std::packaged_task` Template Class

- `std::packaged_task` template class provides wrapper for callable entity (e.g., function or functor) that makes return value available via future
- declaration:

```
template <class R, class... Args>
  class packaged_task<R(Args...)>;
```

- template parameters `R` and `Args` specify return type and arguments for callable entity
- similar to `std::function` except return value of wrapped function made available via future
- packaged task often used as thread function
- movable but ***not copyable***
- `get_future` member retrieves future associated with packaged task
- `get_future` can be called ***only once***

`std::packaged_task` Members

Construction, Destruction, and Assignment

Name	Description
constructor	construct object
destructor	destroy object
`operator``=`	move assignment

Other Functions

Name	Description
`valid`	check if task object currently associated with shared state
`swap`	swap two task objects
`get_future`	get future associated with promised result
`operator``()`	invoke function
`make_ready_at_thread_exit`	invoke function ensuring result ready only once current thread exits
`reset`	reset shared state, abandoning any previously stored result

Example: Packaged Task

```
#include <iostream>
#include <thread>
#include <future>
#include <utility>
#include <chrono>

int getMeaningOfLife() {
    // Let the suspense build before providing the answer.
    std::this_thread::sleep_for(std::chrono::milliseconds(
      1000));
    // Return the answer.
    return 42;
}

int main() {
    std::packaged_task<int()> pt(getMeaningOfLife);
    // Save the future.
    auto f = pt.get_future();
    // Start a thread running the task and detach the thread.
    std::thread t(std::move(pt));
    t.detach();
    // Get the result via the future.
    int result = f.get();
    std::cout << "The meaning of life is " << result << '\n';
}
```

Example: Packaged Task With Arguments

```
#include <iostream>
#include <cmath>
#include <thread>
#include <future>

double power(double x, double y) {
    return std::pow(x, y);
}

int main() {
    // invoke task in main thread
    std::packaged_task<double(double, double)> task(power);
    task(0.5, 2.0);
    std::cout << task.get_future().get() << '\n';
    // reset shared state
    task.reset();
    // invoke task in new thread
    auto f = task.get_future();
    std::thread t(std::move(task), 2.0, 0.5);
    t.detach();
    std::cout << f.get() << '\n';
}
```

Section 3.5.9

Atomics

Atomics

- To avoid data races when sharing data between threads, it is often necessary to employ ***synchronization*** (e.g., by using mutexes).
- Atomic types are another mechanism for providing synchronized access to data.
- An operation that is indivisible is said to be **atomic** (i.e., no parts of any other operations can interleave with any part of an atomic operation).
- Most processors support atomic memory operations via special machine instructions.
- Atomic memory operations cannot result in torn reads or torn writes.
- The standard library offers the following types in order to provide support for atomic memory operations:
  - `std::atomic_flag`
  - `std::atomic`
- These types provide a uniform interface for accessing the atomic memory operations of the underlying hardware.

Atomics (Continued)

- An atomic type provides guarantees regarding:
  1. atomicity; and
  2. ordering.
- An ordering guarantee specifies the manner in which memory operations can become visible to threads.
- Several memory ordering schemes are supported by atomic types.
- The default memory order is sequentially consistent (`std::memory_order_seq_cst`).
- Initially, only this default will be considered.

- `std::atomic_flag` provides flag with basic atomic operations
- flag can be in one of two states: set (i.e., true) or clear (i.e., false)
- two operations for flag:
  - **test and set**: set state to true and query previous state
  - **clear**: set state to false
- default constructor initializes flag to ***unspecified*** state
- ***not movable*** and ***not copyable***
- implementation-defined macro `ATOMIC_FLAG_INIT` can be used to set flag to clear state in (static or automatic) initialization using statement of the form "`std::atomic_flag f = ATOMIC_FLAG_INIT;`"
- guaranteed to be ***lock free***
- intended to be used as building block for higher-level synchronization primitives, such as spinlock mutex

std::atomic_flag Members

Member Functions

Member Name	Description
constructor	constructs object
`clear`	atomically sets flag to false
`test_and_set`	atomically sets flag to true and obtains its previous value

Example: Suboptimal Spinlock Mutex

```
#include <iostream>
#include <thread>
#include <atomic>
#include <mutex>

class SpinLockMutex {
public:
    SpinLockMutex() {f_.clear();}
    void lock() {while (f_.test_and_set()) {}}
    void unlock() {f_.clear();}
private:
    std::atomic_flag f_; // true if thread holds mutex
};

SpinLockMutex m;
unsigned long long counter = 0;

void doWork() {
    for (unsigned long long i = 0; i < 100000ULL; ++i)
    {std::lock_guard lock(m); ++counter;}
}

int main() {
    std::thread t1(doWork), t2(doWork);
    t1.join(); t2.join();
    std::cout << counter << '\n';
}
```

- default memory order is suboptimal (and will be revisited later)

Example: One-Time Wait

```
#include <iostream>
#include <atomic>
#include <thread>
#include <chrono>

// notReady flag initially not set
std::atomic_flag notReady = ATOMIC_FLAG_INIT;
int result = 0;

int main() {
    notReady.test_and_set(); // indicate result not ready
    std::thread producer([](){
        std::this_thread::sleep_for(std::chrono::seconds(1));
        result = -42;
        notReady.clear(); // indicate result ready
    });
    std::thread consumer([](){
        // loop until result ready
        while (notReady.test_and_set()) {}
        std::cout << result << '\n';
    });
    producer.join();
    consumer.join();
}
```

- This is ***not*** a particularly good use of `atomic_flag`.

The std::atomic Template Class

- `std::atomic` class provides types with atomic operations
- declaration:

```
template <class T> struct atomic;
```

- provides object of type `T` with atomic operations
- has partial specializations for integral types and pointer types
- full specializations for all fundamental types
- in order to use class type for `T`, `T` must be trivially copyable and bitwise equality comparable
- not required to be lock free
- on most popular platforms `atomic` is lock free when `T` is built-in type
- ***not move constructible*** and ***not copy constructible***
- assignable but assignment operator returns value not reference
- most operations have memory order argument
- default memory order is SC (`std::memory_order_seq_cst`)

Basic

Member Name	Description
constructor	constructs object
operator=	atomically store value into atomic object
is_lock_free	check if atomic object is lock free
store	atomically replaces value of atomic object with given value
load	atomically reads value of atomic object
operator T	obtain result of load
exchange	atomically replaces value of atomic object with given value and obtain value of previous value
compare_exchange_weak	similar to exchange_strong but may fail spuriously
compare_exchange_strong	atomically compare value of atomic object to given value and perform exchange if equal or load otherwise

`std::atomic` Members (Continued 1)

Fetch

Member Name	Description
`fetch_add`	atomically adds given value to value stored in atomic object and obtains value held previously
`fetch_sub`	atomically subtracts given value from value stored in atomic object and obtains value held previously
`fetch_and`	atomically replaces value of atomic object with bitwise AND of atomic object's value and given value, and obtains value held previously
`fetch_or`	atomically replaces value of atomic object with bitwise OR of atomic object's value and given value, and obtains value held previously
`fetch_xor`	atomically replaces value of atomic object with bitwise XOR of atomic object's value and given value, and obtains value held previously

`std::atomic` Members (Continued 2)

Increment and Decrement

Member Name	Description
operator++	atomically increment the value of atomic object by one and obtain value after incrementing
operator++(**int**)	atomically increment the value of atomic object by one and obtain value before incrementing
operator--	atomically decrement the value of atomic object by one and obtain value after decrementing
operator--(**int**)	atomically decrement the value of atomic object by one and obtain value after decrementing

Compound Assignment

Member Name	Description
operator+=	atomically adds given value to value stored in atomic object
operator-=	atomically subtracts given value from value stored in atomic object
operator&=	atomically performs bitwise AND of given value with value stored in atomic object
operator|=	atomically performs bitwise OR of given value with value stored in atomic object
operator^=	atomically performs bitwise XOR of given value with value stored in atomic object

Constants

Member Name	Description
is_always_lock_free	indicates if type always lock free

Example: Atomic Counter

```
#include <iostream>
#include <vector>
#include <thread>
#include <atomic>

class AtomicCounter {
public:
    AtomicCounter() : c_(0) {}
    int operator++() {return ++c_;}
    int get() const {return c_.load();}
private:
    std::atomic<int> c_;
};

AtomicCounter counter;

void doWork() {
    for (int i = 0; i < 10000; ++i) {++counter;}
}

int main() {
    std::vector<std::thread> v;
    for (int i = 0; i < 10; ++i)
        {v.emplace_back(doWork);}
    for (auto& t : v) {t.join();}
    std::cout << counter.get() << '\n';
}
```

Example: Atomic Increment With Compare and Swap

```
#include <atomic>

template <class T>
void atomicIncrement(std::atomic<T>& x) {
    T curValue = x;
    while (!x.compare_exchange_weak(curValue,
      curValue + 1)) {}
}
```

Example: Counting Contest

```
#include <iostream>
#include <vector>
#include <atomic>
#include <thread>

constexpr int numThreads = 10;
std::atomic ready(false);
std::atomic done(false);
std::atomic startCount(0);

void doCounting(int id) {
    ++startCount;
    while (!ready) {}
    for (volatile int i = 0; i < 20000; i++) {}
    bool expected = false;
    if (done.compare_exchange_strong(expected, true))
      {std::cout << "winner: " << id << '\n';}
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < numThreads; ++i)
      {threads.emplace_back(doCounting, i);}
    while (startCount != numThreads) {}
    ready = true;
    for (auto& t : threads) {t.join();}
}
```

An Obligatory Note on `volatile`

- `volatile` qualifier not useful for multithreaded programming
- `volatile` qualifier makes *no guarantee of atomicity*
- can create object of `volatile`-qualified type whose size is sufficiently large that no current processor can access object atomically
- some platforms may happen to guarantee memory operations on (suitably-aligned) `int` object to be atomic, but in such cases this is normally true *even without `volatile` qualifier*
- `volatile` qualifier *does not adequately address issue of memory consistency*
- `volatile` qualifier does not imply use of memory barriers or other mechanisms needed for memory consistency
- optimizer and hardware might reorder operations (on non-`volatile` objects) across operations on `volatile` objects

Section 3.5.10

Atomics and the Memory Model

Semantics of Multithreaded Programs

- To be able to reason about the behavior of a program, we must know:
  - the ***order*** in which the operations of the program are performed; and
  - when the effects of each operation become ***visible*** to other operations in the program, which may be performed in different threads.
- In a single-threaded program, the ordering of operations and when the effects of operations become visible is quite intuitive.
- In a multi-threaded program, this matter becomes ***considerably more complicated***.
- In what follows, we examine the above matter more closely (which essentially relates to the memory model).

Happens-Before Relationships

- For two operations A and B performed in the ***same or different*** threads, A is said to **happen before** B if the effects of A become ***visible*** to the thread performing B before B is performed.
- The happens-before relationship is ***not equivalent*** to "happens earlier in time".
- If operation A happens earlier in time than operation B, this does not imply that the effects of A must be ***visible*** to the thread performing B before B is performed, due to the effects of caches, store buffers, and so on, which ***delay*** the visibility of results.
- Happening earlier in time is only a necessary but not sufficient condition for a happens-before relationship to exist.
- Happens-before relationships are ***not always transitive***.
- In the absence of something known as a dependency-ordered-before relationship (to be discussed later), which arise relatively less frequently, happens-before relationships are ***transitive*** (i.e., if A happens before B and B happens before C then A happens before C).

"Earlier In Time" Versus Happens Before

- Consider the multithreaded program (with two threads) shown below, where `x` and `y` are integer variables, ***initially zero***.

Thread 1 Code	Thread 2 Code
`x = 1; // A`	`y = x; // B`

- Suppose that the run-time platform is such that memory operations on `x` are ***atomic*** so the program is data-race free.
- Consider what happens when the program executes with the particular timing shown below, where ***operation A occurs earlier in time than operation B***.

Time ↓	Thread 1 (on Core 1)	Thread 2 (on Core 2)
	`x = 1; //A`	
		`y = x; //B`

- The value read for `x` in operation B will not necessarily be 1.

Sequenced-Before Relationships

- Given two operations A and B performed in the *same* thread, the operation A is **sequenced before** B if A precedes B in program order (i.e., source-code order).
- Sequenced-before relationships are *transitive* (i.e., if A is sequenced before B, and B is sequenced before C, then A is sequenced before C).
- Example: In the code below, statement A is sequenced before statement B; B is sequenced before statement C; and, by transitivity, A is sequenced before C.

```
x = 1;    // A
y = 2;    // B
z = x + 1; // C
```

- Example:
  - Consider the line of code below, which performs (in order) the following operations: 1) multiplication, 2) addition, and 3) assignment.

    ```
    y = a * x + b; // (y = ((a * x) + b);
    ```

  - Multiplication is sequenced before addition.
  - Addition is sequenced before assignment.
  - Thus, by transitivity, multiplication is sequenced before assignment.

Sequenced-Before Relationships (Continued)

- For two operations A and B in the ***same*** thread, if A is ***sequenced before*** B then A ***happens before*** B.
- In other words, program order establishes happens-before relationships for operations ***within a single thread***.
- A sequenced-before relationship is essentially an ***intra-thread happens-before*** relationship. (Note that "intra" means "within".)
- Example: In the code below, statement A is sequenced before statement B. Therefore, A happens before B. Similarly, B happens before statement C, and A happens before C.

```
x = 1;     // A
y = 2;     // B
z = x + 1; // C
```

Inter-Thread Happens-Before Relationships

- Establishing whether a happens-before relationship exists between operations in different threads is somewhat more complicated than the same-thread case.
- Inter-thread happens-before relationships establish happens-before relationships for operations in ***different*** threads.
- For two operations A and B in ***different*** threads, if A **inter-thread happens before** B then A happens before B.
- Inter-thread happens-before relationships are ***transitive*** (i.e., if A inter-thread happens before B and B inter-thread happens before C then A inter-thread happens before C).
- Some form of ***synchronization*** is required to establish an inter-thread happens-before relationship.
- The various forms that this synchronization may take will be introduced on later slides.

Summary of Happens-Before Relationships

- For two operations A and B in either the ***same or different*** threads, A happens before B if:
  1. A and B are in the ***same*** thread and A is sequenced before (i.e., intra-thread happens before) B; or
  2. A and B are in ***different*** threads and A inter-thread happens before B.
- In other words, A happens before B if A either intra-thread happens before or inter-thread happens before B.
- Intra-thread happens-before (i.e., sequenced-before) relationships are ***transitive***.
- Inter-thread happens-before relationships are ***transitive***.
- Happens-before relationships are ***mostly but not always transitive***.
- A happens-before relationship is important because it tells us if the result of one operation ***can be seen*** by a thread performing another operation.

Synchronizes-With Relationships

- A variety of relationships can imply an inter-thread happens-before relationship, with one being the synchronizes-with relationship.
- For two operations A and B in ***different*** threads, if A **synchronizes with** B then A ***inter-thread happens before*** B.
- Example:
  - Consider the two-threaded program shown below, with the shared variable `x` of type **`int`**, where `x` is initially zero.

Thread 1 Code

```
x = 1;
// A (call of foo)
foo();
```

Thread 2 Code

```
bar();
// B (return from bar)
assert(x == 1);
```

  - Suppose that the call of the function `foo` is known to ***synchronize with*** the return from the function `bar`, which implies that A synchronizes with B.
  - Since A synchronizes with B, A must inter-thread happen before B, which implies that ***A happens before B***.
  - Therefore, the assertion in thread 2 ***can never fail***.

Examples of Synchronizes-With Relationships

- ***Thread creation.*** The completion of the constructor for a `thread` object T synchronizes with the start of the invocation of the thread function for T.
- ***Thread join.*** The completion of the execution of a thread function for a `thread` object T synchronizes with (the return of) a `join` operation on T.
- ***Mutex unlock/lock.*** All prior `unlock` operations on a mutex M synchronize with (the return of) a `lock` operation on M.
- ***Atomic.*** A suitably tagged atomic write operation W on a variable `x` synchronizes with a suitably tagged atomic read operation on `x` that reads the value stored by W (where the meaning of "suitably tagged" will be discussed later).

Synchronizes-With Relationship: Thread Create and Join

```cpp
#include <thread>
#include <cassert>

int x = 0;

void doWork() {
    // A1 (start of thread execution)
    assert(x == 1); // OK: M1 synchronizes with A1
    x = 2;
    // A2 (end of thread execution)
}

int main() {
    x = 1;
    std::thread t(doWork); // M1 (completion of constructor)
    t.join(); // M2 (return from join)
    assert(x == 2); // OK: A2 synchronizes with M2
}
```

- since construction of thread (`M1`) synchronizes with start of thread function execution (`A1`), `M1` happens before `A1` implying that assertion in `doWork` cannot fail
- since completion of execution of thread function (`A2`) synchronizes with `join` operation (`M2`), `A2` happens before `M2` implying that assertion in `main` cannot fail

Synchronizes-With Relationship: Mutex Lock/Unlock

Shared Data

```
std::mutex m;
int x = 0;
int y = 0;
```

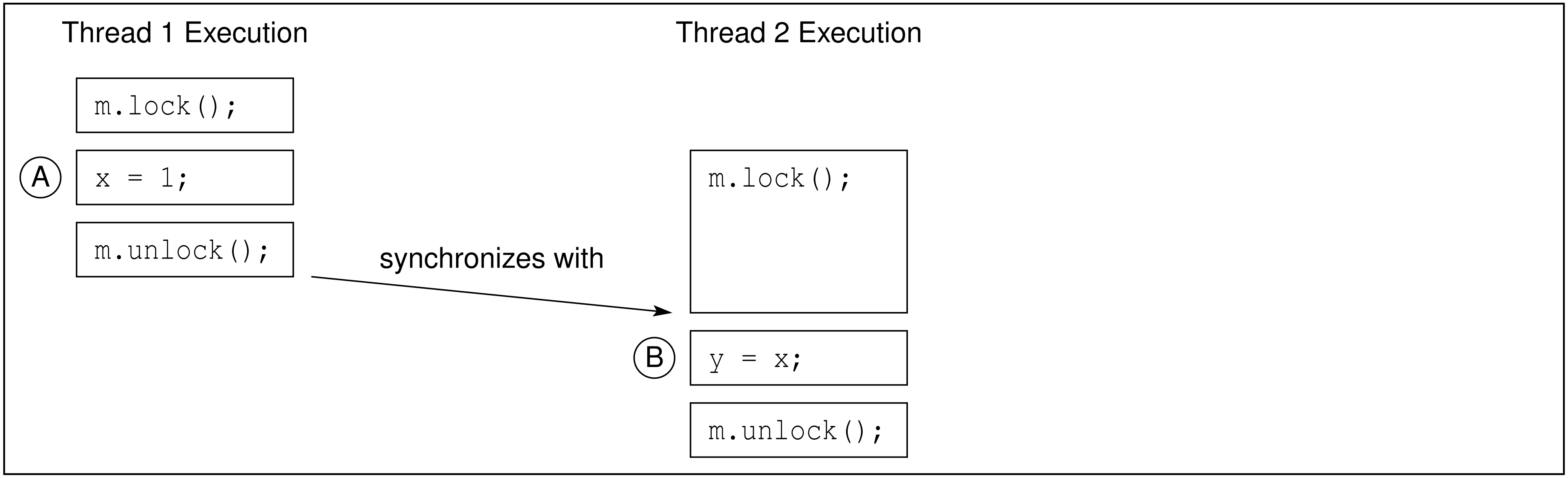
Thread 1 Code

```
m.lock();
x = 1;
m.unlock();
```

Thread 2 Code

```
m.lock();
y = x;
m.unlock();
```

Execution



- since unlock synchronizes with lock, A happens before B; thus, for timing shown, B must see `1` for `x`

- Most operations on atomic types allow a memory order to be specified.
- Example:

```
std::atomic<int> x = 0;
x.store(42, std::memory_order_seq_cst);
int y = x.load(std::memory_order_seq_cst);
```

- The following memory orders are supported:
  - sequentially consistent (`std::memory_order_seq_cst`)
  - acquire-release (`std::memory_order_acq_rel`)
  - acquire (`std::memory_order_acquire`)
  - release (`std::memory_order_release`)
  - consume (`std::memory_order_consume`)
  - relaxed (`std::memory_order_relaxed`)
- Read operations can use the orders:
  - sequentially consistent, acquire, consume, and relaxed.
- Write operations can use the orders:
  - sequentially consistent, release, and relaxed.
- Read-modify-write operations can use:
  - all of the orders allowed for read and write operations; and
  - acquire-release.

Memory Models

- Although several memory orders can be employed for operations on atomic types, these orders support ***four basic models***:
  1. sequentially consistent,
  2. acquire release,
  3. consume release, and
  4. relaxed.
- These models differ in the guarantees that they make regarding:
  - whether all writes to all atomic objects become visible to ***all*** threads ***simultaneously*** (i.e., total order for all writes to all atomic objects); and
  - whether operations on atomic objects in different threads can establish a ***synchronization*** relationship (namely, a synchronizes-with or dependency-ordered-before [discussed later] relationship).
- The models listed from strongest (i.e., makes the most guarantees) to weakest (i.e., makes the least guarantees) are:
  1. sequentially consistent,
  2. acquire release,
  3. consume release, and
  4. relaxed.

Memory Models (Continued 1)

- These models are ***hierarchical*** in the sense that each model makes at least all of the same guarantees as its weaker counterparts.
- As we proceed from stronger to weaker models, more guarantees are lost.
- A stronger model may require additional synchronization by hardware, which can ***degrade performance***.
- A weaker model ***may not provide sufficient guarantees*** for the correct functioning of code.
- Using a model that fails to provide sufficient guarantees for correct code behavior will result in ***bugs***.
- Also, as the model is weakened, it becomes more difficult to reason about the behavior of code, leading to ***incomprehensible code*** and an ***increased likelihood of (often very subtle) bugs***.

Modification Order

- All writes to a particular atomic object M (over its lifetime) occur in some particular total order, called its **modification order**.
- Each atomic object has its own well-defined modification order.
- For a particular atomic object M, ***all*** threads in a program are guaranteed to see M change in a manner ***consistent with its modification order***.
- Essentially, this guarantee ensures that, once a given thread has seen a particular value of an atomic object, a subsequent read by that thread cannot retrieve an earlier value of the object.
- If such a guarantee were not made, the memory model would be so weak as to be impractical to use.
- Modification order is primarily a ***conceptual*** tool that is useful for describing memory-model behavior.
- In practice, a thread is unlikely to actually observe every change in the modification order of an object.

Modification Order (Continued)

- For each atomic object M, each thread has its own current position in object's modification order.
- A thread's current position in the modification order of a particular atomic object need not be the same for all threads.
- A read from an atomic object M by a thread T can ***optionally*** move T's current position to a later position in the modification order of M and then returns the value at the current position.
- A write to an atomic object M by a thread T appends the value to be written to the modification order of M and updates T's current position in the modification order of M to correspond to the value written.
- An read-modify-write operation A on an atomic object M reads the ***last*** value in the modification order of M, modifies the value read appropriately, appends the resulting value to the modification order of M, and updates T's current position in the modification order of M to correspond to the value written.

Modification Order Example

- Consider an atomic object M with the modification sequence:
  - 0, 1, 2, 3, 4, 5, 6, 7, 8.
- A thread could, for example, legitimately see M undergo any of the following sequences of updates:
  - 0, 4, 8
  - 8
  - 2, 7
  - 0, 1, 2, 5, 7, 8
  - 0, 1, 2, 3, 4, 5, 6, 7, 8
- A thread would, for example, be guaranteed ***never*** to see M undergo any of the following sequences of updates, as all of these sequences are ***inconsistent*** with the modification order of M:
  - 1, 0
  - 1, 2, 1
  - 42
  - 0, 1, 2, 3, 4, 5, 6, 7, 6, 8

Relative Ordering of Changes to Different Atomic Objects

- Although each atomic object has its own well-defined modification order, it is not necessarily the case that the modification orders for individual objects can be combined into a single total order over ***all*** atomic objects.
- Practically speaking, the reason for this is the delay in the visibility of results introduced by store buffers, caches, and so on.
- If a single total order for writes to all atomic objects is not guaranteed, this implies that the relative order of changes to ***different*** atomic objects need not appear the same to different threads.
- Ensuring the existence of a single total order over all atomic objects would require a significant amount of additional processor synchronization, which can significantly degrade performance.
- Therefore, this guarantee is not required to be made in all cases, the idea being that we only ask for the guarantee when it is needed for correct code behavior.

Modification Order Revisited

- Consider a program with two threads and two shared integer atomic objects `x` and `y`, each having the modification order: 0, 1.
- Suppose that no requirement is imposed to guarantee the existence of a single total order on writes to ***all*** atomic objects.
- Thread 1 could see `x` and `y` change in the following manner, consistent with their stated modification order:

Variable	Updates to Value Seen By Thread
`x`	0 ········ 1 ··························
`y`	0 ·························· 1 ········

- Thread 2 could see `x` and `y` change in the following manner, consistent with their stated modification order:

Variable	Updates to Value Seen By Thread
`x`	0 ·························· 1 ········
`y`	0 ········ 1 ··························

- Observe that thread 1 and thread 2 do not see `x` and `y` change in the same order relative to one another (i.e., thread 1 sees `x` change before `y`, while thread 2 sees `y` change before `x`).

Sequentially-Consistent Model

- The sequentially-consistent model simply corresponds to the default memory model for the language, namely, SC-DRF. (Since data races cannot occur on atomic objects, SC-DRF degenerates into SC for such objects.)
- For the sequentially-consistent model, all memory operations (i.e., read, write, and read-modify-write) must use the sequentially-consistent memory order (`std::memory_order_seq_cst`).
- A ***total ordering*** is guaranteed on all sequentially-consistent writes to ***all*** atomic objects.
- All sequentially-consistent writes to atomic objects must become **visible** to all threads ***simultaneously***.
- A sequentially-consistent write operation W on an atomic object M (in one thread) ***synchronizes with*** a sequentially-consistent operation on M (in another thread) that reads the value written by W.
- This model allows for relatively ***easy reasoning*** about code behavior.

Example: Sequentially-Consistent Model

- shared data:

  x and y are of type std::atomic<**int**> and both are initially zero
- thread 1 code (writes x):

```
x.store(1, std::memory_order_seq_cst);
```

- thread 2 code (writes y):

```
y.store(1, std::memory_order_seq_cst);
```

- thread 3 code (reads x then y):

```
int x1 = x.load(std::memory_order_seq_cst);
int y1 = y.load(std::memory_order_seq_cst);
```

- thread 4 code (reads y then x):

```
int y2 = y.load(std::memory_order_seq_cst);
int x2 = x.load(std::memory_order_seq_cst);
```

- memory order guarantees total order for all writes to all atomic objects
- so, thread 3 and thread 4 must agree about order in which x and y are modified
- not possible to see `x1 == 1` and `y1 == 0` in thread 3 (implying x modified before y) and `x2 == 0` and `y2 == 1` in thread 4 (implying y modified before x)

Example: Sequentially-Consistent Model

```cpp
#include <atomic>
#include <thread>
#include <cassert>

std::atomic<int> x, y, c;

void w_x() {x.store(1, std::memory_order_seq_cst);}

void w_y() {y.store(1, std::memory_order_seq_cst);}

void r_xy() {
    while (!x.load(std::memory_order_seq_cst)) {}
    if (y.load(std::memory_order_seq_cst)) {++c;}
}

void r_yx() {
    while (!y.load(std::memory_order_seq_cst)) {}
    if (x.load(std::memory_order_seq_cst)) {++c;}
}

int main() {
    x = 0; y = 0; c = 0;
    std::thread t1(w_x), t2(w_y), t3(r_xy), t4(r_yx);
    t1.join(); t2.join(); t3.join(); t4.join();
    assert(c != 0); // assertion cannot fail
}
```

- assertion cannot fail: when **`while`** loop in `r_xy` terminates, all threads must see `x` as nonzero; when **`while`** loop in `r_yx` terminates, all threads must see `y` as nonzero; at least one of these must happen before **`if`** statements in both `r_xy` and `r_yx` executed

Acquire-Release Model

- For the acquire-release model, the memory order is chosen as follows:
  - a read operation uses the acquire order (`std::memory_order_acquire`)
  - a write operation uses the release order (`std::memory_order_release`)
  - a read-modify-write operation uses one of the orders allowed for read and write operations, or the acquire-release order (`std::memory_order_acq_rel`), which results in read acquire and write release.
- ***No total ordering*** exists on all writes to ***all*** atomic objects (unlike in the sequentially-consistent model).
- Consequently, threads do not necessarily have to agree on the ***relative order*** in which different atomics objects are modified.
- A write-release operation W on an atomic object M ***synchronizes with*** a read-acquire operation on M that reads the value written by W (or a value written by the release sequence headed by W).
- The acquire-release model is useful for situations that involve ***pairwise synchronization*** of threads, such as with mutexes.
- With the acquire-release model, it is often still possible to reason about code behavior without too much difficulty.

Example: Acquire-Release Model

- shared data:

  x and y are of type `std::atomic<int>` and both are initially zero
- thread 1 code (writes x):

```
x.store(1, std::memory_order_release);
```

- thread 2 code (writes y):

```
y.store(1, std::memory_order_release);
```

- thread 3 code (reads x then y):

```
int x1 = x.load(std::memory_order_acquire);
int y1 = y.load(std::memory_order_acquire);
```

- thread 4 code (reads y then x):

```
int y2 = y.load(std::memory_order_acquire);
int x2 = x.load(std::memory_order_acquire);
```

- no ordering relationship between stores to x and y
- so, thread 3 and thread 4 do not need to agree about order in which x and y are modified
- possible to see `x1 == 1` and `y1 == 0` in thread 3 (i.e., thread 3 sees x change before y) and `x2 == 0` and `y2 == 1` in thread 4 (i.e., thread 4 sees y change before x)

Example: Acquire-Release Model

```
#include <atomic>
#include <thread>
#include <cassert>

std::atomic<int> x, y, c;

void w_x() {x.store(1, std::memory_order_release);}

void w_y() {y.store(1, std::memory_order_release);}

void r_xy() {
    while (!x.load(std::memory_order_acquire)) {}
    if (y.load(std::memory_order_acquire)) {++c;}
}

void r_yx() {
    while (!y.load(std::memory_order_acquire)) {}
    if (x.load(std::memory_order_acquire)) {++c;}
}

int main() {
    x = 0; y = 0; c = 0;
    std::thread t1(w_x), t2(w_y), t3(r_xy), t4(r_yx);
    t1.join(); t2.join(); t3.join(); t4.join();
    assert(c != 0); // assertion can fail
}
```

- assertion can fail: one thread seeing `x` or `y` being nonzero does not imply other thread sees same

Example: Spinlock Mutex Using `std::atomic_flag`

```
#include <iostream>
#include <thread>
#include <atomic>

class SpinLockMutex {
public:
    SpinLockMutex() {f_.clear();}
    void lock() {
        while (f_.test_and_set(std::memory_order_acquire)) {}
    }
    void unlock() {f_.clear(std::memory_order_release);}
private:
    std::atomic_flag f_; // true if thread holds mutex
};

SpinLockMutex m;
unsigned long long counter = 0;

void doWork() {
    for (unsigned long long i = 0; i < 100000ULL; ++i)
    {m.lock(); ++counter; m.unlock();}
}

int main() {
    std::thread t1(doWork), t2(doWork);
    t1.join(); t2.join();
    std::cout << counter << '\n';
}
```

- uses acquire-release model

Example: Spinlock Mutex and std::lock_guard

```
#include <iostream>
#include <thread>
#include <atomic>
#include <mutex>

class SpinLockMutex {
public:
    SpinLockMutex() {f_.clear();}
    void lock() {
        while (f_.test_and_set(std::memory_order_acquire)) {}
    }
    void unlock() {f_.clear(std::memory_order_release);}
private:
    std::atomic_flag f_; // true if thread holds mutex
};

SpinLockMutex m;
unsigned long long counter = 0;

void doWork() {
    for (unsigned long long i = 0; i < 100000ULL; ++i)
    {std::lock_guard lg(m); ++counter;}
}

int main() {
    std::thread t1(doWork), t2(doWork);
    t1.join(); t2.join();
    std::cout << counter << '\n';
}
```

Carries-A-Dependency Relationships

- For two operations A and B performed in the ***same*** thread, A is said to **carry a dependency** to B if the result of A is used as an operand for B (ignoring some special cases).
- Example: In the code below, statement A ***carries a dependency*** to statement B but not statement C.

```
x = 42;    // A
y = x + 1; // B
z = 0;     // C
```

- Note that "carries a dependency to" is a subset of "is sequenced before" (i.e., the former implies the latter).
- The carries-a-dependency-to relationship is ***transitive*** (i.e., if A carries a dependency to B and B carries a dependency to C then A carries a dependency to C).
- Example: In the code below, statement A carries a dependency to statement B; and B carries a dependency to statement C. Therefore, transitively, A carries a dependency to C.

```
x = 42;    // A
y = x + 1; // B
z = 2 * y; // C
```

Dependency-Ordered-Before Relationships

- Another type of synchronization relationship is known as a dependency-ordered-before relationship.
- A write-release operation A is ***dependency ordered before*** a read-consume operation B if B reads the value written by A (or any side effect in the release sequence headed by A).
- For two operations A and B performed in ***different*** threads, if A is dependency ordered before B then A ***inter-thread happens before*** B.
- Thus, dependency-ordered-before relationships can also establish happens-before relationships.

Inter-Thread Happens-Before Relationships Revisited

- The inter-thread happens before relation describes an ***arbitrary concatenation*** of sequenced-before, synchronizes-with, and dependency-ordered-before relations, ***with two exceptions***:
  1. a concatenation is not permitted to end with dependency ordered before followed by (one or more) sequenced before; and
  2. a concatenation is not permitted to consist entirely of sequenced-before relations.
- The first restriction is required since a dependency-ordered-before relationship synchronizes ***only data dependencies***.
- The second restriction is required since inter-thread happens-before relationship must (by definition) involve operations in ***different*** threads.

Consume-Release Model

- For the consume-release model, the memory order is chosen as follows:
  - a write operation uses release order (`std::memory_order_release`)
  - a read operation uses the consume order (`std::memory_order_consume`)
- The consume-release model is identical to the acquire-release model with one important difference, namely the type of synchronization relationship established.
- A write-release operation W is ***dependency ordered before*** a read-consume operation (in a different thread) that reads the value stored by W (or any side effect in the release sequence headed by W).
- In other words, the consume-release model establishes a ***dependency-ordered-before*** relationship, whereas the acquire-release model establishes a ***synchronizes-with*** relationship.
- In this sense, the consume-release model is weaker than the acquire-release model (i.e., less data is synchronized).

Example: Consume-Release Model

```
#include <thread>
#include <atomic>
#include <cassert>

int x = 0;
std::atomic y(0);

void producer() {
    x = 42;
    y.store(1, std::memory_order_release);
}

void consumer() {
    int a;
    while (!(a = y.load(std::memory_order_consume))) {}
    assert(x == 42); // data race
}

int main() {
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join();
    t2.join();
}
```

- program has ***data race*** on `x`; `a` does not carry dependency to `x` so `x = 42` does not necessarily happen before `x` used in assertion
- if consume changed to acquire, no data race and assertion cannot fail

Example: Publishing Data Via Pointer

```
#include <thread>
#include <atomic>
#include <cassert>
#include <string>

std::atomic<std::string*> p(nullptr);
int x = 0;

void producer() {
    std::string* s = new std::string("Hello");
    x = 42;
    p.store(s, std::memory_order_release);
}

void consumer() {
    std::string* s;
    while (!(s = p.load(std::memory_order_consume))) {}
    assert(*s == "Hello");
    // assert(x == 42); would result in data race
}

int main() {
    std::thread t1(producer), t2(consumer);
    t1.join(); t2.join();
}
```

- assertion cannot fail; store to `p` is dependency ordered before load and load carries dependency to `*s` in assertion

Relaxed Model

- For the relaxed model, all memory operations use the relaxed order (`std::memory_order_relaxed`).
- Like in the acquire-release model, ***no total order*** exists on updates to ***all*** atomic objects (collectively).
- Operations on the same variable ***within a single thread*** satisfy a happens-before relationship (i.e., within a single thread, accesses to a single atomic variable must follow program order).
- Unlike in the acquire-release model, ***no inter-thread synchronization*** relationship is established.
- No requirement exists on the ordering relative to other threads.
- The relaxed order is sometime suitable for updating counters (e.g., blind event counters).
- Except in very trivial cases, it can be ***extremely difficult to reason*** about the meaning and/or correctness of code that uses relaxed order.

Behavior of Relaxed Model

- consider atomic memory operations with relaxed order
- for each individual atomic object, all threads have view of updates that is consistent with single modification sequence
- read operation (e.g., `load`):
  - if current position not set, return any element in sequence and set current position to that of returned element
  - otherwise, either leave current position unchanged or move later in sequence and return value at current position
- write operation (e.g., `store`):
  - append value to end of sequence
  - set current position to correspond to appended value
- read-modify-write operation (e.g., increment, decrement, `exchange`, `compare_exchange`):
  - read last value from sequence
  - modify read value as appropriate to obtain new value
  - append new value to end of sequence
  - set current position to correspond to that of appended value
- considerable flexibility in value returned by read

Example: Relaxed Model

```
#include <atomic>
#include <thread>
#include <cassert>

std::atomic<int> x, y, c;

void w_x() {x.store(1, std::memory_order_relaxed);}

void w_y() {y.store(1, std::memory_order_relaxed);}

void r_xy() {
    while (!x.load(std::memory_order_relaxed)) {}
    if (y.load(std::memory_order_relaxed)) {++c;}
}

void r_yx() {
    while (!y.load(std::memory_order_relaxed)) {}
    if (x.load(std::memory_order_relaxed)) {++c;}
}

int main() {
    x = 0; y = 0; c = 0;
    std::thread t1(w_x), t2(w_y), t3(r_xy), t4(r_yx);
    t1.join(); t2.join(); t3.join(); t4.join();
    assert(c != 0); // assertion can fail
}
```

- assertion can fail: one thread seeing `x` or `y` being nonzero does not imply other thread sees same

Example: Blind Event Counters

```
#include <vector>
#include <iostream>
#include <thread>
#include <atomic>

std::atomic<unsigned long long> counter(0);

void doWork() {
    for (long i = 0; i < 100'000L; ++i) {
        counter.fetch_add(1, std::memory_order_relaxed);
    }
}

int main() {
    std::vector<std::thread> workers;
    for (int i = 0; i < 10; ++i) {
        workers.emplace_back(doWork);
    }
    for (auto& t : workers) {
        t.join();
    }
    std::cout << "counter " << counter << '\n';
}
```

- `fetch_add` can use ***relaxed*** order, since only incrementing counter ***blindly*** (i.e., not taking action based on value of counter)
- thread join operations provide synchronization to ensure desired value read for counter when output

Example: Done Flag

```
#include <vector>
#include <thread>
#include <atomic>
#include <chrono>

std::atomic<bool> done;

void doWork() {
    while (!done.load(std::memory_order_relaxed)) {
        // do something here
    }
}

int main() {
    std::vector<std::thread> workers;
    done.store(false, std::memory_order_relaxed); // I hope? ;)
    for (int i = 0; i < 16; ++i) {
        workers.emplace_back(doWork);
    }
    std::this_thread::sleep_for(std::chrono::seconds(5));
    done = true; // not relaxed
    for (auto& t : workers) {
        t.join();
    }
}
```

- `done.store` can be relaxed due to synchronization from thread create
- `done.load` can be relaxed since order not important; different order as if other threads ran at different speeds
- assign to `done` must be sequentially-consistent to prevent assign from floating past join (due to single-thread optimization)

Example: std::shared_ptr Reference Counting

- The copy constructor for `shared_ptr` (which increments a reference count) would look something like:

```
// ...
controlBlockPtr = other->controlBlockPtr;
controlBlockPtr->refCount.fetch_add(1,
  std::memory_order_relaxed);
// ...
```

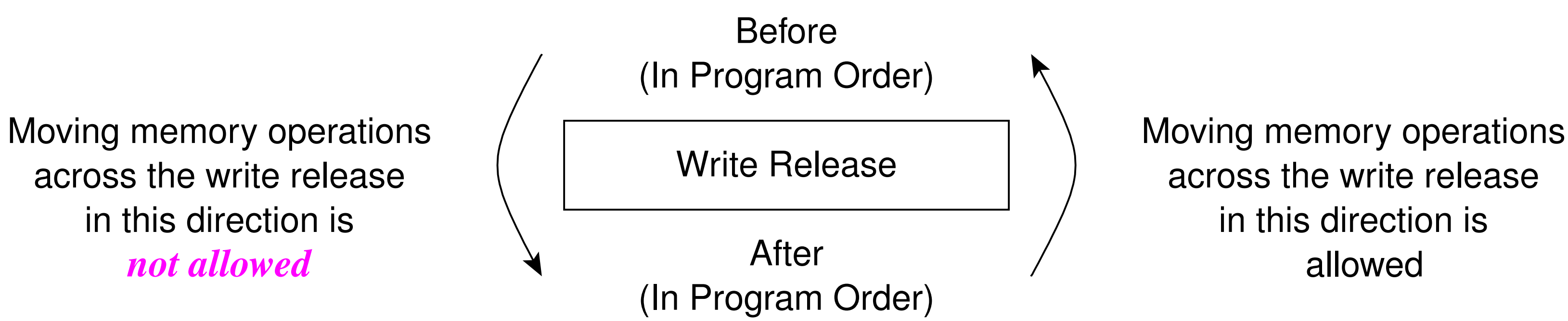
- The destructor for `shared_ptr` (which decrements a reference count) would look something like:

```
// ...
if (!controlBlockPtr->refCount.fetch_sub(1,
  std::memory_order_acq_rel)) {
    delete controlBlockPtr;
}
// ...
```

- The increment operation can use ***relaxed*** order, since ***no action is taken*** based on the reference count value.
- The decrement operation needs to use ***acquire-release*** order so that the decrement cannot float and the correct view of the data is seen by the thread doing the delete (all decrements form a ***synchronization chain***).

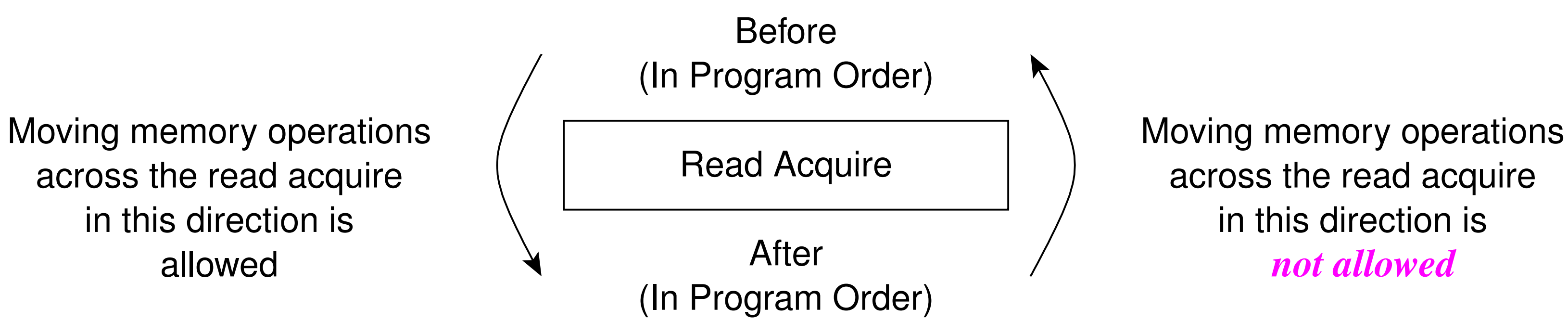
Release Semantics for Memory Operations

- Release semantics is a property that can only apply to operations that ***write to memory*** (i.e., read-modify-write operations or plain writes).
- A write operation that has release semantics is called a **write release**.
- A write release operation W cannot be reordered with any read or write operation that ***precedes*** W in program order (i.e., memory operations cannot be moved from before W to after W).
- The term release semantics originates from mutexes.
- In the context of mutexes, the operations prior to a mutex release operation, which correspond to operations in a critical section, must not be moved after the mutex release operation, as operations after the mutex release operation are not protected by the mutex.



Acquire Semantics for Memory Operations

- Acquire semantics is a property that can only apply to operations that ***read from memory*** (i.e., read-modify-write operations or plain reads).
- A read operation that has acquire semantics is called a **read acquire**.
- A read acquire operation R cannot be reordered with any read or write operation that ***follows*** R in program order (i.e., memory operations cannot be moved from after R to before R).
- The term acquire semantics originates from mutexes.
- In the context of mutexes, the operations following a mutex acquire operation, which correspond to operations in a critical section, must not be moved before the mutex acquire operation, as operations before the mutex acquire operation are not protected by the mutex.



Release Sequences

- A **release sequence** headed by a release operation A on an atomic object M is a maximal contiguous subsequence of side effects in the modification order of M, where the first operation is A, and every subsequent operation
  - is performed by the same thread that performed A, or
  - is an atomic read-modify-write operation.

Release Sequence Example

```
#include <thread>
#include <atomic>
#include <cassert>

int x = 0;
std::atomic y(0);

int main() {
    std::thread t1([](){
        x = 42;
        y.store(1, std::memory_order_release); // A
        y.store(2, std::memory_order_relaxed); // B
    });
    std::thread t2([](){
        int r;
        while ((r = y.load(std::memory_order_acquire)) // C
          < 2) {}
        assert(x == 42);
    });
    t1.join();
    t2.join();
}
```

- stores to `y` in `A` and `B` constitute release sequence headed by store in `A`
- when while loop terminates, load in `C` will have read value written by store in `B` (not store in `A`)
- `A` synchronizes with `C`, since `C` reads value in release sequence headed by `A`
- assertion cannot fail, since `A` happens before `C`

Fences

- A **memory fence** (also known as a **memory barrier**) is an operation that causes the processor and compiler to enforce an ***ordering constraint*** on memory operations issued before and after the fence operation.
- Certain types of memory operations before a fence are guaranteed not to be reordered with certain types of memory operations after the fence.
- A fence may also introduce ***synchronizes-with*** relationships under certain circumstances.
- An **acquire fence** prevents the reordering of any ***read or write*** following the fence (in program order) with any ***read*** prior to the fence (in program order). (That is, a memory operation after the fence cannot be moved before any read operation before the fence.)
- A **release fence** prevents the reordering of any ***read or write*** prior to the fence (in program order) with any ***write*** following the fence (in program order). (That is, a memory operation before the fence cannot be moved after any write operation after the fence.)
- A fence is ***not*** a release or acquire operation. as it does not read/write memory.

- memory fences can be inserted via function `std::atomic_thread_fence`
- declaration:

```
void atomic_thread_fence(std::memory_order order)
  noexcept;
```

- no effect if `order` is `std::memory_order_relaxed`
- acquire fence if `order` is `std::memory_order_acquire` or `std::memory_order_consume`
- release fence if `order` is `std::memory_order_release`
- both acquire and release fence if `order` is `std::memory_order_acq_rel`
- sequentially consistent acquire and release fence if `order` is `std::memory_order_seq_cst`

Fences and Synchronizes-With Relationships

- ***Release fence and acquire fence.*** A release fence A synchronizes with an acquire fence B if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.
- ***Release fence and acquire operation.*** A release fence A synchronizes with an atomic operation B that performs an acquire operation on an atomic object M if there exists an atomic operation X such that A is sequenced before X, X modifies M, and B reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.
- ***Release operation and acquire fence.*** An atomic operation A that is a release operation on an atomic object M synchronizes with an acquire fence B if there exists some atomic operation X on M such that X is sequenced before B and reads the value written by A or a value written by any side effect in the release sequence headed by A.

Example: Incorrect Code Without Fence

```
#include <thread>
#include <atomic>
#include <iostream>

std::atomic ready(false);
int data = 0;

void produce() {
    data = 42; // write to data can move after store in A
    // release fence needed here
    ready.store(true, std::memory_order_relaxed); // A
}

void consume() {
    while (!ready.load(std::memory_order_relaxed)) {} // B
    // acquire fence needed here
    std::cout << data << '\n';
      // read of data can move before load in B
}

int main() {
    std::thread t1(produce);
    std::thread t2(consume);
    t1.join(); t2.join();
}
```

- atomic store (to `ready`) does not synchronize with atomic load (of `ready`), due to relaxed order; results in race on `data`

Example: Correct Code With Fence

```
#include <thread>
#include <atomic>
#include <iostream>

std::atomic ready(false);
int data = 0;

void produce() {
    data = 42;
    std::atomic_thread_fence(std::memory_order_release);
    ready.store(true, std::memory_order_relaxed);
}

void consume() {
    while (!ready.load(std::memory_order_relaxed)) {}
    std::atomic_thread_fence(std::memory_order_acquire);
    std::cout << data << '\n';
}

int main() {
    std::thread t1(produce);
    std::thread t2(consume);
    t1.join(); t2.join();
}
```

- release fence synchronizes with acquire fence, due to atomic load (of `ready`) reading from result of atomic store (to `ready`)

Memory Orders: The Bottom Line

- Use sequentially-consistent order unless there is a compelling case to do otherwise.
- In situations where semantics dictate a clear pairwise synchronization between threads, consider the use of acquire-release order if it can be easily seen to yield correct code.
- Only consider relaxed order in situations where the performance penalty of using a stronger order would be unacceptable.
- ***Be very wary of using relaxed order.*** Even world experts on the C++ memory model acknowledge that this can be tricky.
- Always have any code using relaxed order thoroughly reviewed by people who are extremely knowledgeable about memory models.

Section 3.5.11

References

1. A. Williams. *C++ Concurrency in Action.*
Manning Publications, Shelter Island, NY, USA, 2012.
This is a fairly comprehensive book on concurrency and multithreaded programming in C++. It is arguably the best book available for those who want to learn how to write multithreaded code using C++.

2. M. J. Batty. *The C11 and C++11 Concurrency Model.*
PhD thesis, University of Cambridge, Cambridge, UK, Nov. 2014.
This very well written Ph.D. thesis introduces the C++11/C11 memory model and presents work in mathematically formalizing, refining, and validating this model.

3. M. Batty. *Multicore Programming: C++0x.*
University of Cambridge, Cambridge, UK, Nov. 2010.
This set of slides appear to have been used for part of a course on multicore programming at the University of Cambridge.

References II

4 M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, Burlington, MA, USA, 2008.

A good reference for concurrent programming.

5 S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial.
IEEE Computer, 29(12):66–76, Dec. 1996.

6 S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware.
Communications of the ACM, 53(8):90–101, Aug. 2010.

7 H.-J. Boehm and S. V. Adve. You don't know jack about shared variables or memory models.
Communications of the ACM, 55(2):48–54, Feb. 2012.

8 H.-J. Boehm, Memory Model Rationale, ISO/IEC JTC1/SC22/WG14/N1479, May 2010. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1479.htm`

Talks I

1. Herb Sutter. `atomic<>` Weapons: The C++11 Memory Model and Modern Hardware, C++ and Beyond, Asheville, NC, USA, Aug. 5–8, 2012. Available online at https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2 and https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-2-of-2. (This talk is in two parts.)
2. Herb Sutter. C++ Concurrency, C++ and Beyond, Asheville, NC, USA, Aug. 5–8, 2012. Available online at https://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Herb-Sutter-Concurrency-and-Parallelism.
3. Herb Sutter. Lock-Free Programming (Or, Juggling Razor Blades), CppCon, 2014. Available online at https://youtu.be/c1gO9aB9nbs and https://youtu.be/CmxkPChOcvw. (This talk is in two parts.)

Talks II

4. Hans-J. Boehm. Threads and Shared Variables in C++11. Going Native, Redmond, WA, USA, Feb. 2–3, 2012. Available online at https://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Threads-and-Shared-Variables-in-C-11.
5. Mike Long. Introducing the C++ Memory Model. Norwegian Developers Conference, Oslo, Norway, Jun. 15–19, 2014. Available online at https://vimeo.com/97419179.
6. Herb Sutter. Machine Architecture and You: Things Your Programming Language Never Told You. Northwest C++ Users' Group, Redmond, WA, USA, Sept. 19. 2007. Available online at http://nwcpp.org/september-2007.html.
7. Pablo Halpern. Overview of Parallel Programming in C++, CppCon, Bellevue, WA, USA, Sept. 8, 2014. Available online at https://youtu.be/y0GSc5fKtl8.
8. Valentin Ziegler. C++ Memory Model, Meeting C++, Berlin, Germany, Dec. 6, 2014. Available online at https://youtu.be/gpsz8sc6mNU.

9. Jeff Preshing. How Ubisoft Montreal Develops Games for Multicore — Before and After C++11, CppCon, Bellevue, WA, USA, Sept. 11, 2014. Available online at `https://youtu.be/X1T3IQ4N-3g`.

Part 4

Even More C++

Section 4.1

Undefined Behavior and Other Evil Stuff

Undefined, Unspecified, and Implementation-Defined Behavior

- **undefined behavior**: behavior for which standard imposes no requirements (i.e., anything could happen)
- **unspecified behavior**: behavior, for a well-formed program construct and correct data, that depends on the implementation; implementation is not required to document which behavior occurs; range of possible behaviors usually specified in standard
- **implementation-defined behavior**: behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents (i.e., only know what will happen for a particular implementation)
- ***always avoid undefined behavior*** and ***do not rely on unspecified behavior***; otherwise cannot guarantee correct behavior of program
- ***try to avoid relying on implementation-defined behavior***; otherwise cannot guarantee correct behavior of program across all language implementations (i.e., code will not be portable)

Examples of Undefined Behavior

- dereferencing a null pointer; for example:

```
char* p = nullptr;
char c = *p; // undefined behavior
```

- attempting to modify a string literal or any other const object (excluding mutable data members):

```
const int x = 0;
const_cast<int&>(x) = 42; // undefined behavior
```

- signed integer overflow
- evaluating an expression that is not mathematically defined; for example:

```
double z = 0.0;
double x = 1.0 / z; // undefined behavior
```

- not returning a value from a value-returning function (other than `main`)

```
int& increment(int& x) {
    ++x;
    // undefined behavior
}
```

- multiple definitions of the same entity

Examples of Undefined Behavior (Continued)

- performing pointer arithmetic that yields a result before start of or after end (i.e., one past last element) of an array; for example:

```
int v[10];
int* p = &v[0];
--p; // undefined behavior
```

- using pointers to objects whose lifetime has ended
- left-shifting values by a negative amount; for example:

```
int i = 1;
i << (-3); // undefined behavior
```

- shifting values by an amount greater than or equal to the number of bits in the number; for example:

```
int i = 1;
i << 10000; // undefined behavior
```

- using an automatic variable whose value has not been initialized; for example:

```
void func() {
    int i; ++i; // undefined behavior
}
```

Examples of Unspecified Behavior

- order in which arguments to a function are evaluated; for example:

```
#include <iostream>

int count() {
    static int c = 0;
    return c++;
}

void func(int x, int y) {
    std::cout << x << ' ' << y << '\n';
}

int main() {
    func(count(), count());
      // what values are passed to func?
      // 0, 1; or 1, 0?
}
```

Examples of Implementation-Defined Behavior

- meaning of **#pragma** directive
- nesting limit for **#include** directives
- search locations for `""` and `<>` headers
- sequence of places searched for header
- signedness of char
- sizeof built-in types other than **char**, **signed char**, **unsigned char**
- type of `size_t`, `ptrdiff_t`
- parameters to `main` function
- alignment (i.e., restrictions on the addresses at which an object of a particular type can be placed)
- result of right shift of negative value
- precise types used in various parts of C++ standard library (e.g., actual type named by `vector<T>::iterator`)
- meaning of **asm** declaration
- for more examples, see “Index of implementation-defined behavior” section in C++11 standard

```
#include <iostream>

template <typename Tag>
typename Tag::type saved_private_v;

template <typename Tag, typename Tag::type x>
bool save_private_v = (saved_private_v<Tag> = x);

class Widget {
public:
    Widget(int i) : i_(i) {}
private:
    int i_;
    int f_() const {return i_;}
};

struct Widget_i_ {using type = int Widget::*;};
struct Widget_f_ {using type = int (Widget::*)() const;};

template bool save_private_v<Widget_i_, &Widget::i_>;
template bool save_private_v<Widget_f_, &Widget::f_>;

int main() {
    Widget w(42);
    std::cout << w.*saved_private_v<Widget_i_> << '\n';
    std::cout << (w.*saved_private_v<Widget_f_>)() << '\n';
}
```

Section 4.2

Best Practices, Tips, and Common Pitfalls

Use of `std::istream::eof`

- do not use `std::istream::eof` to determine if earlier input operation has failed, as this will not always work
- `eof` simply returns end-of-file (EOF) flag for stream
- EOF flag for stream can be set during ***successful*** input operation (when input operation takes places just before end of file)
- when stream extractors (i.e., `operator>>`) used, fields normally delimited by whitespace
- to read all data in whitespace-delimited field, must read ***one character beyond*** field in order to know that end of field has been reached
- if field followed immediately by EOF without any intervening whitespace characters, reading one character beyond field will cause EOF to be encountered and EOF bit for stream to be set
- in preceding case, however, EOF being set does not mean that input operation failed, only that stream data ended immediately after field that was read

Example: Incorrect Use of `eof`

- example of ***incorrect*** use of `eof`:

```
#include <iostream>

int main() {
    while (true) {
        int x;
        std::cin >> x;
        // std::cin may not be in a failed state.
        if (std::cin.eof()) {
            // Above input operation may have succeeded.
            std::cout << "EOF encountered\n";
            break;
        }
        std::cout << x << '\n';
    }
}
```

- code incorrectly assumes that `eof` will only return true if preceding input operation has failed
- last field in stream will be incorrectly ignored if not followed by at least one whitespace character; for example, if input stream consists of three character sequence '1', space, '2', program will output:

```
1
EOF encountered
```

Example: Correct Use of `eof`

- to determine if input operation failed, simply check if stream in failed state
- if stream ***already known to be in failed state*** and need to determine specifically if failure due to EOF being encountered, then use `eof`
- example of correct use of `eof`:

```
#include <iostream>

int main() {
    int x;
    // Loop while std::cin not in a failed state.
    while (std::cin >> x) {
        std::cout << x << '\n';
    }
    // Now std::cin must be in a failed state.
    // Use eof to determine the specific reason
    // for failure.
    if (std::cin.eof()) {
        std::cout << "EOF encountered\n";
    } else {
        std::cout << "input error (excluding EOF)\n";
    }
}
```

Use of `std::endl`

- `std::endl` is not some kind of string constant
- `std::endl` is stream manipulator and declared as
  `std::ostream& std::endl(std::ostream&)`
- inserting `endl` to stream always (regardless of operating system) equivalent to outputting single newline character `'\n'` followed by flushing stream
- flushing of stream can incur very substantial overhead; so only flush when strictly necessary

Use of `std::endl` (Continued)

- some operating systems terminate lines with single linefeed character (i.e., `'\n'`), while other operating systems use carriage-return and linefeed pair (i.e., `'\r'` plus `'\n'`)
- existence of `endl` has nothing to do with dealing with handling new lines in operating-system independent manner
- when stream opened in text mode, translation between newline characters and whatever character(s) operating system uses to terminate lines is performed automatically (both for input and output)
- above translation done for all characters input and output and has nothing to do with `endl`

Stream Extraction Failure

- for built-in types, if stream extraction fails, value of target for stream extraction depends on reason for failure
- in following example, what is value of `x` if stream extraction fails:

```
int x;
std::cin >> x;
if (!std::cin) {
    // what is value of x?
}
```

- in above example, `x` may be ***uninitialized*** upon stream extraction failure
- if failure due to I/O error or EOF, target of extraction is ***not modified***
- if failure due to badly formatted data, target of extraction is zero
- if failure due to overflow, target of extraction is closest machine-representable value
- ***common error:*** incorrectly assume that target of extraction will always be initialized if extraction fails
- for class types, also dangerous to assume target of extraction always written upon failure

Stream Extraction Failure (Continued)

```
#include <iostream>
#include <sstream>
#include <limits>
#include <cassert>

int main() {
    int x;

    std::stringstream s0("");
    x = -1;
    s0 >> x;
    // No data; x is not set by extraction.
    assert(s0.fail() && x == -1);

    std::stringstream s1("A");
    x = -1;
    s1 >> x;
    // Badly formatted data; x is zeroed.
    assert(s1.fail() && x == 0);

    std::stringstream
      s2("9999999999999999999999999999999999999999");
    x = -1;
    s2 >> x;
    // Overflow; x set to closest machine-representable value.
    assert(s2.fail() && x == std::numeric_limits<int>::max());
}
```

The abs Function

- Consider a program with the following source listing:

```
#include <iostream>
#include <cstdlib>
int main() {std::cout << abs(-1.5) << '\n';}
```

- The C++ implementation is permitted (but not required) to place the C `abs` function in the global namespace.
- If the implementation does not do this, the above program will fail to compile (avoiding the more troubling problem discussed next).
- If, however, the C++ implementation does do this (which is not uncommon in practice), the above program will compile successfully, but behave unexpectedly when run.
- In particular, the program will output a value of 1, instead of the value of 1.5 that was likely expected by the programmer.
- Since the C `abs` function is declared as `int abs(int)`, the use of this function will introduce a conversion from **`double`** to **`int`**, leading to the unexpected result.

The abs Function (Continued)

- The problems of the previous slide can be easily avoided as follows.
- First, include the header `cmath`, which provides overloads of `std::abs` for various built-in types, including **`double`**.
- Then, invoke the function `std::abs` (instead of `::abs`).
- For example, the following code will behave as expected, outputting the value of 1.5:

```
#include <iostream>
#include <cmath>
int main() {std::cout << std::abs(-1.5) << '\n';}
```

Types of Literals

- When specifying a literal, be careful to use a literal of the correct type, as the type can often be quite important.
- For example, what value will be printed by the following code and (more importantly) why:

```
std::vector<double> values;
values.push_back(0.5);
values.push_back(0.5);
// Compute the sum of the elements in the vector values.
double sum = std::accumulate(values.begin(),
  values.end(), 0);
std::cout << sum << '\n';
```

- Hint: The value printed for `sum` is not 1.
- In order to determine what values will be printed, look carefully at the definition of `std::accumulate`.
- Answer: The value printed for `sum` is 0.

Testing Failure State of Streams

- consider `istream` or `ostream` object `s`
- `!s` is equivalent to `s.fail()`
- `bool(s)` is not equivalent to `s.good()`
- `s.good()` is not the same as `!s.fail()`
- do not use `good` as opposite of `fail` since this is wrong

Member Initialization Order

- data members are initialized in order in which declared
- Example:

```
#include <cassert>

class Widget {
public:
    Widget() : y_(42), x_(y_ + 1) {assert(x_ == 43);}
    int x_;
    int y_;
};

int main() {
    Widget w;
}
```

- what will above code do when run?
- in constructor, `x_` initialized before `y_`, which results in use of `y_` before its initialization
- strictly speaking, undefined behavior
- in practice, likely `x_` will simply have garbage value when body of constructor executes and assertion will fail

Global Object Initialization Order

- be careful about initialization order of global objects
- Example (program with three source files):

```
int main() {
}
```

```
#include <vector>
std::vector<int> v = {1, 2, 3, 4};
```

```
#include <vector>
extern std::vector<int> v;
std::vector<int> w = {v[0], v[1]};
```

- no guarantee that `v` will be constructed before `w`
- bad things will happen if `w` is constructed before `v`
- no guarantee about order of initialization between translation units (i.e., source files [loosely speaking])

Implement Postfix Increment/Decrement via Prefix

- implement postfix increment/decrement in terms of prefix increment/decrement
- ensures that prefix and postfix versions always consistent
- Example:

```
class Counter {
public:
    Counter(int count = 0) : count_(count) {}
    Counter& operator++() {
        ++count_;
        return *this;
    }
    Counter operator++(int) {
        Counter old(*this);
        ++(*this);
        return old;
    }
    // similarly for prefix/postfix decrement
private:
    int count_;
};
```

Sizeof Class Versus Sum of Member Sizes

- compilers can (and do) add padding to classes/structs
- Example:

```
#include <iostream>

class Widget {
// ...
private:
    char c;
    int i;
};

int main() {
    // two numbers printed not necessarily the same
    std::cout << sizeof(char) + sizeof(int) << ' ' <<
      sizeof(Widget) << '\n';
    std::cout << alignof(int) << ' ' <<
      alignof(Widget) << '\n';
}
```

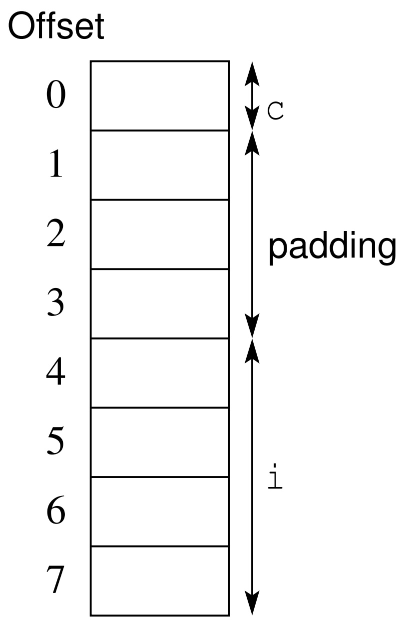
- many processors place alignment restrictions on data (e.g., data type of size n must be aligned to start on address that is multiple of n)
- other factors can also add to size of class/struct (e.g., virtual function table pointer)

Sizeof Class Versus Sum of Member Sizes (Continued)

- consider following type:

```
struct Widget {
    char c;
    int i;
};
```

- suppose that `sizeof(int)` is 4 and `alignof(int)` is 4
- compiler adds padding to structure so that `int` data member is suitably aligned (i.e., offset is multiple of 4)
- memory layout for `Widget` object:



Division/Modulus Operator and Negative Numbers

- for integral operands, division operator yields algebraic quotient with any fractional part discarded (i.e., round towards zero)
- if quotient `a / b` is representable in type of result, `(a / b) * b + a % b` is equal to `a`
- so, assuming `b` is not zero and no overflow, `a % b` equals `a - (a / b) * b`
- result of modulus operator not necessarily nonnegative
- Example:

```
#include <cassert>

int main() {
    assert(5 % 3 == 2);
    assert(5 % (-3) == 2);
    assert((-5) % 3 == -2);
    assert((-5) % (-3) == -2);
}
```

- What is wrong with the following code?

```
void func(const std::string&);
std::string s("one");
const char* p = "two";
func(std::string(s) + std::string(", ") + std::string(p));
func(std::string(p) + std::string(", ") + std::string(s));
```

- *Unnecessary temporaries*!
- Fix:

```
func(s + ", " + p);
func(p + ", "s + s);
```

- What is wrong with the following code?

```
std::vector<std::string> v;
std::string s("one");
v.push_back(std::string(s));
v.push_back(std::string(s + ", two"));
v.push_back(std::string("three"));
v.push_back(std::string());
```

- Again, ***unnecessary temporaries***.
- Fix:

```
v.push_back(s);
v.push_back(s + ", two")
v.emplace_back("three");
v.emplace_back();
```

- What is wrong with this code?

```
class TwoResources {
public:
    TwoResources() : x_(nullptr) : y_(nullptr) {
        x_ = new X;
        y_ = new Y;
    }
    ~TwoResources() {
        delete x_;
        delete y_;
    }
private:
    X* x_;
    Y* y_;
};
```

- If an exception is thrown in a constructor, the object being constructed is deemed not to have started its lifetime and no destructor will ever be called for the object.
- So, for example, if `new Y` throws, `x_` will be leaked.
- Fix:

```
class TwoResources {
public:
    TwoResources() : x_(make_unique<X>()),
      y_(make_unique<Y>()) {}
private:
    unique_ptr<X> x_;
    unique_ptr<Y> y_;
};
```

- What is wrong with the following code?

```
const std::string getMessage() {
    return "Hello";
}
```

- The const return value will ***interact poorly with move semantics***, as the returned object cannot be used as the source for a move operation (since the source for a move operation must be modifiable).
- Fix:

```
std::string getMessage() {
    return "Hello";
}
```

Normally Avoid Using `std::move` When Returning By Value

- What is wrong with the following code?

```
std::vector<int> getVector() {
    std::vector<int> v;
    // calculate v
    return std::move(v);
}
```

- Due to the use of `std::move`, the type of the expression in the return statement does not match the function return type (i.e., `std::vector<int>` versus `std::vector<int>&&`).
- RVO/NRVO can only be applied if the type of the expression in the return statement matches the function return type.
- So, ***RVO/NRVO cannot be applied*** in this case.
- If the types ***would not have matched*** anyways (e.g., a two-element `std::tuple` and a `std::pair`), `std::move` would be reasonable to employ.

- Returning an rvalue reference to an rvalue reference parameter can potentially lead to very subtle bugs.
- Example:

```
std::string&& join(std::string&& s, const char* p) {
    return std::move(s.append(", ").append(p));
}

std::string getMessage() {return "Hello";}

void func() {
    const string& r = join(getMessage(), " World");
    // lifetime of temporary returned by getMessage
    // not extended to lifetime of r since not
    // directly bound to r
    // r now refers to destroyed temporary
}
```

- Fix:

```
std::string join(std::string&& s, const char* p) {
    return std::move(s.append(", ").append(p));
}
```

- Returning by rvalue reference should probably be avoided, except in very special circumstances (such as `std::forward` and `std::move`).

No Explicit Template Arguments to std::make_pair

- Never provide explicit template arguments to `std::make_pair`.
- Let `x` and `y` be objects of type `X` and `Y`, respectively.
- What is wrong with the following code?

```
std::make_pair<X, Y>(x, y)
```

- `make_pair` declared as:

```
template <class T1, class T2>
  pair<V1, V2> make_pair(T1&& x, T2&& y);
```

  where `V1` and `V2` are (except in special case) `std::decay_t<T1>` and `std::decay_t<T2>`, respectively
- If, for example, `X` and `Y` are **`int`**, then `make_pair` has two rvalue reference parameters which cannot bind to the lvalues `x` and `y`.
- Use `make_pair(x, y)` or sometimes `pair<X, Y>(x, y)`.

Prefer Use of `std::make_shared`

- when creating `std::shared_ptr` objects, prefer to use `std::make_shared` (as opposed to explicit use of **`new`** with `shared_ptr`)
- more efficient
- control block and owned object can be allocated together
- one memory allocation instead of two; better cache efficiency
- better exception safety (avoid resource leaks)

Be Careful When Mixing Signed and Unsigned Types

```
#include <cassert>

int main() {
    short ss = -1;
    int si = -1;
    long sl = -1;
    long long sll = -1;
    unsigned short us = 0;
    unsigned int ui = 0;
    unsigned long ul = 0;
    unsigned long long ull = 0;
    // comparison between signed and unsigned types
    assert(ss < ui); // FAILS: ss becomes UINT_MAX
    // comparison between signed and unsigned types
    assert(si < ui); // FAILS: si becomes UINT_MAX
    // comparison between signed and unsigned types
    assert(sl < ul); // FAILS: sl becomes ULONG_MAX
    // comparison between signed and unsigned types
    assert(sll < ull); // FAILS: sll becomes ULONGLONG_MAX
}
```

- be aware of rules for promotions and conversions involving integral types
- if these rules not considered, code may not behave in manner expected

Section 4.3

Idioms

- proxy class provides modified interface to another class

Proxy Class Example

```
#include <iostream>
#include <utility>

class BoolVector;

class Proxy {
public:
    ~Proxy() = default;
    Proxy& operator=(const Proxy&) = default;
    operator bool() const;
    void operator=(bool b);
private:
    friend class BoolVector;
    Proxy(const Proxy&) = default;
    Proxy(BoolVector* v, int i) : v_(v), i_(i) {}
    BoolVector* v_;
    int i_;
};

class BoolVector {
public:
    BoolVector(int n) : n_(n), d_(new unsigned char[(n + 7) / 8]) {
        std::fill_n(d_, (n + 7) / 8, 0);
    }
    ~BoolVector() {delete [] d_;}
    int size() const {return n_;}
    bool operator[](int i) const {return getElem(i);}
    Proxy operator[](int i) {return Proxy(this, i);}
private:
    friend class Proxy;
    bool getElem(int i) const {return (d_[i / 8] >> (i % 8)) & 1;}
    void setElem(int i, bool b) {
        (d_[i / 8] &= ~(1 << (i % 8))) |= (b << (i % 8));
    }
    int n_;
    unsigned char* d_;
};

inline void Proxy::operator=(bool b) {v_->setElem(i_, b);}
inline Proxy::operator bool() const {return v_->getElem(i_);}
```

Proxy Class Example (Continued)

```
#include "proxy_class_example_1.hpp"

int main() {
    BoolVector v(16);
    for (int i = 0; i < v.size(); ++i) {
        v[i] = (i & 1);
    }
    for (int i = 0; i < v.size(); ++i) {
        std::cout << v[i];
    }
    std::cout << '\n';
    const BoolVector& cv = v;
    for (int i = 0; i < cv.size(); ++i) {
        std::cout << cv[i];
    }
    std::cout << '\n';
}
```

Section 4.4

C++ Compatibility

C++ Compatibility

- many changes have been made to C++ language and standard library during evolution of C++ from C++98 to present
- some changes resulted in incompatibilties between different versions of C++ standard
- subsequent slides list some reference material that discusses how C++ standard changed from one version to next
- knowing such changes helps to understand incompatibilities between different versions

Talks I

1. Leor Zolman, An Overview of C++11/14, CppCon, Bellevue, WA, USA, Sept 8, 2014. (This talk is in two parts.) Available online at https://youtu.be/Gycxew-hztI and https://youtu.be/pBI0tS2yfjw.
2. Alisdair Meredith, A Quick Tour of C++14, CppCon, Bellevue, WA, USA, Sept. 11, 2014. Available online at https://youtu.be/fBU1R7jp_TE.
3. Alisdair Meredith, C++17 in Breadth, CppCon, Bellevue, WA, USA, Sept. 19, 2016. (This talk is in two parts.) Available online at https://youtu.be/22jIHfvelZk and https://youtu.be/-rIixnNJM4k.
4. Nicolai Josuttis, C++17: The Language Features, Norwegian Developers Conference, London, UK, Jan. 16–20 2017. Available online at https://youtu.be/pEzV32yRu4U.
5. Nicolai Josuttis, C++17: The Library Features, Norwegian Developers Conference, London, UK, Jan. 16–20 2017. Available online at https://youtu.be/ELwTKHiKZS4.
6. Bryce Lelbach, C++17 Features, C++Now, Aspen, CO, USA, May 16, 2017. Available online at https://youtu.be/LvwXJjRQfHk.

Section 4.5

C Compatibility

C Compatibility

- Although C++ attempted to maintain compatibility with C where possible, there are numerous incompatibilities between the languages.
- Unfortunately, as C++ and C continue to evolve, the number of incompatibilities between these languages continue to grow.
- In practice, many C programs are valid C++ programs and can therefore be compiled with a C++ compiler.
- Some C programs, however, may require a significant number of changes to be valid C++.
- A few examples of incompatibilities between C++ and C are given in what follows.

Conflicts with New Keywords

```
#include <stdio.h>
#include <unistd.h>

/* Delete a file. */
int delete(const char* filename) { /* note function name */
    return unlink(filename);
}

int main(int argc, char** argv) {
    if (argc >= 2) {
        if (delete(argv[1])) {
            printf("cannot delete file\n");
            return 1;
        }
    }
    return 0;
}
```

- C++ introduces many new keywords.
- Some C programs might use some of these keywords as identifiers (e.g., **`new`**, **`delete`**).

Function Declarations Without Arguments

```
#include <stdio.h>

int plusOne(); /* no arguments specified */

int main(int argc, char** argv) {
    printf("%d\n", plusOne(0));
    return 0;
}

int plusOne(int i) {
    return i + 1;
}
```

- In C, a function declaration without arguments implies that the arguments are unspecified.
- In C++, a function declaration without arguments implies that the function takes no arguments.

Implicit Return Type

```
#include <stdio.h>

myfunc() { /* implicit return type */
    return 3;
}

int main(int argc, char **argv) {
    int i;
    i = myfunc();
    printf("%d\n", i);
    return 0;
}
```

- In C, if the return type of a function is not specified, it is treated as **`int`**.
- In C++, the return type of a function must always be explicitly specified.

More Restrictive Conversions Involving **void***

```
int main(int argc, char** argv) {
    int i;
    int* ip;
    void* vp;
    ip = &i;
    vp = ip;
    ip = vp; /* problematic */
    return 0;
}
```

- C provides an implicit conversion from **void*** to any pointer type, while C++ does not.

Scoping Rules for Nested Structs

```
struct outer {
    struct inner {
        int i;
    };
    int j;
};

struct inner a = {1}; /* inner vs. outer::inner */

int main(int argc, char** argv) {
    return 0;
}
```

- C and C++ both allow nested **struct** types, but the scoping rules differ.

Part 5

Libraries

Section 5.1

Boost Libraries

Section 5.1.1

Introduction

Boost Libraries

- Boost libraries are collection of free peer-reviewed portable C++ source libraries
- license encourages both commercial and non-commercial use
- often Boost libraries later adopted by C++ standard
- web site: `http://www.boost.org`

Some Boost Libraries

Containers and Data Structures

Library	Description
Bimap	bidirectional maps (i.e., associative containers in which both types stored in map can be used as key)
Container	standard library containers and extensions
Heap	priority queue data structures
Intrusive	intrusive containers and algorithms
Multi-Array	generic N-dimensional array
Multi-Index	containers that maintain one or more indices with different sorting and access semantics

Iterators

Library	Description
Iterator	concepts that extend C++ standard iterator requirements and components for building iterators based on these concepts; includes several iterator adaptors

Some Boost Libraries (Continued 1)

Math and Numerics

Library	Description
Interval	interval arithmetic
Math	various numeric types and math functions
Multiprecision	extended precision arithmetic types for floating-point, integer, and rational arithmetic
Rational	rational number class

String and Text Processing

Library	Description
Lexical Cast	general literal text conversions, such as converting **`int`** to `std::string` or vice versa
Tokenizer	break a string or other character sequence into a series of tokens

Some Boost Libraries (Continued 2)

Image and Geometry Processing

Library	Description
Geometry	geometric algorithms, primitives, and spatial index
GIL	generic image library
Graph	graph types and algorithms

Input/Output

Library	Description
I/O State Savers	classes for saving/restoring state associated with I/O streams

Miscellaneous

Library	Description
Program Options	process program options via command line or configuration file

Some Boost Libraries (Continued 3)

Concurrent Programming

Library	Description
Fiber	userland threads library
Compute	parallel/GPU computing library
Lockfree	lockfree containers (e.g., stacks and queues)

Section 5.1.2

Boost Container Library

Boost Container Library

- Boost Container library provides support for numerous ***nonintrusive*** containers
- containers provided by library include:
  - enhanced versions of several containers from standard library
  - several non-standard containers

Container Types

Standard Container Types

Type	Description
`vector`	similar to `std::vector`
`list`	similar to `std::list`
`deque`	similar to `std::deque`
`set`	similar to `std::set`
`multiset`	similar to `std::multiset`
`map`	similar to `std::map`
`multimap`	similar to `std::multimap`

Container Types (Continued)

Non-Standard Container Types

Type	Description
`stable_vector`	vector with non-contiguous elements and stable element references
`flat_set`	set based on sorted vector
`flat_multiset`	multiset based on sorted vector
`flat_map`	map based on sorted vector
`flat_multimap`	multimap based on sorted vector
`slist`	singly-linked list
`static_vector`	vector of bounded size with storage for elements that is contiguous and statically allocated
`small_vector`	vector-like container optimized for case of containing few elements

Section 5.1.3

Boost Intrusive Library

Boost Intrusive Library

- Boost Intrusive library provides support for numerous ***intrusive*** and ***semi-intrusive*** containers
- containers provided by library include those based on:
  - linked lists
  - trees
  - hash tables

Container Types

Intrusive Container Types

Type	Description
slist	singly-linked list
list	doubly-linked list
set	set/map based on red-black tree
multiset	multiset/multimap based on red-black tree
rbtree	red-black tree
avl_set	set/map based on AVL tree
avl_multiset	multiset/multimap based on AVL tree
avltree	AVL tree
splay_set	set/map based on splay tree
splay_multiset	multiset/multimap based on splay tree
splaytree	splay tree
sg_set	set/map based on scapegoat tree
sg_multiset	multiset/multimap based on scapegoat tree
sgtree	scapegoat tree

Container Types (Continued)

Semi-Intrusive Container Types

Type	Description
`unordered_set`	unordered set/map based on hash table
`unordered_multiset`	unordered multiset/multimap based on hash table

Base and Member Hooks

- **hook** is class object that must be added to a user's class in order for user's class to be usable with intrusive container
- hook used to encapsulate data used to manage nodes in container (e.g., successor and predecessor links for doubly-linked list)
- two kinds of hooks:
  1. base hook
  2. member hook
- base hook is included in user's class as base class object using ***public*** inheritance
- member hook included in user's class as ***public*** data member

```
#include <iostream>
#include <vector>
#include <boost/intrusive/slist.hpp>

namespace bi = boost::intrusive;

struct Widget : public bi::slist_base_hook<> {
    Widget(int i_) : i(i_) {}
    int i;
};

using WidgetList = bi::slist<Widget>;

int main() {
    std::vector<Widget> buffer;
    for (int i = 0; i < 10; ++i) {buffer.push_back(Widget(i));}
    WidgetList a;
    for (auto&& i : buffer) {a.push_front(i);}
    for (auto i = a.begin(); i != a.end(); ++i) {
        if (i != a.begin()) {std::cout << ' ';}
        std::cout << i->i;
    }
    std::cout << '\n';
    while (!a.empty()) {a.erase_after(a.before_begin());}
}
```

slist With Member Hook

```
#include <iostream>
#include <vector>
#include <boost/intrusive/slist.hpp>

namespace bi = boost::intrusive;

struct Widget {
    Widget(int i_) : i(i_) {}
    int i;
    bi::slist_member_hook<> hook;
};

using WidgetList = bi::slist<Widget, bi::member_hook<Widget,
  bi::slist_member_hook<>, &Widget::hook>>;

int main() {
    std::vector<Widget> buffer;
    for (int i = 0; i < 10; ++i) {buffer.push_back(Widget(i));}
    WidgetList a;
    for (auto&& i : buffer) {a.push_front(i);}
    for (auto i = a.begin(); i != a.end(); ++i) {
        if (i != a.begin()) {std::cout << ' ';}
        std::cout << i->i;
    }
    std::cout << '\n';
    while (!a.empty())
      {a.erase_after(a.before_begin());}
}
```

list With Base Hook

```cpp
#include <iostream>
#include <vector>
#include <boost/intrusive/list.hpp>

namespace bi = boost::intrusive;

struct Widget : public bi::list_base_hook<> {
    Widget(int i_) : i(i_) {}
    int i;
};

using WidgetList = bi::list<Widget>;

int main() {
    std::vector<Widget> buffer;
    for (int i = 0; i < 10; ++i) {buffer.push_back(Widget(i));}
    WidgetList a;
    for (auto&& i : buffer) {a.push_back(i);}
    for (auto i = a.begin(); i != a.end(); ++i) {
        if (i != a.begin()) {std::cout << ' ';}
        std::cout << i->i;
    }
    std::cout << '\n';
    while (!a.empty()) {a.erase(a.begin());}
}
```

```
#include <iostream>
#include <vector>
#include <boost/intrusive/list.hpp>

namespace bi = boost::intrusive;

struct Widget {
    Widget(int i_) : i(i_) {}
    int i;
    bi::list_member_hook<> hook;
};

using WidgetList = bi::list<Widget, bi::member_hook<Widget,
  bi::list_member_hook<>, &Widget::hook>>;

int main() {
    std::vector<Widget> buffer;
    for (int i = 0; i < 10; ++i) {buffer.push_back(Widget(i));}
    WidgetList a;
    for (auto&& i : buffer) {a.push_back(i);}
    for (auto i = a.begin(); i != a.end(); ++i) {
        if (i != a.begin()) {std::cout << ' ';}
        std::cout << i->i;
    }
    std::cout << '\n';
    while (!a.empty()) {a.erase(a.begin());}
}
```

list With Multiple Base Hooks

```
#include <iostream>
#include <vector>
#include <boost/intrusive/list.hpp>

namespace bi = boost::intrusive;

struct Alpha {};
struct Beta {};
struct Widget : public bi::list_base_hook<bi::tag<Alpha>>,
  public bi::list_base_hook<bi::tag<Beta>> {
    Widget(int i_) : i(i_) {}
    int i;
};

int main() {
    std::vector<Widget> buffer;
    for (int i = 0; i < 10; ++i) {buffer.push_back(Widget(i));}
    bi::list<Widget, bi::base_hook<bi::list_base_hook<bi::tag<Alpha>>>>
      a;
    bi::list<Widget, bi::base_hook<bi::list_base_hook<bi::tag<Beta>>>>
      b;
    for (auto&& i : buffer)
      {a.push_back(i); b.push_front(i);}
    for (auto&& w : a) {std::cout << w.i << '\n';}
    std::cout << '\n';
    for (auto&& w : b) {std::cout << w.i << '\n';}
    while (!a.empty()) {a.erase(a.begin());}
    while (!b.empty()) {b.erase(b.begin());}
}
```

```
#include <iostream>
#include <vector>
#include <boost/intrusive/set.hpp>

namespace bi = boost::intrusive;

struct Widget : public bi::set_base_hook<> {
    Widget(int i_) : i(i_) {}
    bool operator<(const Widget& other) const {return i < other.i;}
    int i;
};

int main() {
    int values[] = {1, 3, 5, 7, 9, 0, 2, 4, 6, 8};
    std::vector<Widget> buffer;
    for (auto i : values) {buffer.push_back(Widget(i));}
    bi::set<Widget> a;
    for (auto&& i : buffer) {a.insert(a.end(), i);}
    for (auto&& w : a) {std::cout << w.i << '\n';}
    if (a.find(7) != a.end()) {std::cout << "7 is in set\n";}
    while (!a.empty())
      {a.erase(a.begin());}
}
```

set and list With Base Hooks

```cpp
#include <iostream>
#include <vector>
#include <boost/intrusive/set.hpp>
#include <boost/intrusive/list.hpp>

namespace bi = boost::intrusive;

struct Widget : public bi::set_base_hook<>,
  public bi::list_base_hook<> {
    Widget(int i_) : i(i_) {}
    bool operator<(const Widget& other) const {return i < other.i;}
    int i;
};

int main() {
    int values[] = {1, 3, 5, 7, 9, 0, 2, 4, 6, 8};
    std::vector<Widget> buffer;
    for (auto i : values) {buffer.push_back(Widget(i));}
    bi::set<Widget> a;
    bi::list<Widget> b;
    for (auto&& i : buffer)
      {a.insert(a.end(), i); b.push_back(i);}
    if (a.find(7) != a.end()) {std::cout << "found 7\n\n";}
    for (auto&& w : a) {std::cout << w.i << '\n';}
    std::cout << '\n';
    for (auto&& w : b) {std::cout << w.i << '\n';}
    while (!a.empty()) {a.erase(a.begin());}
    while (!b.empty()) {b.erase(b.begin());}
}
```

Achieving Map Functionality With set

```
#include <iostream>
#include <vector>
#include <boost/intrusive/set.hpp>

namespace bi = boost::intrusive;

struct Widget : public bi::set_base_hook<> {
    Widget(int i_, int j_) : i(i_), j(j_) {}
    int i;
    int j;
};

struct Is_key {
    using type = int;
    const type& operator()(const Widget& w) const
      {return w.i;}
};

int main() {
    int values[] = {1, 3, 5, 7, 9, 0, 2, 4, 6, 8};
    std::vector<Widget> buffer;
    for (auto i : values) {buffer.push_back(Widget(i, -i));}
    bi::set<Widget, bi::key_of_value<Is_key>> a;
    for (auto&& i : buffer) {a.insert(a.end(), i);}
    for (auto&& w : a) {std::cout << w.i << ' ' << w.j << '\n';}
    auto i = a.find(5);
    std::cout << i->j << '\n';
    while (!a.empty()) {a.erase(a.begin());}
}
```

```
#include <iostream>
#include <vector>
#include <boost/intrusive/unordered_set.hpp>

namespace bi = boost::intrusive;

class Widget : public bi::unordered_set_base_hook<> {
public:
    Widget(int value = 0) : value_(value) {}
    int get_value() const {return value_;}
private:
    int value_;
};

bool operator==(const Widget& a, const Widget& b)
  {return a.get_value() == b.get_value();}

std::size_t hash_value(const Widget& a) {return std::size_t(a.get_value());}

int main() {
    std::vector<Widget> widgets;
    for (int i = 0; i < 10; ++i) {widgets.push_back(Widget(i));}
    using bucket_type = bi::unordered_set<Widget>::bucket_type;
    using bucket_traits = bi::unordered_set<Widget>::bucket_traits;
    bucket_type buckets[100];
    bi::unordered_set<Widget> s(bucket_traits(buckets, 100));
    for (auto&& w : widgets) {s.insert(w);}
    for (auto&& i : s) {std::cout << i.get_value() << '\n';}
    if (s.find(7) != s.end()) {std::cout << "found 7\n";}
    return 0;
}
```

Section 5.1.4

Boost Iterator Library

Boost Iterator Library

- Boost iterator library consists of two parts:
  1. system of concepts which extend C++ standard iterator requirements
  2. framework of componenets for building iterators based on these concepts
- tricky to write standard-conforming iterators
- by using Boost Iterator library, can often significantly reduce amount of code needed to implement standard-conforming iterators

Forward Iterator Example: Iterator Class Without Boost (1)

```
#include <type_traits>
#include <iterator>

// singly-linked list node base (for intrusive container)
template <class T> struct slist_node_base {
    slist_node_base(T* next_) : next(next_) {}
    T* next; // pointer to next node in list
};

// single-linked list iterator (const and non-const)
template <class T> class slist_iter {
public:
    using iterator_category = std::forward_iterator_tag;
    using value_type = typename std::remove_const_t<T>;
    using reference = T&;
    using pointer = T*;
    slist_iter(T* node = nullptr) : node_(node) {}
    template <class OtherT, class =
      std::enable_if_t<std::is_convertible_v<OtherT*, T*>>>
      slist_iter(const slist_iter<OtherT>& other) : node_(other.node_) {}
    reference operator*() {return *node_;}
    pointer operator->() {return node_;}
    slist_iter& operator++() {
        node_ = node_->next;
        return *this;
    }
```

[link: SFINAE]

```
    slist_iter operator++(int) {
        slist_iter old(*this);
        node_ = node_->next;
        return old;
    }
    template <class OtherT> bool operator==(const slist_iter<OtherT>& other)
      const {return node_ == other.node_;}
    template <class OtherT> bool operator!=(const slist_iter<OtherT>& other)
      const {return !(*this == other);}
private:
    template <class> friend class slist_iter;
    T* node_; // pointer to list node
};
```

```
#include <type_traits>
#include <boost/iterator/iterator_facade.hpp>

template <class T> struct slist_node_base {
    slist_node_base(T* next_) : next(next_) {}
    T* next; // pointer to next node in list
};

template <class T> class slist_iter : public boost::iterator_facade<
  slist_iter<T>, T, boost::forward_traversal_tag> {
public:
    using base = typename boost::iterator_facade<slist_iter<T>, T,
      boost::forward_traversal_tag>;
    using typename base::reference;
    using typename base::value_type;
    slist_iter(T* node = nullptr) : node_(node) {}
    template <class OtherT, class =
      std::enable_if_t<std::is_convertible_v<OtherT*, T*>>>
      slist_iter(const slist_iter<OtherT>& other) : node_(other.node_) {}
private:
    reference dereference() const {return *node_;}
    template <class OtherT> bool equal(const slist_iter<OtherT>& other) const
      {return node_ == other.node_;}
    void increment() {node_ = node_->next;}
    template <class> friend class slist_iter;
    friend class boost::iterator_core_access;
    T* node_; // pointer to list node
};
```

Forward Iterator Example: User Code

```
#include <iostream>
#include <vector>
#include "iterator_facade_2.hpp"

struct Node : public slist_node_base<Node> {
    Node(Node* next_, int value_) : slist_node_base<Node>(next_),
      value(value_) {}
    int value;
};

int main() {
    constexpr int num_nodes = 10;
    std::vector<Node> nodes; nodes.reserve(num_nodes);
    for (int i = 0; i < num_nodes - 1; ++i)
      {nodes.push_back(Node(&nodes[i + 1], i));}
    nodes.push_back(Node(nullptr, num_nodes - 1));
    slist_iter<Node> begin(&nodes[0]);
    slist_iter<Node> end;
    slist_iter<const Node> cbegin(begin);
    slist_iter<const Node> cend(end);
    for (auto i = cbegin; i != cend; ++i) {std::cout << i->value << '\n';}
    slist_iter<Node> i(begin);
    slist_iter<const Node> ci(cbegin);
    // slist_iter<Node> j(cbegin); // ERROR
    i = begin;
    // i = ci; // ERROR
    ci = cbegin;
    ci = i;
}
```

Random-Access Iterator Example: Iterator Class Without Boost (1)

```
#include <type_traits>
#include <iterator>

// array element iterator
template <class T> class array_iter {
public:
    using iterator_category = typename std::random_access_iterator_tag;
    using value_type = std::remove_const_t<T>;
    using reference = T&;
    using pointer = T*;
    using difference_type = std::ptrdiff_t;
    array_iter(T* ptr = nullptr) : ptr_(ptr) {}
    template <class OtherT, class =
      std::enable_if_t<std::is_convertible_v<OtherT*, T*>>>
      array_iter(const array_iter<OtherT>& other) : ptr_(other.ptr_) {}
    reference operator*() const {return *ptr_;}
    pointer operator->() const {return ptr_;}
    array_iter& operator++() {
        ++ptr_;
        return *this;
    }
    array_iter operator++(int) {
        array_iter old(*this);
        ++ptr_;
        return old;
    }
    array_iter& operator--() {
        --ptr_;
        return *this;
    }
```

Random-Access Iterator Example: Iterator Class Without Boost (2)

```
    array_iter operator--(int) {
        array_iter old(*this);
        --ptr_;
        return old;
    }
    array_iter& operator+=(difference_type n) {
        ptr_ += n;
        return *this;
    }
    array_iter& operator-=(difference_type n) {
        ptr_ -= n;
        return *this;
    }
    reference& operator[](difference_type n) const {return ptr_[n];}
    array_iter operator+(difference_type n) const
      {return array_iter(ptr_ + n);}
    difference_type operator-(const array_iter& other) const
      {return ptr_ - other.ptr_;}
    array_iter operator-(difference_type n) const
      {return array_iter(ptr_ - n);}
    template <class OtherT> bool operator==(const array_iter<OtherT>& other)
      const {return ptr_ == other.ptr_;}
    template <class OtherT> bool operator!=(const array_iter<OtherT>& other)
      const {return ptr_ != other.ptr_;}
    template <class OtherT> bool operator<(const array_iter<OtherT>& other)
      const {return ptr_ < other.ptr_;}
    template <class OtherT> bool operator>(const array_iter<OtherT>& other)
      const {return ptr_ > other.ptr_;}
    template <class OtherT> bool operator<=(const array_iter<OtherT>& other)
      const {return ptr_ <= other.ptr_;}
```

```
    template <class OtherT> bool operator>=(const array_iter<OtherT>& other)
      const {return ptr_ >= other.ptr_;}
private:
    template <class> friend class array_iter;
    T* ptr_; // pointer to array element
};

template <class T>
array_iter<T> operator+(typename array_iter<T>::difference_type n,
  const array_iter<T>& iter) {return array_iter<T>(iter) += n;}
```

```
#include <boost/iterator/iterator_facade.hpp>
#include <type_traits>

// array element iterator
template <class T> class array_iter : public boost::iterator_facade<
  array_iter<T>, T, boost::random_access_traversal_tag> {
public:
    using typename boost::iterator_facade<array_iter<T>, T,
      boost::random_access_traversal_tag>::reference;
    using typename boost::iterator_facade<array_iter<T>, T,
      boost::random_access_traversal_tag>::difference_type;
    array_iter(T* ptr = nullptr) : ptr_(ptr) {}
    template <class OtherT, class =
      std::enable_if_t<std::is_convertible_v<OtherT*, T*>>>
      array_iter(const array_iter<OtherT>& other) : ptr_(other.ptr_) {}
private:
    reference dereference() const {return *ptr_;}
    template <class OtherT> bool equal(const array_iter<OtherT>& other) const
      {return ptr_ == other.ptr_;}
    void increment() {++ptr_;}
    void decrement() {--ptr_;}
    void advance(difference_type n) {ptr_ += n;}
    difference_type distance_to(const array_iter& other) const
      {return other.ptr_ - ptr_;}
    template <class> friend class array_iter;
    friend class boost::iterator_core_access;
    T* ptr_; // pointer to array element
};
```

Random-Access Iterator Example: User Code

```
#include <iostream>
#include <cassert>
#include "iterator_facade_1.hpp"

int main() {
    char buffer[] = "Hello, World!\n";
    std::size_t length = sizeof(buffer) - 1;
    array_iter<char> begin(buffer);
    array_iter<char> end(buffer + length);
    array_iter<const char> cbegin = begin;
    array_iter<const char> cend = end;
    assert(begin + length == end);
    assert(cbegin + length == end);
    for (auto i = cbegin; i != cend; ++i)
      {std::cout << *i << '\n';}
    array_iter<char> i(begin);
    array_iter<const char> ci(cbegin);
    // array_iter<char> j(cbegin); // ERROR
    i = begin;
    // i = ci; // ERROR
    ci = cbegin;
    ci = i;
}
```

Section 5.1.5

Miscellaneous Examples

```
#include <iostream>
#include <limits>
#include <boost/math/constants/constants.hpp>

int main() {
    namespace bmc = boost::math::constants;

    std::cout.precision(std::numeric_limits<
      long double>::max_digits10);

    constexpr auto f_pi = bmc::pi<float>();
    constexpr auto d_pi = bmc::pi<double>();
    constexpr auto ld_pi = bmc::pi<long double>();

    std::cout << f_pi << '\n';
    std::cout << d_pi << '\n';
    std::cout << ld_pi << '\n';

    constexpr auto f_e = bmc::e<float>();
    constexpr auto d_e = bmc::e<double>();
    constexpr auto ld_e = bmc::e<long double>();

    std::cout << f_e << '\n';
    std::cout << d_e << '\n';
    std::cout << ld_e << '\n';
}
```

Math Constant π

```
#include <iostream>
#include <boost/math/constants/constants.hpp>

template <class Real>
Real area_of_circle(Real r) {
    namespace bmc = boost::math::constants;
    return bmc::pi<Real>() * r * r;
}

int main() {
    double r;
    while (std::cin >> r) {
        std::cout << area_of_circle(r) << '\n';
    }
}
```

```
#include <cmath>
#include <boost/multiprecision/gmp.hpp>
#include <iostream>

using boost::multiprecision::mpz_int;

mpz_int factorial(const mpz_int& n) {
    mpz_int result = 1;
    for (mpz_int i = n; i >= 2; --i) {
        result *= i;
    }
    return result;
}

int main() {
    std::cout << factorial(200) << '\n';
}

/* Output:
7886578673647905035523632139321850622951359776871732632947
4253324435944996340334292030428401198462390417721213891963
8830257642790242637105061926624952829931113462857270763317
2373969889439224456214516642402540332918641312274282948532
7752424240757390324032125740557956866022603190417032406235
1700858796178922222789623703897374720000000000000000000000
00000000000000000000000000
*/
```

multi_array Example

```
#include <boost/multi_array.hpp>
#include <iostream>
#include <iomanip>

int main() {
    using Array2 = boost::multi_array<int, 2>;
    int num_rows = 5;
    int num_cols = 7;
    Array2 a(boost::extents[num_rows][num_cols]);
    for (int row = 0; row < num_rows; ++row) {
        for (int col = 0; col < num_cols; ++col) {
            a[row][col] = num_cols * row + col;
        }
    }
    Array2 b(a);
    assert(b.shape()[0] == num_rows && b.shape()[1] == num_cols);
    Array2 c;
    c.resize(boost::extents[b.shape()[0]][b.shape()[1]]);
    c = b;
    for (int row = 0; row < num_rows; ++row) {
        for (int col = 0; col < num_cols; ++col) {
            if (col) {std::cout << ' ';}
            std::cout << std::setw(2) << c[row][col];
        }
        std::cout << '\n';
    }
}
```

2-D Array Class With multi_array

```
#include <boost/multi_array.hpp>
#include <iostream>
#include <iomanip>

template <class T>
class array2 {
public:
    using value_type = T;
    array2(int num_rows = 0, int num_cols = 0) :
      a_(boost::extents[num_rows][num_cols]) {}
    array2(const array2& other) : a_(other.a_) {}
    array2& operator=(const array2& other) {
        if (this != &other) {
            a_.resize(boost::extents[other.a_.shape()[0]][
              other.a_.shape()[1]]);
            a_ = other.a_;
        }
        return *this;
    }
    int num_rows() const {return a_.shape()[0];}
    int num_cols() const {return a_.shape()[1];}
    const value_type& operator()(int row, int col) const
      {return a_[row][col];}
    value_type& operator()(int row, int col) {return a_[row][col];}
private:
    using array = boost::multi_array<T, 2>;
    array a_;
};
```

```
template <class T>
std::ostream& operator<<(std::ostream& out, const array2<T>& a) {
    auto width = out.width();
    for (int row = 0; row < a.num_rows(); ++row) {
        for (int col = 0; col < a.num_cols(); ++col) {
            if (col) {out << ' ';}
            out << std::setw(width) << a(row, col);
        }
        out << '\n';
    }
    return out;
}

int main() {
    array2<int> a(5, 7);
    for (int row = 0; row < a.num_rows(); ++row) {
        for (int col = 0; col < a.num_cols(); ++col) {
            a(row, col) = a.num_cols() * row + col;
        }
    }
    array2<int> b(a);
    std::cout << "a:\n" << std::setw(2) << a;
    std::cout << "b:\n" << std::setw(2) << b;
}
```

Program Options Example

```
#include <iostream>
#include <string>
#include <boost/program_options.hpp>

int main(int argc, char** argv) {
    namespace po = boost::program_options;
    po::options_description desc("Allowed options");
    desc.add_options()
      ("help,h", "Print help information.")
      ("count,c", po::value<int>()->default_value(1), "Specify count.")
      ("file,f", po::value<std::string>(), "Specify file name.");
    po::variables_map vm;
    try {
        po::store(po::parse_command_line(argc, argv, desc), vm);
        po::notify(vm);
    } catch (po::error& e) {
        std::cerr << "usage:\n" << desc << '\n';
        return 1;
    }
    if (vm.count("help")) {std::cout << desc << "\n"; return 1;}
    if (vm.count("file")) {
        std::cout << "file: " << vm["file"].as<std::string>() << '\n';
    }
    if (vm.count("count")) {
        std::cout << "count: " << vm["count"].as<int>() << '\n';
    }
    return 0;
}
```

Rational Numbers Example

```
#include <iostream>
#include <cassert>
#include <boost/rational.hpp>
#include <exception>

int main() {
    using boost::rational;
    const rational<int> zero;
    rational<int> three(3);
    rational<int> ninth(1, 9);
    rational<int> third(1, 3);
    auto result = three * ninth;
    assert(result == third);
    try {
        std::cout << three / zero << '\n';
    } catch (const boost::bad_rational& e) {
        std::cout << "bad rational " << e.what() << '\n';
    }
    // rational<int> x(1.5); // ERROR: no matching call
    // result = 3.0; // ERROR: no matching call
    result = 42;
    assert(result == rational<int>(42));
    std::cout << result << '\n';
}
```

Section 5.1.6

References

References I

1. Boost C++ Libraries Web Site, `http://www.boost.org`.
2. Boost Library Incubator Web Site, `http://www.blincubator.com`.
3. B. Schaling, The Boost C++ Libraries, `http://theboostcpplibraries.com`. [This is an online version of Schaling's book on Boost.]

Talks I

1. Boris Schaling. Containers in Boost. C++ Now, 2013. Available online at `https://youtu.be/FM-fUjhoCp0`.
2. Boris Schaling. Boost.Graph for Beginners. C++ Now, 2013. Available online at `https://youtu.be/uYvBH7TZlFk`.
3. Nat Goodspeed. The Fiber Library. C++ Now, 2016. Available online at `https://youtu.be/gcNphOWuUb0`.
4. Kyle Lutz. Boost.Compute: A library for GPU/parallel computing. C++ Now, 2015. Available online at `https://youtu.be/q7oCblCtTT8`.

Section 5.2

Computational Geometry Algorithms Library (CGAL)

Computational Geometry Algorithms Library (CGAL)

- very powerful open-source C++ library for geometric computation
- used by many commercial organizations, such as: British Telecom, Boeing, France Telecom, GE Health Care, The MathWorks
- very well documented (extensive manual, more than 4000 pages)
- provides data types for representing various geometric objects, such as:
  - points, lines, planes, polygons
  - Voronoi diagrams
  - 2D, 3D and dD triangulations
  - polygon meshes
  - kinetic data structures
- provides algorithms for manipulating these data types
- available for Microsoft Windows and Unix/Linux platforms
- some Linux distributions already have packages for CGAL (e.g., Fedora packages: `CGAL`, `CGAL-devel`, `CGAL-demos-source`)
- web site: `http://www.cgal.org`
- online manual (latest version): `http://www.cgal.org/Manual/latest`

- provides support for polygon meshes
- can read/write polygon mesh data in various common formats
- built-in support for several subdivision schemes
- by using CGAL, can greatly simplify amount of effort required to implement methods using subdivision surfaces or wavelet transforms for polygon meshes
- in CGAL manual, most relevant material is that pertaining to:
  - 2D and 3D linear geometry kernels
  - 3D polyhedral surfaces
  - 3D surface subdivision methods

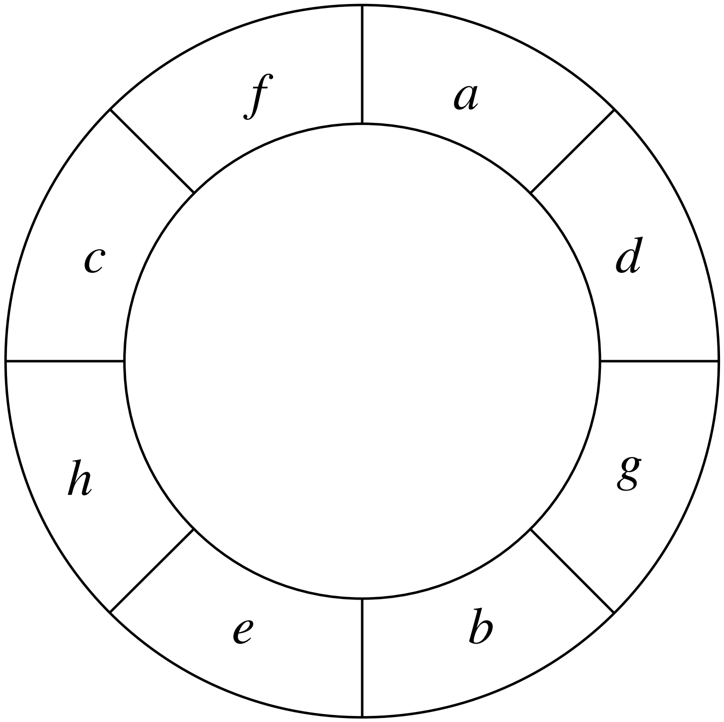
Handles

- **handle**: object used to reference element stored in some data structure (i.e., object can be dereferenced to obtain access to element)
- for data structure storing elements of type `T`, handle type might be:
  - simple pointer (i.e., `T*`)
  - smart pointer (i.e., user-defined type that behaves like pointer)
- examples of handle types:
  - types used to access vertices, facets, halfedges of polygon mesh

Linear Sequences Versus Circular Sequences

a	d	g	b	e	h	c	f

Linear Sequence



Circular Sequence

- linear sequence:
  - has well defined first and last element
  - fits well with iterator model
- circular sequence:
  - does not have well defined first and last element
  - does not fit well with iterator model

Circulators

- *iterators* are very useful, but intended for use with linear sequences of elements (i.e., sequences with well-defined first and last element)
- often want iterator-like functionality for circular sequences of elements
- **circulator**: object that allows iteration over elements in circular sequence of elements
- examples of circulator types:
  - type to allow iteration over all halfedges incident on vertex in polygon mesh
  - type to allow iteration over all halfedges incident on facet in polygon mesh
- circulators come in const and mutable (i.e., non-const) forms
- mutable circulator can be used to modify referenced element, while const circulator cannot

Section 5.2.1

Geometry Kernels

Real Number Types

- **`float`**: single-precision floating point type
- **`double`**: double-precision floating point type
- `Interval_nt`: interval-arithmetic type
- `MP_Float`: arbitrary-precision floating-point type

MP_Float Class

- MP_Float is arbitrary-precision floating-point type
- additions, subtractions, and multiplications computed exactly
- does not provide division or square root (which is not typically problematic as division rarely needed and square root almost always avoided in geometric computation)
- no roundoff error
- no overflow error unless astronomically large numbers involved (arbitrary length mantissa; integral-valued double exponent can overflow, but extremely unlikely)
- very slow, can require considerable memory (unbounded)
- default constructor does not initialize to particular value
- stream inserter (i.e., **operator**<<) for MP_Float first converts MP_Float to **double** and then outputs result
- stream extracter (i.e., **operator**>>) for MP_Float first reads **double** and then converts to MP_Float

MP_Float Example

```
#include <iostream>
#include <CGAL/MP_Float.h>

int main() {
    CGAL::MP_Float x;
    CGAL::MP_Float y;
    if (!(std::cin >> x >> y)) {return 1;}
    if (x < y) {
        std::cout << x << " is less than " << y << '\n';
    }
    CGAL::MP_Float z = -(x + y) * (x - y) + x;
    std::cout << z << '\n';
}
```

`Interval_nt` Class

- declared as: `template <bool M = true> Interval_nt<M>`
- `M` indicates if safe rounding mode enabled
- if safe rounding mode enabled, rounding mode always restored to round towards zero (required by C++); must be careful if safe rounding mode not used
- when safe rounding mode not used, faster but need to worry about things like compiler options like `-frounding-math`
- `using Interval_nt_advanced = Interval_nt<false>;` (i.e., `Interval_nt_advanced` is `Interval_nt` with safe rounding mode disabled)
- interval-arithmetic number type (internally uses floating-point type)
- represents interval $[a, b]$
- every arithmetic operation performed twice, once while rounding towards $-\infty$ to produce result a' and once while rounding towards $+\infty$ to produce result b'
- true answer must lie on interval $[a', b']$
- approximately twice of time cost of built-in floating-point type

Geometry Kernels

- represent geometric objects (e.g., point, line, line segment, ray, plane, triangle, circle,)
- points in 2 or 3 dimensions
- provide operations on geometric objects (e.g., intersection, composition)
- allow certain conditions to be tested involving geometric objects (e.g., collinear, coplanar, equality)

Point Representation

- Cartesian kernels: coordinates represented in Cartesian form
- homogeneous kernels: coordinates represented in homogeneous form

Simple_cartesian and Cartesian Classes

- geometry kernel that represents coordinates in Cartesian form
- declaration:

```
template <class F> Simple_cartesian<F>
```

- declaration:

```
template <class F> Cartesian<F>
```

- `F` field number type (used to represent coordinates)
- `F` often chosen as **double**
- `Cartesian` is reference counted version of `Simple_cartesian`, which allows more efficient copying of objects
- `Cartesian` probably preferred if frequent copying occurs

- geometry kernel that represents coordinates in homogeneous form
- declaration:

```
template <class R> Simple_homogeneous<R>
```

- declaration:

```
template <class R> Homogeneous<R>
```

- R ring number type used for representing numerator and denominator of rational coordinates
- Homogeneous is reference counted version of Simple_homogeneous, which allows more efficient copying of objects
- Homogeneous probably preferred if frequent copying occurs

Constructions

- produces new geometric object from other objects
- result is not one of a small number of enumerable values
- result is numerical (e.g., involves real numbers)
- create line segment from two points
- create triangle from three points
- create plane from three (non-coplanar) points
- create circle from three (non-collinear) points
- find intersection of line and plane
- exact construction: any newly created geometric objects resulting from construction are exactly represented (i.e., no roundoff/overflow error)
- inexact construction: newly created geometric objects are not guaranteed to be exactly represented (e.g., due to roundoff error)
- extremely important to be aware of whether kernel being used provides exact constructions; affects how you write code!!!

Predicates

- does not involve any newly computed numerical data
- result is one of very small set of values, such as boolean or enumerated type
- typically used to make decisions (i.e., affect control flow)
- are three points collinear (true or false)
- are four points coplanar (true or false)
- what is position of point relative to oriented line (left of, right of, or on)
- what is position of point relative to oriented circle (inside, outside, or on)
- exact predicate: result of test is guaranteed to be correct (i.e., result determined as if by exact computation)
- inexact predicate: result of test may be incorrect (e.g., due to roundoff/overflow error)
- extremely important to be aware of whether kernel being used provides exact predicates; affects how you write code!!!

Kernel Member Types: Basic Types

Member Type	Description
`FT`	field number type (e.g., **`double`**)
`RT`	ring number type (e.g., **`int`**)
`Boolean`	boolean type (**`bool`** or `Uncertain<`**`bool`**`>`)
`Sign`	sign (`Sign` or `Uncertain<Sign>`)
`Comparison_result`	comparison result (`Comparison_result` or `Uncertain<Comparison_result>`)
`Orientation`	orientation (`Orientation` or `Uncertain<Orientation>`)
`Oriented_side`	oriented side (`Oriented_side` or `Uncertain<Oriented_side>`)
`Bounded_side`	bounded side (`Bounded_side` or `Uncertain<Bounded_side>`)
`Angle`	angle (`Angle` or `Uncertain<Angle>`)

Member Type	Description
`Point_2`	point in two dimensions
`Vector_2`	vector in two dimensions
`Direction_2`	direction in two dimensions
`Line_2`	line in two dimensions
`Ray_2`	ray in two dimensions
`Segment_2`	line segment in two dimensions
`Triangle_2`	triangle in two dimensions
`Iso_rectangle_2`	axis-aligned rectangle in two dimensions
`Circle_2`	circle in two dimensions

Member Type	Description
Point_3	point in three dimensions
Vector_3	vector in three dimensions
Direction_3	direction in three dimensions
Iso_cuboid_3	axis-aligned cuboid in three dimensions
Line_3	line in three dimensions
Ray_3	ray in three dimensions
Circle_3	circle in three dimensions
Sphere_3	sphere in three dimensions
Segment_3	line segment in three dimensions
Plane_3	plane in three dimensions
Triangle_3	triangle in three dimensions
Tetrahedron_3	tetrahedron in three dimensions

Kernel Selection

- coordinate representation
- exact or inexact constructions
- exact or inexact predicates
- in practice, almost always require exact predicates
- if code well designed, need for exact constructions can usually be avoided
- for `T` chosen as any numeric type that has roundoff/overflow error (e.g., **`float`**, **`double`**, **`long double`**), the following kernels do not provide exact constructions or exact predicates:

```
Simple_cartesian<T>
Cartesian<T>
Simple_homogeneous<T>
Homogeneous<T>
```

- class to convert kernel with inexact predicates into one with exact predicates
- declared as:

```
template <class K> Filtered_kernel<K>
```

- `K` is kernel from which to make filtered kernel
- predicates of `K` replaced by predicates using numeric type `Interval_nt`
- if interval arithmetic can yield reliable answer, result used
- otherwise, exception thrown and caught by class and predicate using `MP_Float` used
- for exact predicates with `Simple_cartesian<double>`, use: `Filtered_kernel<Simple_cartesian<double>>` or equivalently `Exact_predicates_inexact_constructions_kernel`
- `Exact_predicates_inexact_constructions_kernel` very commonly used

Writing Custom Exact Predicates

- exact predicate cannot at any point rely on a computation that is not exact
- no floating point arithmetic (since it has roundoff error)
- no integer arithmetic that might overflow
- no inexact constructions
- no inexact predicates
- `Filtered_predicate` may be helpful

Filtered_predicate Class

- adapter for predicate functors for producing efficient exact predicates
- declared as:

```
template <class EP, class FP, class CE, class CF>
Filtered_predicate<EP, FP, CE, CF>
```

- `EP` is exact predicate (typically uses arbitrary-precision type such as `MP_Float`)
- `FP` is filtering predicate (typically uses interval-arithmetic type like `Interval_nt`)
- `CE` and `CF` are function objects for converting arguments of unfiltered predicate to types used by exact and filtering predicates
- must be careful about operation used in unfiltered predicate being plugged into `Filtered_kernel`
- for kernel ring number type `RT`, can safely use addition, subtraction, multiplication
- can also safely use `sign`

Execution of Filtered Predicate

- execution of code for filtered predicate functor proceeds as follows:
  1. invoke unfiltered (i.e., original) predicate functor for numeric type `CGAL::Interval_nt<false>`
     if any operation on interval arithmetic type yields uncertain result (e.g., `CGAL::sign`), exception is thrown, with thrown exception being caught by filtered predicate functor
  2. if no exception thrown (so that unfiltered functor returns normally), return return value of unfiltered functor (and we are done); otherwise, continue
  3. invoke unfiltered predicate functor for numeric type `CGAL::MP_Float`
  4. return return value of unfiltered functor

Filtered Predicate Example

```
#include <CGAL/Cartesian.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Interval_nt.h>
#include <CGAL/Filtered_predicate.h>
#include <CGAL/Cartesian_converter.h>

template <class K>
struct Test_orientation_2 {
   using RT = typename K::RT;
   using Point_2 = typename K::Point_2;
   using result_type = typename K::Orientation;
   result_type operator()(const Point_2& p, const Point_2& q,
     const Point_2& r) const {
       RT prx = p.x() - r.x();
       RT pry = p.y() - r.y();
       RT qrx = q.x() - r.x();
       RT qry = q.y() - r.y();
       return CGAL::sign(prx * qry - qrx * pry);
   }
};

using Kernel = CGAL::Cartesian<double>;
using Ia_kernel = CGAL::Cartesian<CGAL::Interval_nt<false>>;
using Exact_kernel = CGAL::Cartesian<CGAL::MP_Float>;
using Test_orientation = CGAL::Filtered_predicate<
  Test_orientation_2<Exact_kernel>,
  Test_orientation_2<Ia_kernel>,
  CGAL::Cartesian_converter<Kernel, Exact_kernel>,
  CGAL::Cartesian_converter<Kernel, Ia_kernel>
  >;

int main() {
   double big = 1e50;
   Kernel::Point_2 p(0.0, 0.0), q(1.0, 1.0), r(2.0 * big, 2.0 * big);
   Test_orientation orientation;
   std::cout << orientation(p, q, r) << "\n";
}
```

Filtered Predicate Example (Continued)

- for example on previous slide, execution of filtered predicate functor proceeds as follows:
  1. invoke `Test_orientation_2<Cartesian<CGAL::Interval_nt<false>>>` functor with points $([0,0],[0,0])$, $([1,1],[1,1])$, $([2\cdot 10^{50}, 2\cdot 10^{50}], [2\cdot 10^{50}, 2\cdot 10^{50}])$
  2. `CGAL::sign` called for $[-1.55414\cdot 10^{85}, 1.55414\cdot 10^{85}]$, which results in exception being thrown
  3. exception caught by filtered predicate code
  4. invoke `Test_orientation_2<Cartesian<CGAL::MP_Float>>` functor with points $(0,0)$, $(1,1)$, $(2\cdot 10^{50}, 2\cdot 10^{50})$
  5. `CGAL::sign` called for 0, resulting in return value of 0
  6. filtered predicate returns 0
- critically important that `RT` used for all arithmetic operations and not **`double`** (or **`float`**); otherwise, arithmetic computation done using wrong numeric type, which will prevent predicate from being correct (i.e., exact)

Section 5.2.2

Polygon Meshes

Polyhedron_3 Class

- represents polyhedral surface (i.e., polygon mesh), which consists of vertices, edges, and facets and incidence relationship amongst them
- each edge represented by pair of halfedges
- declaration for `Polyhedron_3` class:

```
template <class Kernel,
  class PolyhedronItems = CGAL::Polyhedron_items_3,
  template <class T, class I>
    class HalfedgeDS = CGAL::HalfedgeDS_default,
  class Alloc = CGAL_ALLOCATOR(int)>
class Polyhedron_3;
```

- `Kernel` is geometry kernel, which specifies such things as how points are represented and provides basic geometric operations/predicates (e.g., `CGAL::Cartesian<double>` and `CGAL::Filtered_kernel<CGAL::Cartesian<double>>`)
- `PolyhedronItems` specifies data types for representing vertices and facets (in many cases, default will suffice)
- `HalfedgeDS` specifies halfedge data structure for representing polygon mesh and `Alloc` specifies allocator (defaults should suffice)

Polyhedron_3 Type Members

Basic Types

Type	Description
Vertex	vertex type
Halfedge	halfedge type
Facet	facet type
Point_3	point type (for vertices)

Handles

Type	Description
Vertex_const_handle	const handle to vertex
Vertex_handle	handle to vertex
Halfedge_const_handle	const handle to halfedge
Halfedge_handle	handle to halfedge
Facet_const_handle	const handle to facet
Facet_handle	handle to facet

Iterators

Type	Description
`Vertex_const_iterator`	const iterator over all vertices
`Vertex_iterator`	iterator over all vertices
`Halfedge_const_iterator`	const iterator over all halfedges
`Halfedge_iterator`	iterator over all halfedges
`Facet_const_iterator`	const iterator over all facets
`Facet_iterator`	iterator over all facets
`Edge_const_iterator`	const iterator over all edges (every other halfedge)
`Edge_iterator`	iterator over all edges (every other halfedge)

Circulators

Type	Description
`Halfedge_around_vertex_const_circulator`	const circulator of halfedges around vertex (CW)
`Halfedge_around_vertex_circulator`	circulator of halfedges around vertex (CW)
`Halfedge_around_facet_const_circulator`	const circulator of halfedges around facet (CCW)
`Halfedge_around_facet_circulator`	circulator of halfedges around facet (CCW)

Size

Name	Description
size_of_vertices	get number of vertices
size_of_halfedges	get number of halfedges
size_of_facets	get number of facets

Iterators

Name	Description
vertices_begin	iterator for first vertex in mesh
vertices_end	past-the-end vertex iterator
halfedges_begin	iterator for first halfedge in mesh
halfedges_end	past-the-end halfedge iterator
facets_begin	iterator for first facet in mesh
facets_end	past-the-end facet iterator
edges_begin	iterator for first edge in mesh
edges_end	past-the-end edge iterator

Polyhedron_3 Function Members (Continued 1)

Combinatorial Predicates

Name	Description
is_closed	true if no border edges (no boundary)
is_pure_triangle	true if all facets are triangles
is_pure_quad	true if all facets are quadrilaterals

Border Halfedges

Name	Description
normalized_border_is_valid	true if border is normalized
normalize_border	sort halfedges such that non-border edges precede border edges (i.e., normalize border)
size_of_border_halfedges	get number of border halfedges (border must be normalized)
size_of_border_edges	get number of border edges (border must be normalized)
border_halfedges_begin	halfedge iterator starting with border edges (border must be normalized)
border_edges_begin	edge iterator starting with border edges (border must be normalized)

- `Facet` type represents facet (i.e., face) in polyhedral surface
- actual class type to which `Facet` corresponds depends on choice of `PolyhedronItems` template parameter for `Polyhedron_3` class
- depending on actual class type to which `Facet` refers, level of functionality offered by `Facet` class may differ (e.g., available function members may differ)
- `Facet` class may contain following optional information:
  - plane equation (corresponding to plane containing facet)
  - handle for halfedge that is incident on facet
- some member functions in `Facet` class provide access to halfedge-around-facet circulator
- halfedge-around-facet circulator may be either forward or bidirectional

Facet Function Members

Operations Available If Facet Plane Supported

Name	Description
plane	get plane equation

Operations Available If Facet Halfedge Supported

Name	Description
halfedge	get halfedge incident on facet
facet_begin	get circulator of halfedges around facet (CCW)
set_halfedge	set incident halfedge
facet_degree	get degree of facet (i.e., number of edges on boundary of facet)
is_triangle	true if facet is triangle
is_quad	true if facet is quadrilateral

Polyhedron_3::Vertex

- `Vertex` type represents vertex in polyhedral surface
- actual class type to which `Vertex` corresponds depends on choice of `PolyhedronItems` template parameter for `Polyhedron_3` class
- depending on actual class type to which `Vertex` refers, level of functionality offered by `Vertex` class may differ (e.g., available function members may differ)
- `Vertex` class may contain following optional information:
  - point (corresponding to vertex position)
  - handle for halfedge that is incident on vertex
- some member functions in `Vertex` class provide access to halfedge-around-vertex circulator
- halfedge-around-vertex circulator may be either forward or bidirectional

Vertex Function Members

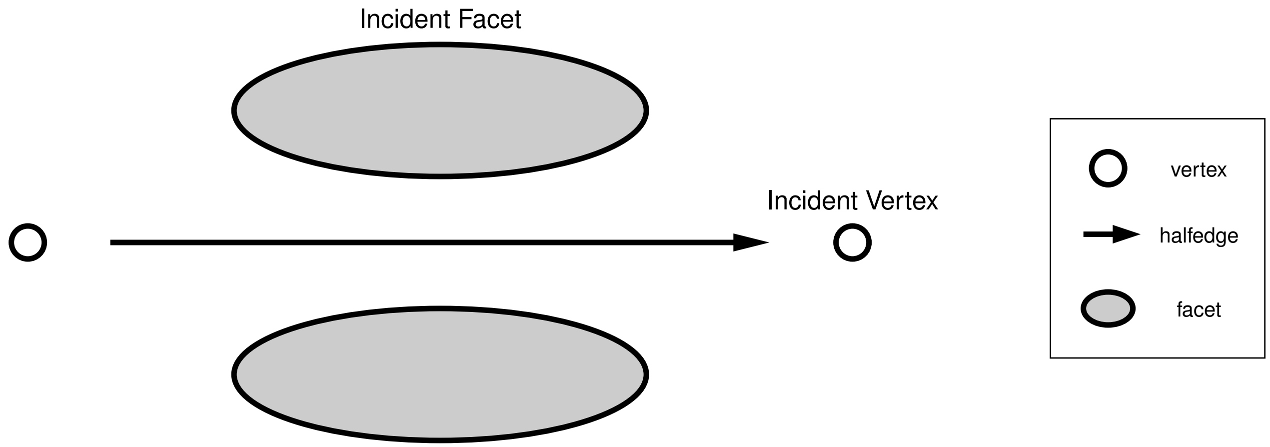
Operations Available If Vertex Point Supported

Name	Description
point	get point associated with vertex

Operations Available If Vertex Halfedge Supported

Name	Description
halfedge	get halfedge incident on vertex
vertex_begin	circulator of halfedges around vertex (CW)
set_halfedge	set incident halfedge
vertex_degree	get valence of vertex
is_bivalent	true if vertex has valence two
is_trivalent	true if vertex has valence three

- `Halfedge` type represents halfedge in polyhedral surface
- actual class type to which `Halfedge` corresponds depends on choice of `PolyhedronItems` template parameter for `Polyhedron_3` class
- depending on actual class type to which `Halfedge` refers, level of functionality offered by `Halfedge` class may differ (e.g., available function members may differ)
- each halfedge directly associated with one vertex and one facet, referred to as incident vertex and incident facet, respectively
- **incident vertex** is vertex at ***terminal end*** of halfedge
- **incident facet** is facet to ***left*** of halfedge



- halfedge contains:
  - handle for next halfedge around incident facet in CCW direction
  - handle for opposite halfedge
- together, these two handles allow for efficient iteration around:
  - halfedges incident on facet in CCW direction ***only***; and
  - halfedges incident on vertex in CW direction ***only***
- halfedge may optionally contain:
  - handle for ***previous*** halfedge around incident facet in CCW direction
- addition of this optional handle allows for efficient iteration around:
  - halfedges incident on facet in ***both*** (CW and CCW) directions; and
  - halfedges incident on vertex in ***both*** (CW and CCW) directions
- halfedge may also contain following optional information:
  - handle for incident vertex
  - handle for incident facet
- if halfedge class provides `prev` member function, halfedge-around-vertex and halfedge-around-facet circulators are bidirectional; otherwise, they are forward only

`Halfedge` Function Members

Adjacency Queries

Name	Description
`opposite`	get opposite halfedge
`next`	get next halfedge incident on same facet in CCW order
`prev`	get previous halfedge incident on same facet in CCW order
`next_on_vertex`	get next halfedge incident on same vertex in CW order
`prev_on_vertex`	get previous halfedge incident on same vertex in CW order

Circulators

Name	Description
`vertex_begin`	get halfedge-around-vertex circulator for incident vertex (CW order)
`facet_begin`	get halfedge-around-facet circulator for incident facet (CCW order)

Border Queries

Name	Description
`is_border`	true if border halfedge
`is_border_edge`	true if associated edge on border

Vertex Valence Queries

Name	Description
`vertex_degree`	get valence of incident vertex
`is_bivalent`	true if incident vertex has valence two
`is_trivalent`	true if incident vertex has valence three

Facet Degree Queries

Name	Description
`facet_degree`	get degree of incident facet
`is_triangle`	true if incident facet is triangle
`is_quad`	true if incident facet is quadrilateral

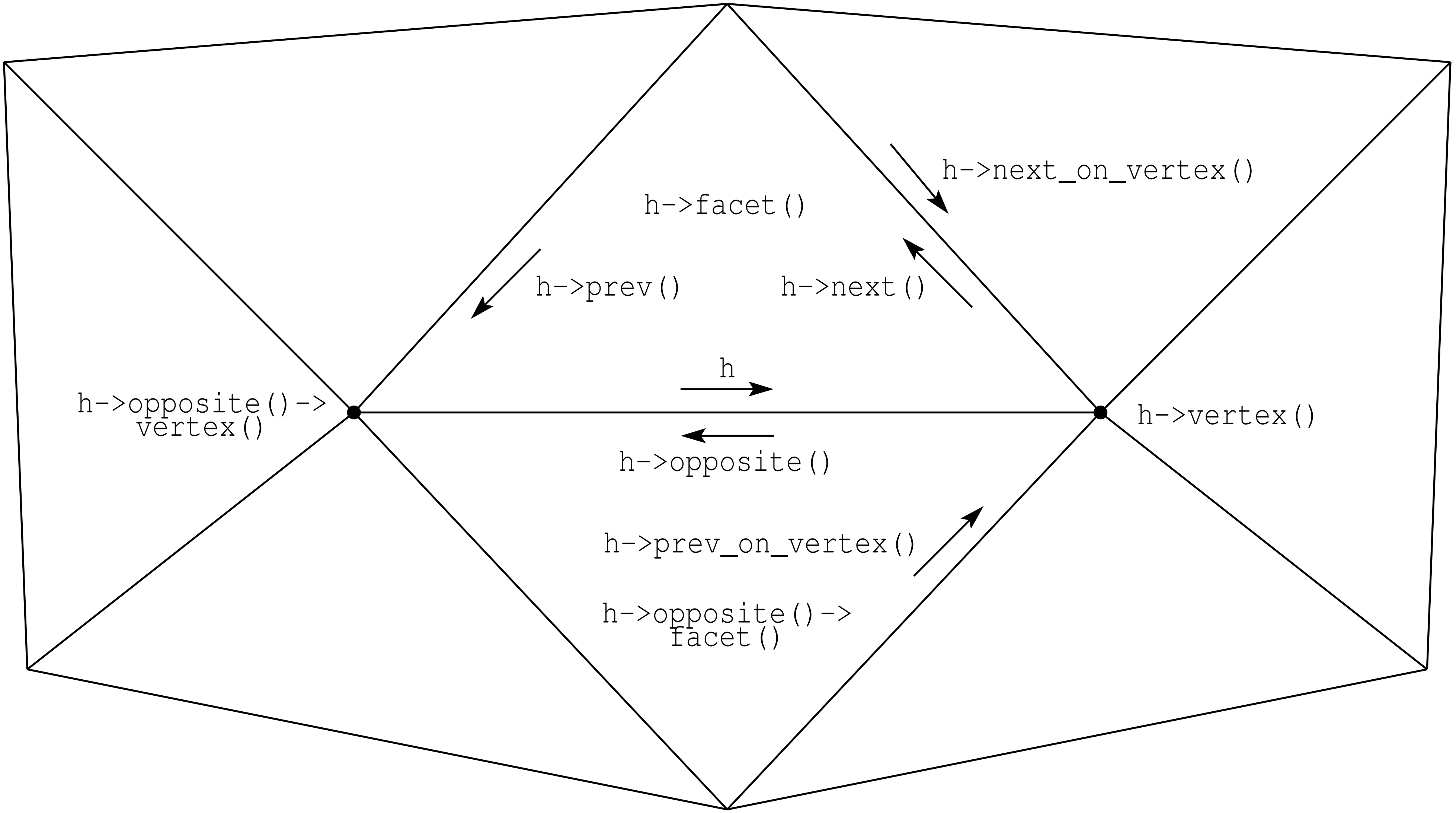
Operations Available If Halfedge Vertex Supported

Name	Description
`vertex`	get handle for incident vertex of halfedge

Operations Available If Halfedge Facet Supported

Name	Description
`facet`	get handle for incident facet of halfedge

Adjacency Example



`Polyhedron_3` I/O

- **`operator`**<< and **`operator`**>> are overloaded for I/O
- read and write polygon mesh data in OFF format

Polyhedron_3 Gotchas

- be mindful of operations on `Polyhedron_3` that may invalidate handles, iterators, or circulators
- halfedge-around-vertex circulators and halfedge-around-facet circulators iterate in opposite directions (i.e., CCW versus CW)
- be careful about const correctness (e.g., const versus mutable handles/iterators/circulators)
- some `Polyhedron_3` operations only valid if border normalized (e.g., `size_of_border_halfedges`, `size_of_border_edges`)
- exactly ***one*** of two halfedges associated with border edge is border halfedge

Section 5.2.3

Surface Subdivision Methods

- several functions provided for performing subdivision of polygon meshes (represented by `Polyhedron_3`)
- generic subdivision functions apply specific topologic refinement rule but allow arbitrary geometric refinement rule
- specific subdivision functions apply specific subdivision method
- contained in `CGAL::Subdivision_method_3` namespace

Subdivision Functions

Generic Subdivision Methods

Function	Description
`PQQ`	perform primal quadrilateral quadrisection with arbitrary geometric refinement rule
`PTQ`	perform primal triangle quadrisection with arbitrary geometric refinement rule
`DQQ`	perform dual quadrilateral quadrisection with arbitrary geometric refinement rule
`Sqrt3`	perform $\sqrt{3}$ topologic refinement with arbitrary geometric refinement rule

Specific Subdivision Methods

Function	Description
`CatmullClark_subdivision`	perform Catmull-Clark subdivision
`Loop_subdivision`	perform Loop subdivision
`DooSabin_subdivision`	perform Doo-Sabin subdivision
`Sqrt3_subdivision`	perform Kobbelt $\sqrt{3}$ subdivision

Section 5.2.4

Example Programs

Mesh Generation Program: `meshMake`

- This program generates a simple triangle mesh corresponding to a tetrahedron.
- First, a polygon mesh corresponding to a tetrahedron is constructed.
- Then, the resulting mesh is written to standard output in Object File Format (OFF).

Mesh Information Program: `meshInfo`

- This program extracts some basic information from a polygon mesh.
- First, a polygon mesh is read from standard input in Object File Format (OFF).
- Then, various information is extracted from the mesh, including:
  - the type of mesh (e.g., triangle, quadrilateral, or general)
  - the number of vertices, edges, faces, and halfedges in the mesh
  - the minimum, maximum, and average valence of vertices in the mesh
  - the number of nonplanar faces in the mesh
- The above information is printed to standard output.

Mesh Subdivision Program: `meshSubdivide`

- This program performs subdivision on a polygon mesh.
- First, a mesh is read from standard input in Object File Format (OFF).
- Next, the mesh is refined using the given number of iterations of the specified subdivision method.
- Finally, the refined mesh is written to standard output in OFF.
- Several subdivision schemes are supported, including: Loop, Catmull-Clark, Doo-Sabin, and Kobbelt $\sqrt{3}$.

Section 5.3

OpenGL Utility Toolkit (GLUT)

OpenGL Utility Toolkit (GLUT)

- simple windowing API for OpenGL
- intended to be used with small to medium sized OpenGL programs
- language binding for C
- window-system independent
- supports most mainstream operating systems (Microsoft Windows, Linux/Unix)
- provides window management functionality (e.g., creating/destroying windows, displaying/resizing windows, and querying/setting window attributes)
- allows for user input (e.g., via keyboard, mouse)
- routines for drawing common wireframe/solid 3-D objects such as sphere, torus, and well-known teapot model
- register callback functions to handle various types of events (e.g., display, resize, keyboard, special keyboard, mouse, timer, idle) and then loop processing events
- open-source implementation of GLUT called Freeglut is available from `http://sourceforge.net/projects/freeglut`

Event-Driven Model

- event-driven model: flow of program determined by events (e.g., mouse clicks, key presses)
- application making use of event-driven model performs some initialization and then enters an event-processing loop for duration of execution
- each iteration of event-processing loop does following:
  1. wait for event
  2. process event
- many libraries for building graphical user interfaces (GUIs) employ event-driven model
- GLUT uses event-driven model

Structure of GLUT Application

1. initialize GLUT library by calling `glutInit`
2. set display mode (via `glutInitDisplay`)
3. perform any additional initialization such as:
   - create windows (via `glutCreateWindow`)
   - register callback functions for handling various types of events (e.g., via `glutDisplayFunc`, `glutReshapeFunc`, `glutKeyboardFunc`)
   - setup initial OpenGL state (e.g., depth buffering, shading, lighting, clear color)
4. enter main event-processing loop by calling `glutMainLoop` [Note that `glutMainLoop` never returns.]

GLUT Header Files

- OpenGL and GLUT header files in `GL` (or `GLUT`) directory
- to use GLUT, need to include `glut.h` in `GL` (or `GLUT`) directory
- header file `glut.h` also includes all necessary OpenGL header files (e.g., `gl.h`, `glu.h`, `glext.h`)

Event Types

Event Type	Description
display	window contents needs to be displayed
overlay display	overlay plane contents needs to be displayed
reshape	window has been resized
keyboard	key has been pressed
mouse	mouse button has been pressed or released
motion	mouse moved within window while one or more buttons pressed
passive motion	mouse moved within window while no buttons pressed
visibility	visibility of window has changed (covered versus uncovered)
entry	mouse has left or entered window
special keyboard	special key has been pressed (e.g., arrow keys, function keys)

Event Types (Continued)

Event Type	Description
spaceball motion	spaceball translation has occurred
spaceball rotate	spaceball rotation has occurred
spaceball button	spaceball button has been pressed or released
button box	button box activity has occurred
dials	dial activity has occurred
tablet motion	tablet motion has occurred
tablet button	table button has been pressed or released
menu status	menu status change
idle	no event activity has occurred
timer	timer has expired

Functions

Initialization

Function	Description
`glutInit`	initialize GLUT library
`glutInitWindowSize`	set initial window size for `glutCreateWindow`
`glutInitWindowPosition`	set initial window position for `glutCreateWindow`
`glutInitDisplayMode`	set initial display mode

Beginning Event Processing

Function	Description
`glutMainLoop`	enter GLUT event-processing loop

Window Management

Function	Description
`glutCreateWindow`	create top-level window
`glutCreateSubWindow`	create subwindow
`glutSetWindow`	set current window
`glutGetWindow`	get current window
`glutDestroyWindow`	destroys specified window
`glutPostRedisplay`	mark current window as needing to be redisplayed
`glutSwapBuffers`	swaps buffers of current window if double buffered (flushes graphics output via `glFlush`)
`glutPositionWindow`	request change to position of current window
`glutReshapeWindow`	request change to size of current window
`glutFullScreen`	request current window to be made full screen
`glutSetWindowTitle`	set title of current top-level window
`glutSetIconTitle`	set title of icon for current top-level window
`glutSetCursor`	set cursor image for current window

Menu Management

Function	Description
`glutCreateMenu`	create new pop-up menu
`glutSetMenu`	set current menu
`glutGetMenu`	get current menu
`glutDestroyMenu`	destroy specified menu
`glutAddMenuEntry`	add menu entry to bottom of current menu
`glutAddSubMenu`	add submenu trigger to bottom of current menu
`glutChangeToMenuEntry`	change specified menu item in current menu into menu entry
`glutChangeToSubMenu`	change specified menu item in current menu into submenu trigger
`glutRemoveMenuItem`	remove specified menu item
`glutAttachMenu`	attach mouse button for current window to current menu
`glutDetachMenu`	detach attached mouse button from current window

Callback Registration

Function	Description
`glutDisplayFunc`	sets display callback for current window
`glutReshapeFunc`	sets reshape callback for current window
`glutKeyboardFunc`	sets keyboard callback for current window
`glutMouseFunc`	sets mouse callback for current window
`glutMotionFunc`	set motion callback for current window
`glutPassiveMotionFunc`	set passive motion callback for current window
`glutVisibilityFunc`	set visibility callback for current window
`glutEntryFunc`	set mouse enter/leave callback for current window
`glutSpecialFunc`	sets special keyboard callback for current window
`glutIdleFunc`	set global idle callback
`glutTimerFunc`	registers timer callback to be triggered in specified number of milliseconds

Functions (Continued 4)

State Retrieval

Function	Description
glutGet	retrieves simple GLUT state (e.g., size or position of current window)
glutDeviceGet	retrieves GLUT device information (e.g., keyboard, mouse, spaceball, tablet)
glutGetModifiers	retrieve modifier key state when certain callbacks generated (i.e., state of shift, control, and alt keys)

Font Rendering

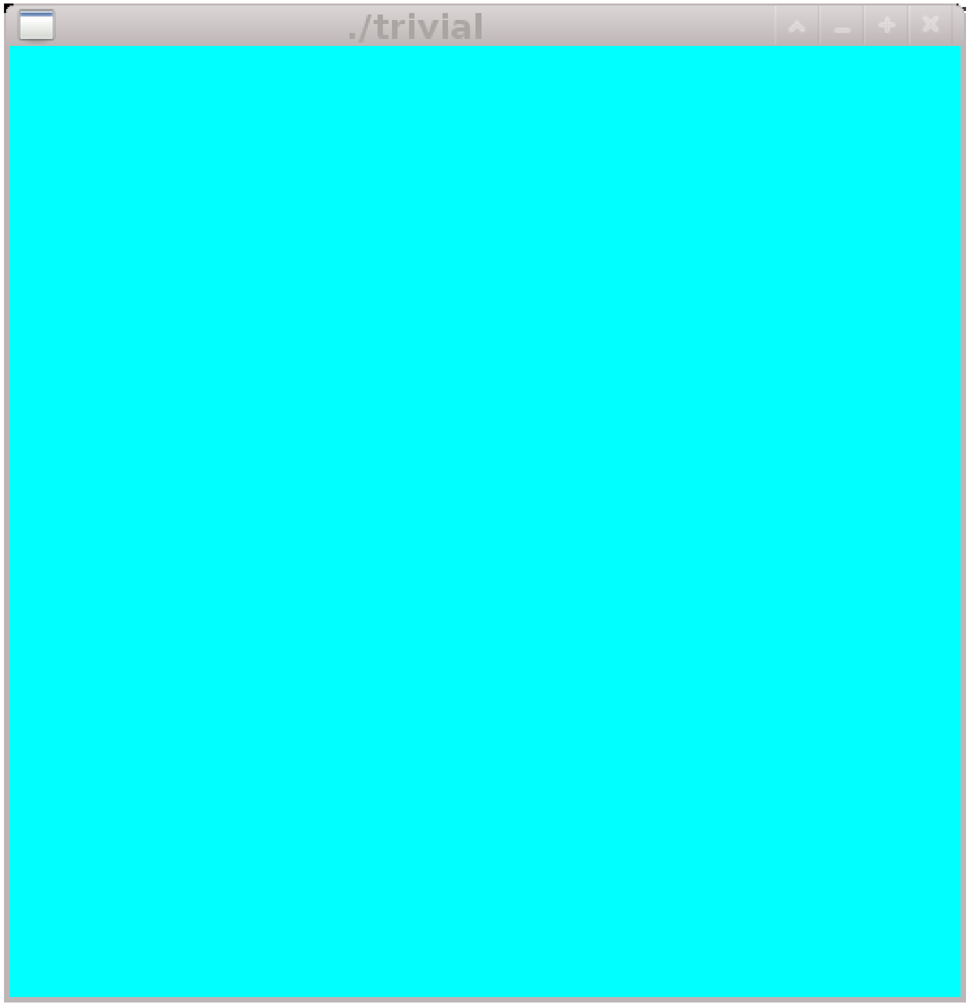
Function	Description
glutBitmapCharacter	renders bitmap character using OpenGL
glutBitmapWidth	get width of bitmap character
glutStrokeCharacter	renders stroke character using OpenGL
glutStrokeWidth	get width of stroke character

Geometric Object Rendering

Function	Description
`glutSolidSphere`	render solid sphere
`glutWireSphere`	render wireframe sphere
`glutSolidCube`	render solid cube
`glutWireCube`	render wireframe cube
`glutSolidCone`	render solid cone
`glutWireCone`	render wireframe cone
`glutSolidTorus`	render solid torus
`glutWireTorus`	render wireframe torus
`glutSolidOctahedron`	render solid octahedron
`glutWireOctahedron`	render wireframe octahedron
`glutSolidTetrahedron`	render solid tetrahedron
`glutWireTetrahedron`	render wireframe tetrahedron
`glutSolidTeapot`	render solid teapot
`glutWireTeapot`	render wireframe teapot

Minimalist GLUT Program

- minimalist program using GLUT
- create window that is cleared to particular color



```cpp
// Create a window that is cleared to a particular color
// when drawn.

#include <GL/glut.h>

void display() {
    glClearColor(0.0, 1.0, 1.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glutSwapBuffers();
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(512, 512);
    glutCreateWindow(argv[0]);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

GLUT References

1. M. J. Kilgard. *The OpenGL Utility Toolkit (GLUT): Programming Interface (API Version 3)*, Nov. 1996.
   Available from `http://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf`.
2. R. S. Wright, B. Lipchak, and N. Haemel. *OpenGL SuperBible*. Addison-Wesley, Upper Saddle River, NJ, USA, 4th edition, 2007.
3. GLUT home page:
   `http://www.opengl.org/resources/libraries/glut`
4. GLUT manual (HTML format):
   `http://www.opengl.org/resources/libraries/glut/spec3/spec3.html`

Section 5.4

OpenGL Framework (GLFW) Library

OpenGL Framework (GLFW) Library

- lightweight open-source windowing API for OpenGL, OpenGL ES, and Vulkan
- language binding for C
- window-system independent
- supports most mainstream operating systems (e.g., Microsoft Windows, OS X, and Linux/Unix)
- provides window management functionality (e.g., creating/destroying windows, displaying/resizing windows, and querying/setting window attributes)
- allows for user input (e.g., via keyboard, mouse, and joystick)
- allows application to register callback functions to handle various types of events (e.g., window refresh, window resize, keyboard, and mouse) and then loop processing events
- web site: `http://www.glfw.org`

GLFW Versus GLUT

- GLFW and modern GLUT (e.g., FreeGLUT) offer somewhat similar functionality
- GLFW allows greater control over event processing loop
- GLFW has clipboard support
- GLFW supports dragging and dropping of files/directories in window
- GLUT has much longer history than GLFW (which can make code examples and tutorials using GLUT relatively easier to find)
- GLUT has built-in support for rendering text and some basic geometric objects
- GLUT has primitive support for menus

Event-Driven Model

- event-driven model: flow of program determined by events (e.g., mouse clicks and key presses)
- application making use of event-driven model performs some initialization and then enters event-processing loop for duration of execution
- each iteration of event-processing loop does following:
    1. wait for event
    2. process event
- many libraries for building graphical user interfaces (GUIs) employ event-driven model
- GLFW uses event-driven model

Structure of GLFW Application

1. initialize GLFW library by calling `glfwInit`
2. perform any additional initialization such as:
   - select type of OpenGL (or OpenGL ES) context to be used for subsequently created windows (via `glfwWindowHint`)
   - create windows (via `glfwCreateWindow`)
   - register callback functions for handling various types of events (e.g., via `glfwRefreshCallback`, `glfwSetWindowSizeCallback`, `glfwSetCharCallback`)
   - configure initial OpenGL state (e.g., depth buffering and clear color) and shaders
3. enter main event-processing loop, which repeatedly calls `glfwWaitEvents`, `glfwPollEvents`, or other similar functions
4. cleanup GLFW library by calling `glfwTerminate`

GLFW Header Files

- GLFW header files in directory `GLFW`
- to use GLFW, need to include `glfw3.h`:

```
#include <GLFW/glfw3.h>
```

- header file `glfw3.h` also includes all necessary OpenGL header files (e.g., `gl.h`, `glu.h`, `glext.h`)
- if using OpenGL extension loading library (such as GLEW), header for this library should be included before `glfw3.h`

Event Types

Keyboard, Mouse, and Joystick Events

Event Type	Description
key	key has been pressed, released, or repeated
character	character has been typed without modifiers
character with modifiers	character has been typed with modifiers
mouse button	mouse button has been pressed or released
cursor position	cursor has moved
cursor enter	cursor has entered or left client area of window
scroll	scrolling device has been used (e.g., mouse wheel or touchpad scrolling area)
joystick	joystick has been connected or disconnected
drop	files/directories have been dropped on window

Event Types (Continued 1)

Framebuffer, Window, and Monitor Events

Event Type	Description
framebuffer size	framebuffer has been resized
window close	window has been closed
window refresh	window contents need to be redrawn
window size	window size has changed
window position	window position has changed
window iconify	window has been iconified or deiconified
window focus	window focus has changed (i.e., been gained or lost)
monitor	monitor has been connected or disconnected

Other Events

Event Type	Description
error	error has occurred in GLFW library

Functions

Initialization and Termination

Function	Description
`glfwInit`	initialize GLFW library
`glfwTerminate`	cleanup GLFW library

Version

Function	Description
`glfwGetVersion`	get version of GLFW library
`glfwGetVersionString`	get version string of GLFW library

Window Creation and Destruction

Function	Description
`glfwCreateWindow`	create window and its associated OpenGL or OpenGL ES context
`glfwDestroyWindow`	destroy window and its associated context
`glfwDefaultWindowHints`	reset all window hints to their default values
`glfwWindowHint`	set window hints for subsequently created windows

Functions (Continued 1)

Setting and Querying Window Attributes

Function	Description
glfwWindowShouldClose	get close flag for specified window
glfwSetWindowShouldClose	set close flag for specified window
glfwSetWindowTitle	set title of specified window
glfwSetWindowIcon	set icon for specified window
glfwGetWindowPos	get position of client area of specified window
glfwSetWindowPos	set position of client area of specified window
glfwGetWindowSize	get size of client area of specified window
glfwSetWindowSize	set size of client area of specified window
glfwSetWindowSizeLimits	set size limits of client area of specified window
glfwSetWindowAspectRatio	set required aspect ratio of client area of specified window

Functions (Continued 2)

Setting and Querying Window Attributes (Continued)

Function	Description
`glfwGetFramebufferSize`	get size of framebuffer of specified window
`glfwGetWindowFrameSize`	get size of frame of window
`glfwGetWindowMonitor`	get monitor that specified window uses for full-screen mode
`glfwSetWindowMonitor`	set monitor that specified window uses for full-screen mode
`glfwGetWindowAttrib`	get attribute of specified window
`glfwGetWindowUserPointer`	get user pointer of specified window
`glfwSetWindowUserPointer`	set user pointer of specified window

Window Management

Function	Description
glfwIconifyWindow	iconifies specified window
glfwRestoreWindow	restores (i.e., deiconifies) specified window
glfwMaximizeWindow	maximizes specified window
glfwShowWindow	make specified window visible
glfwHideWindow	hide specified window
glfwFocusWindow	bring specified window to front and give it input focus
glfwSwapBuffers	swap front and back buffers of specified window when rendering with OpenGL or OpenGL ES
glfwSwapInterval	set swap interval for current OpenGL or OpenGL ES context

Functions (Continued 4)

Callback Registration

Function	Description
`glfwSetErrorCallback`	sets error callback function
`glfwSetWindowPosCallback`	sets window-position callback function for specified window
`glfwSetWindowSizeCallback`	sets window-size callback function for specified window
`glfwSetWindowCloseCallback`	sets window-close callback function for specified window
`glfwSetWindowRefreshCallback`	sets window-refresh callback function for specified window
`glfwSetWindowFocusCallback`	sets window-focus callback function for specified window

Functions (Continued 5)

Callback Registration (Continued 1)

Function	Description
`glfwSetWindowIconifyCallback`	sets window-iconify callback function for specified window
`glfwSetFramebufferSizeCallback`	sets callback function for framebuffer size event
`glfwSetKeyCallback`	sets (physical) key callback function for specified window
`glfwSetCharCallback`	sets character callback function for specified window
`glfwSetCharModsCallback`	sets character-with-modifiers callback function for specified window
`glfwSetMouseButtonCallback`	sets mouse-button callback function for specified window
`glfwGetMonitorCallback`	set monitor configuration callback function

Functions (Continued 6)

Callback Registration (Continued 2)

Function	Description
`glfwSetCursorPosCallback`	sets cursor-position callback function for specified window
`glfwSetCursorEnterCallback`	sets cursor-boundary-crossing callback function for specified window
`glfwSetScrollCallback`	sets scroll callback function for specified window
`glfwSetDropCallback`	sets file-drop callback function for specified window
`glfwSetJoystickCallback`	sets joystick-configuration callback function

Event Handling

Function	Description
glfwPostEmptyEvent	post empty event to event queue
glfwPollEvents	process any pending events and return immediately
glfwWaitEvents	wait until at least one event is pending, then process all pending events and return
glfwWaitEventsTimeout	wait until at least one event pending or timeout expires, then process any pending events and return

Timing

Function	Description
glfwGetTime	get value of timer in seconds
glfwSetTime	set value of timer
glfwGetTimerValue	get value of timer in clock ticks
glfwGetTimerFrequency	get frequency of clock tick

Keyboard, Mouse, Joystick, and Cursor

Function	Description
`glfwGetInputMode`	get value of input option for specified window (e.g., cursor, sticky keys/buttons)
`glfwSetInputMode`	set input option for specified window
`glfwGetKeyName`	get localized name of specified printable key
`glfwGetKey`	get last reported state of keyboard key for specified window
`glfwGetMouseButton`	get last reported state of mouse button for specified window
`glfwGetCursorPos`	get position of cursor relative to client area of specified window
`glfwSetCursorPos`	set position of cursor relative to client area of specified window

Keyboard, Mouse, Joystick, and Cursor (Continued)

Function	Description
`glfwCreateCursor`	create custom cursor
`glfwCreateStandardCursor`	creates cursor with standard shape
`glfwDestroyCursor`	destroys cursor
`glfwSetCursor`	set cursor for use in specified window
`glfwJoystickPresent`	test if joystick is present
`glfwGetJoystickAxes`	get values of all axes of specified joystick
`glfwGetJoystickButtons`	get state of all buttons of specified joystick
`glfwGetJoystickName`	get name of specified joystick

Clipboard

Function	Description
`glfwGetClipboardString`	gets contents of clipboard as string
`glfwSetClipboardString`	sets clipboard to specified string

Functions (Continued 10)

Monitor Management

Function	Description
`glfwGetMonitors`	get currently connected monitors
`glfwGetPrimaryMonitor`	get primary monitor
`glfwGetMonitorPos`	get position of specified monitor's viewport on virtual screen
`glfwGetMonitorPhysicalSize`	get physical size of specified monitor
`glfwGetMonitorName`	get name of specified monitor
`glfwGetVideoModes`	get available video modes for specified monitor
`glfwGetVideoMode`	get current video mode of specified monitor
`glfwSetGamma`	set gamma for specified monitor
`glfwGetGammaRamp`	get current gamma ramp for specified monitor
`glfwSetGammaRamp`	set current gamma ramp for specified monitor

Contexts and Extensions

Function	Description
`glfwMakeContextCurrent`	make context of specified window current for calling thread
`glfwGetCurrentContext`	get window whose context is current on calling thread
`glfwExtensionSupported`	tests if specified API extension is supported by current OpenGL or OpenGL ES context
`glfwGetProcAddress`	get address of specified OpenGL or OpenGL ES core or extension function (if supported) for current context

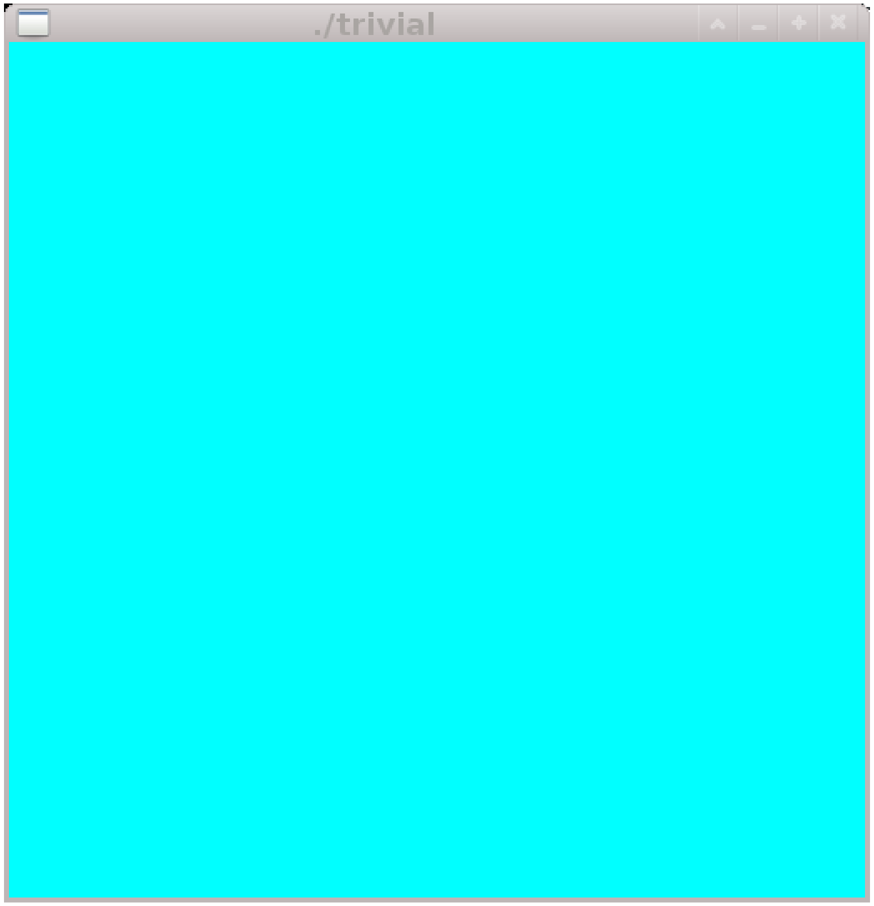
Functions (Continued 12)

Vulkan

Function	Description
`glfwVulkanSupported`	tests if Vulkan loader has been found
`glfwGetRequiredInstanceExtensions`	get Vulkan instance extensions required by GLFW
`glfwGetInstanceProcAddress`	get address of specified Vulkan instance function
`glfwGetPhysicalDevicePresentationSupport`	test if specified queue family can present images
`glfwCreateWindowSurface`	create Vulkan surface for specified Window

Minimalist GLFW Program

- minimalist program using GLFW
- create window that is cleared to particular color



Minimalist GLFW Program: Source Code

```
#include <cstdlib>
#include <GLFW/glfw3.h>

void display(GLFWwindow* window) {
    glClearColor(0.0, 1.0, 1.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glfwSwapBuffers(window);
}

int main(int argc, char** argv) {
    if (!glfwInit()) {return EXIT_FAILURE;}
    glfwSwapInterval(1);
    GLFWwindow* window = glfwCreateWindow(512, 512, argv[0],
      nullptr, nullptr);
    if (!window) {
        glfwTerminate();
        return EXIT_FAILURE;
    }
    glfwMakeContextCurrent(window);
    glfwSetWindowRefreshCallback(window, display);
    while (!glfwWindowShouldClose(window)) {
        glfwWaitEvents();
    }
    glfwTerminate();
    return EXIT_SUCCESS;
}
```

GLFW References

1. GLFW Reference Manual, http://www.glfw.org/docs/latest

Section 5.5

OpenGL Mathematics (GLM) Library

OpenGL Mathematics (GLM) Library

- open-source mathematics library for graphics software based on OpenGL Shading Language (GLSL)
- intended for use with OpenGL
- written in C++
- developed by Christophe Riccio
- provides classes and functions with similar naming conventions and functionality as in GLSL
- web site: `http://glm.g-truc.net`

GLM Header Files

- library has numerous header files
- header files under `glm` directory
- all header files for core GLM functionality can be included by including header file `glm.hpp`
- for matrix transformation functionality, include `gtc/matrix_transform.hpp`
- for string conversion functionality, include `gtx/string_cast.hpp`
- for type value functionality, include `gtc/type_ptr.hpp`
- all identifiers placed in namespace `glm`

Types

- provides vector and matrix types similar to GLSL
- vector types:
  - `vec2`, `vec3`, `vec4`
  - `bvec2`, `bvec3`, `bvec4`
  - `ivec2`, `ivec3`, `ivec4`
  - `uvec2`, `uvec3`, `uvec4`
  - `dvec2`, `dvec3`, `dvec4`
- matrix types:
  - `mat2x2`, `mat2x3`, `mat2x4`, `mat2`, `mat3x2`, `mat3x3`, `mat3x4`, `mat3`, `mat4x2`, `mat4x3`, `mat4x4`, `mat4`
  - `dmat2x2`, `dmat2x3`, `dmat2x4`, `dmat2`, `dmat3x2`, `dmat3x3`, `dmat3x4`, `dmat3`, `dmat4x2`, `dmat4x3`, `dmat4x4`, `dmat4`

- provides GLSL functions (e.g., `inverse` and `transpose`)
- provides functions that offer functionality similar to legacy OpenGL/GLU functions (e.g., `rotate`, `scale`, `translate`, `frustum`, `ortho`, `lookAt`, `perspective`, `pickMatrix`, `project`, and `unProject`)

Code Example: Basic Usage

```
#include <iostream>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/string_cast.hpp>
#include <cmath>

int main() {
    glm::mat4 mv(1.0f);
    mv = mv * glm::lookAt(glm::vec3(0.0f, 0.0f, 0.0f),
      glm::vec3(1.0f, 0.0f, 0.0f),
      glm::vec3(0.0f, 0.0f, 1.0f));
    mv = mv * glm::translate(mv, glm::vec3(1.0f, 1.0f, 1.0f));
    mv = mv * glm::rotate(mv, glm::radians(90.0f),
      glm::vec3(0.0f, 0.0f, 1.0f));
    mv = mv * glm::scale(mv, glm::vec3(1.0f, 1.0f, 2.0f));
    glm::mat4 p = glm::perspective(glm::radians(90.0f), 1.0f,
      1.0f, 2.0f);
    glm::mat4 mvp = p * mv;
    glm::vec4 v(1.0f, -1.0f, -1.0f, 1.0f);
    std::cout << glm::to_string(glm::vec3(mv * v)) << '\n';
    std::cout << glm::to_string(glm::vec3(mvp * v)) << '\n';
    std::cout << glm::radians(180.0f) << '\n';
    std::cout << glm::degrees(M_PI) << '\n';
}
```

```
#include <GL/glew.h>
#include <GL/gl.h>
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>

void setUniform(GLint loc) {
    glm::mat4 m(1.0f);
    // ...
    glUniform4fv(loc, 4, glm::value_ptr(m));
}
```

Section 5.6

Open Graphics Library (OpenGL)

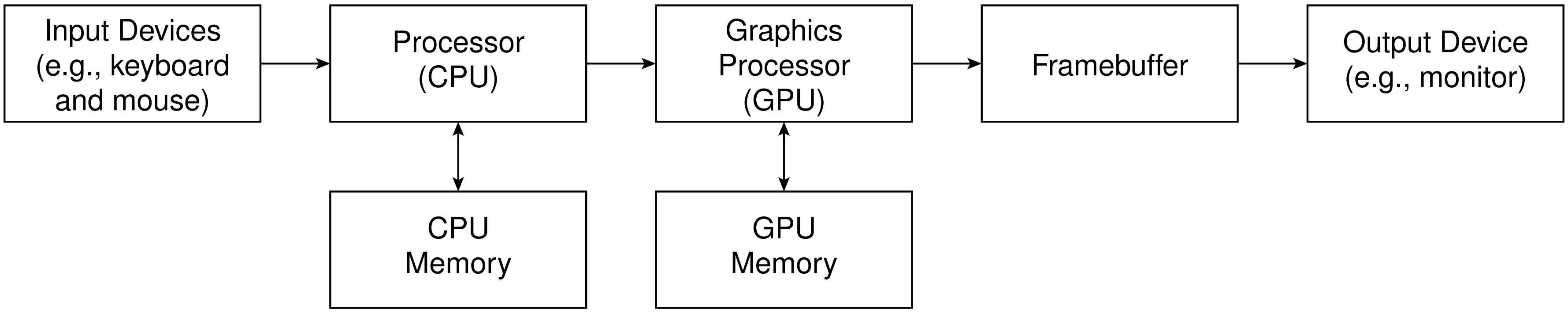
Open Graphics Library (OpenGL)

- application programming interface (API) for high-performance high-quality 2-D and 3-D graphics rendering
- most widely adopted 2-D and 3-D graphics API in industry
- bindings for numerous programming languages (i.e., C, Java, and Fortran)
- focus exclusively on C language binding herein
- window-system and operating-system independent
- available on all mainstream systems (e.g., Microsoft Windows, OS X, and Linux/Unix)
- vendor-neutral, controlled by independent consortium with many organizations as members (including companies such as Intel, NVIDIA, and AMD)
- official web site: `http://www.opengl.org`
- OpenGL ES provides (simplified) subset of OpenGL API for embedded systems (e.g., mobile phones, game consoles, personal navigation devices, personal media players, automotive systems, settop boxes)

OpenGL Functionality

- geometric primitives include points, line segments, and triangles
- arrange geometric primitives in 3-D space and select desired vantage point for viewing composed scene
- calculate colors of objects (e.g., by explicit assignment, lighting, texture mapping, or combination thereof)
- convert mathematical description of objects to pixels on screen (i.e., rasterization)
- can eliminate hidden parts of objects (via depth buffering), perform antialiasing, and so on
- some functionality relies on shaders provided by application program
- only concerned with rendering
- no mechanism provided for creating windows or obtaining user input (e.g., via mouse or keyboard)
- another library must be used in conjunction with OpenGL in order to manage windows and handle user input

Modern OpenGL



- main responsibility of application is to provide graphics data to GPU
- application program running on CPU sends graphics data to GPU
- programs running on GPU called **shaders** control rendering
- GPU performs all rendering
- high performance achieved by offloading rendering work to GPU, with GPU being highly specialized for rendering
- image formed and stored in framebuffer
- shaders written in OpenGL Shading Language (GLSL)
- application program uses OpenGL to compile and link shader source code to yield executable shader program that runs on GPU

OpenGL State Machine

- OpenGL is state machine
- OpenGL functions can be roughly classified into two categories:
  1. primitive generation
  2. state management
- primitive-generation functions:
  - produce graphics output if primitive is visible
  - how vertices are processed and appearance of primitive controlled by OpenGL state
- state-management functions:
  - enabling/disabling OpenGL functionality (e.g., depth buffering)
  - configuring shader programs
  - setting/querying shader variables

Contexts and Profiles

- feature that may be removed in future version of OpenGL is said to be **deprecated**
- **profile** defines subset of OpenGL functionality targeted to specific application domains
- two profiles: core and compatibility
- **core profile** provides functionality mandated by particular version of OpenGL (which does not include deprecated and removed features)
- **compatibility profile** restores support for all functionality that has been removed from OpenGL
- all OpenGL implementations must support core profile, but are not required to support compatibility profile
- for given profile, two types of contexts: full or forward compatible
- **forward compatible context** does not support deprecated features from profile
- **full context** supports deprecated features from profile

Header Files

- header files for OpenGL located in `GL` (or `OpenGL`) directory
- definitions necessary for OpenGL can be found in header file `gl.h`
- above header file provides definitions of all constants and data types (e.g., `GLint` and `GLfloat`) and function declarations for OpenGL
- on some platforms, in order to access newer OpenGL functionality, may need to include `glew.h` (typically in `GL` directory) before `gl.h`
- normally, OpenGL used in conjunction with another helper library such as GLFW or GLUT
- other helper libraries also have header files of their own that must be included
- often header files for helper libraries include `gl.h`

Types

Type	Description
`GLboolean`	boolean
`GLbyte`	8-bit signed two's complement integer
`GLubyte`	8-bit unsigned integer
`GLchar`	8-bit character
`GLshort`	16-bit signed two's complement integer
`GLushort`	16-bit unsigned integer
`GLint`	32-bit signed two's complement integer
`GLuint`	32-bit unsigned integer
`GLfloat`	single-precision floating-point value
`GLdouble`	double-precision floating-point value

- OpenGL types do not necessarily correspond to similarly named C types (e.g., `GLint` is not necessarily **`int`**)

Function Naming Conventions

- all OpenGL functions begin with `gl`
- some OpenGL commands have numerous variants that differ in number and type of parameters
- such commands are named using following pattern:

  $$generic\_name\ N\ T\ V$$

  where $generic\_name$ is generic name of function, N is digit (i.e., 2, 3, 4) indicating number of components, T is one or two letters indicating data type of components, V is either nothing or letter v to indicate component data specified as individual values or as vector (i.e., pointer to array), respectively

Number N

2	(x,y)
3	(x,y,z)
4	(x,y,z,w)

Data Type T

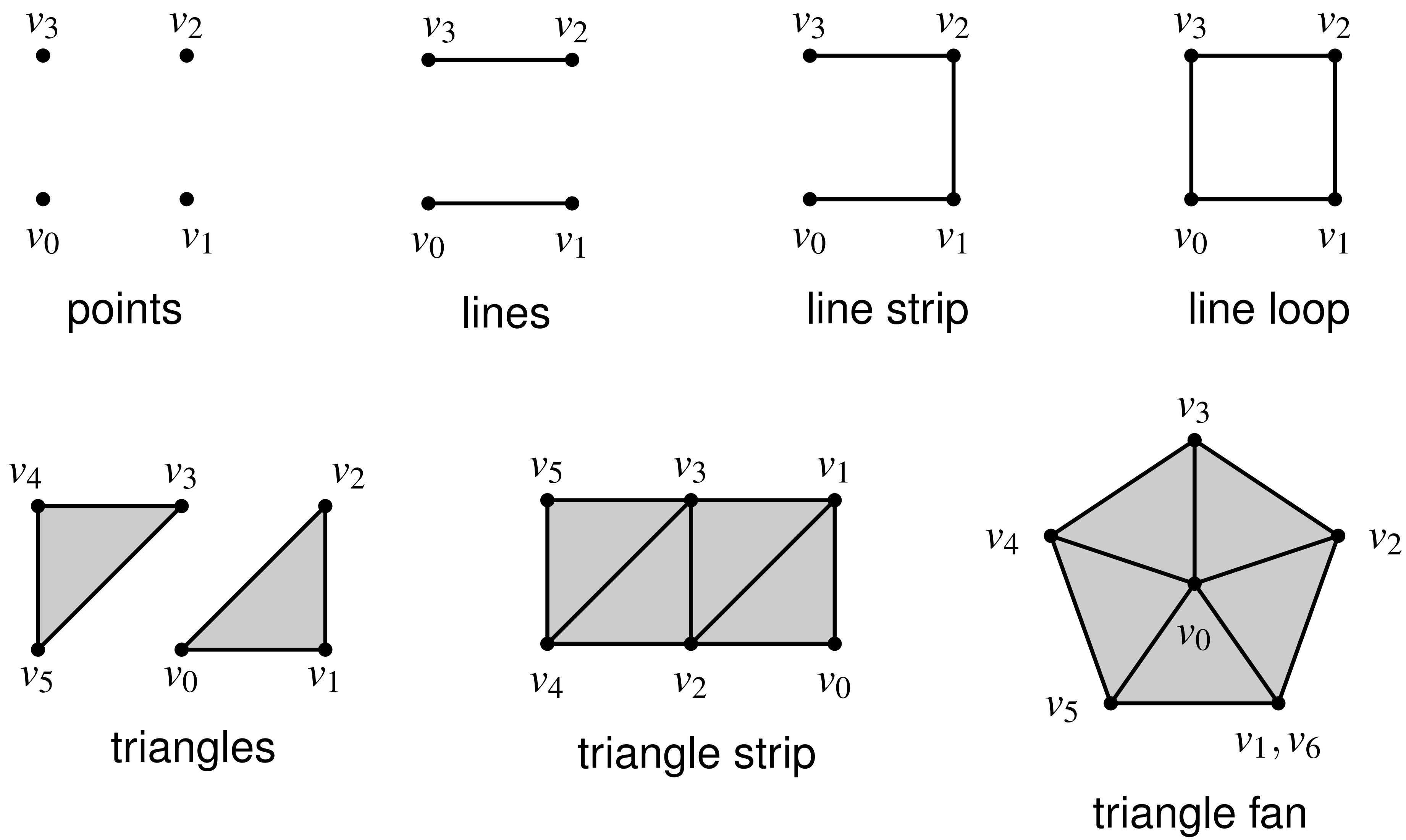
`b`	GLbyte	`ub`	GLubyte
`s`	GLshort	`us`	GLushort
`i`	GLint	`ui`	GLuint
`f`	GLfloat	`d`	GLdouble

- `glUniform3f`: specific version of generic `glUniform` function that takes data in form of three `GLfloat` parameters
- `glUniform3fv`: specific version of generic `glUniform` function that takes data in form of pointer to array of triples of `GLfloat` values

Representing Geometric Objects

- geometric objects represented using vertices
- each vertex has variety of attributes, such as:
  - positional coordinates
  - color
  - texture coordinates
  - surface normal
  - any other data associated with point in space
- position represented using homogeneous coordinates
- vertex data must be stored in vertex buffer objects (VBOs)
- VBOs must be associated with vertex array objects (VAOs)
- VAOs/VBOs allow application program to transfer data to GPU ***once*** and then select between different data on GPU by activating different VAOs/VBOs

Geometric Primitives

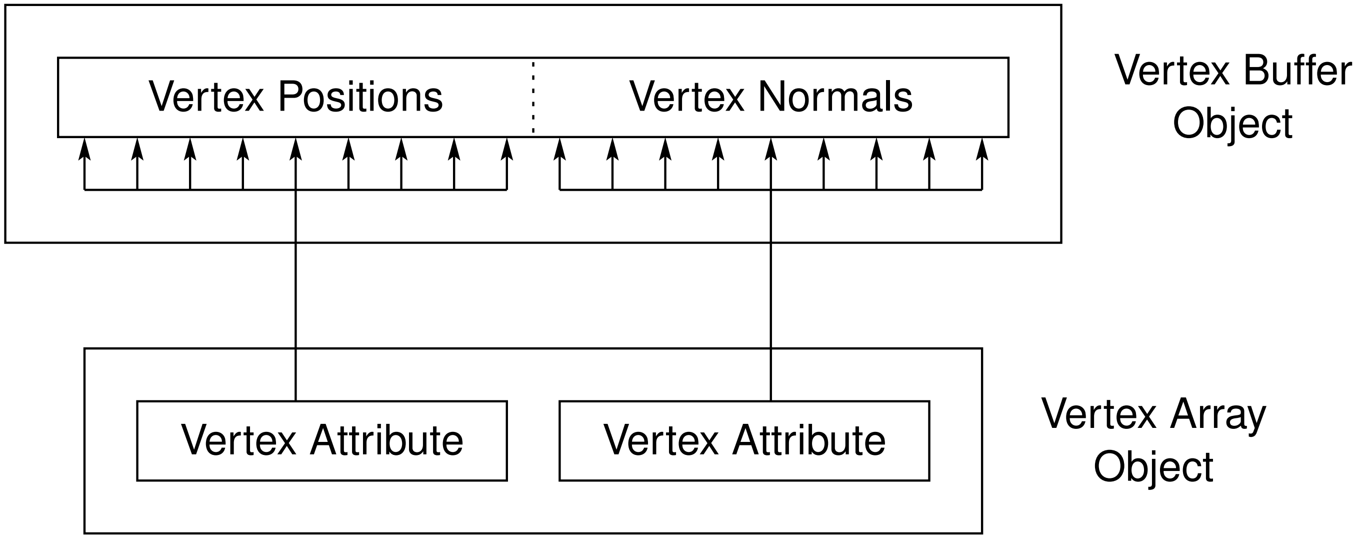


- all primitives specified by vertices

Provoking Vertex

- each primitive has provoking vertex
- one of two conventions can be used to determine provoking vertex: first vertex or last vertex
- for example, with last vertex convention, provoking vertex for triangle is third (i.e., last) vertex of triangle
- convention defaults to last vertex
- convention can be set with `glProvokingVertex`
- provoking vertex becomes important, for example, when using flat interpolation

Vertex Array and Vertex Buffer Objects (VAOs and VBOs)



- vertex buffer objects (VBOs) store vertex attributes (e.g., positions, normals, colors, and texture coordinates)
- storage for VBOs resides in GPU memory
- vertex array objects (VAOs) allow data stored in VBOs to be associated with vertex attributes for vertex shader
- VAOs specify layout (e.g., offset and stride) and format (e.g., type) of data in VBOs
- to render primitives need VAO (which, in turn, is associated with one or more VBOs)

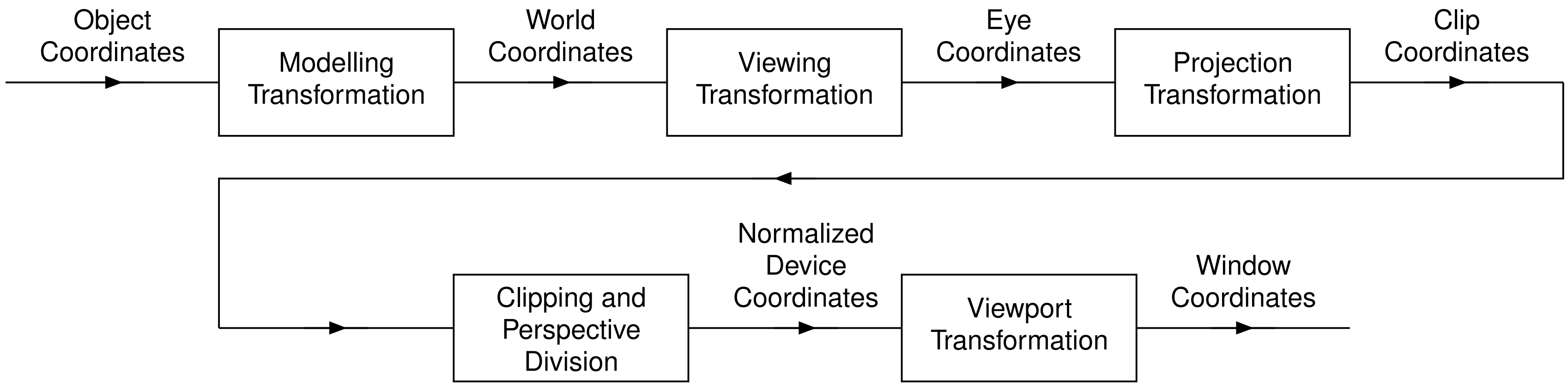
Vertex Array Objects (VAOs)

- VAOs store data for geometric object
- VAO identified by name, which is integer of type `GLuint`
- create one or more VAOs by generating VAO names via `glGenVertexArrays`
- VAO initialized as follows:
  1. bind specific VAO for initialization via `glBindVertexArray`
  2. update VBOs associated with VAO, and specify layout and format of VBO data and its correspondence with vertex attributes for rendering via `glVertexAttribPointer`
- data in VAO rendered as follows:
  1. bind VAO for use in rendering via `glBindVertexArray`
  2. draw content of currently enabled arrays via `glDrawArrays`
- only enabled attributes will be used for rendering (where attributes are enabled with `glEnableVertexAttribArray`)

Vertex Buffer Objects (VBOs)

- vertex buffer objects (VBOs) provide means to transfer data to GPU memory
- vertex data must be stored in VBO associated with VAO
- each VBO associated with name, which is integer of type GLuint
- generate VBO names via `glGenBuffers`
- bind specific VBO for initialization via `glBindBuffer` (after first binding associated VAO)
- allocate underlying storage for VBO (and optionally load data into VBO) via `glBufferData`
- load data into VBO via `glBufferSubData`

Coordinate Systems

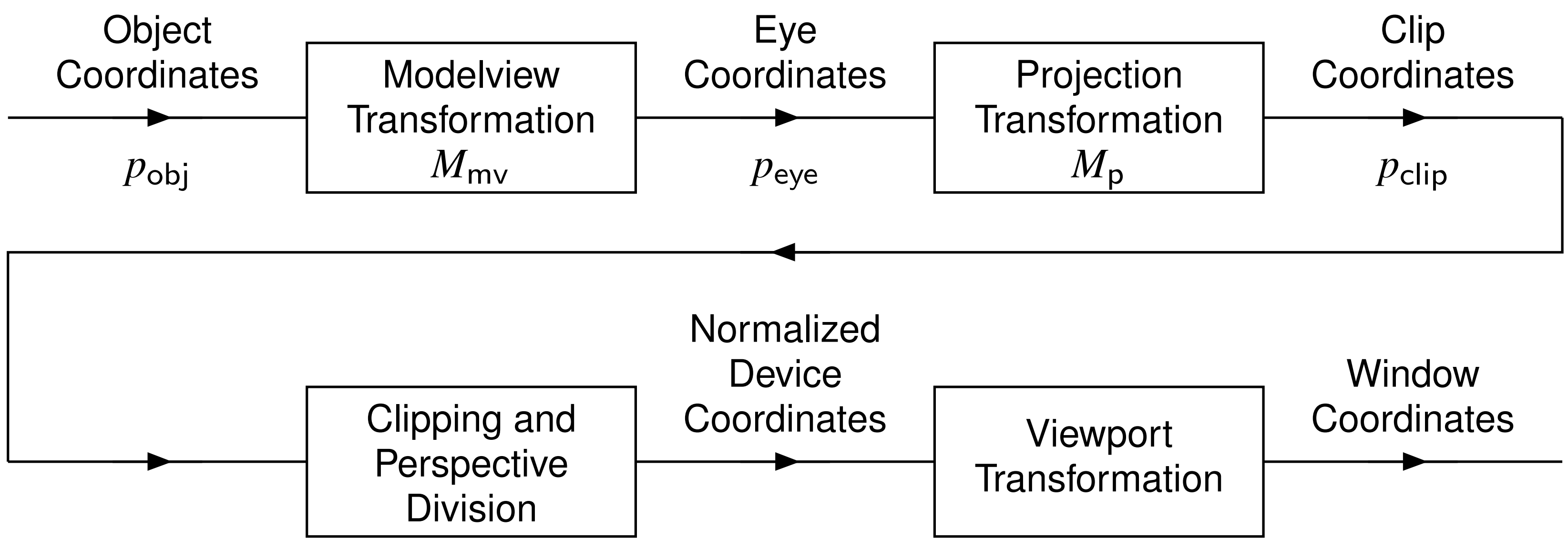


- **object coordinates**: coordinates of object relative to its local origin
- **world coordinates**: coordinates of three-dimensional environment (i.e., world) being rendered
- **eye coordinates**: coordinates relative to camera from which world is being viewed
- **clip coordinates**: coordinates normalized such that viewing volume falls in $[-1,1]\times[-1,1]\times[-1,1]$
- **normalized device coordinates**: result of converting clip coordinates to Cartesian coordinates by perspective division (i.e., dividing by w coordinate)
- **window coordinates**: coordinates relative to graphics window

OpenGL Camera

- appearance of rendered scene determined by camera position, orientation, and viewing volume
- camera positioned at origin
- camera oriented to point in negative z direction with positive y axis pointing up
- orthographic projection in direction of z axis with clipping planes $x = -1$, $x = 1$, $y = -1$, $y = 1$, $z = -1$, and $z = 1$
- viewing volume is $[-1,1] \times [-1,1] \times [-1,1]$ (i.e., cube centered at origin with sides of length 2)
- different camera position, orientation, and viewing volume can be achieved by employing transformations
- perspective projection accomplished by applying transformation that warps viewing volume into frustum

Transformations

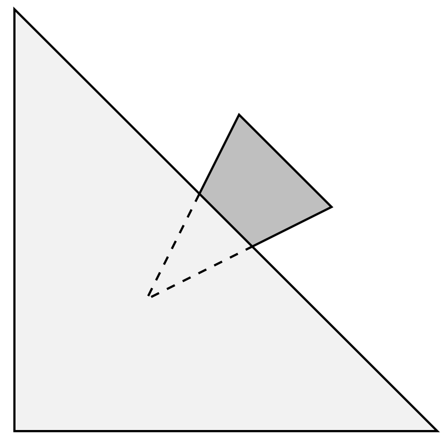


- often modelling and viewing transformations combined into single transformation called modelview transformation
- $p_{\text{eye}} = M_{\text{mv}} p_{\text{obj}}$
- $p_{\text{clip}} = M_{\text{p}} p_{\text{eye}} = M_{\text{p}} M_{\text{mv}} p_{\text{obj}}$
- clip coordinates and normalized device coordinates still retain depth (i.e., z) information in order to facilitate depth buffering

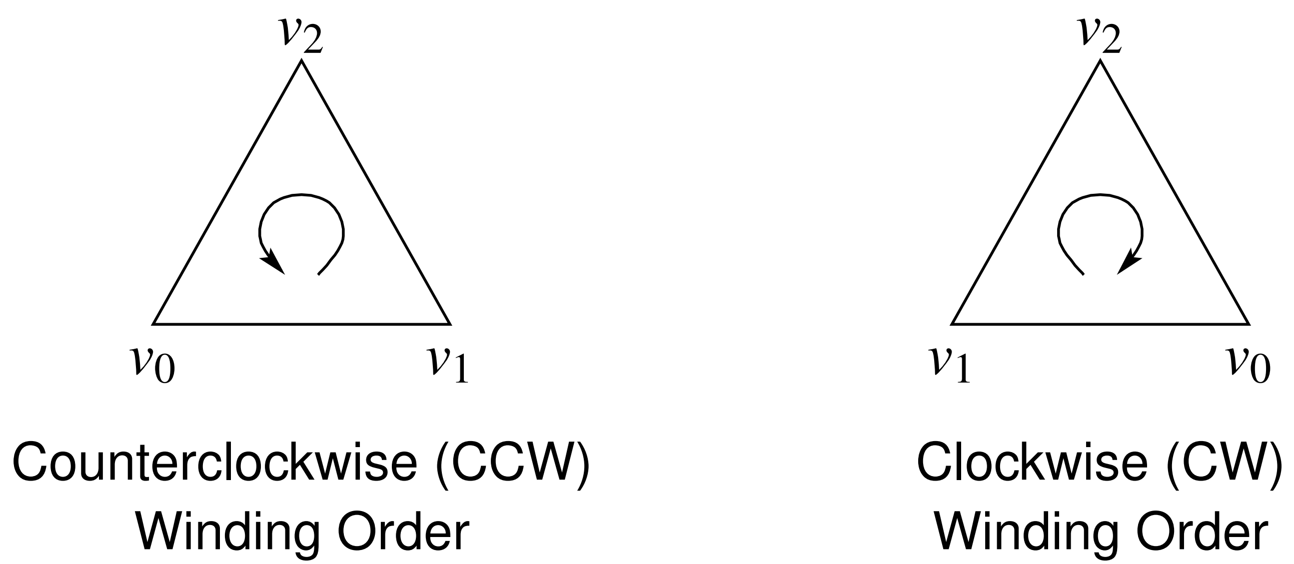
Transformations (Continued)

- viewport transformation determines drawable region within window
- viewport transformation set via `glViewport`

Depth Buffering



- in above figure, darker triangle is partially occluded by lighter triangle from vantage point of camera
- in OpenGL, camera always pointing in direction of negative z axis
- therefore, z coordinate can be used to determine distance of fragment from eye, with lesser value (i.e., closer to $-\infty$) corresponding to greater distance
- if depth buffering enabled, fragment not drawn if its z coordinate less than z coordinate of previously drawn pixel



- winding order used to distinguish front and back sides of triangles
- which winding order corresponds to front side of triangle specified via `glFrontFace`
- which side (or sides) of triangle should be culled specified via `glCullFace`
- if face culling enabled, culled side of triangles not rendered

State Management

- `glEnable` and `glDisable` used to enable and disable specific functionality

Value	Meaning
`GL_CULL_FACE`	if enabled, cull polygons based on their winding in window coordinates (e.g., do not render backs of faces)
`GL_DEPTH_TEST`	if enabled, do depth comparisons and update depth buffer
`GL_LINE_SMOOTH`	if enabled, draw lines with antialiasing

Other Functions

Function	Description
`glClear`	clear buffer to preset values
`glClearColor`	specify clear values for color buffers

Program Structure

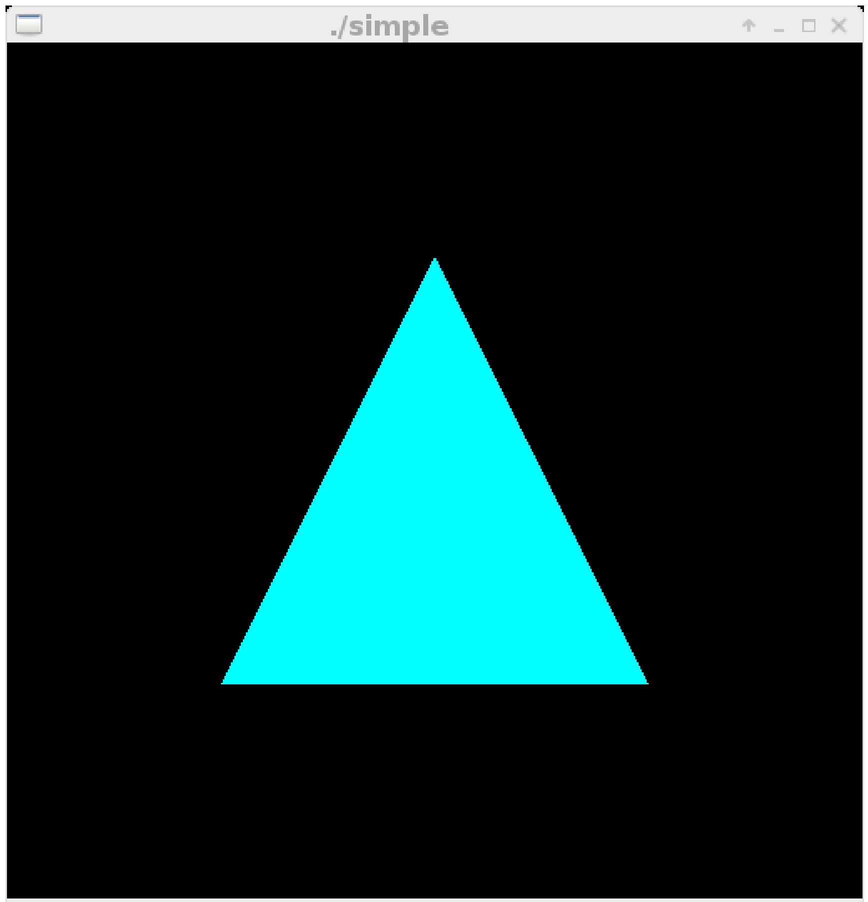
- program typically consists of steps like following:
  1. create window associated with OpenGL context
  2. initialize shaders (e.g., compile and link) and other OpenGL state (e.g., depth buffering and clear color)
  3. initialize data to be drawn
  4. register callback functions to process events
  5. enter main event-processing loop, which repeatedly waits for event of interest and then handles it by invoking appropriate callback function
- events of interest typically include such things as:
  - request to redraw window
  - window-resize notification
  - keyboard input
  - mouse-button press/release

Section 5.6.1

Simple OpenGL Program

OpenGL Application Program Example

- consider very simple OpenGL application program (which utilizes GLFW)
- draws triangle in window
- rendered output shown below



Header Files

```
#include <cstdlib>
#include <string>
#include <GL/glew.h>
#include <GLFW/glfw3.h>
```

Main Function

```
GLuint vao = 0;

void fatalError() {
    glfwTerminate();
    std::exit(EXIT_FAILURE);
}

int main(int argc, char** argv) {
    if (!glfwInit()) {return EXIT_FAILURE;}
    GLFWwindow* window = makeWindow(512, 512, argv[0]);
    if (!window) {fatalError();}
    glfwMakeContextCurrent(window);
    glewExperimental = GL_TRUE;
    if (glewInit() != GLEW_OK) {fatalError();}
    GLuint program = makeProgram(vShaderSource,
      fShaderSource);
    if (!program) {fatalError();}
    glUseProgram(program);
    glClearColor(0.0, 0.0, 0.0, 0.0);
    GLuint vbo;
    makeVao(program, vao, vbo);
    glfwSetWindowRefreshCallback(window, refresh);
    while (!glfwWindowShouldClose(window))
      {glfwWaitEvents();}
    glfwTerminate();
    return EXIT_SUCCESS;
}
```

```
GLFWwindow* makeWindow(int width, int height,
  const std::string& title) {
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
    glfwWindowHint(GLFW_OPENGL_PROFILE,
      GLFW_OPENGL_CORE_PROFILE);
    GLFWwindow* window = glfwCreateWindow(width, height,
      title.c_str(), nullptr, nullptr);
    return window;
}
```

```
const std::basic_string<GLchar> vShaderSource = R"(
#version 330
in vec3 aPosition;
void main() {
    gl_Position = vec4(aPosition, 1.0);
}
)";

const std::basic_string<GLchar> fShaderSource = R"(
#version 330
out vec4 fColor;
void main() {
    fColor = vec4(0.0, 1.0, 1.0, 1.0);
}
)";
```

```
GLuint compileShader(GLuint type,
  const std::basic_string<GLchar>& source) {
    GLuint shader = glCreateShader(type);
    if (!shader) {return 0;}
    const GLchar* cp = &source[0];
    GLint len = source.size();
    glShaderSource(shader, 1, &cp, &len);
    glCompileShader(shader);
    GLint status = GL_FALSE;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
    if (status != GL_TRUE)
      {glDeleteShader(shader); return 0;}
    return shader;
}
```

```
GLuint makeProgram(
  const std::basic_string<GLchar>& vShaderSource,
  const std::basic_string<GLchar>& fShaderSource) {
    GLuint vShader = compileShader(GL_VERTEX_SHADER,
      vShaderSource);
    if (!vShader) {return 0;}
    GLuint fShader = compileShader(GL_FRAGMENT_SHADER,
      fShaderSource);
    if (!fShader) {glDeleteShader(vShader); return 0;}
    GLuint program = glCreateProgram();
    GLint status = GL_FALSE;
    if (program) {
        glAttachShader(program, vShader);
        glAttachShader(program, fShader);
        glLinkProgram(program);
        glGetProgramiv(program, GL_LINK_STATUS, &status);
    }
    glDeleteShader(vShader);
    glDeleteShader(fShader);
    if (!program) {return 0;}
    if (status != GL_TRUE)
      {glDeleteProgram(program); return 0;}
    return program;
}
```

Initialize Vertex Array Object (VAO)

```
void makeVao(GLuint program, GLuint& vao,
  GLuint& vbo) {
    static const GLfloat vertices[][3] = {
        {-0.50, -0.50, 0.0},
        { 0.50, -0.50, 0.0},
        { 0.00, 0.50, 0.0}
    };
    glGenVertexArrays(1, &vao);
    glGenBuffers(1, &vbo);
    glBindVertexArray(vao);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
      vertices, GL_STATIC_DRAW);
    GLuint aPosition = glGetAttribLocation(program,
      "aPosition");
    glVertexAttribPointer(aPosition, 3, GL_FLOAT, GL_FALSE,
      0, 0);
    glEnableVertexAttribArray(aPosition);
}
```

Window Refresh Callback

```
GLuint vao = 0;

void refresh(GLFWwindow* window) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindVertexArray(vao);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glfwSwapBuffers(window);
}
```

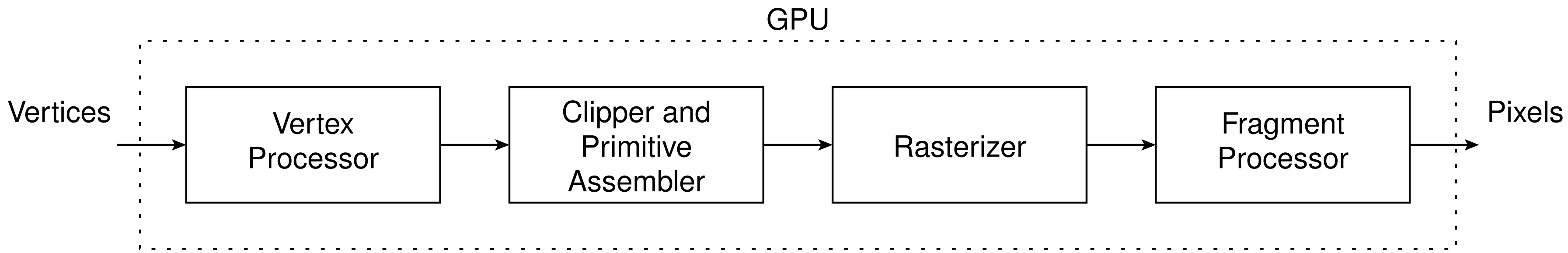
Section 5.6.2

Shaders

Shaders

- shader is user-defined program that runs on GPU and provides functionality associated with some particular stage of rendering pipeline
- shaders written in OpenGL Shading Language (GLSL)
- as of OpenGL 3.1, application program must provide shaders as no default shaders provided (in core profile)
- several types of shaders:
  - vertex shader
  - tessellation control shader
  - tessellation evaluation shader
  - geometry shader
  - fragment shader
  - compute shader
- each type of shader performs specific type of task on GPU

Rendering Pipeline and Shaders



- each type of shader performs distinct task within rendering pipeline
- vertex shader (which is associated with vertex processor block) provides any last geometric transformation of vertices before being fed to remainder of rendering pipeline
- geometry shader (which is associated with vertex processor block) generates actual primitives to be rendered based on primitives received from previous pipeline stage
- fragment shader (which is associated with fragment processor block) provides color to each pixel in framebuffer

OpenGL Shader Language (GLSL)

- shaders written in GLSL
- GLSL is portable multiplatform C-like language
- GLSL borrows heavily from C syntax
- provides simplified subset of C language with numerous modifications:
  - adds new data types, such as matrix and vector types
  - adds overloaded operators and constructors
- supports C and C++ style comments
- GLSL keywords cannot be used as identifiers
- names beginning with “`gl_`” prefix reserved by GLSL

Reserved Keywords

```
attribute        inout       mat4x4
const            float       vec2
uniform          int         vec3
varying          void        vec4
layout           bool        ivec2
centroid         true        ivec3
flat             false       ivec4
smooth           invariant   bvec2
noperspective    discard     bvec3
break            return      bvec4
continue         mat2        uint
do               mat3        uvec2
for              mat4        uvec3
while            mat2x2      uvec4
switch           mat2x3      lowp
case             mat2x4      mediump
default          mat3x2      highp
if               mat3x3      precision
else             mat3x4      sampler1D
in               mat4x2      sampler2D
out              mat4x3      sampler3D
```

Reserved Keywords (Continued)

```
samplerCube             usamplerCube
sampler1DShadow         usampler1DArray
sampler2DShadow         usampler2DArray
samplerCubeShadow       sampler2DRect
sampler1DArray          sampler2DRectShadow
sampler2DArray          isampler2DRect
sampler1DArrayShadow    usampler2DRect
sampler2DArrayShadow    samplerBuffer
isampler1D              isamplerBuffer
isampler2D              usamplerBuffer
isampler3D              sampler2DMS
isamplerCube            isampler2DMS
isampler1DArray         usampler2DMS
isampler2DArray         sampler2DMSArray
usampler1D              isampler2DMSArray
usampler2D              usampler2DMSArray
usampler3D              struct
```

plus other keywords added since OpenGL 3.3

The #version Directive

- `#version` directive specifies which version of GLSL should be used to compile/link shader
- if `#version` directive specified, must be first statement in source
- if no `#version` directive given, version 1.10 is assumed
- `#version` directive takes two parameters (with second being optional):
  1. integer specifying GLSL version (scaled by a factor of 100)
  2. profile name, which can be either `core` or `compatibility` with `core` being default
- for OpenGL 3.3 and above, corresponding GLSL version matches OpenGL version (e.g., OpenGL 4.1 uses GLSL 4.1); for earlier OpenGL versions, relationship between OpenGL and GLSL versions as follows:

OpenGL Version	GLSL Version
2.0	1.10
2.1	1.20
3.0	1.30
3.1	1.40
3.2	1.50

- for example, to specify use of GLSL 3.30 with core profile:

```
#version 330
```

Basic Types

Scalar and Void Types

Type	Description
`void`	dummy type for functions without return value
`bool`	boolean type
`int`	signed integer type
`uint`	unsigned integer type
`float`	single-precision floating-point type

Vector of **`float`** Types

Type	Description
`vec2`	two-component vector of **`float`**
`vec3`	three-component vector of **`float`**
`vec4`	four-component vector of **`float`**

Vector of **`bool`** Types

Type	Description
`bvec2`	two-component vector of **`bool`**
`bvec3`	three-component vector of **`bool`**
`bvec4`	four-component vector of **`bool`**

Vector of **int** Types

Type	Description
ivec2	two-component vector of **int**
ivec3	three-component vector of **int**
ivec4	four-component vector of **int**

Vector of **uint** Types

Type	Description
uvec2	two-component vector of **uint**
uvec3	three-component vector of **uint**
uvec4	four-component vector of **uint**

Matrix of **float** Types

Type	Description	Type	Description
mat2	2×2 matrix of **float**	**mat3x2**	3×2 matrix of **float**
mat3	3×3 matrix of **float**	**mat3x3**	same as **mat3**
mat4	4×4 matrix of **float**	**mat3x4**	3×4 matrix of **float**
mat2x2	same as **mat2**	**mat4x2**	4×2 matrix of **float**
mat2x3	2×3 matrix of **float**	**mat4x3**	4×3 matrix of **float**
mat2x4	2×4 matrix of **float**	**mat4x4**	same as **mat4**

Basic Types (Continued 2)

- numerous sampler types
- numerous other types added since OpenGL 3.3
- matrix types stored in column-major order
- no pointer types
- **`const`** qualifier similar to C
- **`struct`** can be used to construct user-defined types

Operators

- standard C/C++ arithmetic and logical operators
- operators overloaded for matrix and vector types
- for two operands of vector type, multiplication operator performs component-wise multiplication
- for two operands of matrix type or one operand of matrix type and one of vector type, multiplication operator performs standard matrix/vector multiplication
- example:

```
mat4 a; mat4 b; mat4 c;
vec4 u; vec4 v; vec4 w;
// ...
v = a * u; // standard matrix-vector multiplication
c = a * b; // standard matrix-vector multiplication
w = u * v; // component-wise multiplication
```

Operators (Continued 1)

- first, second, third, and fourth components of vector (if they exist) can be selected by:
  - subscripting operator with subscripts 0, 1, 2, and 3, respectively; or
  - selection operator with `x`, `y`, `z`, and `w`, respectively; or
  - selection operator with `r`, `g`, `b`, and `a`, respectively; or
  - selection operator with `s`, `t`, `p`, and `q`, respectively
- example:

```
vec3 v;
// ...
float x = v.x;
float y = v.y;
float z = v.z;
```

- components of matrices can be accessed by subscripting operator
- single subscripting on matrix results in column of matrix
- double subscripting on matrix results in element of matrix
- example:

```
mat2 a;
// ...
vec2 v = a[0];
float f = a[0][0];
```

Operators (Continued 2)

- can also form vectors by selecting multiple elements from vector (e.g., swizzling and smearing)
- example:

```
vec4 v; vec4 u;
vec3 a;
// ...
u = v.wzyx; // vec4(v.w, v.z, v.y, v.x)
u = v.xxyy; // vec4(v.x, v.x, v.y, v.y)
a = v.xyz; // vec3(v.x, v.y, v.z)
u = a.xxxx; // vec4(a.x, a.x, a.x, a.x)
```

Control Flow

- selection statements
  - **`if`**
  - **`if-else`**
  - ternary operator
  - **`switch`**
- looping statements
  - **`for`**
  - **`while`**
  - **`do-while`**
- also has **`break`** and **`continue`**
- no `goto` statement
- only in fragment shader: **`discard`** statement

- numerous built-in functions provided (e.g., `abs`, `sin`, `cos`, `sqrt`)
- user-defined functions are supported
- recursion not allowed
- function overloading supported (including for user-defined functions)
- **`return`** statement to return from function

Constructors

- constructor is function with same name as type
- used to create value of type named by function
- constructor parameters for matrix types specified in column-major order
- example:

```
vec3 v3 = vec3(1.0, 2.0, 3.0);
mat2 m2 = mat2(1.0, 2.0, 3.0, 4.0);
  // first column of m2 is 1.0, 2.0
  // second column of m2 is 3.0, 4.0
mat4 m4 = mat4(1.0); // identity matrix
vec4 v4 = vec4(0.0); // zero vector
const int lut[3] = int[3](1, 2, 4);
vec2 va[2] =
  vec2[](vec2(1.0, 2.0), vec2(3.0, 4.0));
bool b = bool(1);
```

Conversions

- number of implicit conversions allowed, some of which identified below
- integer types (e.g., **`int`** and **`uint`**) can be implicitly converted to **`float`**
- each integer vector type (e.g., **`ivec4`**) can be implicitly converted to floating-point vector type of same dimension (e.g., **`vec4`**)
- floating-point type ***cannot*** be implicitly converted to integer type
- unsigned integer type (e.g., **`uint`**) ***cannot*** be implicitly converted to signed integer type (e.g., **`int`**)
- example:

```
int i; uint ui; float f; vec4 v4; ivec4 iv4;
// ...
f = i; // OK
// i = f; // ERROR: no implicit conversion
i = int(f); // OK
// iv4 = v4; // ERROR: no implicit conversion
iv4 = ivec4(v4); // OK
// i = 0u; // ERROR: no implicit conversion
// i = ui; // ERROR: no implicit conversion
i = int(ui); // OK
```

Angle and Trigonometric Functions

Function	Description
`radians`	convert from degrees to radians
`degrees`	convert from radians to degrees
`sin`	sine function
`cos`	cosine function
`tan`	tangent function
`asin`	arcsine function
`acos`	arccosine function
`atan`	arctangent function

Built-In Functions (Continued 1)

Exponential Functions

Function	Description
`pow`	exponentiation function
`exp`	base-e exponentiation function
`log`	natural logarithm function
`exp2`	base-2 exponentiation function
`log2`	base-2 logarithm function
`sqrt`	square-root function
`inversesqrt`	reciprocal of square-root function

Built-In Functions (Continued 2)

Common Functions

Function	Description
`abs`	absolute-value function
`sign`	signum function
`floor`	floor function
`ceil`	ceiling function
`fract`	fractional-part function
`mod`	modulo function
`min`	minimum of two values
`max`	maximum of two values
`clamp`	clamp value to specified range
`mix`	affine combination of two values
`step`	step function
`smoothstep`	smooth step function

Geometric Functions

Function	Description
`length`	length of vector
`distance`	distance between two points
`dot`	dot product
`cross`	cross product
`normalize`	get vector of unit length
`faceforward`	get vector that points in same direction as reference vector
`reflect`	get vector that points in direction of reflection
`refract`	get vector that points in direction of refraction

Built-In Functions (Continued 4)

Fragment Processing Functions

Function	Description
`dFdx`	partial derivative of argument with respect to x
`dFdy`	partial derivative of argument with respect to y
`fwidth`	sum of absolute value of derivatives in x and y

Matrix Functions

Function	Description
`matrixCompMult`	multiply matrices component-wise

Texture Lookup

Function	Description
`texture2D`	perform 2D texture lookup
`textureCube`	perform cubemap texture lookup

Vector Relational Functions

Function	Description
`lessThan`	component-wise less-than comparison
`lessThanEqual`	component-wise less-than-or-equal comparison
`greaterThan`	component-wise greater-than comparison
`greaterThanEqual`	component-wise greater-than-or-equal comparison
`equal`	component-wise equality comparison
`notEqual`	component-wise inequality comparison
`any`	any component is true
`all`	all components are true
`not`	component-wise logical complement operation

- shader parameters (i.e., input and output variables of shaders) and function parameters can be qualified with **in** and **out** qualifiers
- parameter declared **in**:
  - value given to parameter will be copied into parameter when function called
  - function may then modify parameter but changes will not affect caller
  - essentially pass-by-value semantics
- parameter declared **out**:
  - parameter will not have its value initialized by caller so initial value of parameter at start of function is undefined
  - function must modify parameter
  - after function's execution is complete, value of parameter will be copied into variable that user specified when calling function
- default qualifier is **in**
- example:

```
float foo(float x, int i, out int n);
float calculate(in float x, in float y, in int n);
```

- example:

```
void calc(float x, int i, out float y, out int j) {
    // at this point, y and j are undefined
    y = ++x;
    j = ++i;
}

void func() {
    float a = 0.0;
    int b = 0;
    float c = 0.0;
    int d = 0;
    calc(a, b, c, d);
    // a and b are unchanged by function call
    // c is 1.0, d is 1
}
```

The **uniform** Qualifier

- global variables and interface blocks can be declared with **`uniform`** qualifier
- **`uniform`** qualifier indicates that value of variable does not change across multiple shader invocations during rendering of single primitive (i.e., during `glDraw*` call)
- uniform variables form linkage between shader and application program
- used to declare variables shared between shader and application program (e.g., projection matrix, light source position, material color)
- uniform variable cannot be modified in shader
- uniform variable can only be modified by application program
- uniform variable can be used in multiple shaders (e.g., vertex and fragment shaders)
- if used in multiple shaders, must have identical declaration in each
- example:

```
uniform mat4 projectionMatrix;
uniform mat4 modelViewMatrix;
```

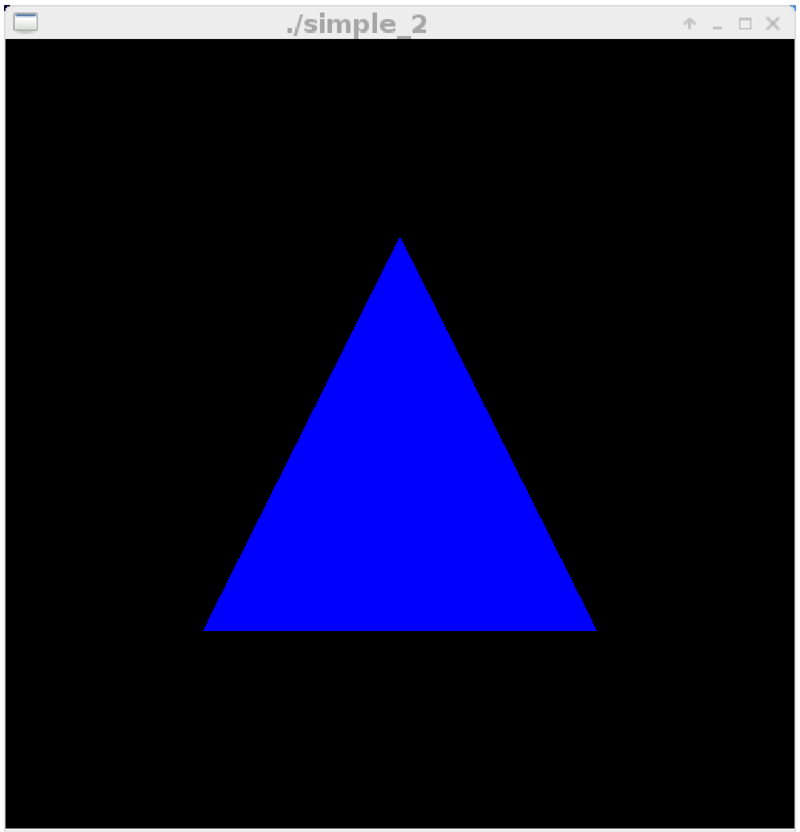
Interpolation Qualifiers

- outputs from and inputs to shader can be qualified with interpolation qualifier
- interpolation qualifier controls how value of particular variable is interpolated
- interpolation qualifiers: **`smooth`**, **`noperspective`**, **`flat`**
- **`smooth`** qualifier: perspective-correct interpolation is performed
- **`noperspective`** qualifier: linear interpolation is performed
- **`flat`** qualifier: no interpolation is performed (i.e., value taken from provoking vertex of primitive)
- default qualifier is **`smooth`**
- example:

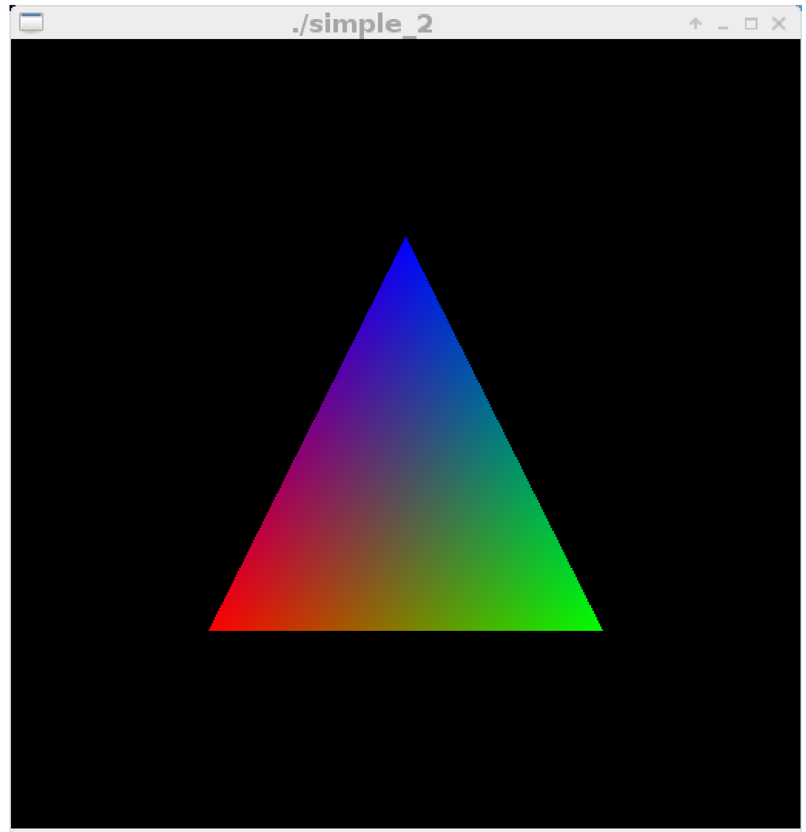
```
flat out vec4 color;
```

Interpolation Example

- single triangle rendered with vertices having color attributes of red, green, and blue, with provoking vertex being last vertex



Without Interpolation (I.e., Flat)

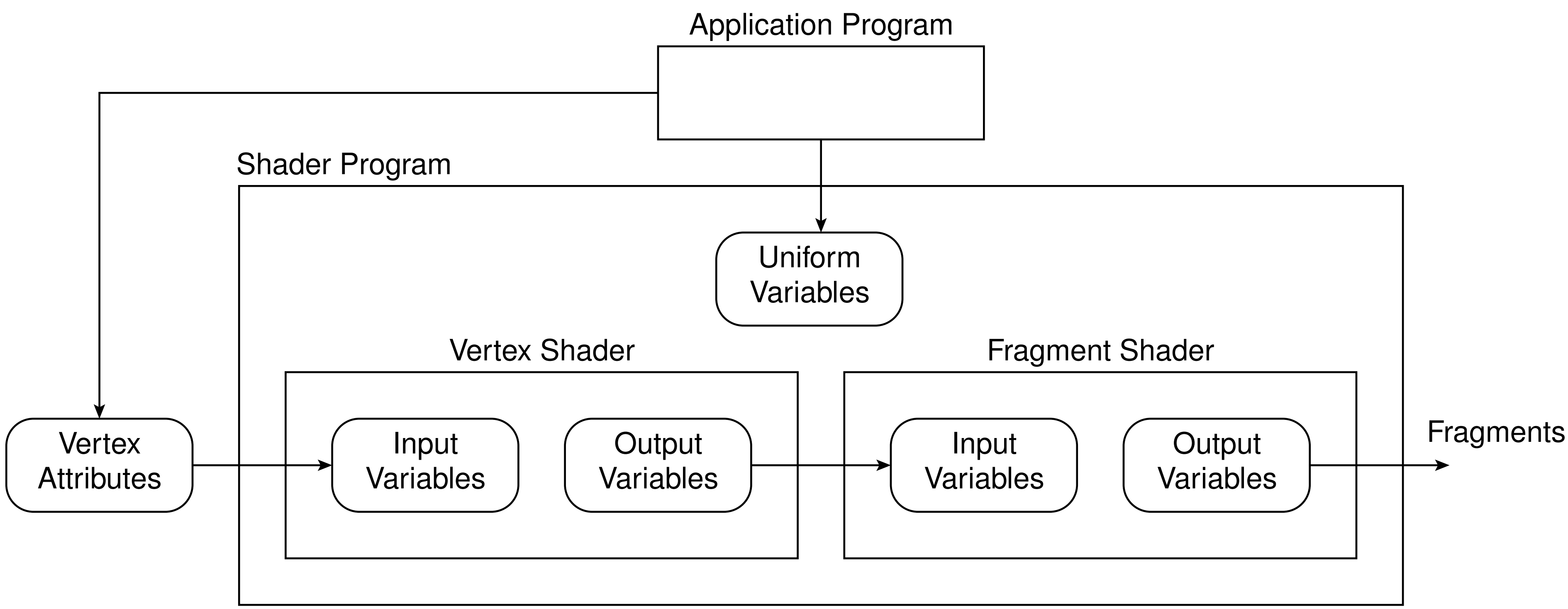


With Interpolation (E.g., Smooth)

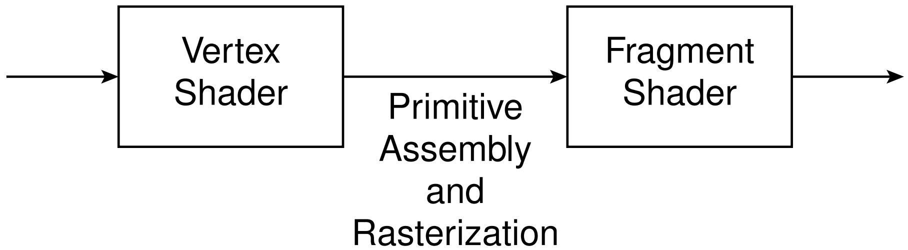
Layout Qualifiers

- layout qualifiers used to specify how storage for variable allocated amongst other things
- layout qualifiers (e.g., `location`) provided by using **`layout`** keyword
- `location` layout qualifier can be used to specify location associated with variable
- vertex shaders allow input layout qualifiers on input variable declarations
- example: following will establish vertex shader input `vPosition` to be copied in from location number 1:
  ```
  layout(location = 1) in vec4 vPosition;
  ```
- example: following will establish vertex shader input `colors` copied in from location numbers 6, 7, and 8:
  ```
  layout(location = 6) in vec4 colors[3];
  ```

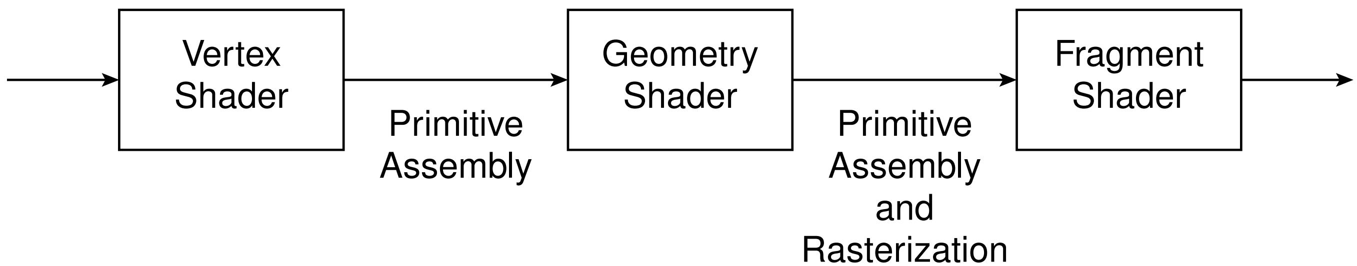
Configuration with Vertex and Fragment Shaders



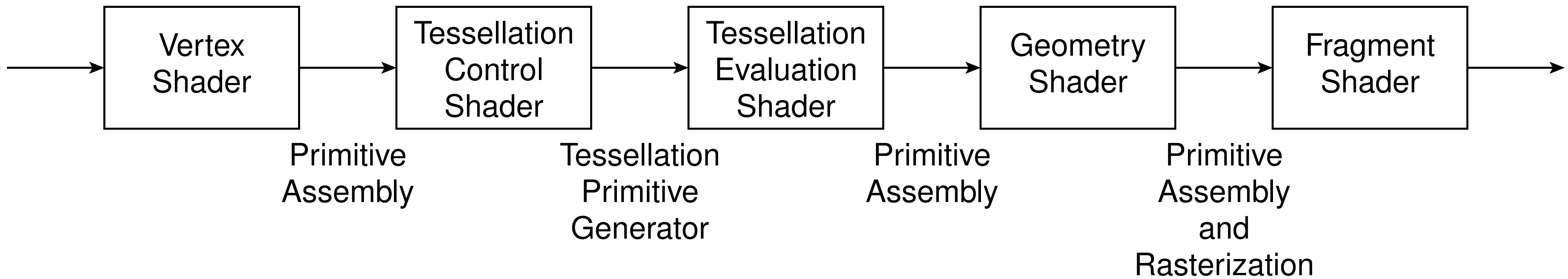
Various Configurations of Shaders



Vertex and Fragment Shaders



Vertex, Geometry, and Fragment Shaders



Vertex, Tessellation, Geometry, and Fragment Shaders

Vertex Shaders

- vertex shader is programmable shader stage in rendering pipeline that handles processing of individual vertices
- vertex shader provided with vertex attribute data (e.g., position, normal, color, and texture coordinates) from VAO from drawing command
- for each vertex in input vertex stream, produces one vertex for output vertex stream
- must be one to one correspondence between input vertices and output vertices
- processes each vertex independently
- some uses of vertex shaders include:
  - vertex position transformation using modelview and projection matrices
  - normal transformation and (if needed) normalization
  - texture coordinate generation and transformation
  - per-vertex lighting
  - color computation

Vertex Shader Inputs and Outputs

- built-in input variables:
  - **int** `gl_VertexID`: index of vertex currently being processed
  - **int** `gl_InstanceID`: index of current instance when doing some form of instanced rendering
- other inputs associated with vertex attributes from VAO/VBO
- built-in output variables:
  - **vec4** `gl_Position`: clip-space output position of current vertex
  - **float** `gl_PointSize`: pixel width/height of point being rasterized; only has meaning for point primitives
  - **float** `gl_ClipDistance[]`: distance from vertex to each user-defined clipping half-space
- vertex shader must set `gl_Position`

Vertex Shader Example

```
// use version 3.30 of GLSL (core profile)
#version 330

// input attribute variable for vertex position
in vec4 aPosition;

// uniform variable for modelview-projection matrix
uniform mat4 uModelViewProjMatrix;

void main() {
    // set output position for vertex
    gl_Position = uModelViewProjMatrix * aPosition;
}
```

Fragment Shaders

- fragment shader is programmable shader stage that processes fragment generated by rasterization into set of colors and single depth value
- for each sample of pixels covered by primitive, fragment is generated
- each fragment has window space position, some other values, and all of interpolated per-vertex output values from last vertex processing stage
- takes single fragment as input and produces single fragment as output
- some uses of fragment shaders include:
  - per-fragment lighting
  - computing colors and texture coordinates per fragment
  - texture application (texture and bump mapping)
  - environment mapping
  - fog computation

Fragment Shader Inputs and Outputs

- built-in input variables:
  - **`vec4`** `gl_FragCoord`: location of fragment in window space
  - **`bool`** `gl_FrontFacing`: indicates if fragment was generated by front face of primitive (only triangles can have back face)
  - **`int vec2`** `gl_PointCoord`: location within point primitive that defines position of fragment relative to side of point
- other input variables correspond to outputs of previous shader stage
- built-in output variables:
  - **`float`** `gl_fragDepth`: depth of fragment which defaults to `gl_FragCoord.z`
- ***`vec4` output variable for fragment color***

Fragment Shader Example

```
// use version 3.30 of GLSL (core profile)
#version 330

// output variable for color
out vec4 fColor;

void main() {
    // set output color to white
    fColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Geometry Shaders

- controls processing of primitives between vertex shader (or optional tessellation stage) and fixed-function vertex post-processing stage
- use of geometry shader optional
- takes single primitive as input and outputs zero or more primitives
- some uses of geometry shaders include:
  - layered rendering
  - transform feedback

Geometry Shader Inputs

- one input primitive per geometry shader invocation
- type of input primitives specified by layout qualifier, which is one of: `points`, `lines`, `lines_adjacency`, `triangles`, `triangles_adjacency`
- number of input vertices determined by input primitive type (e.g., three for triangles)
- per-vertex inputs available as members of elements in array `gl_in`:
  - `vec4 gl_Position`: vertex position
  - `float gl_PointSize`: pixel width/height of point being rasterized; only used for point primitive
  - `float gl_ClipDistance[]`: distance to clipping planes
- `gl_in` contains N elements (with indices starting from 0), where N is number of vertices in input primitive
- each shader input produced by previous pipeline stage is always array with one element per vertex
- per-primitive inputs:
  - `gl_PrimitiveIDIn`: current input primitive's ID
  - `gl_InvocationID`: current instance

Geometry Shader Outputs

- type of output primitive generated specified by layout qualifier, which is one of: `points`, `line_strip`, `triangle_strip`
- can generate zero or more output primitives
- maximum number of vertices that can be generated specified by `max_vertices` layout qualifier
- per-vertex outputs:
  - **`vec4`** `gl_Position`: vertex position
  - **`float`** `gl_PointSize`: pixel width/height of point being rasterized; only used for point primitive
  - **`float`** `gl_ClipDistance[]`: distance to clipping planes
- per-primitive outputs:
  - **`vec4`** `gl_PrimitiveID`: primitive ID to pass to fragment shader
- `EmitVertex` called to process vertex outputs after all per-vertex outputs set
- after `EmitVertex` called, output variables have undefined values
- `EndPrimitive` called to signal end of primitive in order to start next output primitive
- not required to call `EndPrimitive` after last output primitive

Geometry Shader Example (Passthrough)

```
// use version 3.30 of GLSL (core profile)
#version 330

// input primitives are triangles
layout(triangles) in;

// input variable for color
in vec3 vColor[];

// output primitives are triangle strips
// at most three vertices will be generated
layout(triangle_strip, max_vertices = 3) out;

// output variable for color
out vec3 gColor;

void main() {
    // for each vertex of input triangle...
    for (int i = 0; i < 3; ++i) {
        // set position and color of output vertex
        gl_Position = gl_in[i].gl_Position;
        gColor = vColor[i];
        // mark vertex as finished
        EmitVertex();
    }
    EndPrimitive(); // optional
}
```

Using Shader Programs

- shaders need to be compiled and linked to yield executable shader program
- OpenGL provides compiler and linker
- normally, program should have vertex and fragment shaders
- to generate executable shader program:
  1. create program via `glCreateProgram`
  2. for each shader in program:
     1. create shader via `glCreateShader`
     2. load shader source via `glShaderSource`
     3. compile shader source to object code via `glCompileShader` and check status of compile via `glGetShaderiv`
     4. attach shader object code to program via `glAttachShader`
  3. link program `glLinkProgram` and check status of link via `glGetProgramiv`
- shader program currently in use selected via `glUseProgram`
- shader and program can be deleted when no longer needed via `glDeleteShader` and `glDeleteProgram`

Identifying Shader Variables in Application

- application program needs means to refer to attribute and uniform variables in shaders (e.g., in order to associate data with such variables)
- each attribute and uniform variable has integer identifier known as **location**
- location used as means to unambiguously name shader variable
- GLSL provides mechanism to force variable to have particular location via `location` layout qualifier
- location of variable can be queried by name (which is most useful when `location` layout qualifier not employed)
- can force attribute variable to use particular location via `glBindAttribLocation` prior to linking shader program

Identifying Shader Variables in Application (Continued)

- get location of shader variable via `glGetAttribLocation`
- example: query location of attribute variable `aPosition`:

```
GLuint program; // shader program ID
// ...
GLint loc = glGetAttribLocation(program,
  "aPosition");
```

- get location of uniform variable via `glGetUniformLocation`
- example: query location of uniform variable `uModelViewProjMatrix`:

```
GLuint program; // shader program ID
// ...
GLint loc = glGetUniformLocation(program,
  "uModelViewProjMatrix");
```

Associating Data in VAO with Attribute Variable

- application program needs to be able to associate shader attribute variable with data source (namely, data in VBO of VAO)
- to associate data in (VBO of) VAO with attribute variable in vertex shader, call `glVertexAttribPointer` when VAO/VBO containing attribute data is bound
- invocation of `glVertexAttribPointer` specifies:
  - location of vertex attribute variable
  - number of components per vertex attribute (e.g., 1, 2, 3, or 4)
  - type of each component (e.g., `GL_FLOAT` or `GL_DOUBLE`)
  - whether fixed-point values should be normalized (e.g., to $[-1, 1]$ for signed values and $[0, 1]$ for unsigned values)
  - stride (i.e., byte offset) between consecutive vertex attributes in array
  - offset of first component of first vertex attribute in array
- to enable use of attribute data associated with VAO, call `glEnableVertexAttribArray` when VAO containing attribute data is bound

Example: Associating Data in VAO with Attribute Variable

Part of Vertex Shader

```
in vec3 aPosition;
```

Part of Application Program

```
GLuint program; // program ID
GLuint vao; // VAO ID
GLuint vbo; // VBO ID
GLuint offset; // offset of data in VBO
GLsizei stride; // stride of data in VBO
// ...
GLint loc = glGetAttribLocation(program,
  "aPosition");
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE,
  stride, reinterpret_cast<GLvoid*>(offset));
glEnableVertexAttribArray(loc);
```

Accessing Uniform Variables from Application Program

- application program needs to be able to access uniform variables in shader
- application can only write to uniform variables since data flows in one direction only (i.e., from application to shader)
- uniform variable identified by location
- to modify uniform variable, must know its location
- modify uniform variable via `glUniform*` (which identifies variable to change by its location)
- example:

Part of Shader

```
uniform float uTime;
```

Part of Application Program

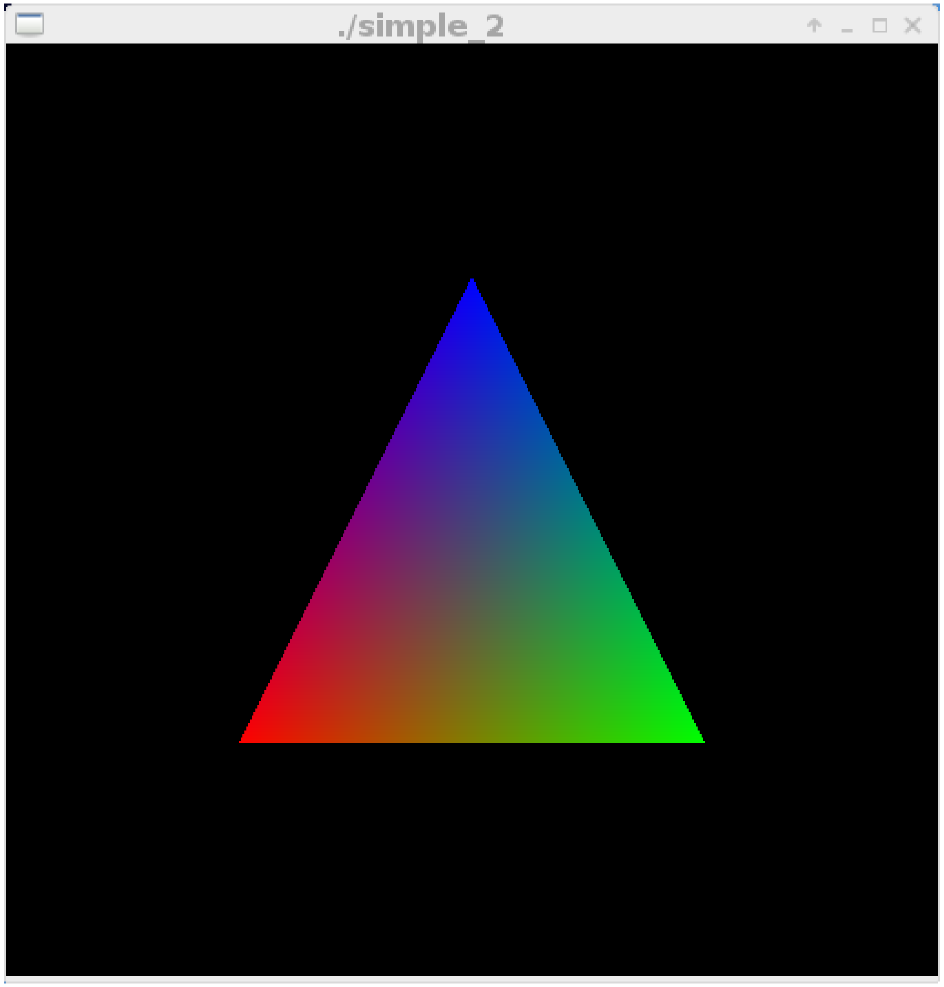
```
GLuint program; // shader program ID
// ...
GLint loc = glGetUniformLocation(program, "uTime");
glUniform1f(loc, 1.5f);
```

Section 5.6.3

Shader Examples

Simple: Shader Example

- vertex shader provided with two attributes per vertex (position and color)
- want smooth interpolation of color across faces
- rending output shown below for mesh consisting of single triangle



Vertex Shader

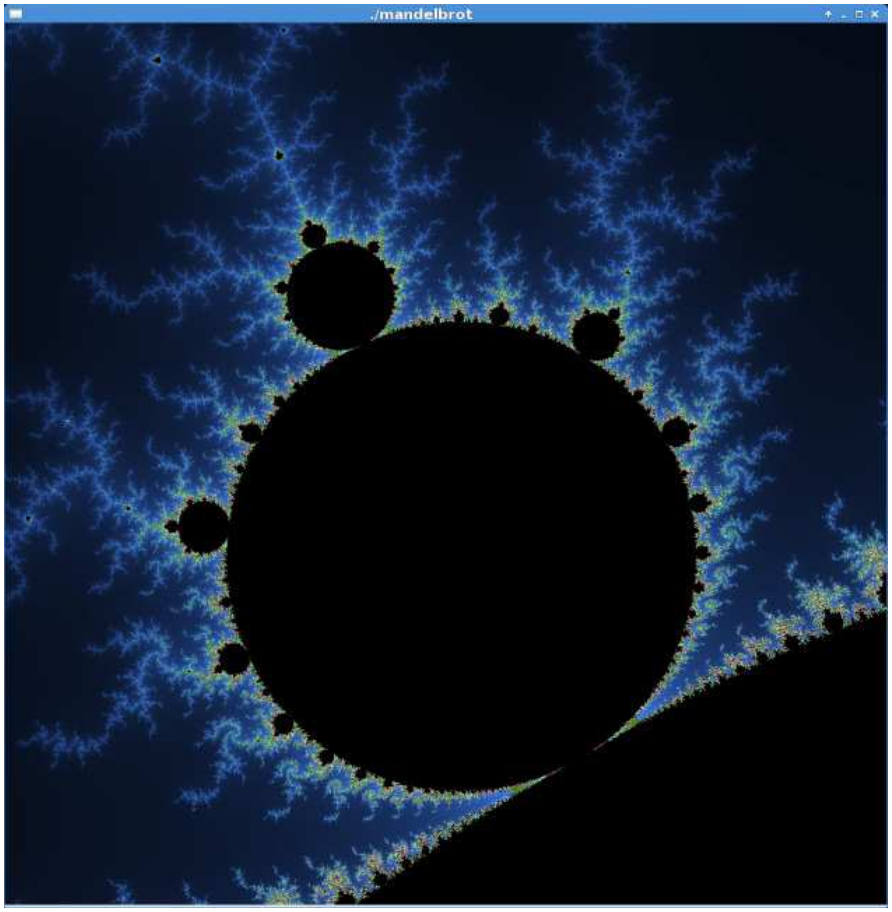
```
#version 330

in vec4 aPosition; // input vertex position attribute
in vec4 aColor; // input vertex color attribute

out vec4 vColor; // output vertex color (interpolated)

// uniform variable for modelview-projection
// matrix product
uniform mat4 uModelViewProjMatrix;

void main() {
    vColor = aColor;
    gl_Position = uModelViewProjMatrix * aPosition;
}
```

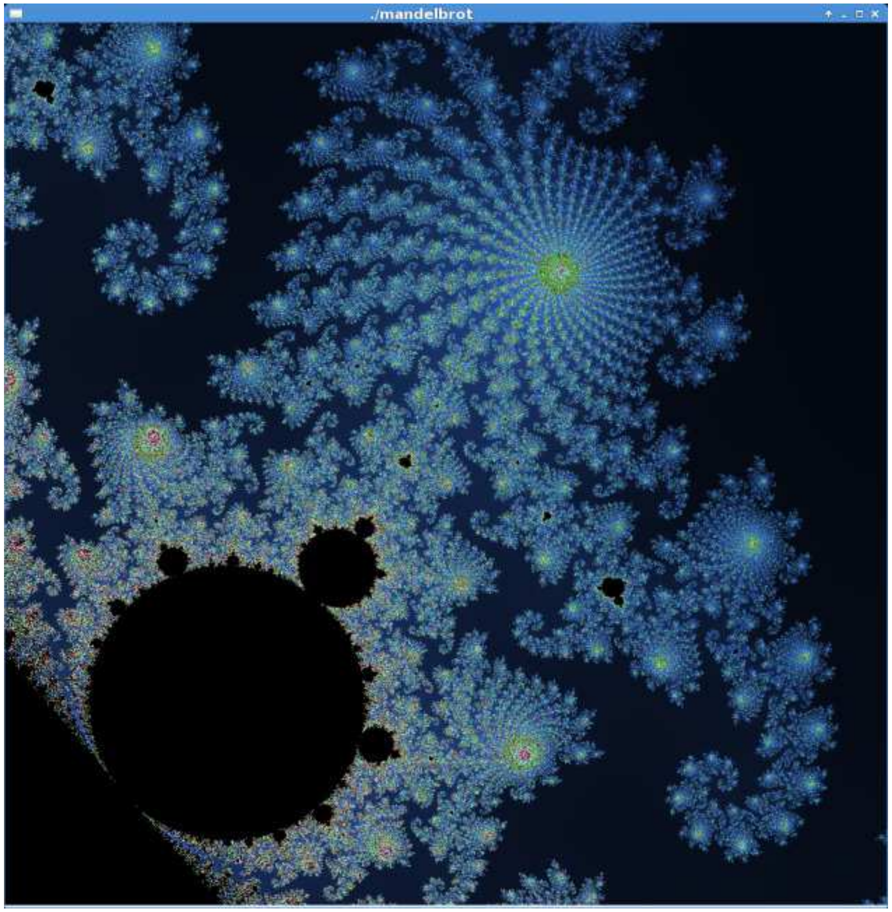
Fragment Shader

```
#version 330

in vec4 vColor; // input color (interpolated)

out vec4 fColor; // output fragment color

void main() {
    fColor = vColor;
}
```

Mandelbrot: Shader Example

- render triangles to cover entire drawing area and texture map Mandelbrot set onto triangles using fragment shader
- some examples of rendering results shown below

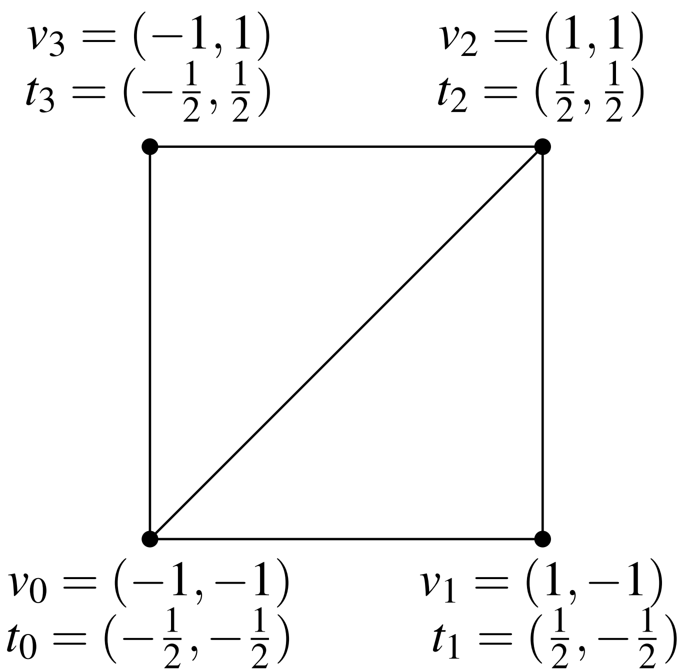


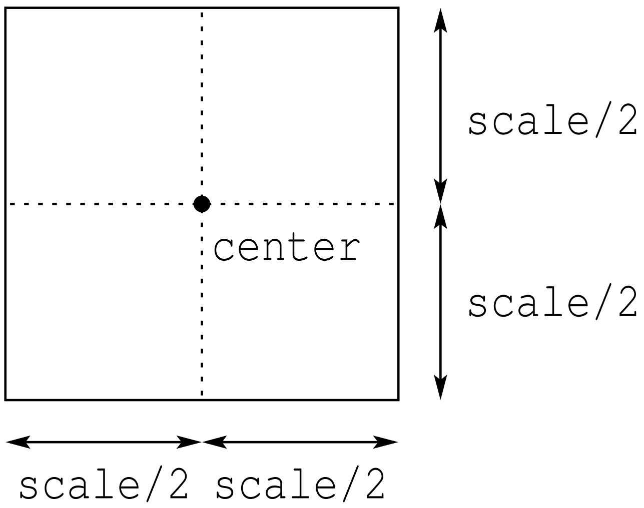


- **Mandelbrot set**: set of all complex numbers c such that sequence $z_0, z_1, z_2, \ldots$ does not tend toward infinity, where

$$z_n = \begin{cases} z_{n-1}^2 + c & \text{if } n \geq 1 \\ c & \text{if } n = 0 \end{cases}$$

- associate rectangular region in complex plane with graphics viewport
- for point corresponding to each pixel in viewport, determine number of steps in above iterative process for which result does not become too large (i.e., tending towards infinity)
- assign color to each pixel depending on obtained iteration count





- application program simply renders two triangles that cover full extent of viewport ($\{v_k\}$ are positional coordinates; $\{t_k\}$ are texture coordinates)
- texture coordinate region $[-\frac{1}{2}, \frac{1}{2}] \times [-\frac{1}{2}, \frac{1}{2}]$ corresponds to full viewport
- square region in complex plane of width/height `scale` centered at point `center` is mapped onto region $[-\frac{1}{2}, \frac{1}{2}] \times [-\frac{1}{2}, \frac{1}{2}]$ in texture coordinates

Mandelbrot: Vertex Shader

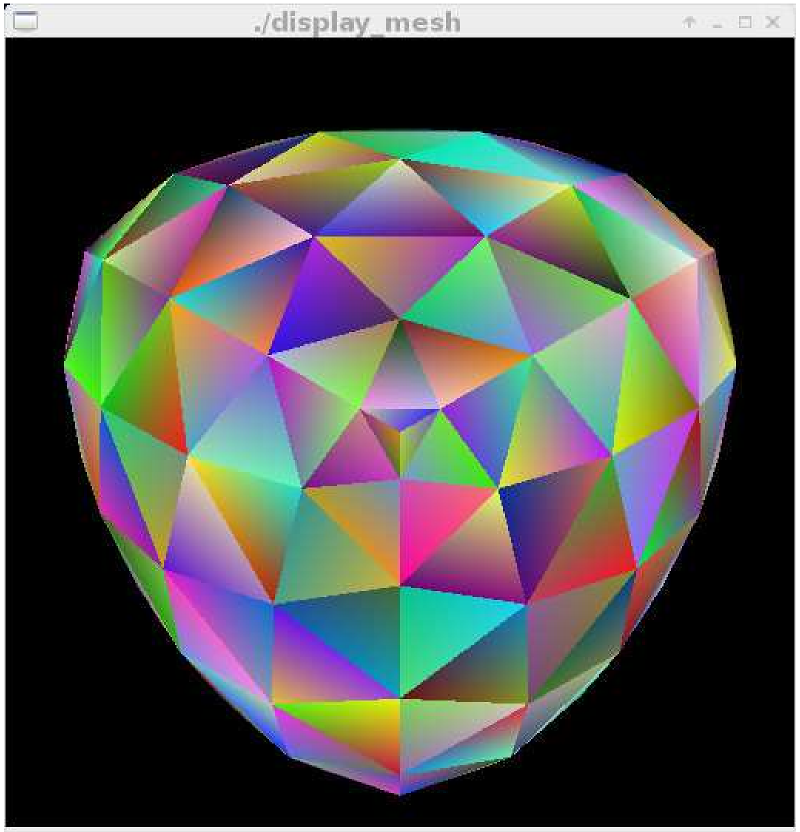
```
#version 330

in vec3 aPosition; // position vertex attribute
in vec3 aTexCoord; // texture-coordinate vertex attribute

out vec3 vTexCoord; // texture coordinate (interpolated)

void main() {
    vTexCoord = aTexCoord;
    gl_Position = vec4(aPosition, 1.0);
}
```

Mandelbrot: Fragment Shader

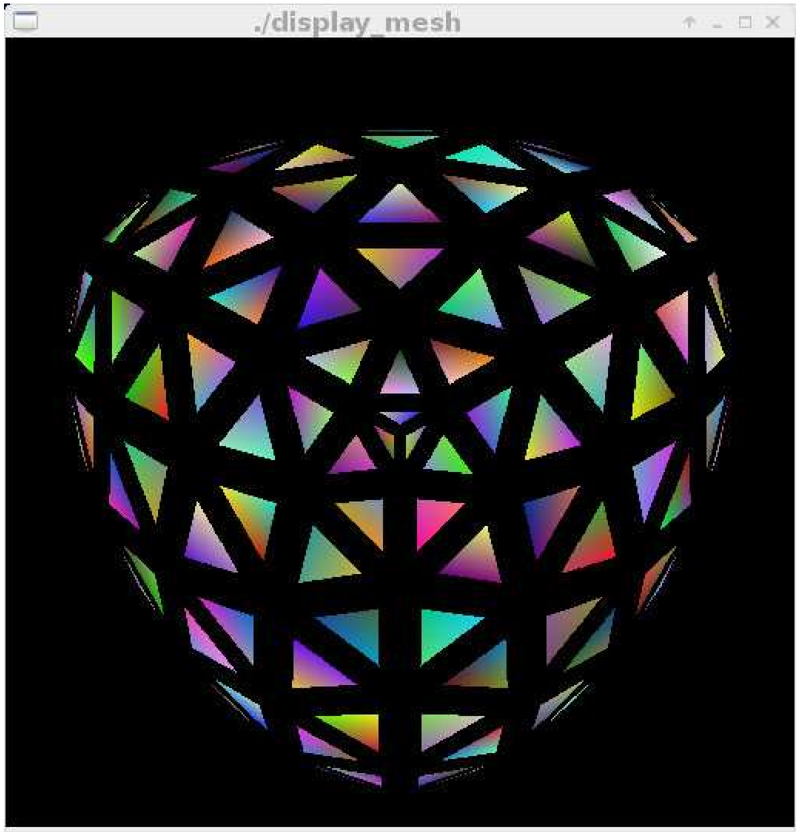
```
#version 330

in vec3 vTexCoord; // texture coordinates

out vec4 fColor; // vertex color

uniform vec2 center; // center of viewing region
uniform float scale; // width/height of viewing region
uniform int maxIters; // maximum iteration count

int mandelbrot(vec2 c) {
    vec2 z = vec2(0.0, 0.0);
    int i;
    for (i = 0; i < maxIters; ++i) {
        z = vec2(z.x * z.x - z.y * z.y + c.x, 2.0 * z.x * z.y + c.y);
        if (length(z) > 2.0) {break;}
    }
    return i;
}

float lookup(float x, float c) {return c * mod(x, 1.0 / c);}

void main() {
    int i = mandelbrot(vec2(scale * vTexCoord.x + center.x,
      scale * vTexCoord.y + center.y));
    float t = float(i) / maxIters;
    fColor = vec4(lookup(t, 2.0), lookup(t, 4.0), lookup(t, 8.0), 1.0);
}
```

ShrinkFace: Shader Example

- use geometry shader to shrink triangles sent to rendering pipeline
- triangles contracted towards their centroid so that triangles that were originally touching now have gap between them
- example rendering results are shown below

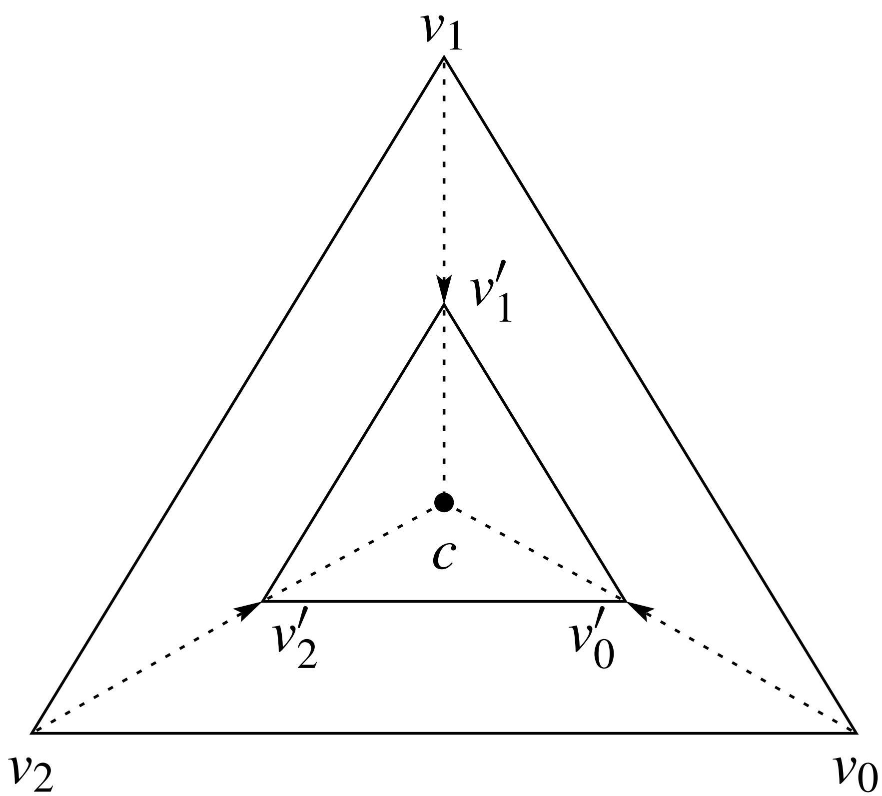


Rendered Normally



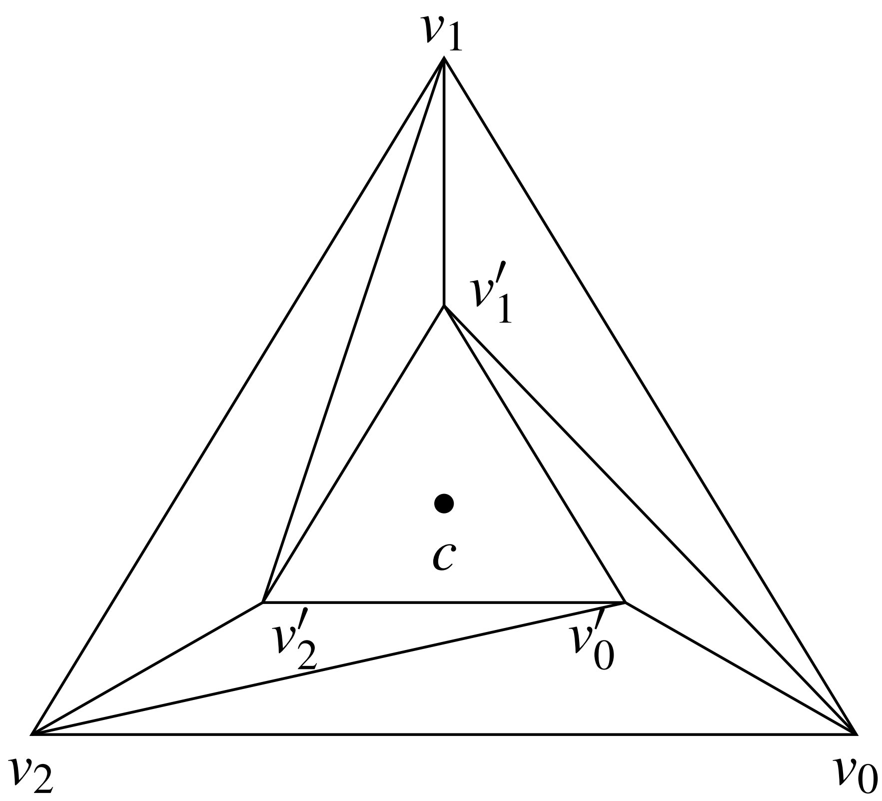
Rendered with Shrunk Faces

ShrinkFace: Triangle Shrinking



- each vertex v_k moved in direction of centroid c to new position $v'_k = \frac{1}{2}(v_k + c)$ (i.e., midpoint of v_k and c)
- gap formed by shrinking of triangle is filled with new triangles drawn in black

ShrinkFace: Gap Filling



- gap can be filled with triangle strip with vertices: v_2, v'_2, v_1, v'_1, v_0, v'_0, v_2, v'_2

ShrinkFace: Vertex Shader

```
#version 330

in vec3 aPosition; // position vertex attribute
in vec3 aColor; // color vertex attribute

out vec3 vColor; // color (interpolated)

void main() {
    gl_Position = vec4(aPosition, 1.0);
    vColor = aColor;
}
```

ShrinkFace: Geometry Shader

```
#version 330

layout(triangles) in; // triangle primitives as input
in vec3 vColor[]; // input vertex colors

layout(triangle_strip, max_vertices=11) out;
  // triangle strips as output; at most 11 vertices
out vec3 gColor; // output color (interpolated)

uniform mat4 uModelViewProjMatrix;
  // modelview-projection matrix product
```

ShrinkFace: Geometry Shader (Continued)

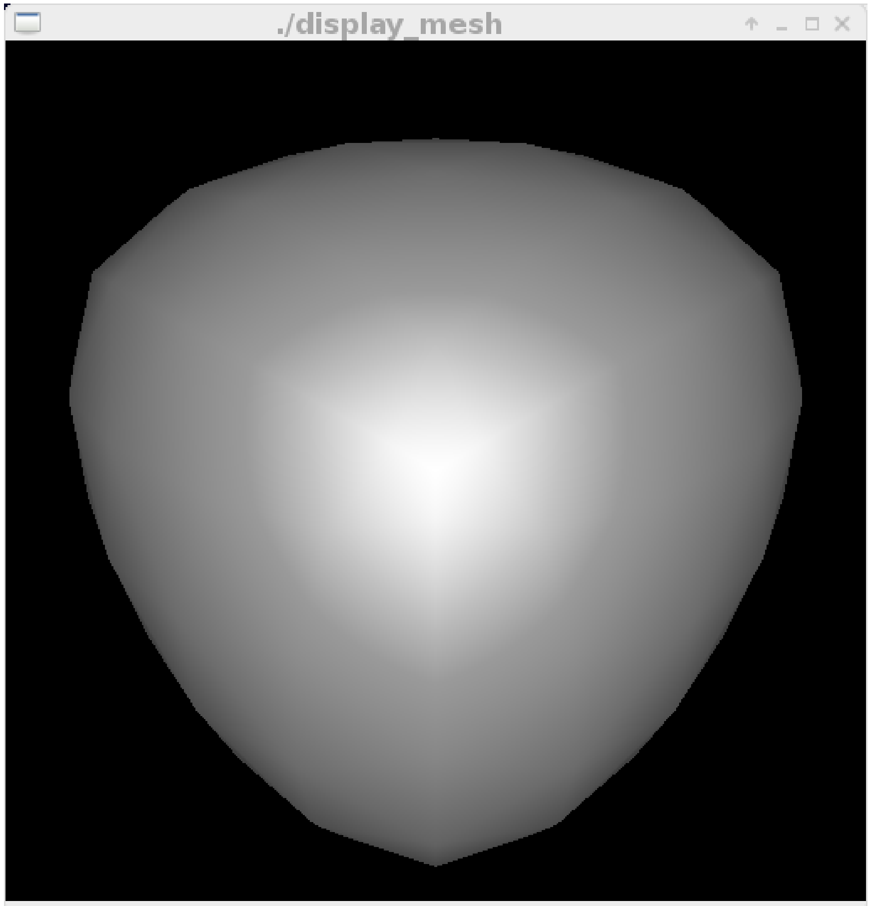
```
void main() {
    vec3 v[6];
    for (int i = 0; i < 3; ++i) {v[i] = gl_in[i].gl_Position.xyz;}

    // compute centroid of triangle
    vec3 c = (v[0] + v[1] + v[2]) / 3.0;

    // compute vertices of shrunk triangle and generate
    // triangle strip consisting only of shrunk triangle
    for (int i = 0; i < 3; ++i) {
        v[i + 3] = c + 0.5 * (v[i] - c);
        gl_Position = uModelViewProjMatrix * vec4(v[i + 3], 1.0);
        gColor = vColor[i];
        EmitVertex();
    }
    EndPrimitive();

    // generate triangle strip to fill gap between triangles
    // introduced by shrinking
    const int lut[] = int[](2, 5, 1, 4, 0, 3, 2, 5);
    for (int i = 0; i < 8; ++i) {
        gl_Position = uModelViewProjMatrix * vec4(v[lut[i]], 1.0);
        gColor = vec3(0.0, 0.0, 0.0);
        EmitVertex();
    }
}
```

ShrinkFace: Fragment Shader

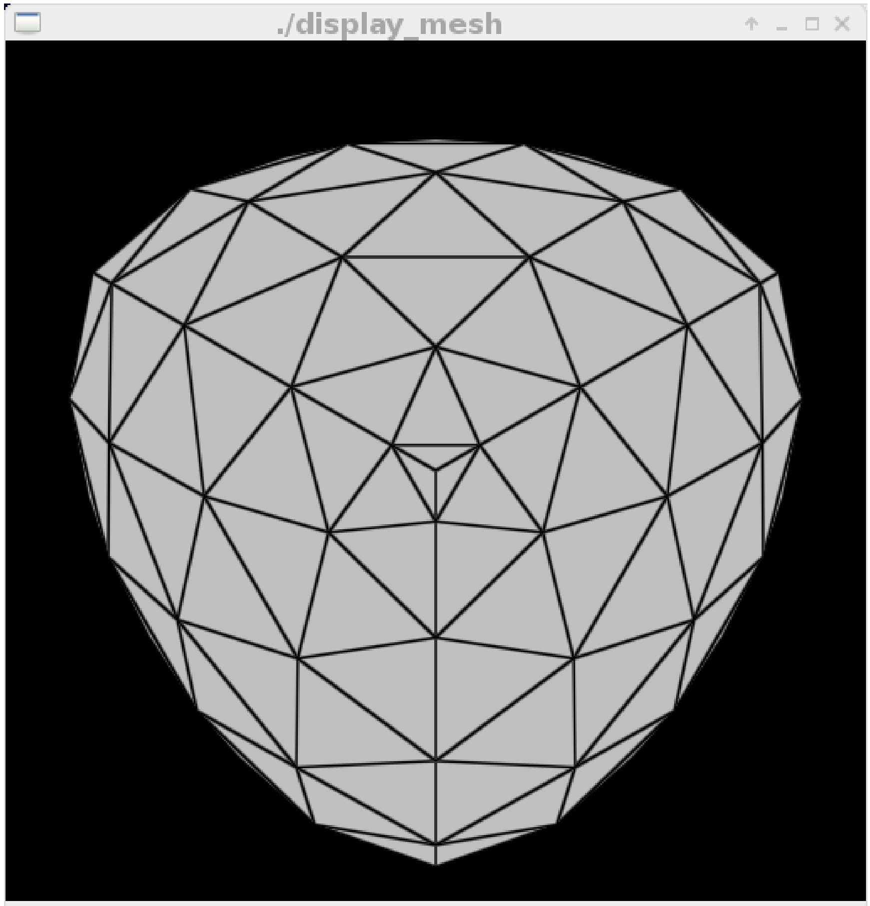
```
#version 330

in vec3 gColor; // input color

out vec4 fColor; // output color

void main() {
    fColor = vec4(gColor, 1.0);
}
```

Wireframe: Shader Example

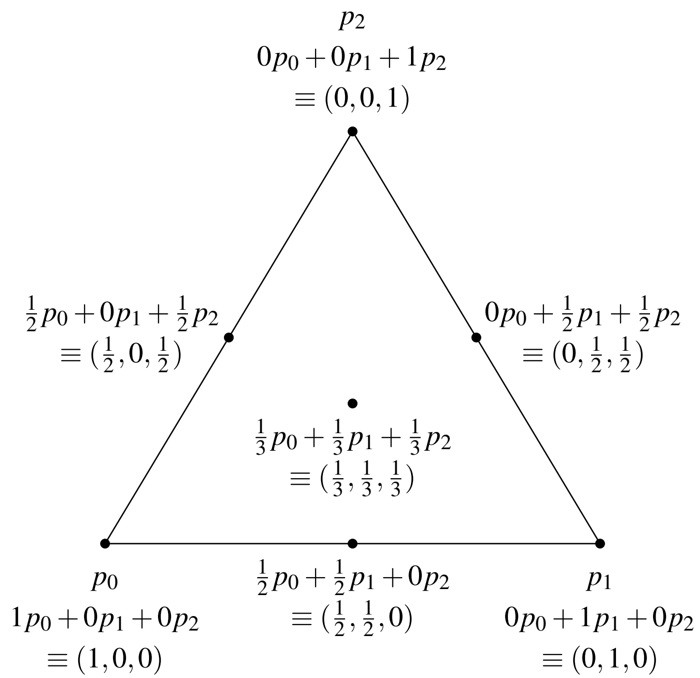
- use geometry shader to assist in superimposing wireframe on rendered surface
- example rendering output shown below



Without Edges Shown



Edges Shown Using Shader



- points on edge of triangle must have exactly one or two barycentric coordinates equal to zero, while points in the interior must have three nonzero coordinates
- if at least one of barycentric coordinates is small, must be in vicinity of edge
- if in vicinity of edge, use different color

Wireframe: Vertex Shader

```
#version 330

in vec3 aPosition; // position vertex attribute
in vec3 aColor; // color vertex attribute

out vec3 vColor; // output color (interpolated)

uniform mat4 uModelViewProjMatrix;
  // modelview-projection matrix product

void main() {
    gl_Position = uModelViewProjMatrix * vec4(aPosition, 1.0);
    vColor = aColor;
}
```

Wireframe: Geometry Shader

```
#version 330

layout(triangles) in; // triangles as input
in vec3 vColor[]; // vertex colors

layout(triangle_strip, max_vertices=3) out;
  // triangle strips as output; at most 3 vertices
out vec3 gColor; // output color
noperspective out vec3 gBaryCoord;
  // output barycentric coordinates (interpolated)

void main() {
    const vec3 lut[3] = vec3[3](
      vec3(1.0, 0.0, 0.0),
      vec3(0.0, 1.0, 0.0),
      vec3(0.0, 0.0, 1.0));
    for (int i = 0; i < 3; ++i) {
        gl_Position = gl_in[i].gl_Position;
        gBaryCoord = lut[i];
        gColor = vColor[i];
        EmitVertex();
    }
}
```

Wireframe: Fragment Shader

```
#version 330

in vec3 gColor; // input color
noperspective in vec3 gBaryCoord;
  // input barycentric coordinates

out vec4 fColor; // output color

void main() {
    const vec3 edgeColor = vec3(0.0, 0.0, 0.0);
    const float edgeWidth = 1.0;
    vec3 d = fwidth(gBaryCoord);
    vec3 a3 = smoothstep(vec3(0.0), d * edgeWidth, gBaryCoord);
    float v = min(min(a3.x, a3.y), a3.z);
    fColor = vec4(mix(edgeColor, gColor, v), 1.0);
}
```

- upper threshold for `smoothstep` chosen relative to approximate gradient magnitude so thickness of edges in wireframe same regardless of triangle size
- simpler code for calculating `a3` shown below would cause thickness of edges in wireframe to depend on triangle size, which would be less aesthetically pleasing:

```
vec3 a3 = smoothstep(vec3(0.0), vec3(0.02), gBaryCoord);
```

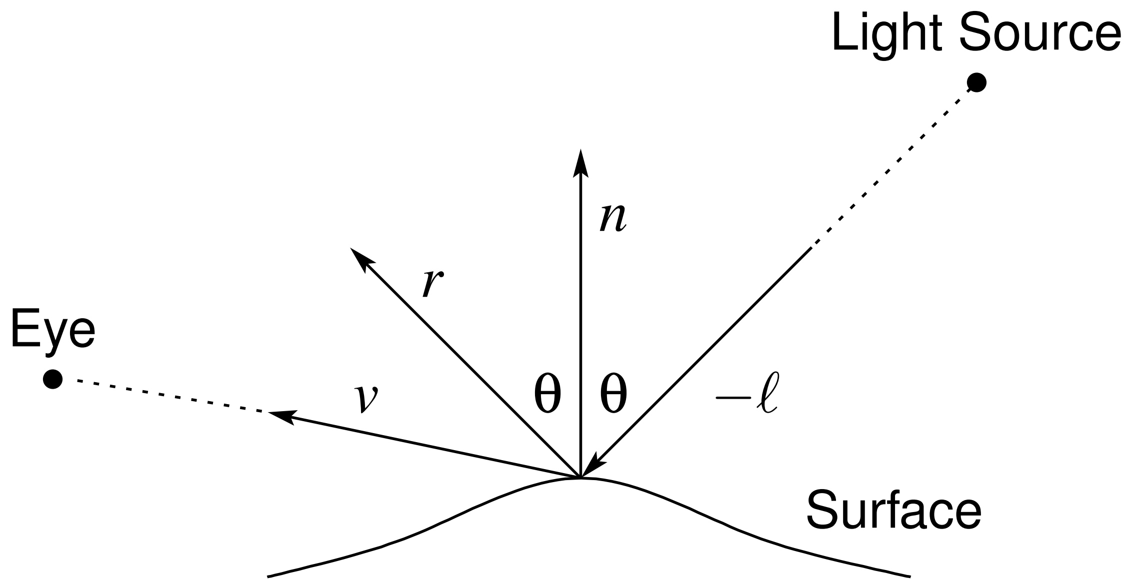
Ambient-Diffuse-Specular (ADS) Lighting Model

- light properties:
  - ℓ_a: ambient component of light source
  - ℓ_d: diffuse component of light source
  - ℓ_s: specular component of light source
- material properties:
  - k_a: ambient reflection constant
  - k_d: diffuse reflection constant
  - k_s: specular reflection constant
  - α: shininess constant
- vectors:
  - ℓ: unit vector vector in direction from point on surface to light source
  - n: unit normal at point on surface
  - v: unit vector in direction from point on surface to viewer
  - r: unit vector in direction that perfectly reflected light ray would take from this point on surface (i.e., $r = 2(l \cdot n)n - \ell$)
- illumination i of point on surface given by:

$$i = k_a \ell_a + \max\{(\ell \cdot n), 0\} k_d \ell_d + \max\{(r \cdot v)^\alpha, 0\} u(\ell \cdot n) k_s \ell_s$$

  where u is unit-step function

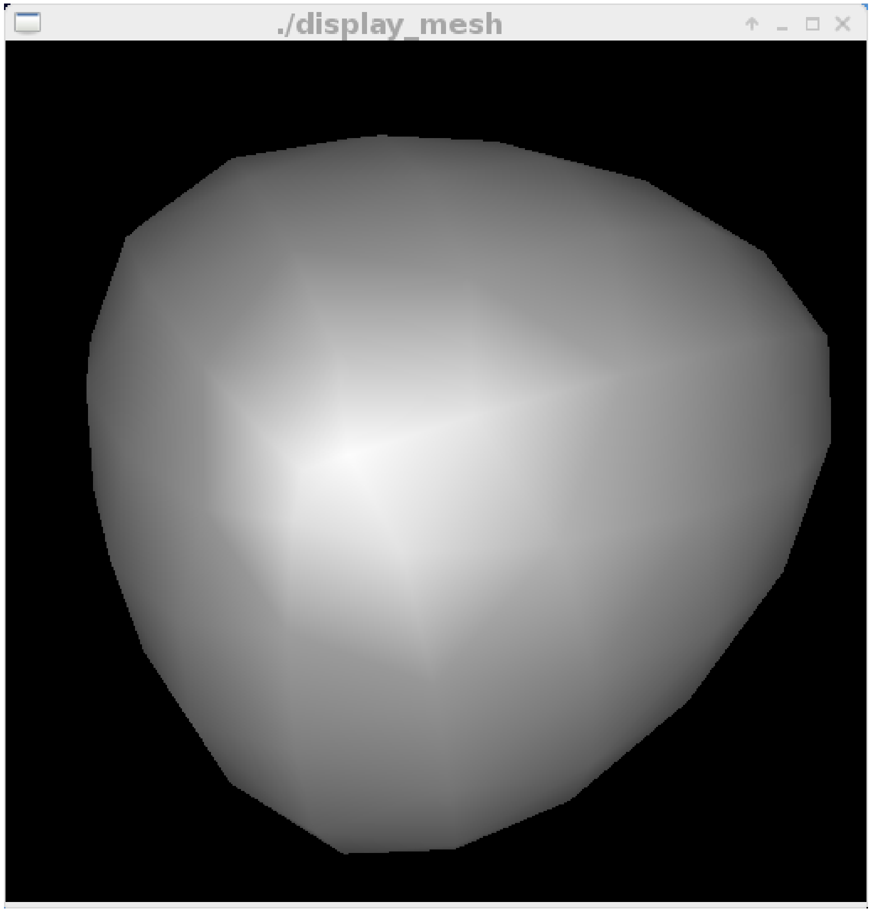
ADS Lighting Model: Diagram



- ℓ: unit vector vector in direction from point on surface to light source
- n: unit normal at point on surface
- v: unit vector in direction from point on surface to viewer
- r: unit vector in direction that perfectly reflected light ray would take from this point on surface

Per-Vertex Lighting: Shader Example

- per-vertex lighting using ambient-diffuse-specular (ADS) model
- example rendering result shown below



Per-Vertex Lighting: Vertex Shader

```
#version 330

in vec3 aPosition; // position vertex attribute
in vec3 aNormal; // normal vertex attribute

out vec3 vColor; // output color (interpolated)

uniform mat4 uModelViewMatrix; // modelview matrix
uniform mat3 uNormalMatrix; // normal transformation matrix
uniform mat4 uModelViewProjMatrix;
  // modelview-projection matrix product

struct LightSourceParams {
    vec4 position; // position
    vec3 ambient; // ambient component
    vec3 diffuse; // diffuse component
    vec3 specular; // specular component
};
uniform LightSourceParams uLight; // light parameters

struct MaterialParams {
    vec3 ambient; // ambient reflectance
    vec3 diffuse; // diffuse reflectance
    vec3 specular; // specular reflectance
    float shininess; // specular exponent
};
uniform MaterialParams uMaterial; // material parameters
```

Per-Vertex Lighting: Vertex Shader (Continued)

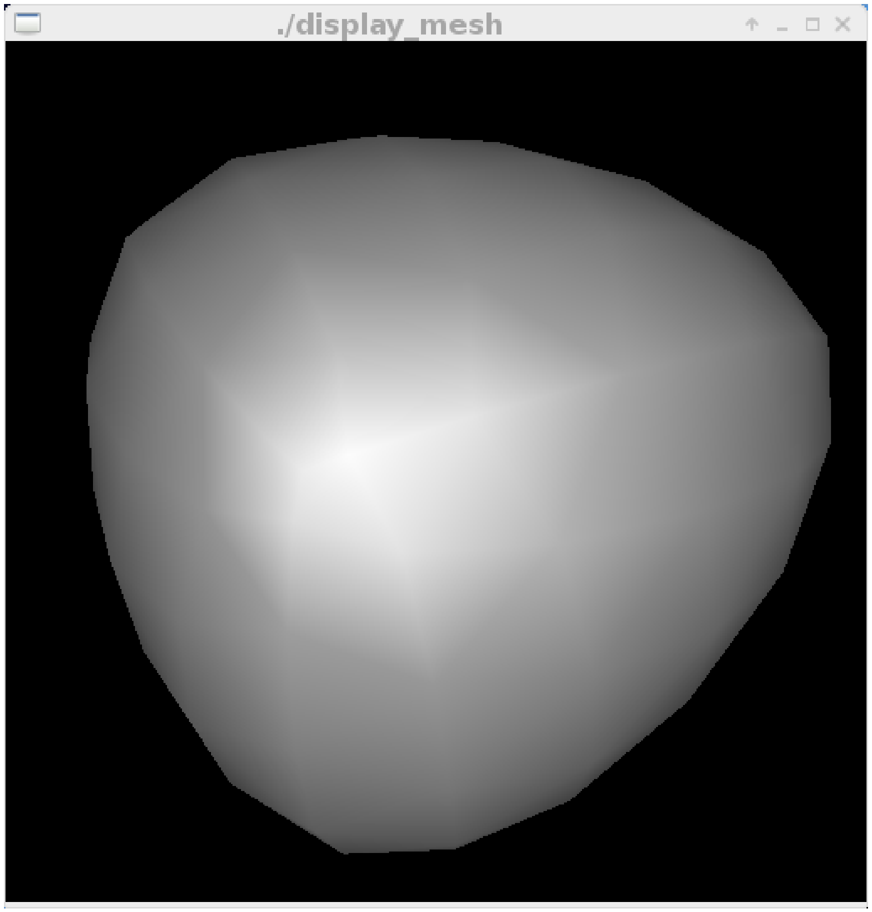
```
vec3 ads(vec4 position, vec3 normal) {
    vec3 s = normalize(vec3(uLight.position - position));
    vec3 v = normalize(-position.xyz);
    vec3 r = reflect(-s, normal);
    float sn = dot(s, normal);
    vec3 ambient = uLight.ambient * uMaterial.ambient;
    vec3 diffuse = uLight.diffuse * uMaterial.diffuse *
      max(sn, 0.0);
    diffuse = clamp(diffuse, 0.0, 1.0);
    vec3 specular = (sn > 0.0) ? (uLight.specular *
      uMaterial.specular * pow(max(dot(r, v), 0.0),
      uMaterial.shininess)) : vec3(0.0);
    specular = clamp(specular, 0.0, 1.0);
    return clamp(ambient + diffuse + specular, 0.0, 1.0);
}

void main() {
    vec3 eyeNorm = normalize(uNormalMatrix * aNormal);
    vec4 eyePos = uModelViewMatrix * vec4(aPosition, 1.0);
    vColor = ads(eyePos, eyeNorm);
    gl_Position = uModelViewProjMatrix *
      vec4(aPosition, 1.0);
}
```

Per-Vertex Lighting: Fragment Shader

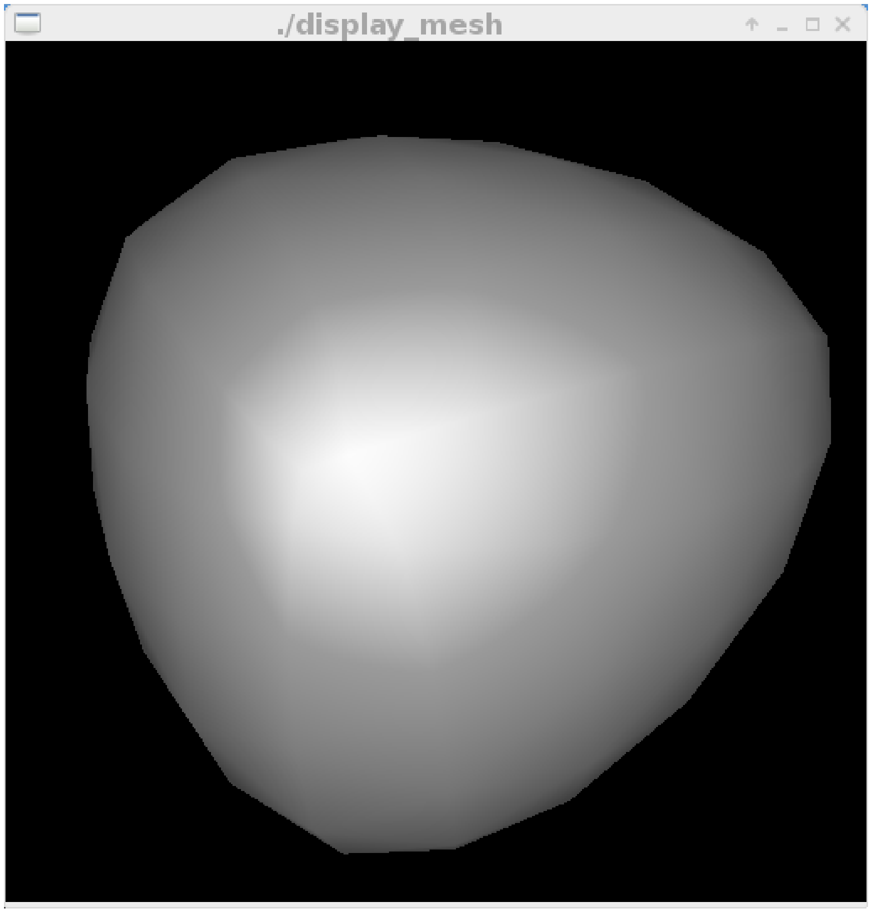
```
#version 330

in vec3 vColor; // input color

out vec4 fColor; // output color

void main() {
    fColor = vec4(vColor, 1.0);
}
```

Per-Fragment Lighting: Shader Example

- per-fragment lighting using ambient-diffuse-specular (ADS) model
- example rendering result shown along with per-vertex lighting result for comparison



Per-Vertex Lighting



Per-Fragment Lighting

Per-Fragment Lighting: Vertex Shader

```
#version 330

in vec3 aPosition; // position vertex attribute
in vec3 aNormal; // normal vertex attribute

out vec3 vPosition; // output position (interpolated)
out vec3 vNormal; // output normal (interpolated)

uniform mat4 uModelViewMatrix; // modelview matrix
uniform mat3 uNormalMatrix; // normal transformation matrix
uniform mat4 uModelViewProjMatrix;
  // modelview-projection matrix product

void main() {
    vNormal = normalize(uNormalMatrix * aNormal);
    vPosition = vec3(uModelViewMatrix * vec4(aPosition, 1.0));
    gl_Position = uModelViewProjMatrix * vec4(aPosition, 1.0);
}
```

Per-Fragment Lighting: Fragment Shader

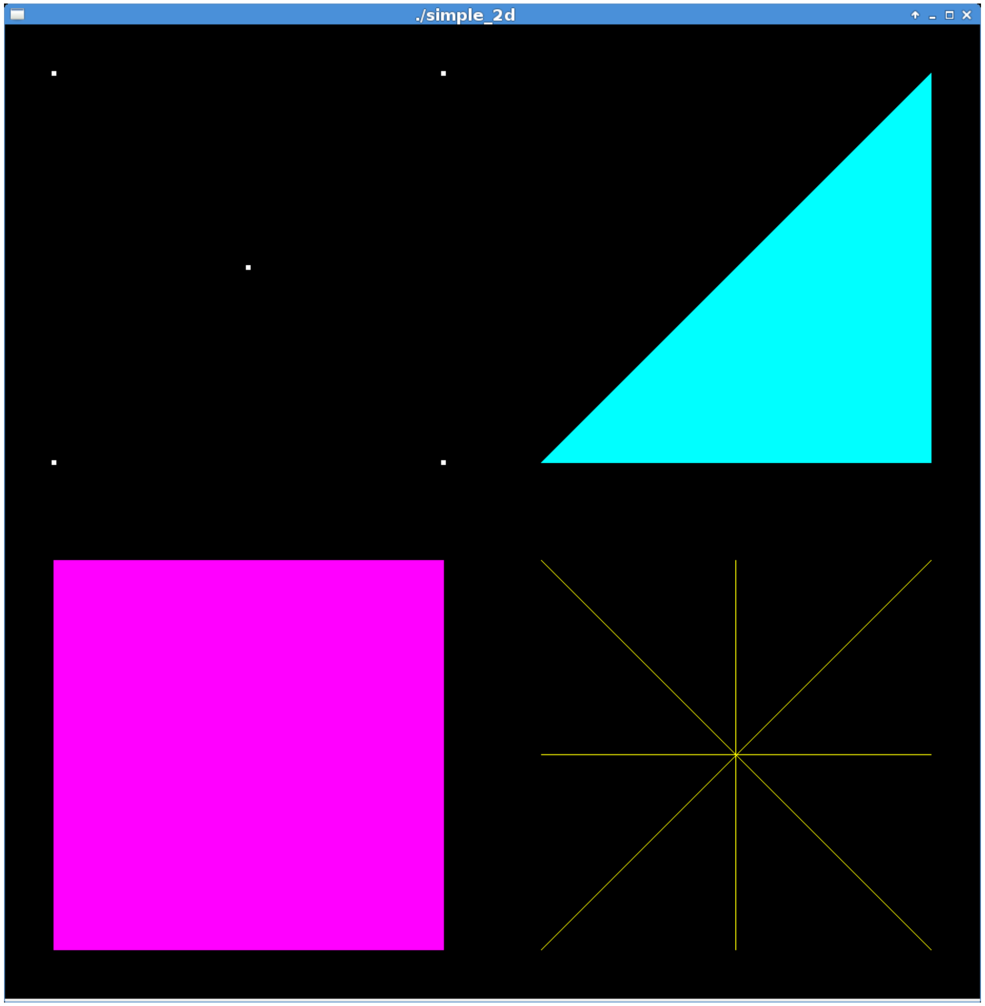
```
#version 330

in vec3 vNormal; // input normal
in vec3 vPosition; // input position

out vec4 fColor; // output color

struct LightSourceParams {
    vec4 position; // position
    vec3 ambient; // ambient component
    vec3 diffuse; // diffuse component
    vec3 specular; // specular component
};
uniform LightSourceParams uLight; // light parameters

struct MaterialParams {
    vec3 ambient; // ambient reflectance
    vec3 diffuse; // diffuse reflectance
    vec3 specular; // specular reflectance
    float shininess; // specular exponent
};
uniform MaterialParams uMaterial; // material parameters
```

```
vec3 ads(vec4 position, vec3 normal) {
    vec3 s = normalize(vec3(uLight.position - position));
    vec3 v = normalize(-position.xyz);
    vec3 r = reflect(-s, normal);
    float sn = dot(s, normal);
    vec3 ambient = uLight.ambient * uMaterial.ambient;
    vec3 diffuse = uLight.diffuse * uMaterial.diffuse *
      max(sn, 0.0);
    diffuse = clamp(diffuse, 0.0, 1.0);
    vec3 specular = (sn > 0.0) ? uLight.specular *
      uMaterial.specular * pow(max(dot(r, v), 0.0),
      uMaterial.shininess) : vec3(0.0);
    specular = clamp(specular, 0.0, 1.0);
    return clamp(ambient + diffuse + specular, 0.0, 1.0);
}

void main() {
    fColor = vec4(ads(vec4(vPosition, 1.0), vNormal), 1.0);
}
```

Section 5.6.4

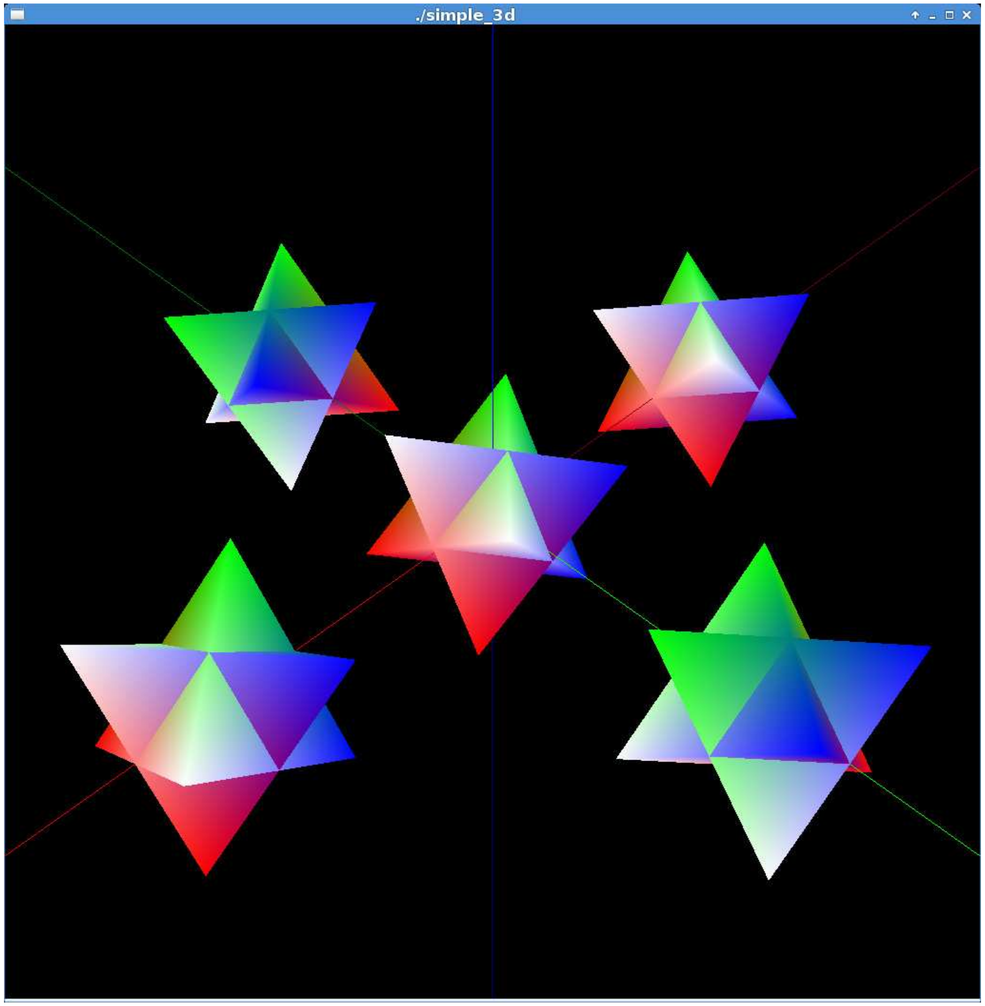
OpenGL Example Programs

- simple 2-D graphics
- draws points, lines, triangle, and quadrilateral



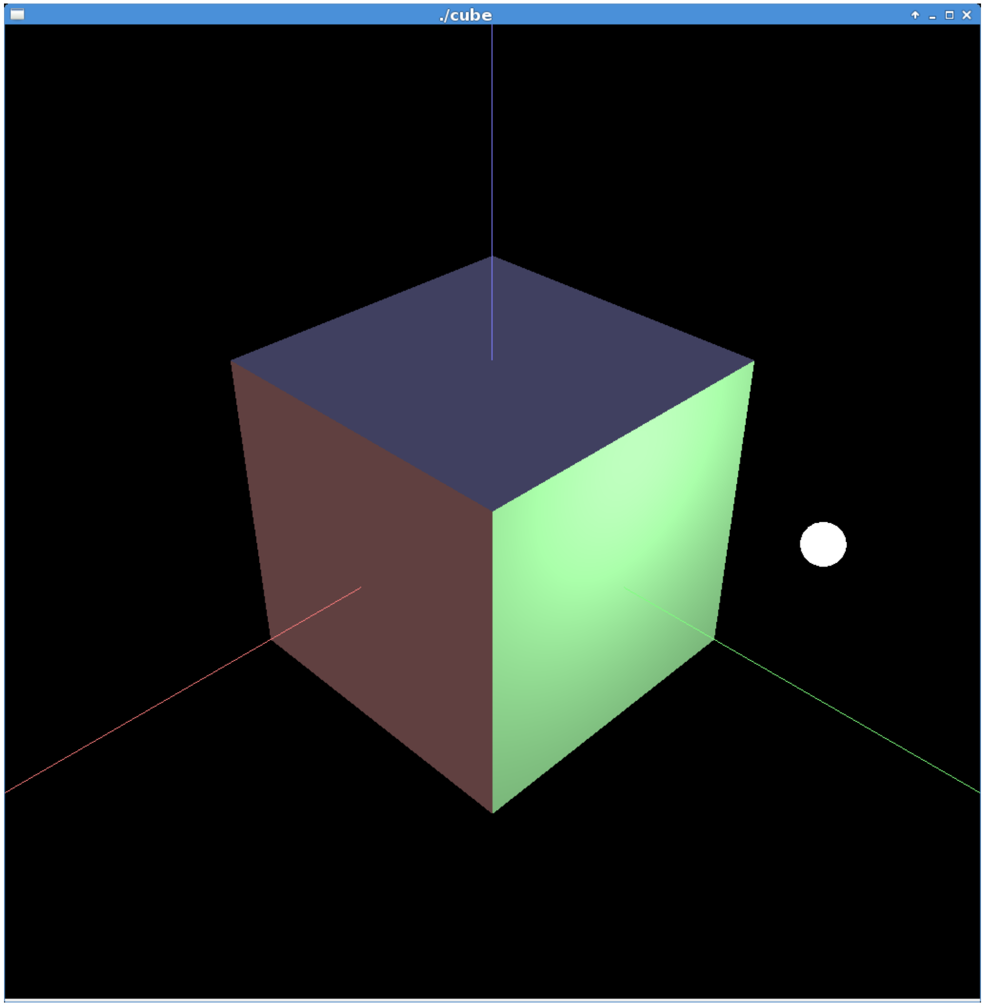
OpenGL Example Program: `simple_3d`

- simple 3-D graphics
- draws and animates several simple polyhedra



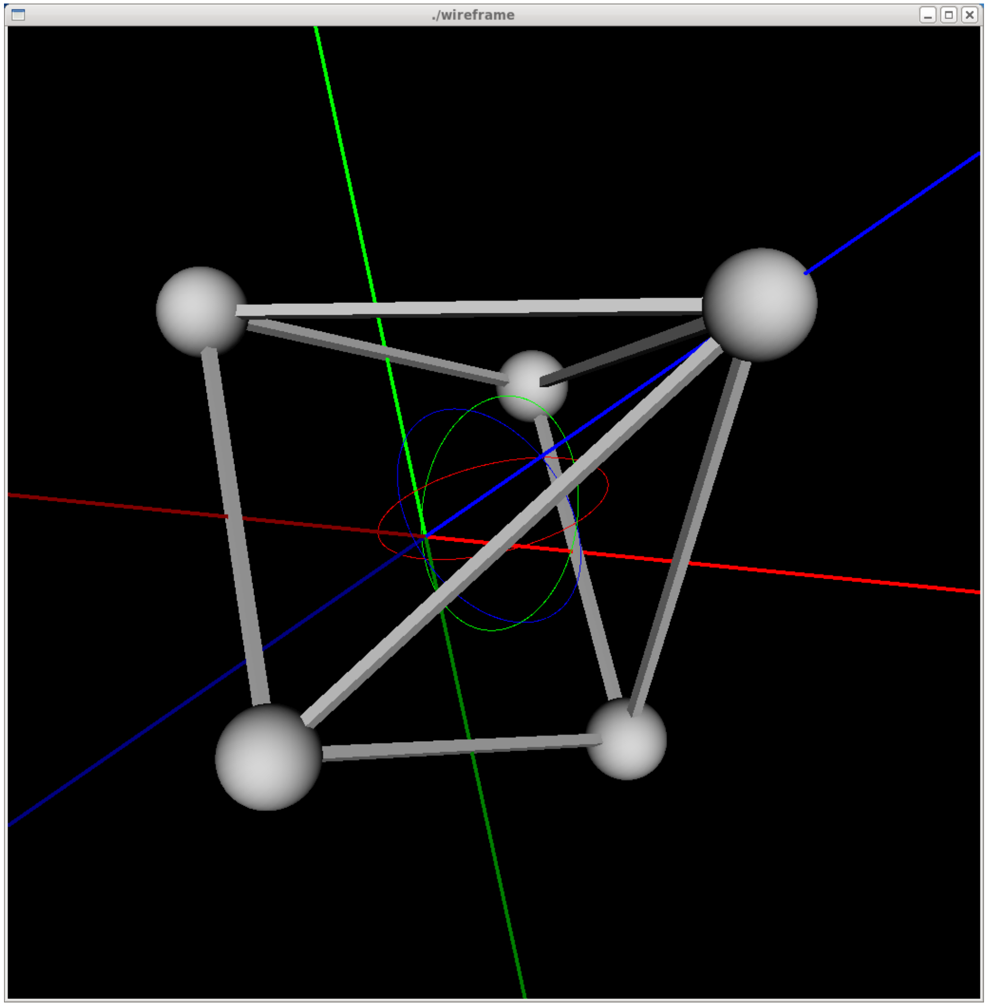
OpenGL Example Program: cube

- 3-D graphics with lighting
- draws cube with lighting



OpenGL/CGAL Example Program: `wireframe`

- wireframe mesh viewer
- allows polygon mesh to viewed as wireframe



Section 5.6.5

References

1. D. Shreiner, G. Sellers, J. Kessenich, and B. Licea-Kane. *OpenGL Programming Guide.*
   Addison-Wesley, Upper Saddle River, NJ, USA, 8th edition, 2013.
2. R. S. Wright Jr., N. Haemel, G. Sellers, and B. Lipchak. *OpenGL Superbible.*
   Addison-Wesley, Upper Saddle River, NJ, USA, 5th edition, 2011.
3. E. Angel and D. Shreiner. *Interactive Compute Graphics — A Top-Down Approach with Shader-Based OpenGL.*
   Addison-Wesley, Boston, MA, USA, 6th edition, 2012.
4. M. Bailey and S. Cunningham. *Graphics Shaders — Theory and Practice.*
   CRC Press, Boca Raton, FL, USA, 2nd edition, 2012.
5. R. J. Rost. *OpenGL Shading Language.*
   Addison-Wesley, Boston, MA, USA, 2nd edition, 2006.

6. D. Wolff. *OpenGL 4.0 Shading Language Cookbook.* Packt Publishing, Birmingham, UK, 2011.
7. The OpenGL graphics system: A specification (version 4.4 (core profile)), Mar. 2014.
8. The OpenGL shading language — language version 4.40, June 2014.
9. OpenGL Web Site, `http://www.opengl.org`.
10. OpenGL Software Development Kit (SDK), `https://www.opengl.org/sdk` (full documentation on each OpenGL function can be found at `http://www.opengl.org/sdk/docs/man`).
11. Khronos Group on YouTube, `https://www.youtube.com/user/khronosgroup`.

Talks I

1. Ed Angel and Dave Shreiner, An Introduction to OpenGL Programming, SIGGRAPH 2013, Available online at `https://youtu.be/6-9XFm7XAT8`.

Software

- OpenGL Extension Wrangler Library (GLEW)

  `http://glew.sourceforge.net`
  `http://www.opengl.org/sdk/libs/GLEW`
- OpenGL FrameWork (GLFW) Library

  `http://www.glfw.org`
- OpenGL Utility Toolkit (GLUT) Library

  `http://sourceforge.net/projects/freeglut`
- OpenGL Mathematics (GLM) Library

  `http://glm.g-truc.net`
- Qt Library

  `http://www.qt.io`
  `http://www.qt.io/developers`

Section 5.7

Other Libraries

Numerical Libraries I

- Eigen
  - C++ library for linear algebra
  - web site: `http://eigen.tuxfamily.org`
- Lapack++
  - C++ library for high-performance linear-algebra computations
  - C++ wrapper for LAPACK and BLAS
  - web site: `http://lapackpp.sourceforge.net`
- Armadillo
  - C++ library for linear algebra
  - web site: `http://arma.sourceforge.net`
- GNU Scientific Library
  - C library for numerical analysis
  - web site: `http://www.gnu.org/software/gsl`
- GNU Multiprecision Library
  - C library for arbitrary-precision arithmetic
  - web site: `http://gmplib.org`

Numerical Libraries II

- Boost.uBLAS
  - C++ library for numerical computation
  - web site: `http://www.boost.org/doc/libs/release/libs/numeric/ublas`
- Boost.Rational
  - C++ rational number library
  - web site: `www.boost.org/doc/libs/release/libs/rational`
- Boost.Interval
  - C++ interval arithmetic library
  - web site: `www.boost.org/doc/libs/release/libs/numeric/interval/doc/interval.htm`
- Boost.Math
  - C++ library
  - provides math constants, GCD, LCM, quaternions, and more
  - web site: `http://www.boost.org/doc/libs/release/libs/math`
- Linear Algebra Package (LAPACK)
  - Fortran library for numerical computing

Numerical Libraries III

- web site: `http://www.netlib.org/lapack`

- Basic Linear Algebra Subprograms (BLAS)
  - de facto API for publishing libraries to perform basic linear algebra operations
  - written in Fortran
  - web site: `http://www.netlib.org/blas`

Part 6

Programming

Section 6.1

Good Programming Practices

Formatting, Naming, Documenting

- Be consistent with the ***formatting*** of the source code (e.g., indentation strategy, tabs versus spaces, spacing, brackets/parentheses).
- Avoid a formatting style that runs against common practices.
- Be consistent in the ***naming conventions*** used for identifiers (e.g., names of objects, functions, namespaces, types) and files.
- Avoid bizarre naming conventions that run against common practices.
- ***Comment*** your code. If code is well documented, it should be possible to quickly ascertain what the code is doing without any prior knowledge of the code.
- Use ***meaningful names*** for identifiers (e.g., names of objects, functions, types, etc.). This improves the readability of code.
- Avoid ***magic literal constants***. Define a constant object and give it a meaningful name.

```
const int maxTableSize = 100;
std::vector<TableEntry> table(maxTableSize);
```

- If a program requires that certain ***constraints on user input*** be satisfied in order to work correctly, do not assume that these constraints will be satisfied. Instead, always check them.
- Always handle errors ***gracefully***.
- Provide ***useful*** error messages.
- Always ***check return codes***. Even if the operation/function theoretically cannot fail (under the assumption of bug-free code), in practice it may fail due to a bug.
- If an operation is performed that can fail, check the ***status of the operation*** to ensure that it did not fail (even if you think that it should not fail). For example, check for error conditions on streams.
- If a function can fail, always check its ***return value***.

Simplicity

- Do not ***unnecessarily complicate*** code. Use the simplest solution that will meet the needs of the problem at hand.
- Do not impose ***bogus limitations***. If a more general case can be handled without complicating the code and this more general case is likely to be helpful to handle, then handle this case.
- Do not ***unnecessarily optimize*** code. Highly optimized code is often much less readable. Also, highly optimized code is often more difficult to write correctly (i.e., without bugs). Do not write grossly inefficient code that is obviously going to cause performance problems, but do not optimize things beyond avoiding gross inefficiencies that you know will cause performance problems.

Code Duplication

- Avoid ***duplication*** of code. If similar code is needed is more than place, put the code in a function. Also, utilize templates to avoid code duplication.
- The avoidance of code duplication has many advantages.
  1. It simplifies code understanding. (Understand once, instead of n times.)
  2. It simplifies testing. (Test once, instead of n times.)
  3. It simplifies debugging. (Fix bugs in one place, instead of n places.)
  4. It simplifies code maintenance. (Change code in one place, instead of n places.)
- Make good use of the available ***libraries***. Do not reinvent the wheel. If a library provides code with the needed functionality, use the code in the library.

- Avoid ***multiple returns paths*** (i.e., multiple points of exit) in functions when they serve to complicate (rather than simplify) code structure.
- Avoid the use of ***global objects***. For example, use static data members instead of global objects. In well designed code, global objects are rarely needed.
- Ensure that the code is ***const correct***.
- If an object does not need to change, make it const. This improves the readability of code. This also helps to ensure const correctness of code.
- Avoid bringing many unknown identifiers into scope. For example, avoid constructs like:

```
using namespace std;
```

  Only bring identifiers into scope if you need them.

Miscellany

- Do not rely on ***undefined/unspecified/implementation-defined behavior***. Do not rely on any behavior that is not promised by the language. Do not rely on undocumented features of libraries. That is, do not write code in a way that it may only work on certain computing platforms or when the moon is full.
- Enable ***compiler warning messages***. Pay attention to warning messages issued by the compiler.
- Learn how to use a ***source-level debugger***. There will be times when you will absolutely need it.
- Be careful to avoid using references, pointers, iterators that do not reference valid data. Always be clear about which operations invalidate references, pointers, and iterators.

Testing: Preconditions and Postconditions

- **precondition**: condition that must be true before function is called
- for example, precondition for function that computes square root of x: $x \geq 0$
- **postcondition**: condition that must be true after function is called
- for example, postcondition for function that removes entry from table of size n: new size of table $n - 1$
- whenever feasible, check for violations of preconditions and postconditions for functions
- if precondition or postcondition is violated, terminate program immediately in order to help in localizing bug (e.g., by calling `std::abort` or `std::terminate`)

Testing

- The single most important thing when writing code is that it does the job it was intended to do ***correctly***. That is, there should not be any bugs.
- ***Test*** your code. If you do not spend as much time testing your code as you do writing it, you are likely not doing enough testing.
- Tests should exercise as much of the code as possible (i.e., provide good ***code coverage***).
- Design and structure your code so that it is easy to test. In other words, testing should be considered ***during design***.
- Your code will have bugs. Design your code so that it will help you to isolate bugs. Use ***assertions***. Use ***preconditions*** and ***postconditions***.
- Design your code so that is modular and can be written and tested ***in pieces***. The first testing of the software should never be testing the entire software as a whole.
- Often in order to adequately test code, one has to write separate ***specialized test code***.

Code Examples

- subscripting operator for 1-D array class:

```
template <class T>
const T& Array_1<T>::operator[](int i) const {
    // Precondition: index is in allowable range
    assert(i >= 0 && i < data_.size());
    return data_[i];
}
```

- function taking pointer parameter:

```
int stringLength(const char* ptr) {
    // Precondition: pointer is not null
    assert(ptr);
    // Code to compute and return string length.
    // ...
}
```

- function that modifies highly complicated data structure:

```
void modifyDataStructure(Type& dataStructure) {
    // Precondition: data structure is in valid state
    assert(isDataStructureValid(dataStructure));
    // Complicated code to update data structure.
    // ...
    // Postcondition: data structure is in valid state
    assert(isDataStructureValid(dataStructure));
}
```

Section 6.2

Algorithms

Software Performance

- two most basic performance measures, which are often of most interest:
  1. time complexity
  2. space complexity
- **time complexity**: amount of time required to execute code
- **space complexity**: amount of memory needed for code execution
- normally must consider both time and space complexities, since one type of complexity can often be traded off for other
- from practical standpoint, real-world time and memory usage are what matter most (as opposed to some approximate theoretical measures of code complexity)
- need techniques that can provide guidance when designing software so that more likely that later implementation (of design) will have acceptable performance
- many factors can potentially impact performance, including:
  - CPU instruction count
  - cache efficiency
  - degree of parallelism and concurrency
  - resource utilization (e.g., memory, disk, and network)

Random-Access Machine (RAM) Model

- algorithms can be measured in machine-independent way using random-access machine (RAM) model
- model assumes single processor
- instructions executed sequentially with no concurrent operations
- elementary types: integer and floating point numbers
- each elementary operation takes one time unit
- elementary operations include:
  - arithmetic operations (e.g., addition, subtraction, multiplication, division) on elementary types
  - loads and stores of elementary types
  - branch operations (e.g., conditional branch, jump)
  - subroutine call
- loops and subroutines are not considered elementary operations, but rather as composition of numerous elementary operations
- each memory access takes one time unit
- unbounded amount of memory available

Worst-Case, Average, and Amortized Complexity

- complexity expressed as function of input problem size
- **worst-case complexity**: gives upper bound on complexity of algorithm for any input of given size
- **average complexity**: gives average complexity of algorithm in statistical sense if probability measure assigned to all inputs of given size
- often algorithm may only approach worst-case complexity for very small fraction of possible inputs, in which case average complexity might be more practically useful than worst-case complexity
- sometimes algorithm may be invoked many times and cost of single invocation difficult to determine in isolation (e.g., time complexity of `push_back` member function of `std::vector`)
- **amortized complexity**: complexity per invocation of algorithm evaluated over sequence of invocations
- amortized complexity makes guarantee about total expense of sequence of invocations of algorithm, rather than single invocation (e.g., `push_back` member function of `std::vector` takes amortized constant time)

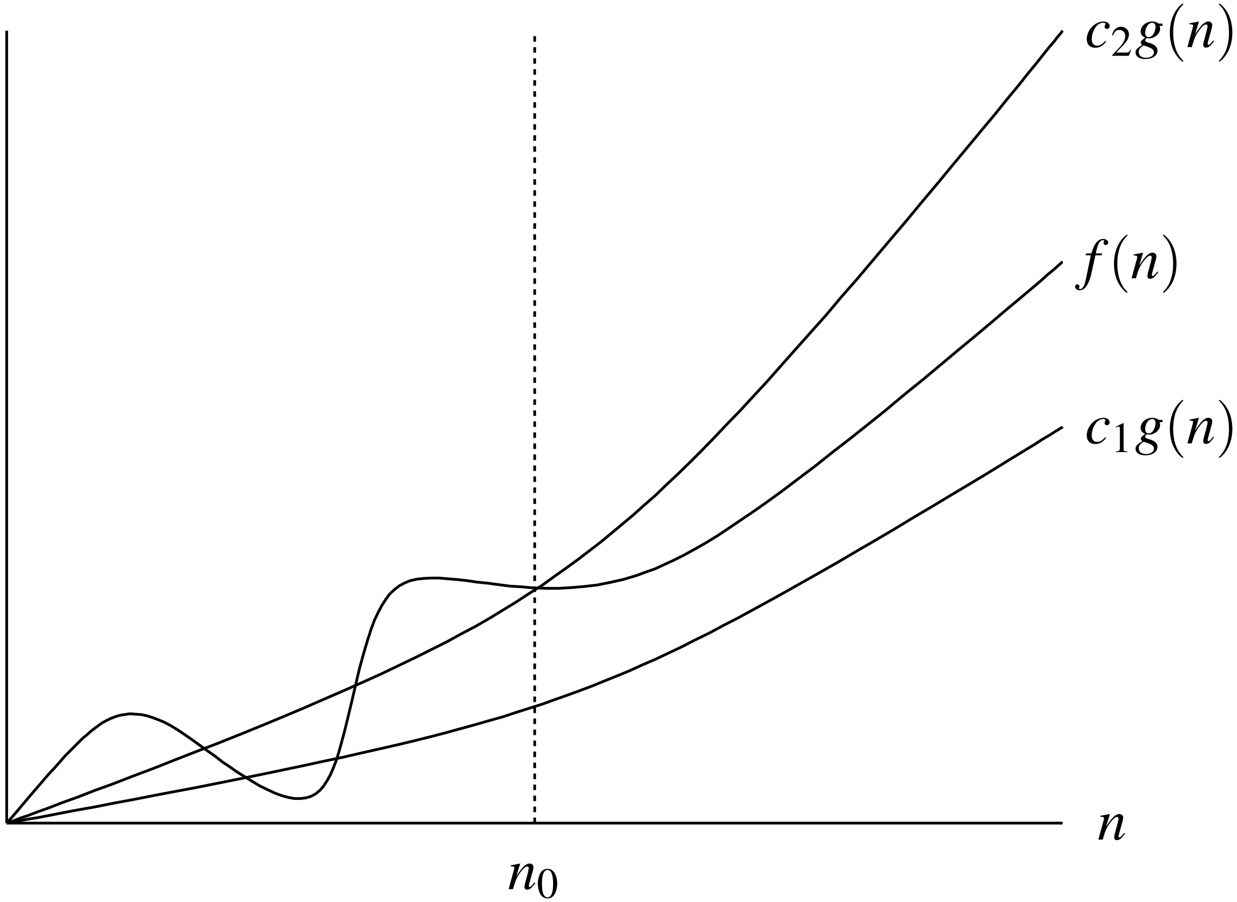
Asymptotic Analysis of Algorithms

- asymptotic analysis deals with behavior of algorithm as problem size becomes arbitrarily large
- **asymptotic complexity**: complexity of algorithm in limit as problem size becomes arbitrarily large
- often interested in:
  - asymptotic time complexity
  - asymptotic space complexity
- asymptotic time and space complexities of algorithm often much easier to determine than exact running time and memory usage
- often (but not always!) algorithm that is asymptotically more efficient will be best choice for all but very small inputs
- asymptotic notation (to be discussed next) provides way to describe functions that is very useful for asymptotic analysis

Big-Theta (Θ) Notation

- **big-theta (Θ) notation**: for function g, $\Theta(g)$ denotes set of all functions f for which positive constants c_1, c_2, and n_0 exist such that
$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \quad \text{for all } n \ge n_0$$
- functions in $\Theta(g)$ grow asymptotically at ***same*** rate as g (to within constant factor)
- effectively, $f(n)$ is sandwiched between $c_1 g(n)$ and $c_2 g(n)$ for sufficiently large n (i.e., $n \ge n_0$)
- used to provide (asymptotic) ***lower and upper bounds*** on function, each to within constant factor (provides asymptotically tight bound)
- if $f \in \Theta(g)$, then for sufficiently large n, $f(n)$ equals $g(n)$ to within constant factor
- examples:
  - $f(n) = an^2 + bn + c$ where a, b, c are constants and $a > 0$; $f \in \Theta(n^2)$ but $f \notin \Theta(n)$ and $f \notin \Theta(n^3)$
  - $f(n) = \sum_{i=0}^{d} a_i n^i$ where $\{a_i\}$ are constants and $a_d > 0$; $f \in \Theta(n^d)$ but $f \notin \Theta(n^{d+1})$ and $f \notin \Theta(n^{d-1})$

Big-Theta (Θ) Notation (Continued)

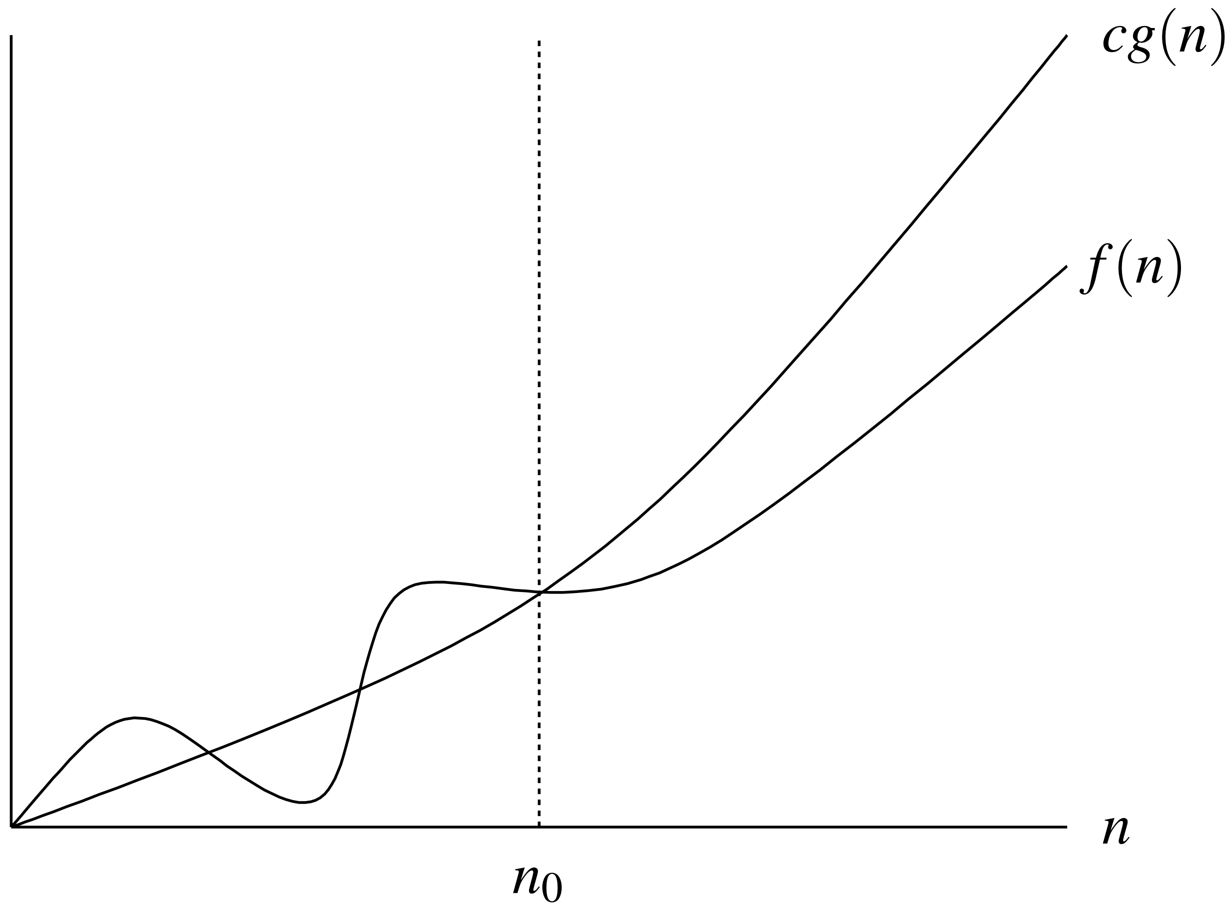


- $f \in \Theta(g)$
- for $n \geq n_0$, $f(n)$ is ***lower bounded*** by $c_1g(n)$ and ***upper bounded*** by $c_2g(n)$
- asymptotically, f grows at ***same*** rate as g to within constant factor

Big-Oh (O) Notation

- **big-oh (O) notation**: for function g, $O(g)$ denotes set of all functions f for which positive constants c and n_0 exist such that
$$0 \leq f(n) \leq cg(n) \quad \text{for all } n \geq n_0$$
- functions in $O(g)$ grow asymptotically at rate ***at most*** that of g (to within constant factor)
- used to provide (asymptotic) ***upper bound*** on function to within constant factor
- if $f \in O(g)$, then for sufficiently large n, $f(n)$ is less than or equal to $g(n)$ to within constant factor
- since $\Theta(g(n)) \subset O(g(n))$, $f(n) \in \Theta(g(n))$ implies $f(n) \in O(g(n))$
- often used to bound worst-case running time of algorithm
- examples:
  - $f(n) = 3n^2 + 2n + 1$; $f \in O(n^2)$ and $f \in O(n^3)$ but $f \notin O(n)$
  - $f(n) = 5n + 42$; $f \in O(n)$ and $f \in O(n^2)$ but $f \notin O(1)$
  - $f(n) = \sum_{i=0}^{d} a_i n^i$ where $\{a_i\}$ are constants and $a_d > 0$; $f \in O(n^d)$ and $f \in O(n^{d+1})$ but $f \notin O(n^{d-1})$

Big-Oh (O) Notation (Continued)



- $f \in O(g)$
- for $n \geq n_0$, $f(n)$ is ***upper bounded*** by $cg(n)$
- asymptotically, f grows at rate ***no greater than*** that of g to within constant factor

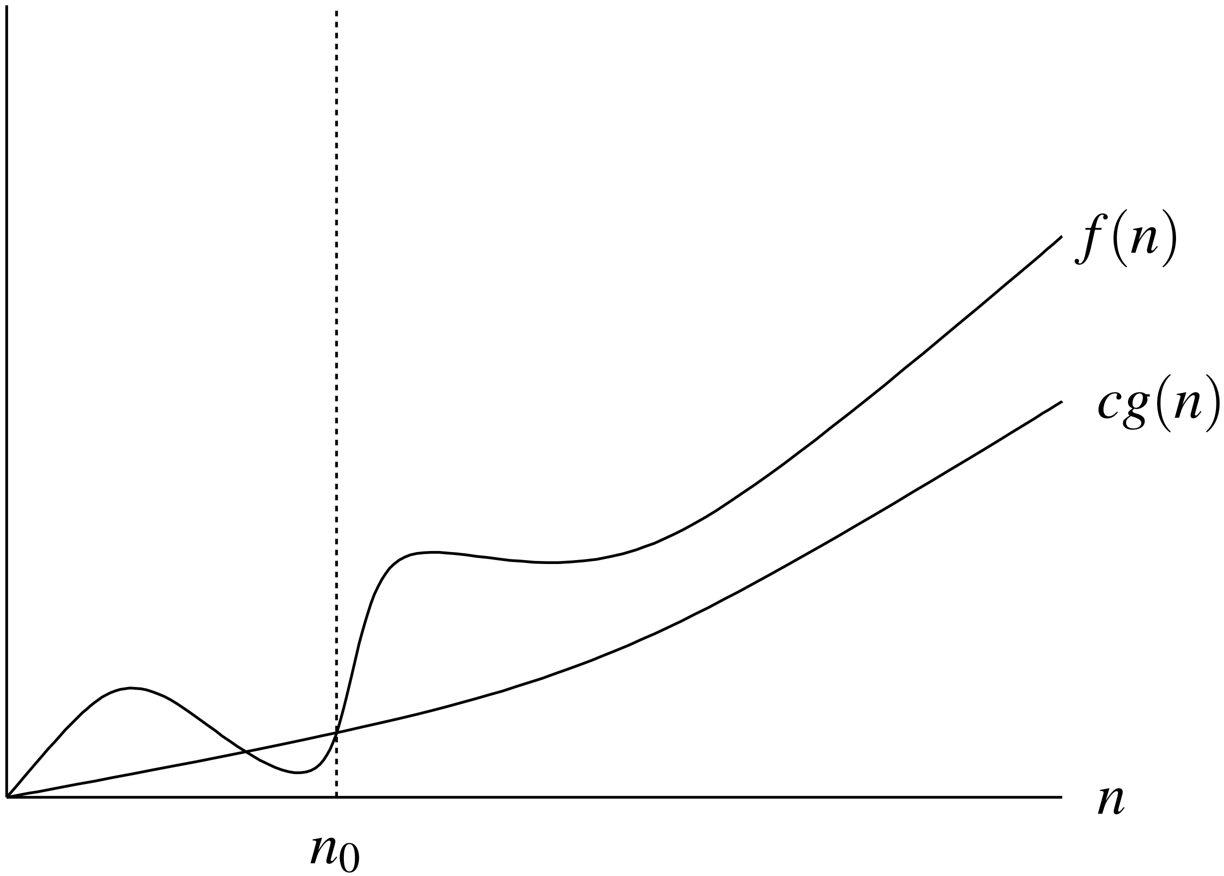
Big-Omega (Ω) Notation

- **big-omega (Ω) notation**: for function g, $\Omega(g)$ denotes set of all functions f for which positive constants c and n_0 exist such that

$$0 \leq cg(n) \leq f(n) \quad \text{for all } n \geq n_0$$

- functions in $\Omega(g)$ grow asymptotically at rate ***at least*** that of g (to within constant factor)
- used to provide (asymptotic) ***lower bound*** on function to within constant factor
- if $f \in \Omega(g)$, then for sufficiently large n, $f(n)$ is greater than or equal to $g(n)$ to within constant factor
- since $\Theta(g(n)) \subset \Omega(g(n))$, $f(n) \in \Theta(g(n))$ implies $f(n) \in \Omega(g(n))$
- examples:
  - $f(n) = 5n^3 + n$; $f \in \Omega(n^3)$ and $f \in \Omega(n^2)$ but $f \notin \Omega(n^4)$
  - $f(n) = an^2 + bn + c$ where a, b, c are constants and $a > 0$; $f \in \Omega(n^2)$ and $f \in \Omega(n)$ but $f \notin \Omega(n^3)$
  - $f(n) = \sum_{i=0}^{d} a_i n^i$ where $\{a_i\}$ are constants and $a_d > 0$; $f \in \Omega(n^d)$ and $f \in \Omega(n^{d-1})$ but $f \notin \Omega(n^{d+1})$

Big-Omega (Ω) Notation (Continued)



- $f \in \Omega(g)$
- for $n \geq n_0$, $f(n)$ ***lower bounded*** by $cg(n)$
- asymptotically, f grows at rate ***no less than*** that of g to within constant factor

- **small-oh (o) notation**: for function g, $o(g)$ denotes set of all functions f such that, for any positive constant c, positive constant n_0 exists such that

$$0 \le f(n) < cg(n) \quad \text{for all } n \ge n_0$$

- functions in $o(g)$ grow asymptotically at ***strictly lesser*** rate than g (to within constant factor)
- used to provide ***upper bound*** on function that is ***not asymptotically tight***
- $f \in o(g)$ implies that $f(n)$ becomes insignificant relative to $g(n)$ as n becomes arbitrarily large (i.e., $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$)
- examples:
  - $f(n) = 3n^3 + 2n + 1$; $f \in o(n^5)$ and $f \in o(n^4)$ but $f \notin o(n^3)$
  - $f(n) = 2n^2$; $f \notin o(n^2)$ but $f \in O(n^2)$
  - $f(n) = \sum_{i=0}^{d} a_i n^i$ where $\{a_i\}$ are constants and $a_d > 0$; $f \in o(n^{d+1})$ and $f \in o(n^{d+2})$ but $f \notin o(n^d)$ and $f \notin o(n^{d-1})$

Small-Omega (ω) Notation

- **small-omega (ω) notation**: for function g, $\omega(g)$ denotes set of all functions f such that, for any positive constant c, positive constant n_0 exists such that

$$0 \leq cg(n) < f(n) \quad \text{for all } n \geq n_0$$

- functions in $\omega(g)$ grow asymptotically at ***strictly greater*** rate than g (to within constant factor)
- used to provide ***lower bound*** on function that is ***not asymptotically tight***
- $f \in \omega(g)$ implies that $f(n)$ becomes arbitrarily large relative to $g(n)$ as n becomes arbitrarily large (i.e., $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$)
- examples:
  - $f(n) = 3n^2$; $f \in \omega(n)$ but $f \notin \omega(n^2)$
  - $f(n) = an^2 + bn + c$ where a, b, c are constants and $a > 0$; $f \in \omega(n)$ and $f \in \omega(1)$ but $f \notin \omega(n^2)$ and $f \notin \omega(n^3)$
  - $f(n) = \sum_{i=0}^{d} a_i n^i$ where $\{a_i\}$ are constants and $a_d > 0$; $f \in \omega(n^{d-1})$ but $f \notin \omega(n^d)$ and $f \notin \omega(n^{d+1})$

Asymptotic Notation in Equations and Inequalities

- when asymptotic notation stands alone on right-hand side of equation, equal sign means set membership
- for example:
  - $f(n) = \Theta(g(n))$ means $f(n) \in \Theta(g(n))$
- more generally, when asymptotic notation appears in formula, interpreted as placeholder for some anonymous function
- for example:
  - $3n^2 + 2n + 1 = 3n^2 + \Theta(n)$ means $3n^2 + 2n + 1 = 3n^2 + f(n)$ where $f(n)$ is some function in $\Theta(n)$ (i.e., $f(n) = 2n + 1 \in \Theta(n)$)
- using asymptotic notation in this way can help to reduce clutter in formulas

Properties of Θ, O, and Ω

- sum of functions:
  - if $f_1 \in \Theta(g)$ and $f_2 \in \Theta(g)$, then $f_1 + f_2 \in \Theta(g)$
  - if $f_1 \in O(g)$ and $f_2 \in O(g)$, then $f_1 + f_2 \in O(g)$
  - if $f_1 \in \Omega(g)$ and $f_2 \in \Omega(g)$, then $f_1 + f_2 \in \Omega(g)$
- multiplication by constant:
  - for all positive functions f and all positive constants a, $af \in \Theta(f)$, $af \in O(f)$, and $af \in \Omega(f)$
- product of functions:
  - for all positive functions f_1, f_2, g_1, g_2, if $f_1 \in \Theta(g_1)$ and $f_2 \in \Theta(g_2)$, then $f_1 f_2 \in \Theta(g_1 g_2)$
  - for all positive functions f_1, f_2, g_1, g_2, if $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, then $f_1 f_2 \in O(g_1 g_2)$
  - for all positive functions f_1, f_2, g_1, g_2, if $f_1 \in \Omega(g_1)$ and $f_2 \in \Omega(g_2)$, then $f_1 f_2 \in \Omega(g_1 g_2)$
- examples:
  - if $f \in \Theta(n)$, then $nf(n) \in \Theta(n^2)$
  - if f and g are positive functions in $\Theta(1)$, then $f + g \in \Theta(1)$

Additional Remarks

- $\log_2 n \in \Theta(\log_b n)$ for all $b > 1$ (i.e., base of logarithm does not impact asymptotic analysis)

Remarks on Asymptotic Complexity

- one must be careful in interpreting results of asymptotic complexity analysis
- asymptotic complexity only considers algorithm behavior when problem size becomes ***arbitrarily large***
- for example: for problems of size $n < 10^{10}$, algorithm A with time complexity $f(n) = \left(\frac{1}{10^{10}}\right) n^2$ will take less time than Algorithm B with time complexity $g(n) = n$, in spite of fact that $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n)$ (i.e., algorithm A has greater asymptotic complexity than algorithm B)
- asymptotic complexity ***hides constant factors***
- for example: for problems of size n, algorithm A with time complexity $f(n) = n$ is clearly preferable to algorithm B with time complexity $g(n) = 1000n$, but both f and g are in $\Theta(n)$ (i.e., both algorithms have same asymptotic complexity)
- asymptotic complexities can be used for guidance but should not be followed blindly

Some Common Complexities

Name	Complexity
constant	$O(1)$
logarithmic	$O(\log n)$
fractional power	$O(n^c)$, $c \in (0,1)$
linear	$O(n)$
log-linear	$O(n \log n)$
quadratic	$O(n^2)$
cubic	$O(n^3)$
exponential	$O(a^n)$
factorial	$O(n!)$
double exponential	$O(a^{b^n})$

- above complexities listed in order of ***increasing*** (asymptotic) growth rate
- that is, for sufficiently large n,
  $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < n! < 2^{2^n}$

- **recurrence relation** is equation that implicitly defines sequence in terms of itself
- for example, Fibonacci number sequence f is solution to recurrence relation:

$$f(n) = \begin{cases} f(n-1) + f(n-2) & n \geq 2 \\ 1 & n \in \{0, 1\} \end{cases}$$

- recurrence relations often arise when trying to determine complexity of algorithm that employs recursion
- for example, consider time complexity of recursive Fibonacci algorithm:

```
unsigned long long fibonacci(unsigned int n) {
    if (n <= 2) {
        return 1;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

- time complexity T of above algorithm leads to recurrence relation $T(n) = c + T(n-1) + T(n-2)$

Solving Recurrence Relations

- no known general technique for solving recurrence relations
- solving recurrence relations somewhat of an art
- linear constant coefficient difference equations can be solved using z transform
- Master theorem can be used to solve some recurrence relations of form:
  $$f(n) = g(n) + af(n/b)$$
- Akra-Bazzi theorem can be used to solve some recurrence relations of form:
  $$f(n) = g(n) + \sum_{i=0}^{L-1} a_i f(b_i n + h_i(n))$$
- need to be careful about non-integer sequence indices arising in recurrence relations like:
  $$T(n) = \sum_{i=0}^{L-1} a_i T(n/b_i) + f(n)$$
- preceding formula does not make sense if n/b_i is not integer
- in many cases, if this issue ignored, correct asymptotic bound still obtained, although without being correctly justified
- numerous software tools available for solving recurrence relations, such as WolframAlpha and PURRS

Solutions for Some Common Recurrence Relations

Recurrence Relation	Solution
$f(n) = \begin{cases} b + f(n-1) & n \geq 2 \\ a & n = 1 \end{cases}$	$f(n) = b(n-1) + a \in \Theta(n)$
$f(n) = \begin{cases} bn + f(n-1) & n \geq 2 \\ a & n = 1 \end{cases}$	$f(n) = \frac{1}{2}bn(n+1) + b - a \in \Theta(n^2)$
$f(n) = \begin{cases} b + f(\lfloor n/2 \rfloor) & n \geq 2 \\ a & n = 1 \end{cases}$	$f(n) \in \Theta(\log n)$
$f(n) = \begin{cases} b + f(\lceil n/2 \rceil) & n \geq 2 \\ a & n = 1 \end{cases}$	$f(n) \in \Theta(\log n)$
$f(n) = \begin{cases} b + f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) & n \geq 2 \\ a & n = 1 \end{cases}$	$f(n) \in \Theta(n)$
$f(n) = \begin{cases} bn + f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) & n \geq 2 \\ a & n = 1 \end{cases}$	$f(n) \in \Theta(n \log n)$
$f(n) = \begin{cases} c + f(n-1) + f(n-2) & n \geq 3 \\ b & n = 2 \\ a & n = 1 \end{cases}$	$f \in \Theta(2^n)$

Matrix Multiplication Algorithm: Time Complexity

- consider algorithm for multiplying $m \times n$ matrix by $n \times p$ matrix:

```
1  template <class T, int m, int n, int p>
2  void multiply(const T (&a)[m][n], const T (&b)[n][p],
3    T (&c)[m][p]) {
4      for (int i = 0; i < m; ++i) {
5          for (int j = 0; j < p; ++j) {
6              T sum = T(0);
7              for (int k = 0; k < n; ++k) {
8                  sum += a[i][k] * b[k][j];
9              }
10             c[i][j] = sum;
11         }
12     }
13 }
```

- total time cost per line (assuming basic operations on `T` are $O(1)$):

Line	Total Time Cost
4	$c_{4,1}m + c_{4,2}$
5	$m(c_{5,1}p + c_{5,2})$
6	$mp(c_6)$
7	$mp(c_{7,1}n + c_{7,2})$
8	$mpn(c_8)$
10	$mp(c_{10})$

- asymptotic time complexity is $a_1 mnp + a_2 mp + a_3 m + a_4 = \Theta(mnp)$

Matrix Multiplication Algorithm: Space Complexity

- again, consider algorithm for multiplying $m \times n$ matrix by $n \times p$ matrix:

```
template <class T, int m, int n, int p>
void multiply(const T (&a)[m][n], const T (&b)[n][p],
  T (&c)[m][p]) {
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < p; ++j) {
            T sum = T(0);
            for (int k = 0; k < n; ++k) {
                sum += a[i][k] * b[k][j];
            }
            c[i][j] = sum;
        }
    }
}
```

- `a`, `b`, and `c` are references and each effectively incur memory cost of pointer
- `m`, `n`, and `p` are constant expressions are require no storage
- assuming objects of type `T` require $O(1)$ space, each of `a`, `b`, `c`, `i`, `j`, `k`, and `sum`, require $\Theta(1)$ space
- asymptotic space complexity is $\Theta(1)$

Iterative Fibonacci Algorithm: Time Complexity

- consider iterative algorithm for computing nth Fibonacci number:

```
1 unsigned long long fibonacci(unsigned int n) {
2     unsigned long long a[3] = {1, 1, 1};
3     for (int i = 3; i <= n; ++i) {
4         a[0] = a[1];
5         a[1] = a[2];
6         a[2] = a[0] + a[1];
7     }
8     return a[2];
9 }
```

- total time cost per line:

Line	Total Time Cost
2	c_1
3	$(n-2)c_{3,1}+c_{3,2}$
4	$(n-2)c_4$
5	$(n-2)c_5$
6	$(n-2)c_6$
8	c_8

- asymptotic time complexity is $a_1 n + a_2 = \Theta(n)$

Iterative Fibonacci Algorithm: Space Complexity

- again, consider iterative algorithm for computing nth Fibonacci number:

```
unsigned long long fibonacci(unsigned int n) {
    unsigned long long a[3] = {1, 1, 1};
    for (int i = 3; i <= n; ++i) {
        a[0] = a[1];
        a[1] = a[2];
        a[2] = a[0] + a[1];
    }
    return a[2];
}
```

- storage cost per variable:

Variable	Storage Cost
`n`	c_1
`a`	c_2
`i`	c_3

- asymptotic space complexity is $c_1 + c_2 + c_3 = a_2 = \Theta(1)$

Recursive Fibonacci Algorithm: Time Complexity

- consider recursive algorithm for computing nth Fibonacci number:

```
unsigned long long fibonacci(unsigned int n) {
    if (n <= 2) {
        return 1;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

- time cost $T(n)$ satisfies recurrence relation:

$$T(n) = \begin{cases} T(n-1) + T(n-2) + c_1 & n \geq 3 \\ c_2 & n \in \{1, 2\} \end{cases}$$

- asymptotic time complexity is $\Theta(2^n)$

Recursive Fibonacci Algorithm: Space Complexity

- again, consider recursive algorithm for computing nth Fibonacci number:

```
unsigned long long fibonacci(unsigned int n) {
    if (n <= 2) {
        return 1;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

- during recursion, function calls nest to depth of at most $n - 2 = \Theta(n)$
- each invocation of function incurs memory cost for local variable `n`
- each function call also incurs space on stack for return address and possibly other saved state
- asymptotic space complexity S is $S(n) = (n-2)c_1 + c_0 = a_1 n + a_0 = \Theta(n)$

Amdahl's Law

- may want to determine overall speedup that can be achieved by introducing speedup into some part of task
- overall speedup s_{o} of whole task given by

$$s_{\mathrm{o}} = \frac{1}{(1-f_{\mathrm{e}}) + \frac{f_{\mathrm{e}}}{s_{\mathrm{e}}}},$$

  where s_{e} is speedup of part of task that benefits from enhancement and f_{e} is fraction of time consumed by part of task benefitting from enhancement
- preceding result known as **Amdahl's law**
- overall speedup is limited by fraction of time that enhancement can be exploited:

$$s_{\mathrm{o}} \leq \frac{1}{1-f_{\mathrm{e}}} \quad \text{and} \quad \lim_{s_{\mathrm{e}}\to\infty} s_{\mathrm{o}} = \frac{1}{1-f_{\mathrm{e}}}$$

- for example, if $f_e = 25\%$ and $s_e = 2$, then $s_o = 1.1429$

Section 6.2.1

References

References I

1. WolframAlpha Recurrence Relation Solver, https://www.wolframalpha.com/examples/Recurrences.html.
2. Parma University's Recurrence Relation Solver (PURRS), http://www.cs.unipr.it/purrs.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.*
   MIT Press, Cambridge, MA, USA, 3rd edition, 2009.
4. A. Levitin. *Introduction to the Design and Analysis of Algorithms.*
   Pearson, Boston, MA, USA, 3rd edition, 2012.
5. J. Erickson. Algorithms, Jan. 2015.
   Available online from http://www.cs.illinois.edu/~jeffe/teaching/algorithms/.
6. M. Akra and L. Bazzi. On the solution of linear recurrence equations.
   Computational Optimization and Applications, 10(2):195–210, May 1998.

References II

7. M. Drmota and W. Szpankowski. A master theorem for discrete divide and conquer recurrences.
In *Proc. of ACM-SIAM Symposium on Discrete Algorithms*, pages 342–361, 2011.

8. S. Roura. Improved master theorems for divide-and-conquer recurrences.
Communications of the ACM, 48(2):170–205, Mar. 2001.

Section 6.3

Data Structures

Abstract Data Types (ADTs)

- **abstract data type (ADT)** is model for data type where behavior specified from point of view of user of type (i.e., with implementation details hidden)
- ADT specifies:
  - general nature of entity represented by type
  - set of allowable states/values that type can assume
  - set of operations that can be performed on type
  - any preconditions or postconditions for operations
- often, ADT also provides complexity guarantees (e.g., time or space complexity guarantees for various operations)
- for example, (generic) integer type is ADT:
  - can assume integer values
  - provides basic arithmetic operations, relational operations, and so on
  - particular representation used for integers not specified by ADT
- in contrast to ADT, concrete (i.e., non-abstract) data type provides very specific details as to how type is implemented

Container ADTs

- **container ADT** (also called **collection ADT**): stores collection of objects, organized in way that follows some specific access rules
- operations for container ADT often include:
  - clear: remove all elements from container
  - is empty: test if container is empty (i.e., contains no elements)
  - size: query number of elements in container
  - insert: insert element in container
  - remove: remove element from container
  - find: locate element in container if present
- often container ADT provides means to traverse elements in container (e.g., via iterator ADT)
- if elements in container consist of key-value pairs where key used to find corresponding value in container, container said to be **associative**
- if elements in container have well-defined order, container said to be **ordered**; otherwise, **unordered**
- if all elements stored in container of same type, container said to be **homogeneous**; otherwise, **heterogeneous**

Container ADTs (Continued)

- examples of realizations of container ADTs:
  - `std::array`, `std::vector`, `std::list`, `std::forward_list`
  - `std::set`, `std::multiset`, `std::map`, `std::multimap`
  - `std::unordered_set`, `std::unordered_multiset`, `std::unordered_map`, `std::unordered_multimap`
  - `boost::intrusive::slist`, `boost::intrusive::list`
- container ADTs can differ in many ways:
  - number of elements container can store (e.g., one versus multiple)
  - whether values stored by container must be unique
  - associative versus non-associative
  - ordered versus unordered
  - homogeneous versus heterogeneous
  - intrusive versus nonintrusive
  - concurrency properties (e.g., not thread safe, thread safe, lock free)

Iterator ADTs

- **iterator ADT** is ADT used to traverse collection of elements, which are often stored in container
- typically iterator ADT provided as part of container ADT
- operations provided by iterator ADT may include:
  - dereference: access element to which iterator refers
  - next: go to next element
  - previous: go to previous element
  - advance: advance by n elements (where n can be negative for backwards direction)
- iterator specifies order in which elements can be accessed; for example:
  - forward, bidirectional (i.e., forward and backward), random access
- iterator may only permit certain types of element access; for example:
  - read only (const), read and write (non-const), write only (output)
  - one dereference per element or multiple dereferences per element
- examples of realizations of iterator ADT:
  - `iterator` and `const_iterator` types in numerous C++ standard library containers, such as `std::vector` and `std::set`

Container and Iterator Considerations

- are elements in container stored contiguously in memory?
- what is fixed storage overhead of container (if any)?
- what is per-element storage overhead of container (if any)?
- is container limited in size (e.g., container based on fixed size array)?
- is container dynamic (i.e., can it be changed once created) or static?
- can element be inserted at start, end, or arbitrary position in container in worst-case or amortized $O(1)$ time?
- can element be removed at start, end, or arbitrary position in container in $O(1)$ time?
- can element be accessed at start, end, or arbitrary position in $O(1)$ time?
- can element be located in container efficiently (e.g., $O(\log n)$ time or better)?

Container and Iterator Considerations (Continued)

- can container be traversed (e.g., via iterator) efficiently?
- what is storage cost of iterator (e.g., 1 pointer)?
- in what order can iterator access elements (e.g., forward, bidirectional, random access)?
- what circumstances result in element references (e.g., pointers, references, iterators) being invalidated?
- what is per-element and amortized time cost of traversing elements in container?

Section 6.3.1

Lists, Stacks, and Queues

List ADT

- **list ADT** is ADT that stores countable number of ordered values, where same value may occur more than once
- operations for list ADT include:
  - clear: remove all elements from list
  - is empty: test if list empty
  - size: query number of elements in list
  - insert: insert element in list
  - remove: remove element from list
- operations for traversing elements in list (which are often provided via iterator ADT) include:
  - successor: get next element in list
  - predecessor (optional): get previous element in list
- examples of realizations of list ADT:
  - `std::vector`, `std::forward_list`, and `std::list`
  - `boost::intrusive::slist` and `boost::intrusive::list`

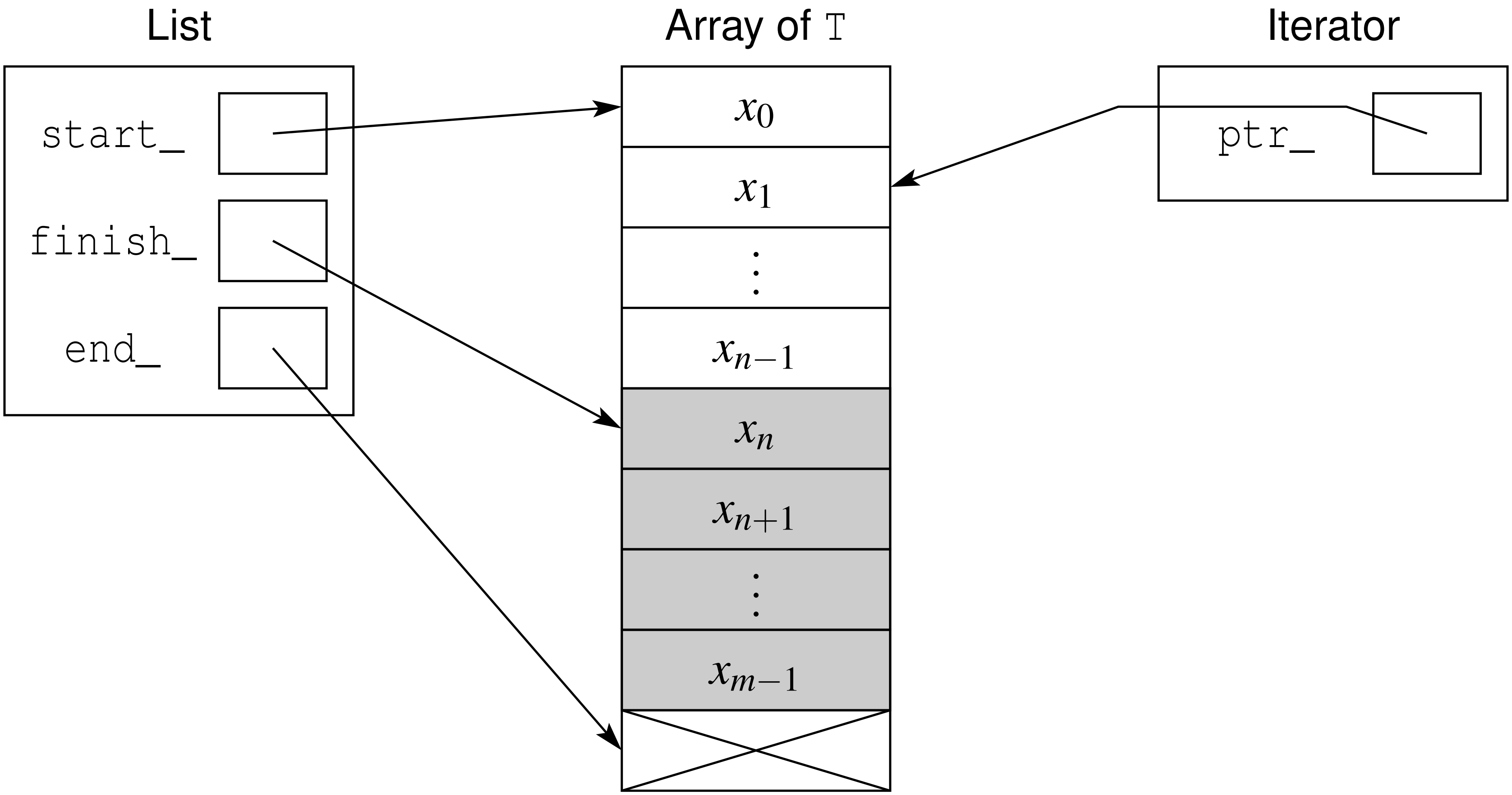
Array-Based Lists

- can represent list with array
- code example:

```
template <class T> class Iterator {
    // ...
    T* ptr_; // pointer to referenced element
};

template <class T> class List {
    // ...
    T* start_; // pointer to start of element data
    T* finish_; // pointer to end of element data
    T* end_; // pointer to end of allocated storage
};
```

- array capacity (i.e., allocated size) is `end_ - start_`
- array size (i.e., number of elements) is `finish_ - start_`

Array-Based Lists: Diagram



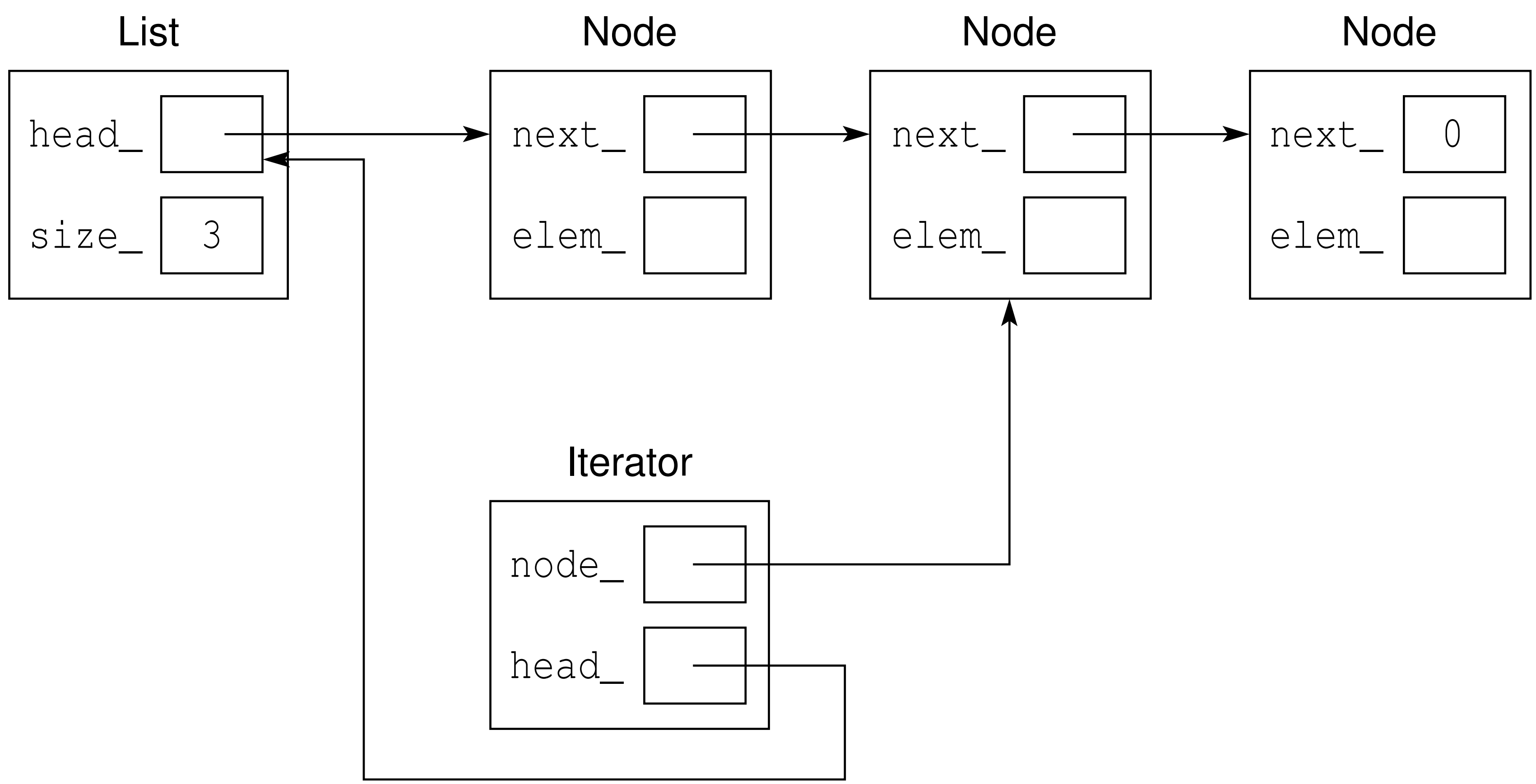
Remarks on Array-Based Lists

- advantages:
  - elements stored contiguously in memory (which is cache friendly)
  - no per-element storage overhead
  - can insert at end of list in amortized $O(1)$ time
  - can remove at end of list in $O(1)$ time
  - can access element in any position in $O(1)$ time
  - (random-access) iterator has storage cost of one pointer
- disadvantages:
  - cannot insert or remove at start or arbitrary position in $O(1)$ time
  - if capacity of array exceeded, memory reallocation and copying required
  - if array can be reallocated, insert at end can only at best guarantee amortized (not worst-case) $O(1)$ time
  - if array reallocated, element references invalidated
- useful when insertion and removal only performed at end of list and stable references to elements not needed

- **singly-linked list** is node-based implementation of list where each node tracks its successor (but not predecessor)
- null pointer used as sentinel value to denote "no such node"; for example, null pointer used to indicate:
  - no successor node for last node in list
  - no head (i.e., first) node for empty list
- for singly-linked list, insertion and removal normally defined to take place at position ***after*** that specified by iterator
- to specify insertion or removal at start of list requires "before-begin" iterator

```
// list node
template <class T> struct Node {
    Node* next_; // pointer to next node in list
    T elem_; // element data
};

// list
template <class T> class List {
    // ...
    Node<T>* head_; // pointer to first node in list
    std::size_t size_; // number of elements in list
};

// iterator
template <class T> class Iterator {
    // ...
    Node<T>* node_; // pointer to node with referenced element
    Node<T>** head_;
      // if before begin, pointer to list head pointer
      // otherwise, null
};
```

Singly-Linked List: Diagram



Remarks on Singly-Linked Lists

- advantages:
  - can insert element after (but not before) particular position in $O(1)$ time
  - can remove element at start of list in $O(1)$ time
  - no capacity exceeded problem like with array
  - reduced memory cost relative to doubly-linked list as consequence of node not tracking predecessor
  - element references are stable
  - can find successor in list in $O(1)$ time
- disadvantages:
  - element data not contiguous in memory
  - has per-element storage overhead (1 pointer for successor)
  - cannot insert element before particular position in $O(1)$ time
  - cannot remove element at arbitrary position in $O(1)$ time
  - cannot efficiently iterate backwards over elements in list
  - cannot find predecessor in list in $O(1)$ time
  - (forward) iterator requires two pointers for state (due to need for "before-begin" iterator)
- typically useful when insertions and removals always performed at start of list

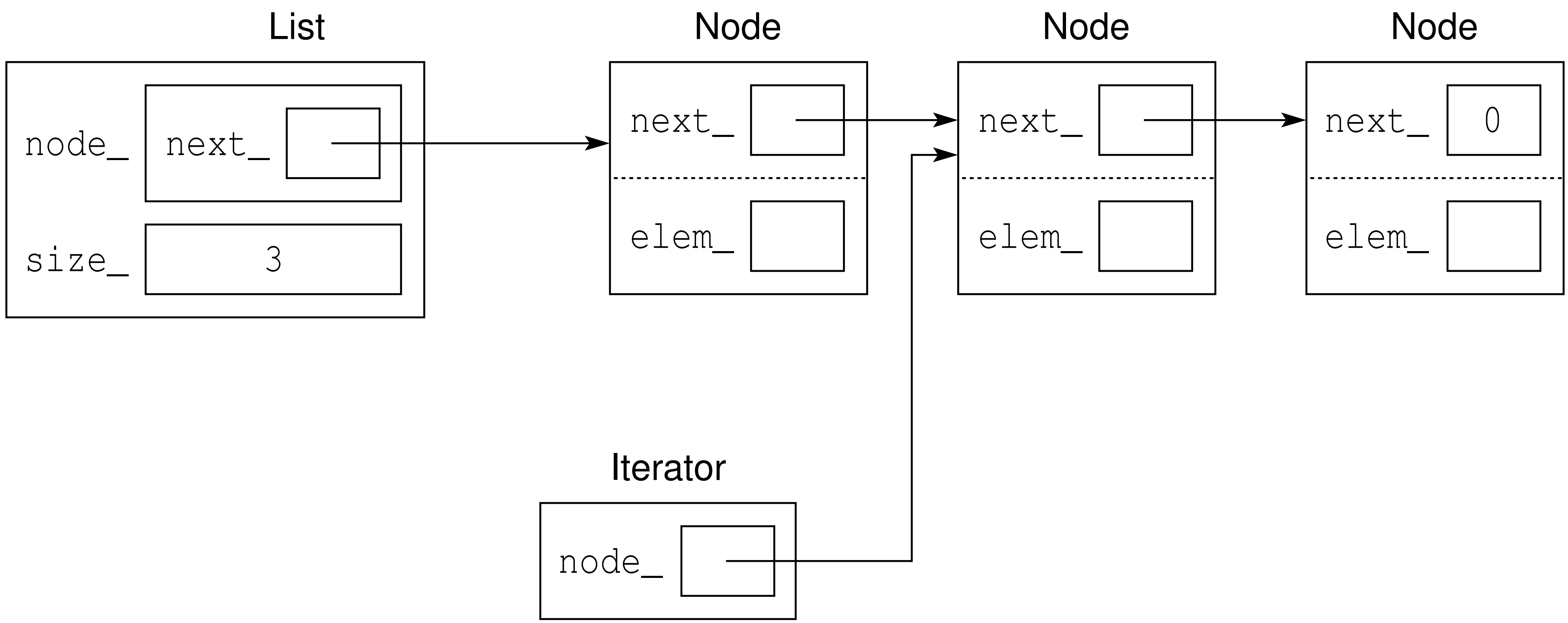
Singly-Linked List With Header Node

- **singly-linked list with header node** is node-based implementation of list where each node tracks its successor (but not predecessor)
- null pointer used as sentinel value to denote “no such node”; for example, null pointer used to indicate:
  - no successor for last node in list
  - no head node for empty list
- header node used as placeholder for one-before start of list (i.e., “before-begin” position)

Singly-Linked List With Header Node: Code

```
// list node base class
struct node_base {
    // ...
    node_base* next_;
};

// list node derived class (with list element)
template <class T> struct node : public node_base {
    T elem_;
};

// list iterator class
template <class T> class slist_iter {
    // ...
    node_base* node_;
};

// list class
template <class T> class list {
    // ...
    node_base node_;
    std::size_t size_;
};
```

Singly-Linked List With Header Node: Diagram



- advantages and disadvantages mostly similar to those of classic singly-linked list
- effectively no memory cost for header node over standard singly-linked list
- use of header node facilitates more efficient iterator type
- in absence of header node, special representation of before-begin iterator needed
- this causes problems for efficient implementation of forward iterator
- use of header node avoids this problem
- (forward) iterator can be implemented with single pointer as state
- typically, singly-linked list with header node used to implement `std::forward_list`

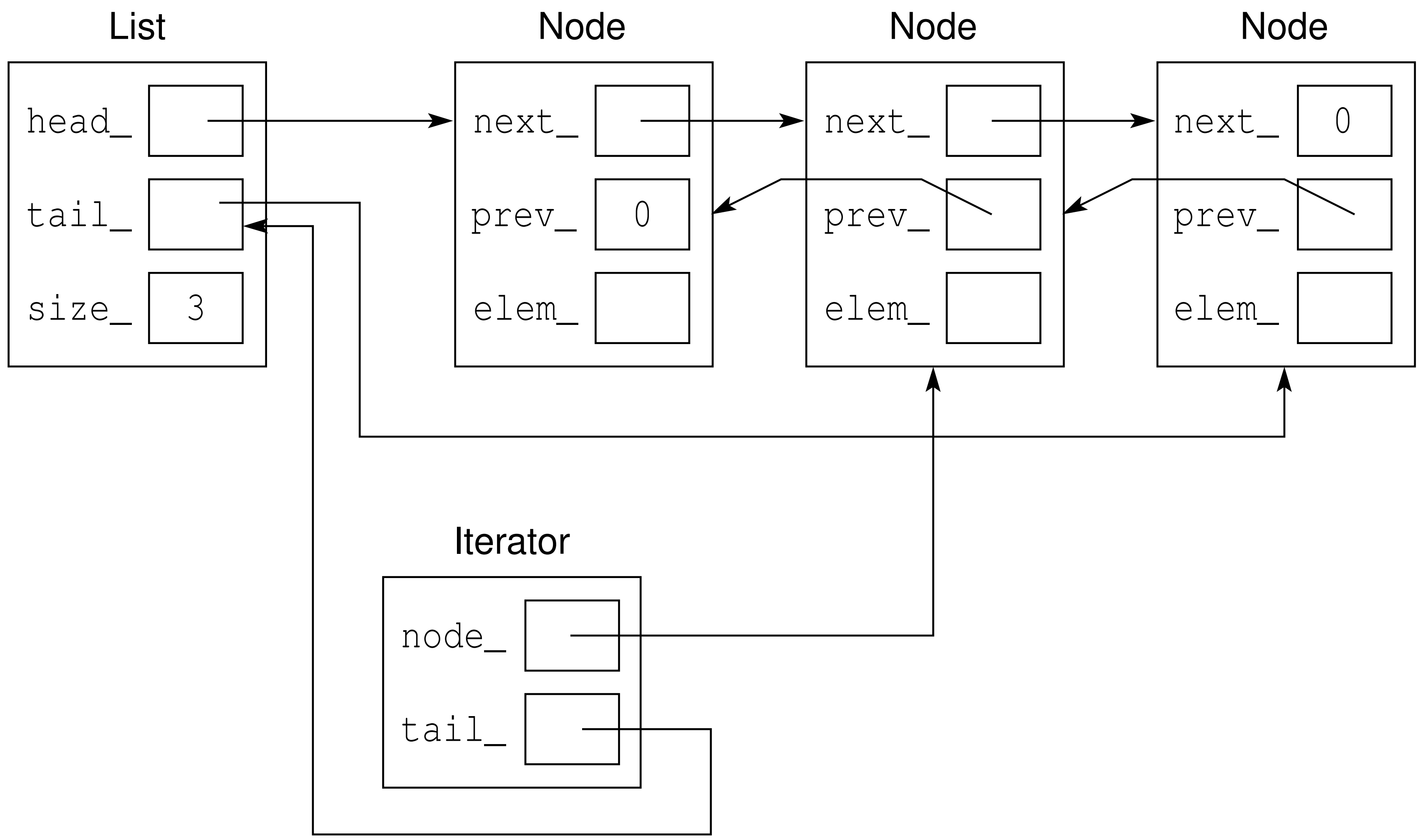
Doubly-Linked Lists

- **doubly-linked list**: node-based implementation of list where each node tracks both its successor and predecessor
- null pointer used as sentinel value to indicate "no such node"; for example, null pointer used to indicate:
  - no successor for last node in list
  - no predecessor for first node in list
  - no head or tail node for empty list

Doubly-Linked Lists: Code

```
// list node class
template <class T> struct Node {
    Node* next_; // pointer to next node in list
    Node* prev_; // pointer to previous node in list
    T elem_; // element
};

// iterator class
template <class T> class Iterator {
    // ...
    Node<T>* node_; // node of referenced element
    Node<T>** tail_; // pointer to tail pointer of list
};

// list class
template <class T> class List {
    // ...
    Node<T>* head__; // pointer to first node in list
    Node<T>* tail_; // pointer to last node in list
    std::size_t size_; // number of elements in list
};
```

Doubly-Linked List: Diagram



Remarks on Doubly-Linked Lists

- advantages:
  - stable references to elements
  - can insert or remove at arbitrary position in $O(1)$ time
  - no capacity-exceeded problem like in array case
  - can find successor and predecessor in $O(1)$ time
  - can efficiently iterate both forwards and backwards over elements in list
- disadvantages:
  - elements not stored contiguously in memory
  - per-element storage overhead (2 pointers)
  - relative to singly-linked list, has greater per-element storage overhead (1 additional pointer for predecessor)
  - iterator storage cost is more than single pointer (i.e., 2 pointers)
- most useful for lists where insertion and removal can happen anywhere in list

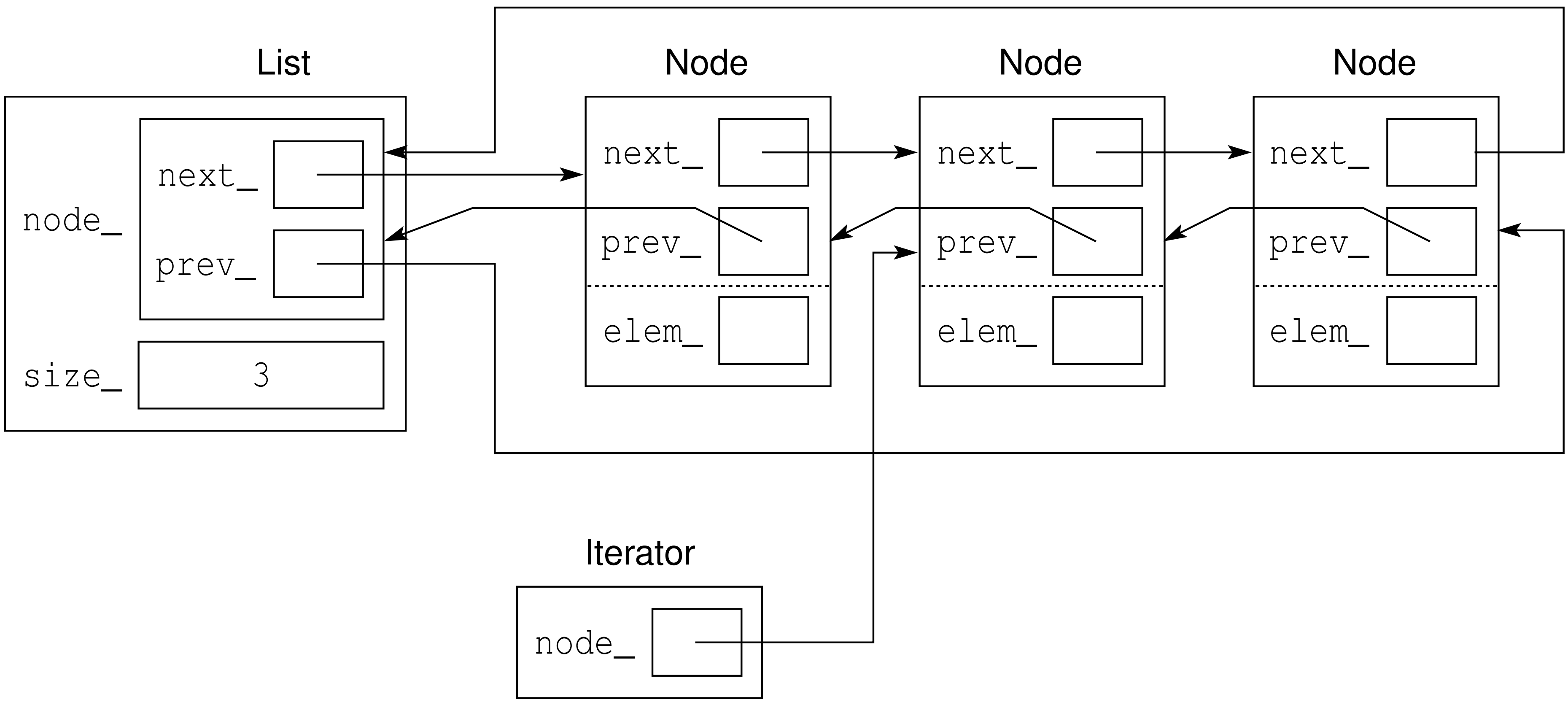
Doubly-Linked List With Sentinel Node

- list has one dummy node called **sentinel node** and zero or more regular (i.e., non-sentinel) nodes
- list object itself has sentinel node as member
- each regular node is associated with list element
- sentinel node is not associated with any list element
- each (regular and sentinel) node has pointer to its successor and predecessor
- if list not empty, successor of sentinel node is node corresponding to first element in list; otherwise, successor is sentinel node itself
- if list not empty, predecessor of sentinel node is node corresponding to last element in list; otherwise, predecessor is sentinel node itself
- thus, sentinel and regular nodes effectively form augmented list that is ***circular***
- augmented list never empty, since always contains sentinel node
- augmented list has has no beginning or end, since circular
- using sentinel node eliminates many special cases for insertion and removal, which leads to simpler and more efficient code

Doubly-Linked List With Sentinel Node: Code

```
// list node base class (which does not have element data)
struct Node_base {
    Node_base* next_; // pointer to next node in list
    Node_base* prev_; // pointer to previous node in list
};

// list node (which has element data)
template <class T> struct Node : public Node_base {
    T elem_; // element data
};

// list
template <class T> class List {
    // ...
    Node_base node_; // sentinel node
};

// list iterator
template <class T> class Iterator {
    // ...
    Node_base* node_; // pointer to referenced node
};
```

Doubly-Linked List With Sentinel Node: Diagram



Remarks on Doubly-Linked Lists With Sentinel Node

- advantages and disadvantages mostly similar to those of classic doubly-linked list
- effectively no memory cost for sentinel node over standard doubly-linked list
- sentinel node effectively makes list circular and always nonempty
- sentinel node eliminates special cases caused by empty list and insertion and removal at start and end of list (simplifying code)
- use of sentinel node facilitates more efficient iterator type
- in absence of sentinel node, null pointer would need to be used to indicate end of list
- this causes problems for efficient implementation of bidirectional iterator (namely, consider predecessor operation for iterator that refers to end of list)
- use of sentinel node avoids this problem
- (bidirectional) iterator can be implemented with single pointer as state
- typically, doubly-linked list with sentinel node used to implement `std::list`

Stack ADT

- **stack ADT** is ADT for container where elements can only be inserted or removed in last-in first-out (LIFO) order
- can only insert and remove elements at top of stack
- operations provided by stack ADT:
  - clear: remove all elements from stack
  - is empty: test if stack is empty
  - top: access element at top of stack (without removing)
  - push: add element to top of stack
  - pop: remove element from top of stack
- **stack overflow**: attempting to perform push operation when insufficient space available for element being added
- **stack underflow**: attempting to perform pop operation when stack empty
- example realizations of stack ADT:
  - `std::stack`
  - `boost::lockfree::stack`

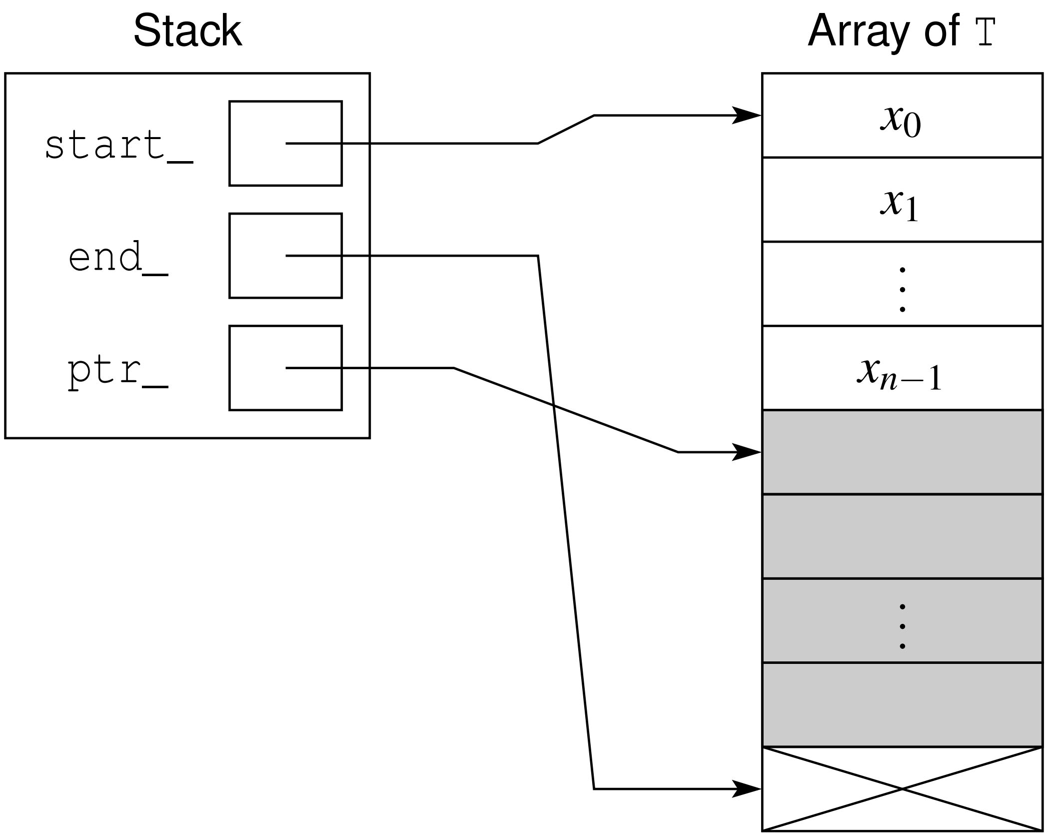
Array Implementation of Stack

- stack can be efficiently implemented using array
- code example:

```
template <class T> class Stack {
    // ...
    T* start_; // pointer to start of element storage
    T* end_; // pointer to end of element storage
    T* ptr_; // pointer to next free slot on stack
};
```

- stack empty if `ptr_` equals `start_`
- stack has reached capacity if `ptr_` equals `end_`
- push operation stores element at `*ptr_` and then increments `ptr_`
- pop operation decrements `ptr_`
- top operation provides access to `ptr_[-1]`
- due to possibility of exceeding array capacity, cannot guarantee each push operation takes constant time; can only hope for amortized (not worst-case) $O(1)$ time
- memory efficient: only per-element storage cost is element data itself
- cache-efficient: element data is contiguous in memory

Array Implementation of Stack: Diagram



Remarks on Array Implementation of Stack

- advantages:
  - elements stored contiguously in memory
  - no per-element storage overhead
- disadvantages:
  - if capacity of array exceeded, must reallocate and copy
  - if array grown, can only guarantee amortized (not worst-case) $O(1)$ time for push
  - if array reallocated, elements references are invalidated

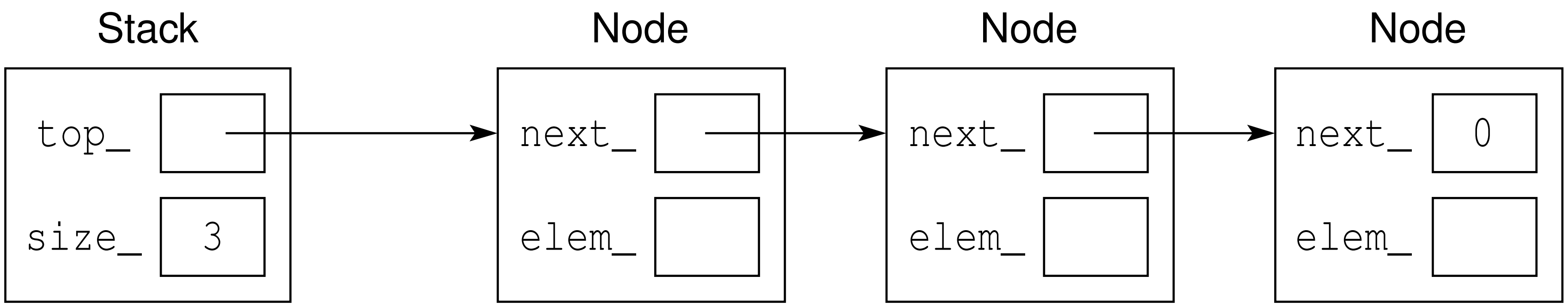
Node-Based Implementation of Stack

- stack can be efficiently implemented using node-based ***singly-linked*** list
- code example:

```
// stack node
template <class T> struct Node {
    Node* next_; // pointer to next node in stack
    T elem_; // element data
};

// stack
template <class T> class Stack {
    // ...
    Node<T>* top_; // pointer to node at top of stack
};
```

- only need list to be singly linked (as opposed to doubly linked), since all insertions and removals performed at start of list (i.e., top of stack)

Node-Based Implementation of Stack: Diagram



Remarks on Node-Based Implementation of Stack

- advantages:
  - no capacity-exceeded problem as in array case
  - can perform push operation in $O(1)$ time in worst case
  - element references are stable
- disadvantages:
  - element data not contiguous in memory
  - has per-element storage overhead (i.e., 1 pointer for successor)
  - relative to array-based implementation, requires more space

Queue ADT

- **queue ADT** is container where elements can only be inserted and removed in first-in first-out (FIFO) order
- elements removed from front (a.k.a. head) of queue
- elements inserted at back (a.k.a. tail) of queue
- operations for queue ADT include:
  - clear: remove all elements from queue
  - is empty: test if queue is empty
  - front: access element at front of queue (without removing)
  - enqueue: insert element at back of queue
  - dequeue: remove element from front of queue
- examples of realizations of queue ADT:
  - `std::queue`
  - `boost::lockfree::queue`
- double-ended queue ADT is similar to queue ADT except allows elements to be inserted or removed at either front or back

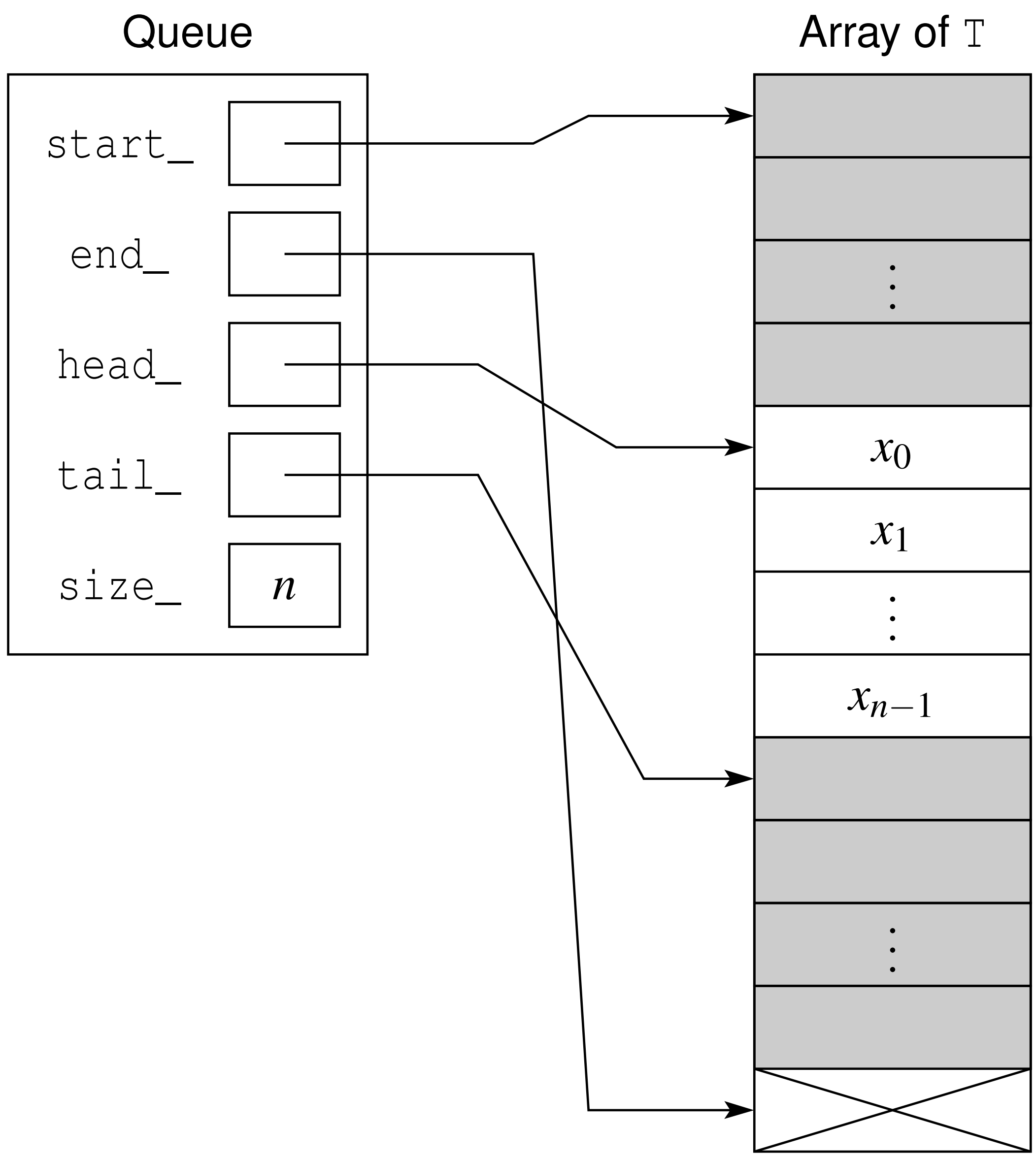
Array Implementation of Queue

- array implementation of bounded queue
- code example:

```
// bounded queue
template <class T> Queue {
    // ...
    T* start_; // start of array for queue elements
    T* end_; // end of array for queue elements
    T* head_; // pointer to element at front of queue
    T* tail_; // pointer to back of queue
    std::size_t size_; // number of entries in queue
};
```

- array used in circular fashion
- queue is empty if `size_` is zero
- queue is full if `size_` equals `max_size`
- if queue not full, enqueue operation places element at `tail_` and then increments `tail_` with wraparound and increments `size_`
- if queue not empty, dequeue operation increments `head_` with wraparound and decrements `size_`
- front operation provides access to `*head_`

Array Implementation of Queue: Diagram



Remarks on Array Implementation of Queue

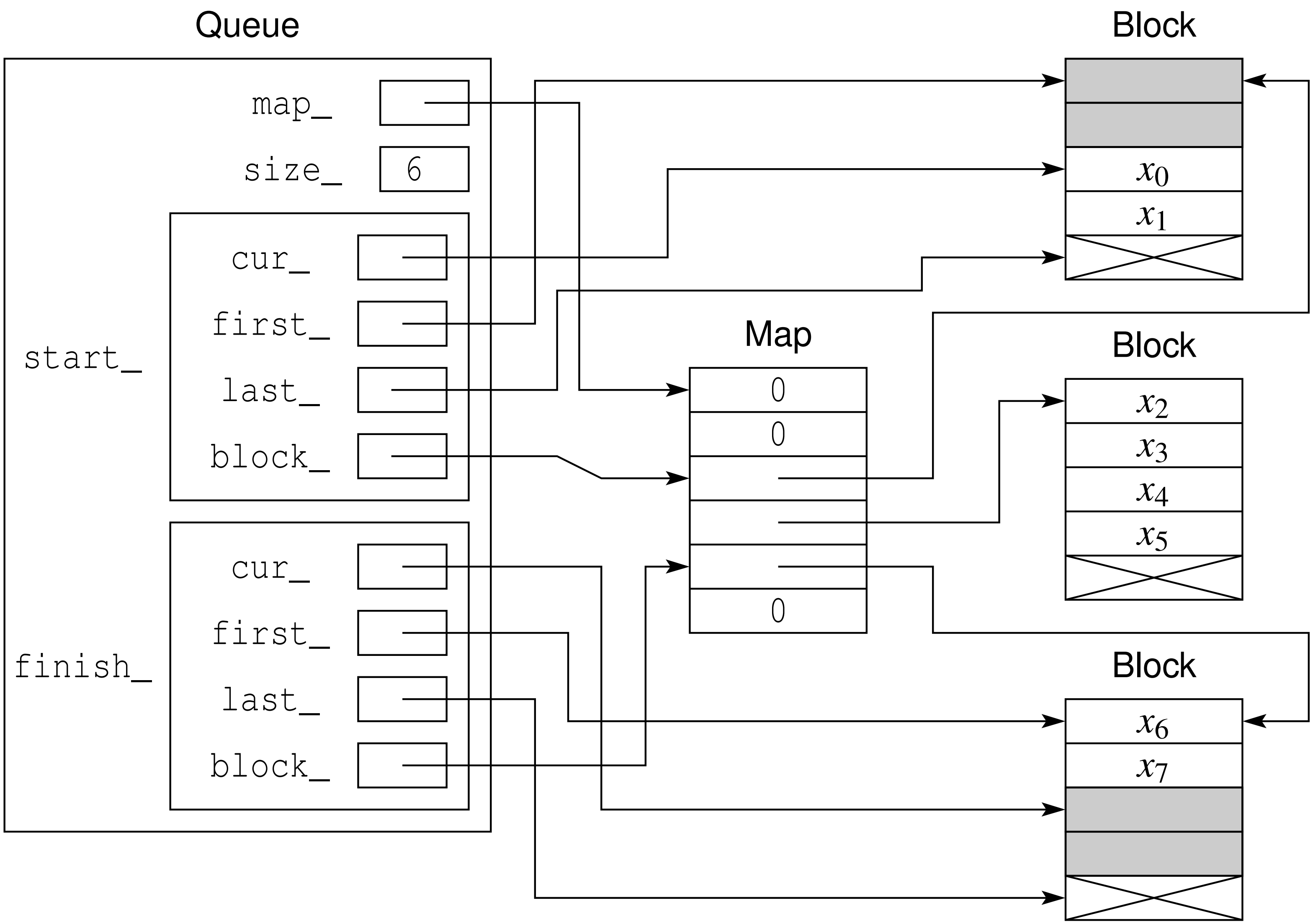
- although only consider queue of bounded size, could extend to unbounded case by using dynamically-resizable array
- advantages:
  - elements stored in contiguous buffer, occupying at most two contiguous regions of memory (i.e., contiguous region with potential hole in middle)
  - can insert and remove in $O(1)$ time
  - can access front element in $O(1)$ time
- disadvantages:
  - queue must be of bounded size
  - relaxing restriction of bounded size raises other issues associated with reallocation of array when capacity exceeded (e.g., worst case enqueue time not $O(1)$, element references not stable)

Array of Arrays Implementation of Queue

- array of arrays can be used to implement (unbounded) queue
- code example:

```
// how many Ts held in each block?
template <class T> constexpr std::size_t block_size
  = sizeof(T) < 512 ? 512 / sizeof(T) : 1;

template <class T> class Iterator {
    // ...
    T* cur_; // pointer to referenced element
    T* first_; // pointer to first element in block
    T* last_; // pointer to end element in block
    T** node_; // pointer to current block
};

template <class T> class Queue {
    // ...
    T** map_; // array of block pointers
    std::size_t size_; // size of map array
    Iterator start_; // iterator for first element in queue
    Iterator finish_; // iterator for end element in queue
};
```

Array of Arrays Implementation of Queue: Diagram



Remarks on Array of Arrays Implementation of Queue

- advantages:
  - elements never change their location so pointers and references to elements are stable
- disadvantages:
  - although each individual block holding element data is contiguous, blocks not contiguous
  - although elements are never relocated by insertions and removals, iterators can be invalidated
- similar data structure used in some implementations of `std::deque`

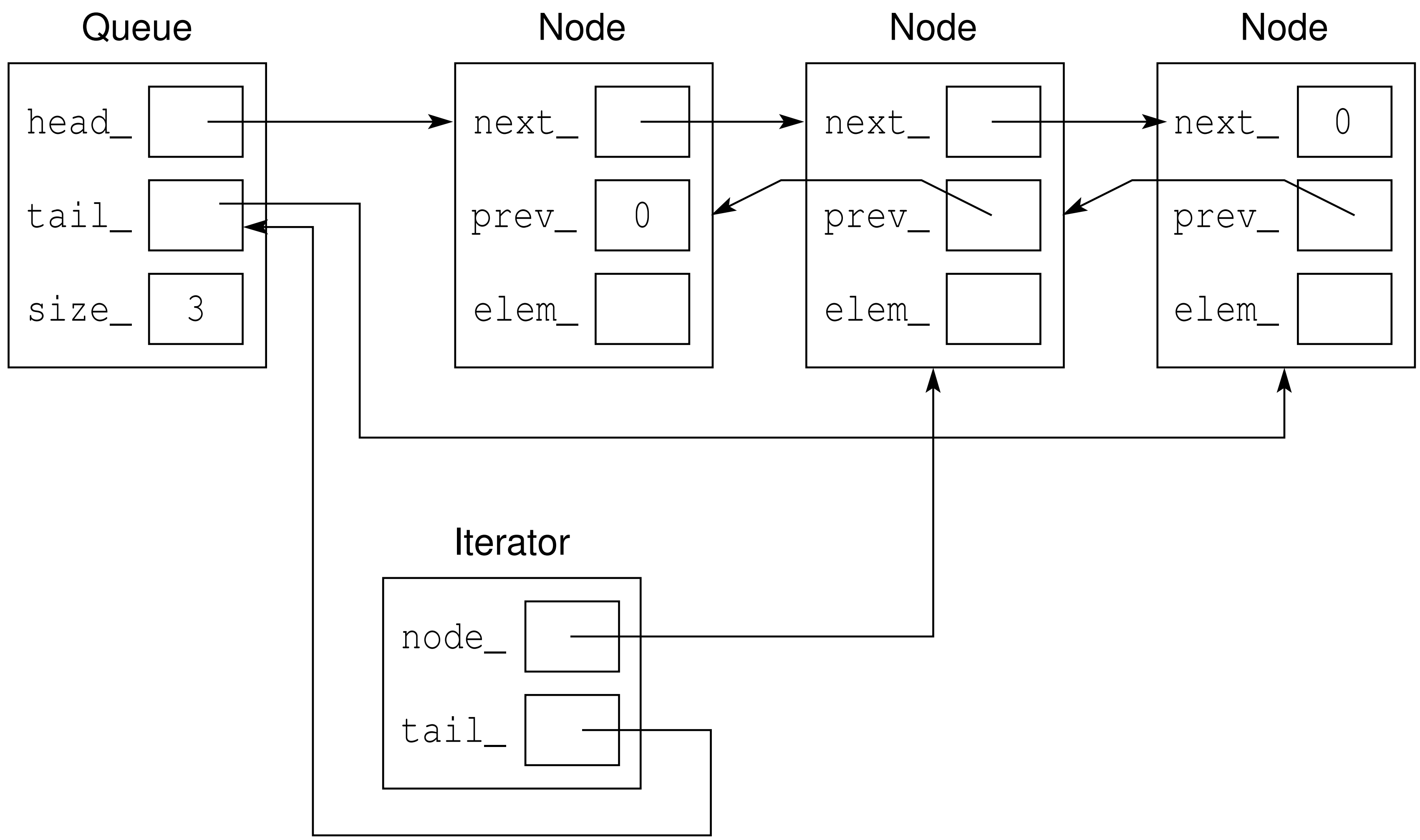
Node-Based Implementation of Queue

- doubly-linked list implementation of queue
- code example:

```
// queue node
template <class T> struct Node {
    Node* next_; // pointer to next entry in queue
    Node* prev_; // pointer to previous entry in queue
    T elem_; // element data
};

template <class T> class Queue {
    // ...
    Node<T>* first_; // first entry in queue
    Node<T>* last_; // last entry in queue
    std::size_t size_; // number of queued elements
};
```

- enqueue operation uses insert operation of linked list to insert element at end of list
- dequeue operation uses remove operation of linked list to remove element at head of list
- front operation provides access to element at head of list

Node-Based Implementation of Queue: Diagram



Remarks on Node-Based Implementation of Queue

- advantages:
  - enqueue and dequeue operations can be performed in $O(1)$ time
  - stable element references
- disadvantages:
  - elements not stored contiguously in memory

Section 6.3.2

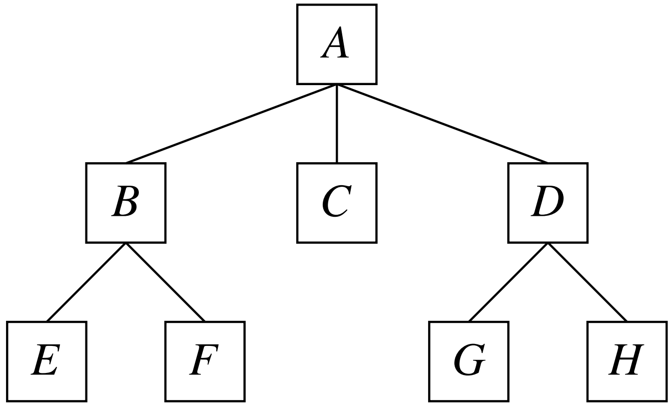
Multiway and Binary Trees

Trees

- tree is non-linear hierarchical data type
- tree consists of zero or more nodes
- except root, each node has parent
- each node has zero or more children
- tree containing no nodes is empty
- node q said to be **parent** of node n if n is child of q
- **root node**: node in tree with no parent
- node q said to be **sibling** of node n if q and n have same parent
- tree said to be **ordered** if linear ordering of children of each node

---

- example:



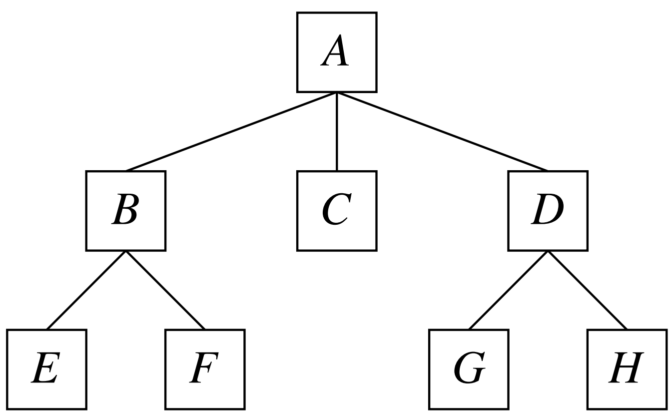
  - A is root node
  - B is child of A
  - A is parent of B
  - C and D are siblings of B

Tree Terminology

- **path** of length k in tree is sequence of $k+1$ nodes $n_0, n_1, \ldots, n_k$ where n_i is parent of n_{i+1}
- node q said to be **ancestor** of node n if q is on path from root node to n
- node q is said to be **descendant** of node n if q on path from n to leaf
- every node is both ancestor and descendant of itself
- node q said to be **proper ancestor** of n if ancestor of, and distinct from, n
- node q is said to be **proper descendant** of n if q is descendant of, and distinct from, n

---

- example:



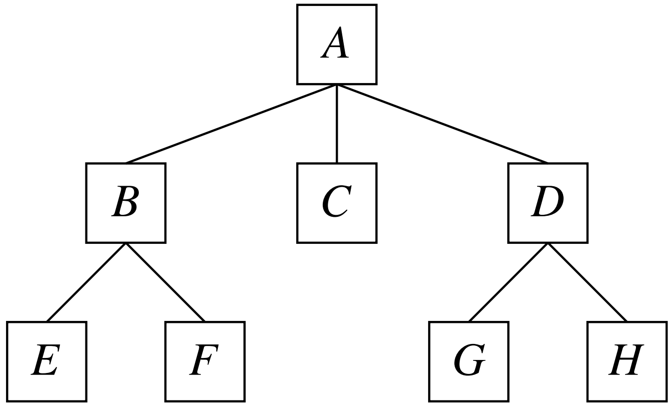
  - A, B, F is path of length 2
  - A and B are proper ancestors of E
  - E and F are proper descendants of B
  - B is ancestor and descendant of B

Tree Terminology (Continued 1)

- **subtree** rooted at node n is tree that consists of n and all of its descendants (e.g., subtree of root is entire tree)
- **degree** of node is number of its children
- **degree** of tree is maximum node degree taken over all nodes in tree
- **internal node** is node that has at least one child
- **external node** (also called **leaf node**) is node that does not have any children

---

- example:

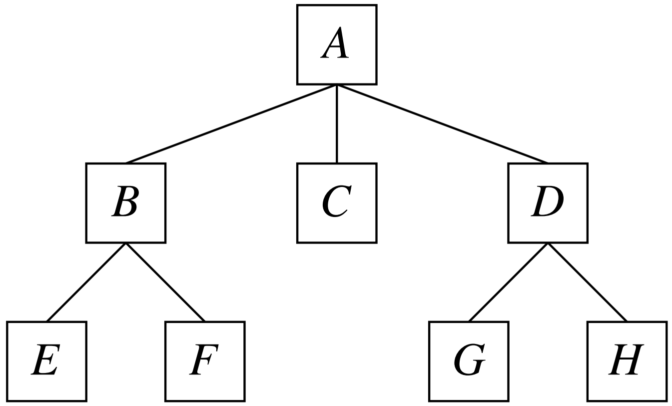


- tree consisting of nodes B, E, and F is subtree associated with node B
- degree of node B is 2
- degree of tree is 3
- A, B, and D are internal nodes
- C, E, F, G, and H are leaf nodes

Tree Terminology (Continued 2)

- **depth** of node (also called **level**) is length of path from root to node (or equivalently, number of proper ancestors of node) (e.g., root node has depth of zero)
- d**th level** of tree is all nodes at depth d in tree
- **height** of node is length of longest path from node to any leaf (e.g., leaf node has height of zero)
- **height** of tree is maximum node height taken over all nodes in tree (i.e., height of root) if tree is nonempty; otherwise, defined to be -1

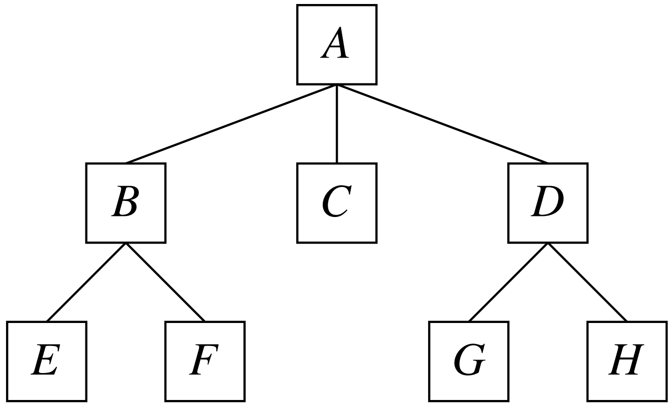
- example:



  - depths of nodes C and E are 1 and 2, respectively
  - nodes B, C, and D are at level 1
  - height of node D is 1
  - height of tree is 2

- **weight** of node n is number of descendant leaf nodes possessed by n
- **weight** of tree is number of leaf nodes in tree (i.e., weight of root node)

- example:

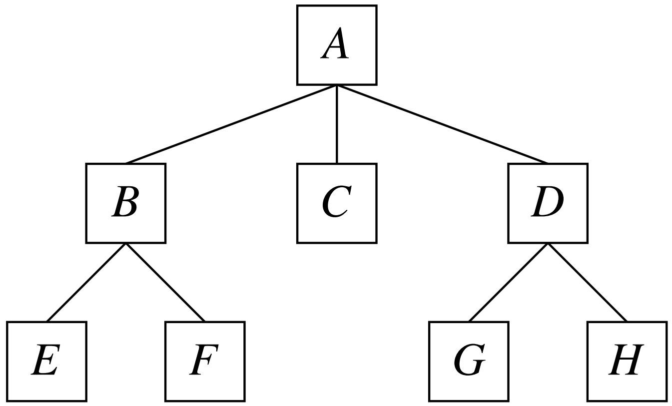


  - weights of nodes B and C are 2 and 0, respectively
  - weight of tree is 5

- **preorder traversal**: node visited before its descendants (i.e., parent before children)
- **postorder traversal**: node visited after its descendants (i.e., children before parent)
- preorder traversal might be used, for example, to print hierarchical document, where nodes correspond to sections in document
- postorder traversal might be used, for example, to compute space used by files in directory and its subdirectories

---

- example:



  - preorder traversal visits nodes in order: A, B, E, F, C, D, G, H
  - postorder traversal visits nodes in order: E, F, B, C, G, H, D, A

- representing directory tree in hierarchical file system
  - each internal node corresponds to directory
  - each leaf node corresponds to file (or empty directory)
- representing arithmetic expressions
  - each internal node corresponds to operator
  - each leaf node corresponds to operand
- representing decision-making process
  - each internal node corresponds to question with yes/no answer
  - each leaf node corresponds to final outcome of decision-making process
- searching for elements in collection
  - nodes correspond to elements in collection
  - nodes positioned in tree based on element keys

Tree ADT

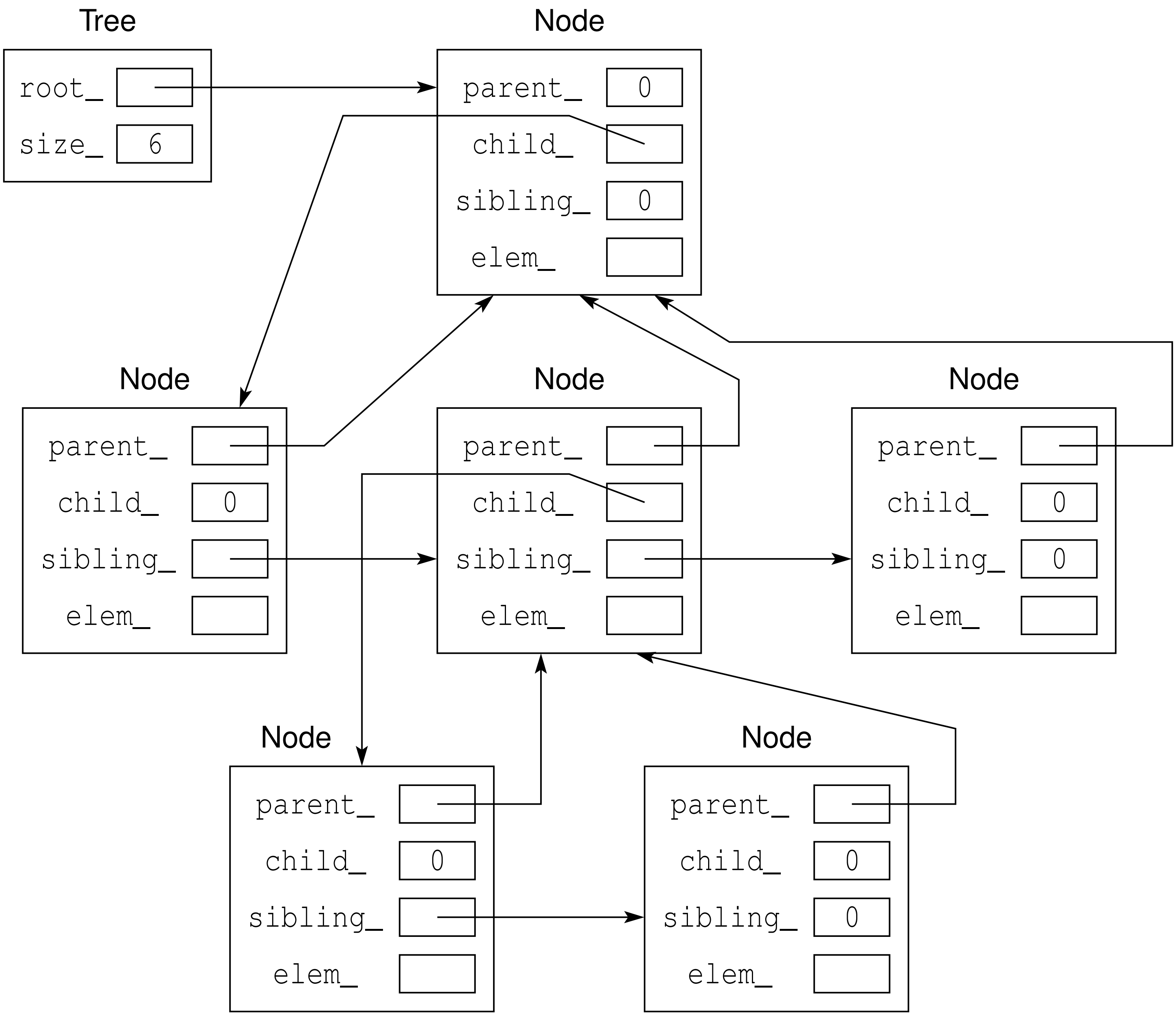
- **tree ADT** provides abstraction of tree data type
- operations provides by tree ADT include:
  - clear: remove all nodes from tree
  - size: get number of nodes in tree
  - is empty: test if tree is empty (i.e., contains no nodes)
  - root: get root node of tree
  - parent: get parent of node (which is not root)
  - children: get children of node
  - is internal: test if node is internal node
  - is external: test if node is external node
  - is root: test if node is root
  - replace: replace element in node
- may provide iterator ADT for traversing tree
- often tree ADT by itself is not particularly useful
- instead, tree ADT typically used to build other more task-specific ADTs (e.g., set ADT, multiset ADT, and so on)

Node-Based Tree Implementation

- node-based implementation of tree
- each node has pointer to first child and next sibling
- subsequent children can be accessed by following sibling pointers from first child
- allows size of node data structure to be constant (i.e., independent of maximum number of children)
- code example:

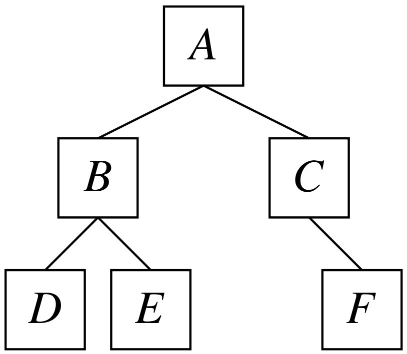
```
template <class T> struct Node {
    Node* parent_; // pointer to parent
    Node* child_; // pointer to first child
    Node *sibling_; // pointer to next sibling
    T elem_;
};

template <class T> class Tree {
    // ...
    Node<T>* root_; // pointer to root node
    std::size_t size_; // number of nodes in tree
};
```

Node-Based Tree Implementation: Diagram



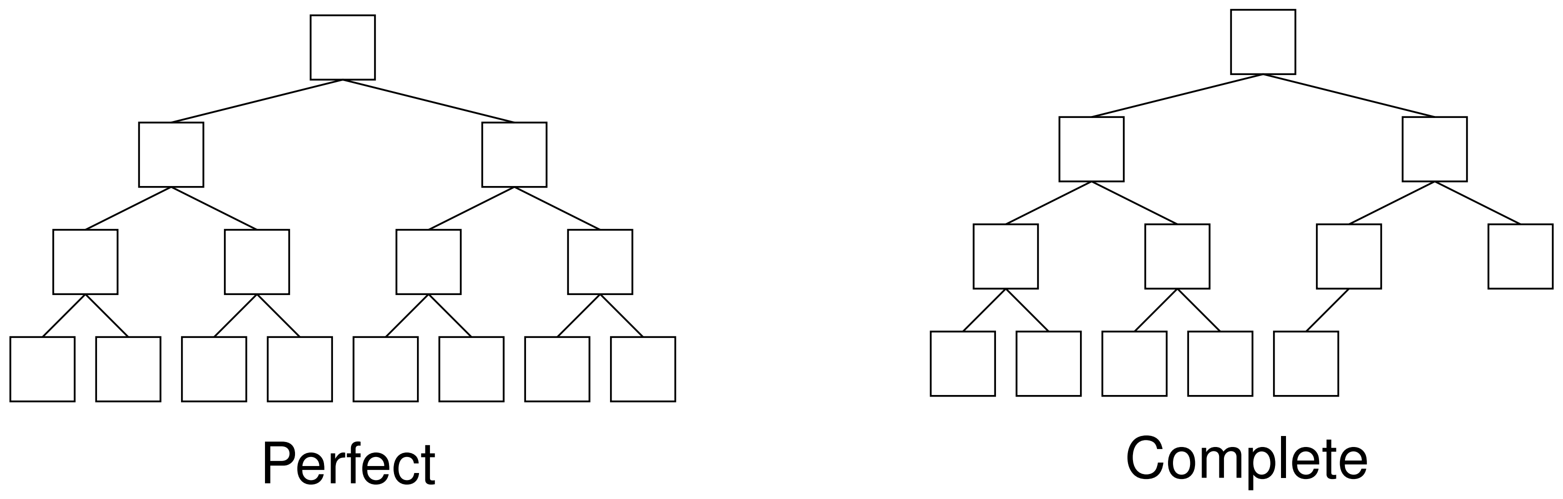
- each internal node has at most two children
- each node, excluding root node, labelled as either left or right child
- **left subtree** is tree rooted at left child
- **right subtree** is tree rooted at right child

- example:



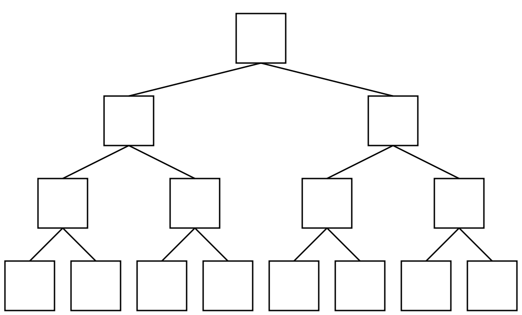
  - root node is A
  - left child of A is B
  - right child of A is C
  - left subtree of A is tree consisting of nodes B, D, and E
  - right subtree of A is tree consisting of nodes C and F

- binary tree said to be **perfect** (or **full**) if each internal node has exactly two children (which results in all leaves being at same level)
- binary tree said to be **complete** if perfect except possibly for deepest level which must be filled from left to right
- perfect implies complete

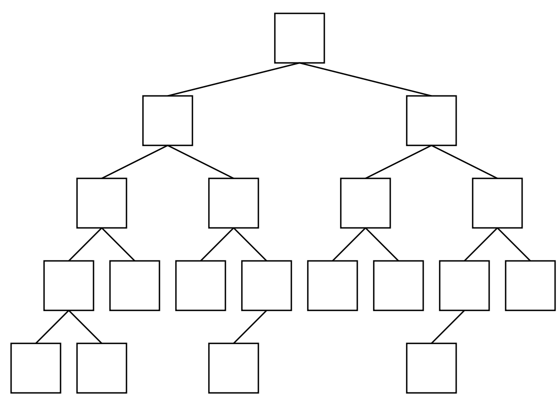


Balanced Binary Trees

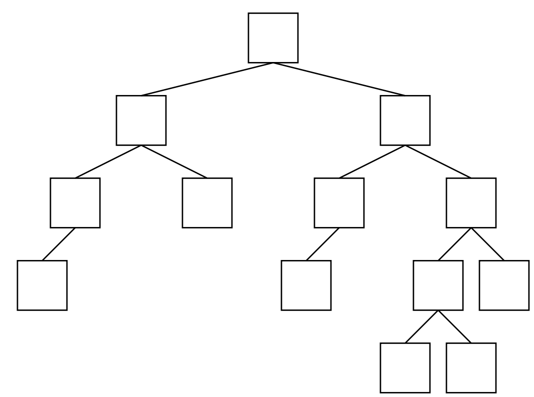
- binary tree said to be **perfectly balanced** all leaf nodes have same depth
- binary tree said to be **strictly balanced** if, for any two leaf nodes, difference in their depth is at most one
- binary tree said to be **height balanced** if height of left and right subtrees of each (interior) node differ by at most one
- perfectly balanced implies strictly balanced (but converse does not hold)
- strictly balanced implies height balanced (but converse does not hold)



Perfectly Balanced



Strictly Balanced

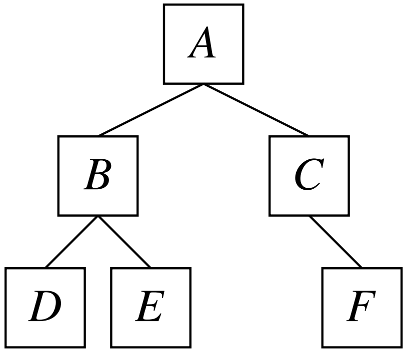


Height Balanced

- **preorder traversal**: visit node, then left subtree, then right subtree
- **postorder traversal**: visit left subtree, then right subtree, then node
- **level-order traversal**: visit nodes from left to right within level from top downwards
- one additional traversal order for binary trees: in order
- **in-order traversal**: visit left subtree, then node, then right subtree

---

- example:



  - preorder traversal order: A, B, D, E, C, F
  - postorder traversal order: D, E, B, F, C, A
  - inorder traversal order: D, B, E, A, C, F
  - level order traversal order: A, B, C, D, E, F

Binary Tree ADT

- **binary tree ADT** provides abstraction of binary tree
- operations provided by binary tree ADT that are common to general (i.e., m-ary) tree ADT include:
  - create: make empty tree
  - root: get root node
  - parent: get parent of node
  - is internal: test if node is internal (i.e., non-leaf)
  - is external: test if node is external (i.e., leaf)
  - is root: test if node is root of tree
  - is empty: test if binary tree is empty (i.e., contains no nodes)
  - size: get number of nodes in tree
  - clear: remove all nodes from tree
  - replace: replace element in node
  - add root: add root node (to empty tree)

Binary Tree ADT (Continued)

- other operations provided by binary tree ADT include:
  - left child: get left child of node
  - right child: get right child of node
  - has left child: test if node has left child
  - has right child: test if node has right child
  - insert left: insert node as left child of node (which must be leaf)
  - insert right: insert node as right child of node (which must be leaf)
  - remove: remove node (which must be leaf)
- may provide iterator ADT for traversing tree

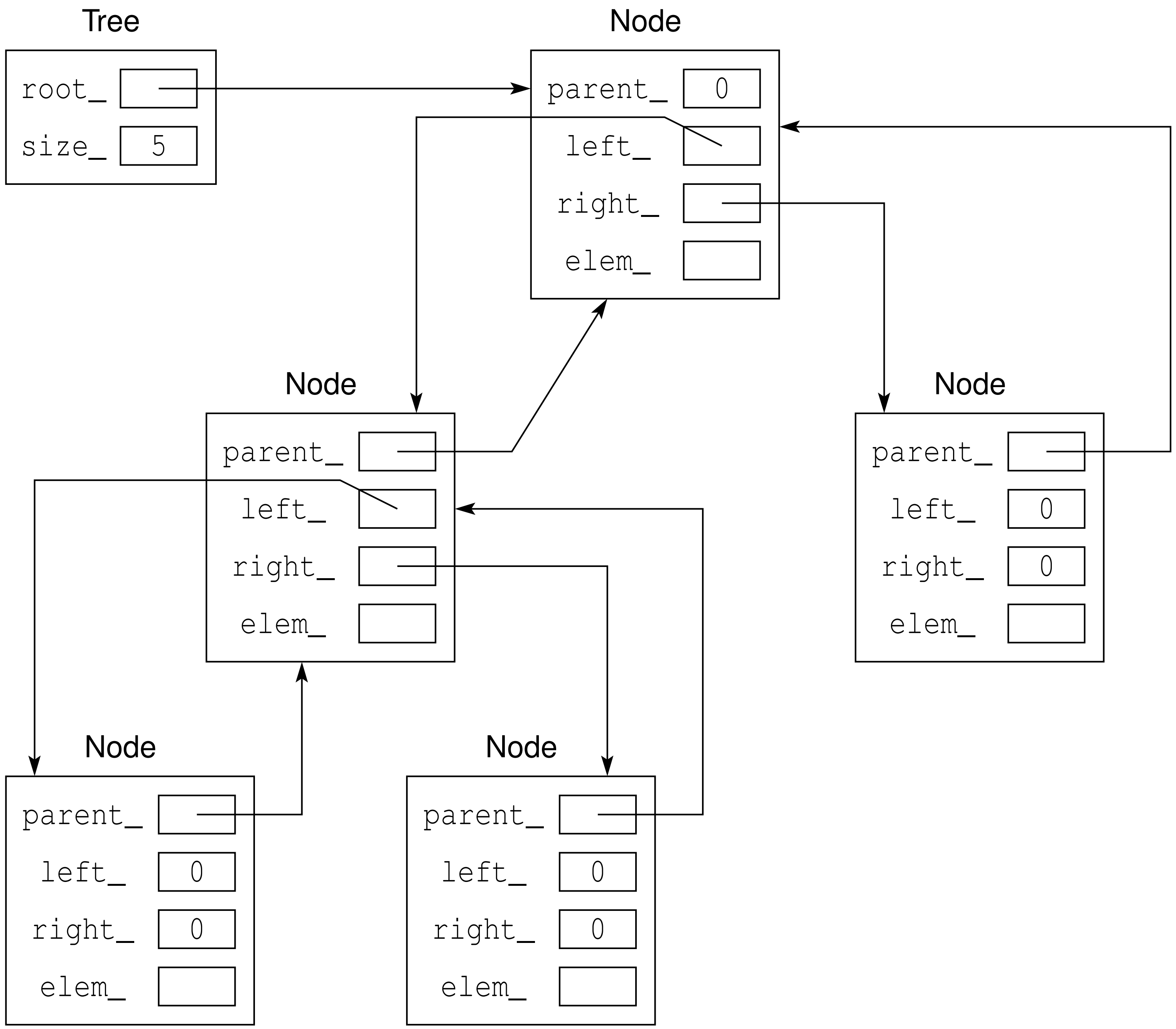
Node-Based Binary Tree

- node-based implementation of binary tree
- node data structure and tree data structure
- null pointer used as sentinel to indicate “no such node” (e.g., child of leaf, parent of root, etc.)
- code example:

```
template <class T> struct Node {
    Node* parent_; // pointer to parent
    Node* left_; // pointer to left child
    Node* right_; // pointer to right child
    T elem_; // element data
};

template <class T> class Tree {
    // ...
    Node<T>* root_; // pointer to root node
    std::size_t size_; // number of nodes in tree
};
```

- node-based implementation preferred for trees that are not complete
- in practice, sentinel node often preferred when iterator functionality must be provided

Node-Based Binary Tree: Diagram



Remarks on Node-Based Binary Tree

- advantages:
  - can handle case of tree that is not complete without gross memory inefficiency
  - can provide stable element references
- disadvantages:
  - has per-element storage overhead (3 pointers: 1 for parent, 1 for first child, and 1 for second child or next sibling)
  - element data not contiguous

- complete binary tree can be implemented using array
- position in array determines position in tree
- let $\operatorname{index}(n)$ denote index of node n
- let $\operatorname{parent}(n)$, $\operatorname{left}(n)$, and $\operatorname{right}(n)$ denote parent, left child, and right child of node n
- root node has index 0; and

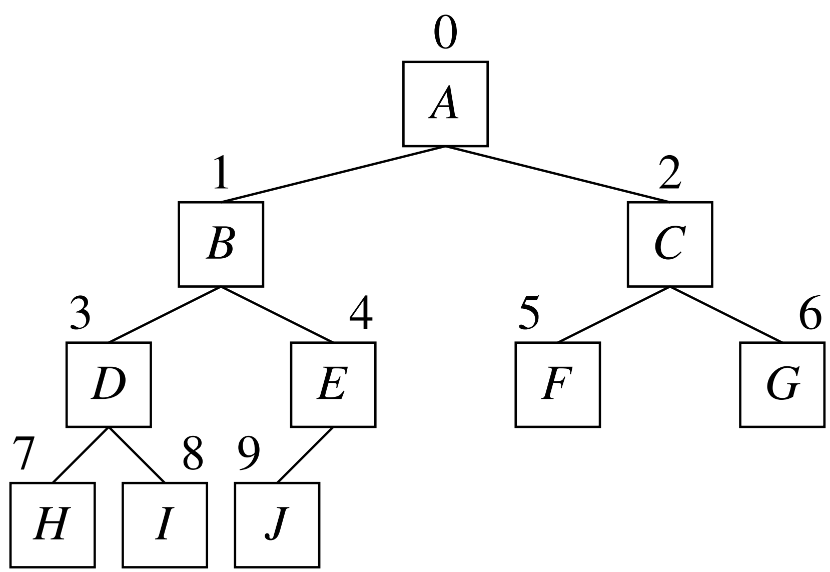
$$\begin{aligned}
\operatorname{index}(\operatorname{left}(n)) &= 2\operatorname{index}(n)+1 \\
\operatorname{index}(\operatorname{right}(n)) &= 2\operatorname{index}(n)+2 \\
\operatorname{index}(\operatorname{parent}(n)) &= \lfloor(\operatorname{index}(n)-1)/2\rfloor
\end{aligned}$$

- code example:

```
template <class T> class Tree {
    // ...
    T* start_; // start of element data
    T* end_; // end of element data
    std::size_t size_; // allocated size of data
};
```

Array-Based Binary Tree: Diagram

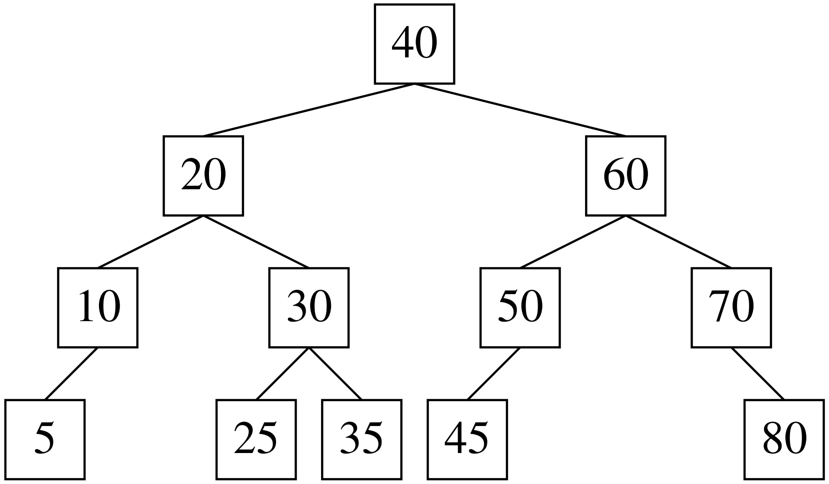
- example of complete tree with nodes labelled with corresponding array indices:



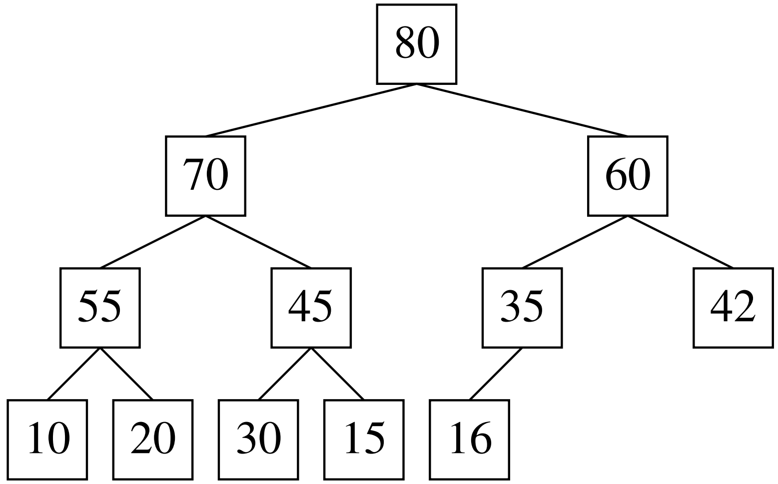
Remarks on Array-Based Binary Tree

- advantages:
  - memory efficient: no per-element storage overhead (i.e., no memory cost for representing connectivity of nodes in tree)
  - cache efficient: element data stored contiguously in memory
- disadvantages:
  - can only handle complete trees
  - although could generalize this approach to handle non-complete tree, would be grossly inefficient in terms of memory usage
  - if array capacity exceeded, costly reallocation and copy required
  - if array reallocation occurs, cannot provide stable references to elements
- array implementation should be preferred for complete trees (unless inability to guarantee stable element references is problematic)

- binary tree is said to have **binary search tree** property if, for each node node n with key k, following holds:
  - every key in left subtree of n is less than or equal to k; and
  - every key in right subtree of n is greater than or equal to k
- for tree of height h, can find element in $O(h)$ time
- example of binary search tree:



- tree said to have **heap property** if, for each node n in tree, following holds:
  - key of n is greater than or equal to key of each descendant of n
- **heap** is tree that satisfies heap property
- inserting or removing node can be done in $O(\log n)$ time without breaking heap property (but may need rearrangement of some nodes)
- example of heap:



Section 6.3.3

Hash Tables

Basic Idea Behind Hash Tables

- rather than navigating through search tree comparing search key to element key, hashing tries to reference element directly in table based on key
- effectively, hashing transforms key into address in table
- basic operations provided by hash table:
  - insert: add element to hash table
  - remove: remove element from hash table
  - find: search for element in hash table based on key
- want above operations to take $O(1)$ time on average
- order of elements in table unimportant

- hash table of size m consists of m **slots** (also called **buckets**) numbered from 0 to $m-1$ (inclusive)
- each element stored in hash table has key and possibly some associated value
- each slot can be empty or contain element data
- slot in which element stored determined by applying hash function to key
- **load factor** is ratio of number of elements in hash table to hash table size
- **collision** said to occur when two distinct keys map to same index in hash table
- often, choosing table size as prime number helps to ensure more uniform distribution of elements over slots

Hash Table Example

- collection of 10-digit employee numbers
- employee number is key
- hash function yields last four digits of employee number

Index	Slot
0000	
0001	0019910001
0002	5919870002
0003	
	⋮
9997	1212009997
9998	
9999	1122339999

Hash Functions

- **hash function**: maps key k of given type to integer in $\{0, 1, \ldots, m-1\}$
- hash function usually specified as composition of two functions:
  1. hash code map
  2. compression map
- **hash code map**: maps key to integer
- **compression map**: maps integer to integer in $\{0, 1, \ldots, m-1\}$
- first hash-code map h_1 applied to key k and then compression map h_2 applied to result to yield hash function $h(k) = h_2(h_1(k))$
- hash function is typically many-to-one mapping (which can therefore result in collisions)
- goal of hash function is to distribute keys uniformly across elements of $\{0, 1, \ldots, m-1\}$, which will reduce likelihood of collisions
- hash function should be fast to compute

Remarks on Hash-Code Maps

- various strategies can be used to generate hash-code map
- integer cast:
  - reinterpret bits of key as integer
- component sum:
  - partition key into integers of fixed size
  - then, sum these integers ignoring overflow
- polynomial accumulation:
  - partition bits of key into sequence of components of fixed length $a_0, a_1, \ldots, a_{n-1}$
  - then, evaluate polynomial $p(z) = \sum_{i=0}^{n-1} a_i z_i^i$ for some fixed value of z, ignoring overflow

Remarks on Compression Maps

- various strategies can be used to generate compression map
- let m denote size of hash table
- m usually chosen to be prime in order to better distribute keys over hash values
- division:
  - $h_2(i) = i \bmod m$
- multiply, add, and divide:
  - $h_2(i) = (ai + b) \bmod m$, where a and b nonnegative integers and $a \bmod m \neq 0$

Collision Resolution by Chaining

- chaining also called closed addressing
- with chaining, collisions handled by allowing multiple elements to be placed in single slot
- elements in slot stored in linked list
- **simple uniform hashing**: keys equally likely to hash into any of slots
- for load factor α, successful and unsuccessful searches take average-case time $\Theta(1+\alpha)$ under assumption of simple uniform hashing
- if insertion of elements already in hash table not allowed, insert operation has worst-case $O(1)$ time
- removal of element has worst-case $O(1)$ time
- can support insert, remove, and search all in $O(1)$ time on average (under assumption of simple uniform hashing)
- hash table cannot fill since each slot can potentially hold any number of elements

Collision Resolution by Open Addressing

- with open addressing, only one element allowed to be stored per slot in table so in case of collision alternate choice must be made for slot to store element
- sequence of indices to consider (in order) when inserting (or searching for) element with given key called **probe sequence**
- examine table at each position in probe sequence until slot for element is found (e.g., empty slot for insertion)
- for each possible key k, probe sequence should be permutation of $\{0,1,\ldots,m-1\}$ so that all slots are reachable
- many possible choices for probe sequence (e.g., linear, quadratic, double hashing, and random hashing)
- load factor α must satisfy $\alpha \leq 1$ (since only one element stored per slot)
- **uniform hashing**: probe sequence of each key equally likely to be any of $m!$ permutations of $\{0,1,\ldots,m-1\}$
- number of probes in unsuccessful search is at most $\frac{1}{1-\alpha}$, assuming uniform hashing

- with linear probing, probe sequence starts at hash value of key and then proceeds as necessary sequentially, wrapping around to beginning of table when end of table reached
- ith value in probe sequence for key k given by $h(k,i) = (h'(k)+i) \bmod m$, where h' is hash function
- suffers from primary clustering, where colliding elements clump together causing future collisions to generate longer sequence of probes
- expected number of probes for insertion or unsuccessful search is $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
- expected number of probes for successful search is $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$

Linear Probing Example

- integer key; hash function $h(k) = k \bmod 13$
- insert 18, 15, 23, 31, 44, and 9 (in order):

0	1	2	3	4	5	6	7	8	9	10	11	12
					18							

0	1	2	3	4	5	6	7	8	9	10	11	12
		15			18							

0	1	2	3	4	5	6	7	8	9	10	11	12
		15			18					23		

0	1	2	3	4	5	6	7	8	9	10	11	12
		15			18	31				23		

0	1	2	3	4	5	6	7	8	9	10	11	12
		15			18	31	44			23		

0	1	2	3	4	5	6	7	8	9	10	11	12
		15			18	31	44		9	23		

k	$h(k)$
18	5
15	2
23	10
31	5
44	5
9	9

Quadratic Probing

- with quadratic probing, distance between probes is determined by quadratic polynomial
- ith value in probe sequence for key k given by $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where h' is hash function and c_1 and c_2 are nonnegative integer constants
- c_1, c_2, and m must be carefully chosen to guarantee successful insertion is possible
- most often $c_1 = 0$ and $c_2 = 1$ and m prime
- must ensure that loading factor $\alpha \leq \frac{1}{2}$ in order to guarantee successful insertion
- eliminates primary clustering, but suffers from secondary clustering

Quadratic Probing Example

- integer key; hash function $h(k) = k \bmod 13$; $c_1 = 0$ and $c_2 = 1$
- insert 18, 15, 23, 31, 44, and 9 (in order):

0	1	2	3	4	5	6	7	8	9	10	11	12
					18							

0	1	2	3	4	5	6	7	8	9	10	11	12
		15			18							

0	1	2	3	4	5	6	7	8	9	10	11	12
		15			18					23		

0	1	2	3	4	5	6	7	8	9	10	11	12
		15			18	31				23		

0	1	2	3	4	5	6	7	8	9	10	11	12
		15			18	31			44	23		

0	1	2	3	4	5	6	7	8	9	10	11	12
9		15			18	31			44	23		

k	$h(k)$
18	5
15	2
23	10
31	5
44	5
9	9

- with double hashing, distance between successive probes determined by secondary hash function
- ith value in probe sequence for key k given by $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$, where h_1 is (primary) hash function and h_2 is secondary hash function
- h_2 must never be zero
- h_2 must be coprime with m for entire hash table to be searched
- for example, could let m be prime and have h_2 always yield strictly positive integer less than m

Double Hashing Example

- integer key; hash function $h(k) = k \bmod 13$; secondary hash function $d(k) = 7 - k \bmod 7$
- insert 18, 15, 23, 31, 44, and 9 (in order):

0	1	2	3	4	5	6	7	8	9	10	11	12
					18							

0	1	2	3	4	5	6	7	8	9	10	11	12
		15			18							

0	1	2	3	4	5	6	7	8	9	10	11	12
		15			18					23		

0	1	2	3	4	5	6	7	8	9	10	11	12
		15			18				31	23		

0	1	2	3	4	5	6	7	8	9	10	11	12
		15			18		44		31	23		

0	1	2	3	4	5	6	7	8	9	10	11	12
	9	15			18		44		31	23		

k	$h(k)$	$d(k)$
18	5	3
15	2	6
23	10	5
31	5	4
44	5	5
9	9	5

Random Hashing

- with random hashing, probe sequence generated by output of pseudorandom number generator seeded by key
- random number generation is relatively expensive
- in practice, double hashing tends to work about as well

Open Addressing: Insertion, Removal, and Search

- with open addressing, removal of elements can be problematic; in earlier linear probing example, consider removal of element with key 31 followed by search for element with key 44
- simplest solution is to distinguish between slot that has always been empty and slot from which element was deleted
- to perform search:
  - probe until:
    - element with query key is found; or
    - empty slot is found; or
    - all slots have been unsuccessfully probed
- to insert element (assuming not already in table):
  - examine successive slots in probe sequence until
    - slot found that is empty or “deleted”
  - if all slots have been unsuccessfully probed, error
  - store element in located slot
- to remove element:
  - remove element and mark occupied slot with special “deleted” marker

Rehashing

- if keep adding elements to hash table, eventually size of table will need to be increased, due to loading factor becoming too large (for good performance or correct behavior)
- **rehashing**: rebuilding hash table with different number of slots
- typical threshold for load factor α for rehashing:
  - 1 for chaining
  - $\frac{1}{2}$ for open addressing

Some Applications of Hash Tables

- dictionary searches (e.g., spelling checkers, natural language understanding)
- accessing tree or graph nodes by name (e.g., city names on geographical maps)
- symbol tables in compilers
- transposition tables used in some games (e.g., chess)

Section 6.3.4

Sets, Multisets, Maps, and Multimaps

Set and Multiset ADTs

- **set ADT** is container that stores collection of unique values
- set can be ordered (i.e., elements have well-defined order) or unordered
- operations provided by set ADT include:
  - clear: remove all elements from set
  - is empty: test if set is empty
  - size: query cardinality of set (i.e., number of elements in set)
  - insert: insert value in set
  - remove: remove value from set
  - find: locate value in set if present (i.e., for testing set membership)
- **multiset ADT** similar to set ADT except that duplicate values allowed
- example realizations of set/multiset ADT:
  - `std::set` and `std::multiset`
  - `std::unordered_set` and `std::unordered_multiset`
  - `boost::intrusive::unordered_set` and `boost::intrusive::unordered_multiset`
  - `boost::intrusive::set` and `boost::intrusive::multiset`

Map and Multimap ADTs

- **map (or associative array) ADT** is container that stores pairs each consisting of key and value, where keys are unique
- each element in map consists of key and value
- operations provided by map ADT include:
  - clear: remove all elements from map
  - is empty: test if map is empty
  - size: query number of elements in map
  - insert: insert element in map
  - remove: remove element from map
  - find: locate element in map if present based on its key
- **multimap ADT** similar to map ADT except that restriction that keys must be unique is dropped
- example realizations of map/multimap ADT:
  - `std::map` and `std::multimap`
  - `std::unordered_map` and `std::unordered_multimap`
  - `boost::intrusive::set` and `boost::intrusive::multiset`

Remarks on Implementation of Sets and Maps

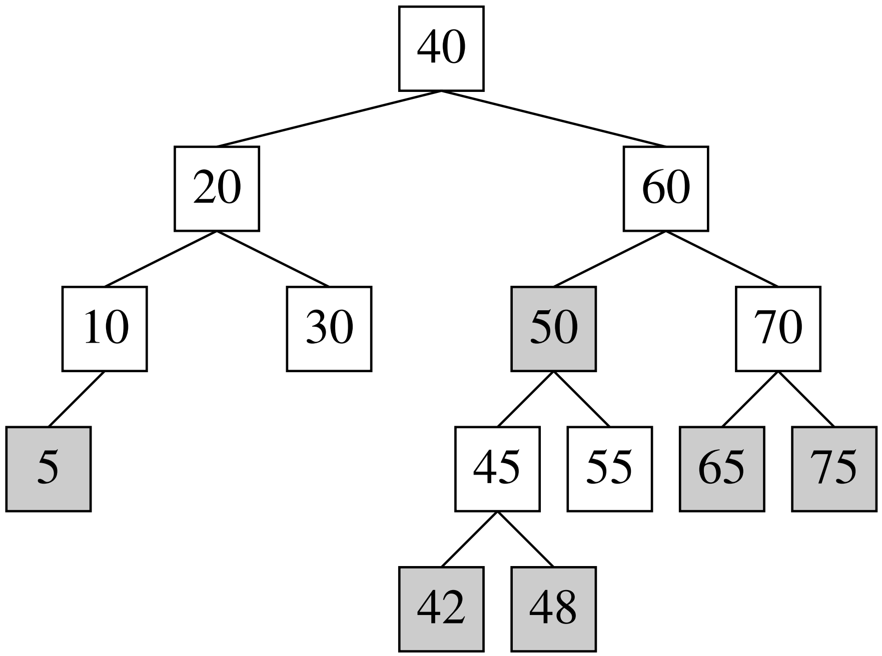
- ordered sets, multisets, maps, and multimaps typically implemented using balanced binary search tree [link: binary search trees]
- unordered sets, multisets, maps, and multimaps typically implemented using hash table

Red-Black Trees

- red-black trees first proposed by Bayer (1972)
- red-black tree is approximately height-balanced binary search tree
- requires one additional field per node, namely, color (i.e., red or black)
- binary search tree with following invariants:
  - each node is either red or black
  - root node is black
  - if node is red, then both of its children are black
  - every path from given node to any of its descendant nil nodes (i.e., null pointer) contains same number of black nodes
- invariants guarantee approximate height balancing
- path from root to farthest leaf no more than twice as long as path from root to nearest leaf
- height h of tree with n nodes is bounded by $h \leq 2\log_2(n+1)$
- invariants maintained by rotation and color flipping operations
- memory cost only 1 additional bit per node (for color), relative to classic binary tree

Red-Black Trees (Continued)

- some C++ standard library implementations use red-black trees for types that provide binary search tree functionality (e.g., `std::set` and `std::map`)
- example realizations of red-black trees:
  - `boost::intrusive::rbtree`, `boost::intrusive::set`, and `boost::intrusive::multiset`
- example of red-black tree (where red nodes are shaded gray):



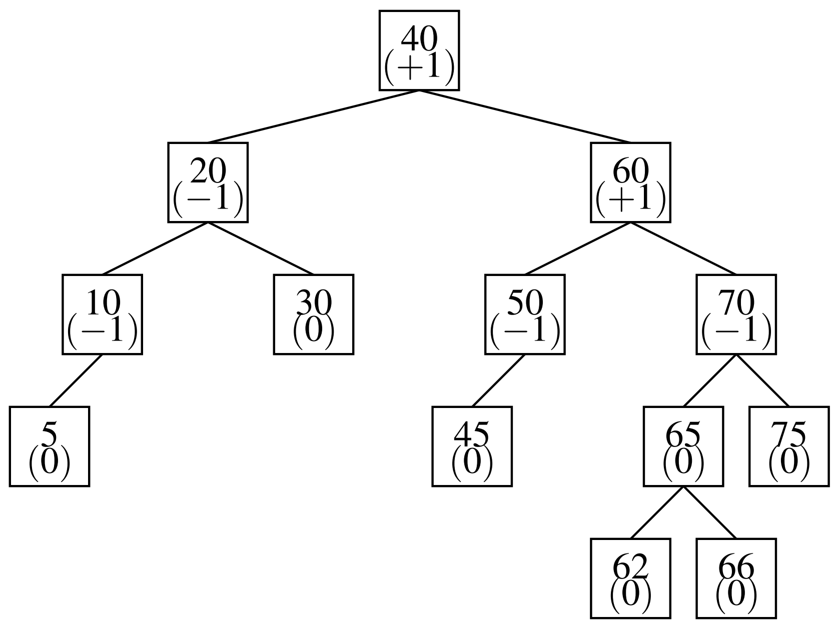
- AVL trees first proposed by Adelson-Velsky and Landis (1962)
- AVL tree is height-balanced binary search tree
- balance factor b of node n is defined as $b = r - \ell$, where ℓ and r are heights of left and right subtrees of n, respectively
- AVL tree is binary search tree such that, for every node n, balance factor b of n satisfies $b \in \{-1, 0, 1\}$ (i.e., for each node in tree, height of left and right subtrees differ by at most one)
- need to store balance factor in each node
- AVL trees more rigidly balanced than red-black trees
- height h of tree with n nodes is bounded by

$$h \leq c \log_2(n+d) + b \approx 1.440 \log_2(n + 1.065) - 0.328,$$

  where $c = \frac{1}{\log_2 \varphi}$, $b = \frac{c}{2} \log_2 5 - 2$, $d = 1 + \frac{1}{\varphi^4 \sqrt{5}}$, and $\varphi = \frac{1+\sqrt{5}}{2}$
- memory cost is 2 bits per node (for balance factor), relative to classic binary tree
- rebalancing achieved by rotation operations

AVL Trees (Continued)

- since AVL trees more rigidly balanced than red-black trees, search operations typically faster in AVL tree
- insertion and removal operations typically slower in AVL tree than in red-black tree, due to more work being required for tree re-balancing
- example realizations of AVL trees:
  - `boost::intrusive::avltree`, `boost::intrusive::avl_set`, and `boost::intrusive::avl_multiset`
- example of AVL tree:



Treaps

- treap is combination of binary search tree and heap
- each node has key and priority
- nodes arranged to form binary search tree with respect to key
- nodes also arranged to form heap with respect to priority
- if priorities chosen randomly, tree will be well balanced with high probability
- treaps provide benefits of balanced search trees, but rebalancing (which is driven by heap property) is less complicated than with some other types of balanced search trees
- example realizations of treaps:
  - `boost::intrusive::treap`, `boost::intrusive::treap_set`, and `boost::intrusive::treap_multiset`

Splay Trees

- splay tree is self-adjusting binary search tree with property that ***searches for more frequently accessed elements can be performed more quickly***
- splay tree keeps more recently accessed elements closer to root
- caching effect comes at cost of tree rebalancing being required each time search is performed
- significant disadvantage of splay tree is that height of tree can become linear in number of elements
- in worst case, insertion, removal, and search operations take amortized $O(\log n)$ time
- example realizations of splay trees:
  - `boost::intrusive::splay_tree`, `boost::intrusive::splay_set`, and `boost::intrusive::splay_multiset`

- scapegoat tree is self-balancing binary search tree
- provides worst-case $O(\log n)$ search time
- provides insertion and removal in amortized $O(\log n)$ time
- unlike other self-balancing binary search trees that provide worst-case $O(\log n)$ search time, scapegoat trees have ***no additional per-node overhead*** compared to regular binary search tree
- rebalancing can potentially be very expensive, although only infrequently
- consequently, insertion and removal operations have worst-case $O(n)$ time
- example realizations of scapegoat trees:
  - `boost::intrusive::sgtree`, `boost::intrusive::sg_set`, and `boost::intrusive::sg_multiset`

Section 6.3.5

Priority Queues

Priority Queue ADT

- **priority queue ADT** is ADT similar to queue except that each element on queue also has corresponding priority
- element at front of queue is always element with highest priority
- operations provided by priority queue ADT include:
  - front: access element at front of queue (i.e., element with highest priority)
  - insert: insert element in queue with specified priority
  - remove: remove element from front of queue (i.e.. element with highest priority)
  - update priority (optional): update priority of element in queue
- if priority queue has **stability property**, elements with equal priority will be removed in FIFO order
- examples of realization of priority queue ADT:
  - `std::priority_queue`,
  - `boost::heap::priority_queue` and `boost::heap::fibonacci_heap`

Remarks on Priority Queue Implementations

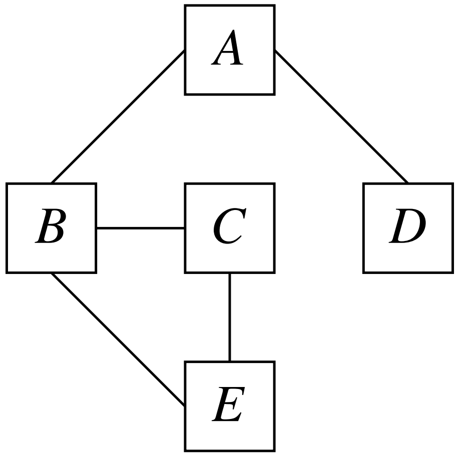
- priority queues typically implemented with heaps [link: heaps]
- heaps can always be constructed to be complete trees
- consequently, can reasonably choose to implement priority queue with either array-based or node-based tree
- in practice, stability often ensured by augmenting priority with integer sequence number, which is incremented with each insertion
- array-based implementation more memory efficient but does not have stable element references (if underlying array can be reallocated)
- node-based implementation can offer stable element references

Section 6.3.6

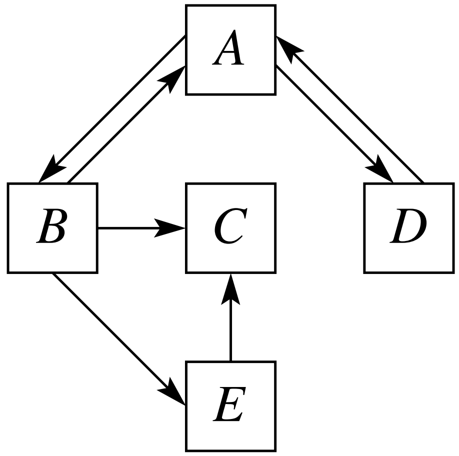
Graphs

Graphs

- graph is concept from discrete mathematics
- collection of nodes and edges
- nodes can be connected by edges
- directed graph: edges are directed (i.e., have direction)
- undirected graph: edges are undirected
- examples of graphs:



Undirected



Directed

Graph ADTs

- **graph ADT** is abstraction of mathematical notion of graph
- operations for graph ADT include:
  - adjacent: tests if edge from one vertex to another
  - neighbours: list all vertices that have edge to another vertex
  - insert vertex: add vertex to graph
  - remove vertex: remove vertex from graph
  - insert edge: add edge from one vertex to another
  - remove edge: remove edge from one vertex to another
  - get vertex value: get value associated with vertex
  - set vertex value: set value associated with vertex
  - get edge value: get value associated with edge
  - set edge value: set value associated with edge
- examples of realization of graph ADT:
  - `boost::adjacency_list` and `boost::adjacency_matrix`

Adjacency Matrix Implementation of Graph

- adjacency (i.e., connectivity) of n nodes can be represented using $n \times n$ binary matrix called **adjacency matrix**
- (i, j)th element of adjacency matrix is 1 if graph has edge from node i to node j and 0 otherwise
- if graph is undirected, adjacency matrix is symmetric and only lower triangular part of matrix need be stored
- if graph is directed, adjacency matrix is not necessarily symmetric, and entire matrix must be stored
- can test adjacency of two nodes (which requires examining element in matrix) in $O(1)$ time
- identifying all neighbours of given node takes $O(n)$ time
- iterating over all edges is slow
- storage cost of adjacency matrix is $\Theta(n^2)$
- high storage cost easier to justify if graph has large number of edges

Adjacency List Implementation of Graph

- adjacency (i.e., connectivity) of v nodes can be represented using v linked lists
- ith list contains node j if and only if graph has edge from node i to node j
- can test adjacency of two nodes (which requires traversing linked list) in worst-case $O(d)$ time, where d is largest valence of nodes in graph
- identifying all neighbours of given node very cheap
- storage cost of adjacency list is $O(v+e)$, where v and e is number of node and edges, respectively

Section 6.3.7

Intrusive Containers

Intrusive Containers

- container said to be **intrusive** if it requires help from elements it intends to store in order to store them
- intrusive container ***directly*** places user's objects in container (***not copies*** of user's objects)
- node pointers exposed to user of container, which allows some operations to be performed more efficiently
- intrusive container ***does not own*** elements it stores
- lifetime of stored object not bound to or managed by container (i.e., lifetime of stored objects managed by user)
- can store element in multiple intrusive containers simultaneously (which is not possible with nonintrusive containers)
- more coupling between code for container and code using container

Shortcomings of Non-Intrusive Containers

- object can only belong to one container
- only copies of objects stored in nonintrusive containers
- creating copies of values can become bottleneck (due to memory allocation and copying)
- noncopyable and nonmovable objects cannot be stored in nonintrusive containers (unless objects can be directly constructed inside container and are guaranteed not to be copied/moved subsequently)
- cannot store derived object in nonintrusive container and retain original type (i.e., copying derived object into container would result in slicing)

Advantages of Intrusive Containers

- same object can be placed in multiple intrusive containers simultaneously
- intrusive containers do not invoke memory management operations since do not own stored elements
- complexity of inserting and removing elements in intrusive containers more predictable since no memory allocation involved
- intrusive containers tend to allow stronger complexity guarantees (since no memory allocation or copying performed)
- intrusive containers offer better exception safety guarantees (since do not need to make copy of element to place in container)
- for intrusive container, computation of iterator from pointer or reference to element is $O(1)$ time operation, which is not usually true for nonintrusive containers (e.g., for nonintrusive linked list, this operation takes $O(n)$ time)

Disadvantages of Intrusive Containers

- in order to use type with intrusive container, must change definition of type
- each type stored in intrusive container needs additional memory to hold information for container
- intrusive containers unavoidably expose some implementation details of container to user
- since some implementation details are exposed, easier to break invariants of container; for example:
  - changing key of element in map
  - corrupting pointers used to link nodes in container
- user must assume responsibility for memory management (since container does not)
- user must manage lifetime of objects placed in container independent from lifetime of container itself, which can be error prone:
  - when destroying container before object, must be careful to avoid resource leaks
  - destroying object while in container, likely to be disasterous since container uses part of object to implement container

Disadvantages of Intrusive Containers (Continued)

- intrusive containers typically not copyable or movable since such containers do not directly perform memory allocation
- analyzing thread safety of program using intrusive containers often more difficult since container contents can be modified without going through container interface

Intrusive Doubly-Linked List

- node-based implementation of list where each node tracks both its successor and predecessor
- value type (which stores user data) and node type (which is used to maintain list) are ***same***
- null pointer used as sentinel value to indicate “no such node” (e.g., no successor/predecessor node or no head/tail node)
- in order for elements of type `T` to be used with container, `T` must include special type as data member (which encapsulates next/previous pointers for linked list)
- uses pointer to member to identify member that holds list node state [link: pointers to members

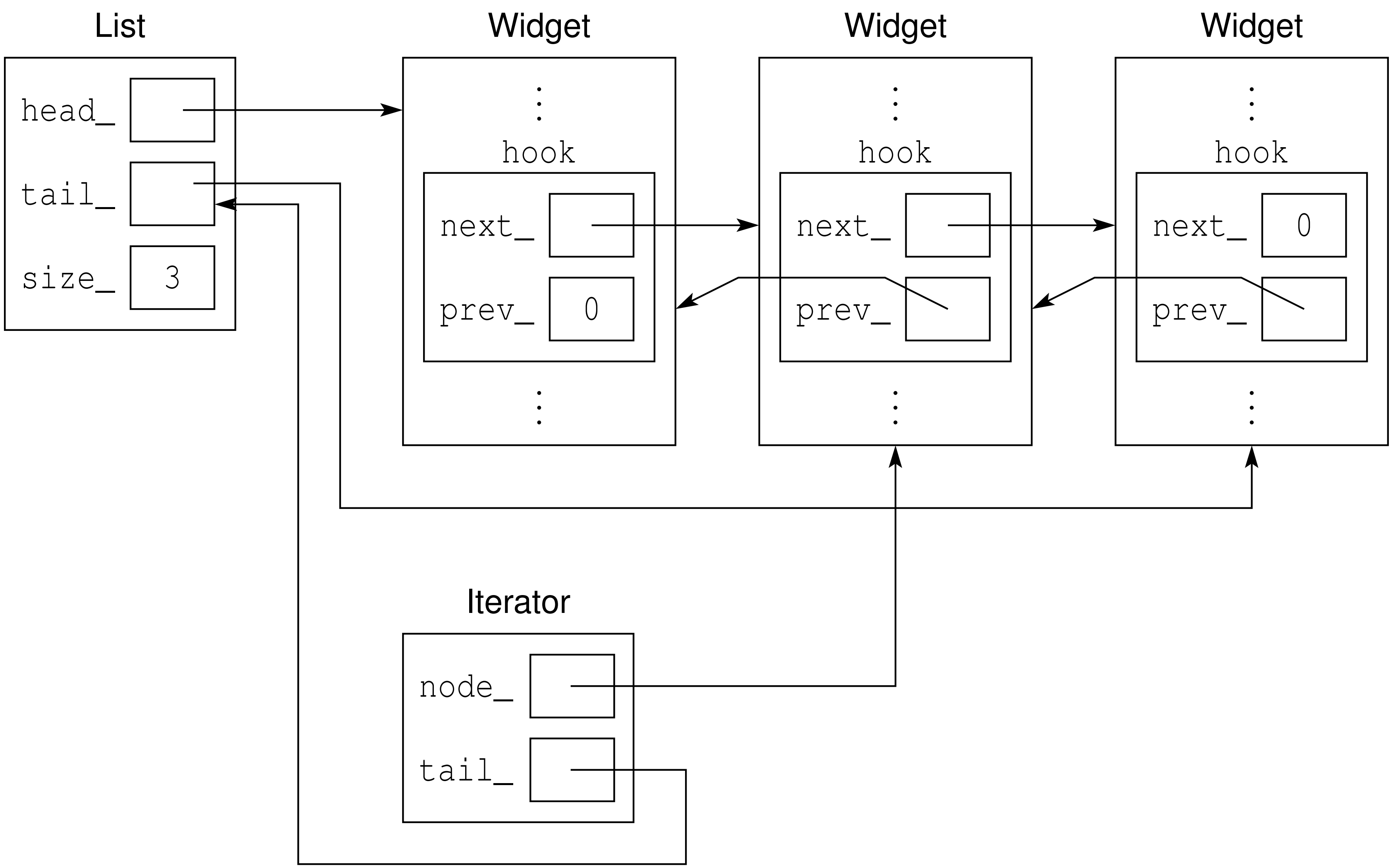
Intrusive Doubly-Linked List: Code

```
// type encapsulating links for list (i.e., part of list node)
template <class T> struct list_hook {
    // ...
    T* next_; // pointer to next node in list
    T* prev_; // pointer to previous node in list
};

// iterator class (C determines if const_iterator)
template <class T, list_hook<T> T::* P, bool C> class list_iterator {
    // ...
    // node_ptr is either T* or const T* depending on C
    node_ptr node_; // pointer to node of referenced element
    node_ptr const* tail_; // pointer to list tail node pointer
};

template <class T, list_hook<T> T::* P> class list {
    // ...
    T* head_; // pointer to first node in list
    T* tail_; // pointer to last node in list
    std::size_t size_; // number of elements in list
};
```

Intrusive Doubly-Linked List: Code (Continued)

```
// list node with user data
struct Widget {
    // ...
    list_hook<Widget> hook; // public
    // ...
};

// type for list of Widget objects
using Widget_list = list<Widget, &Widget::hook>;
```

Intrusive Doubly-Linked List: Diagram



Remarks on Intrusive Doubly-Linked List

- node pointer and value pointer are equivalent (i.e., pointers to next and previous nodes have type `T*`)
- storage cost of iterator is two pointers (but one pointer would be more desirable)
- iterator state requires pointer to list tail pointer in order to handle case of decrementing end iterator (which has null node pointer)
- implementation does not require any non-portable constructs

Intrusive Doubly-Linked List With Sentinel Node

- **intrusive doubly-linked list with sentinel node** is circular doubly-linked list with dummy node that serves as sentinel (instead of using null pointer)
- value type (which stores user data) and node type (which is used to maintain list) are ***distinct***
- in particular, value type contains node type as data member
- effectively sentinel node makes list circular
- in order for elements of type `T` to be used with container, `T` must include special type as data member (which encapsulates next/previous pointers for linked list)

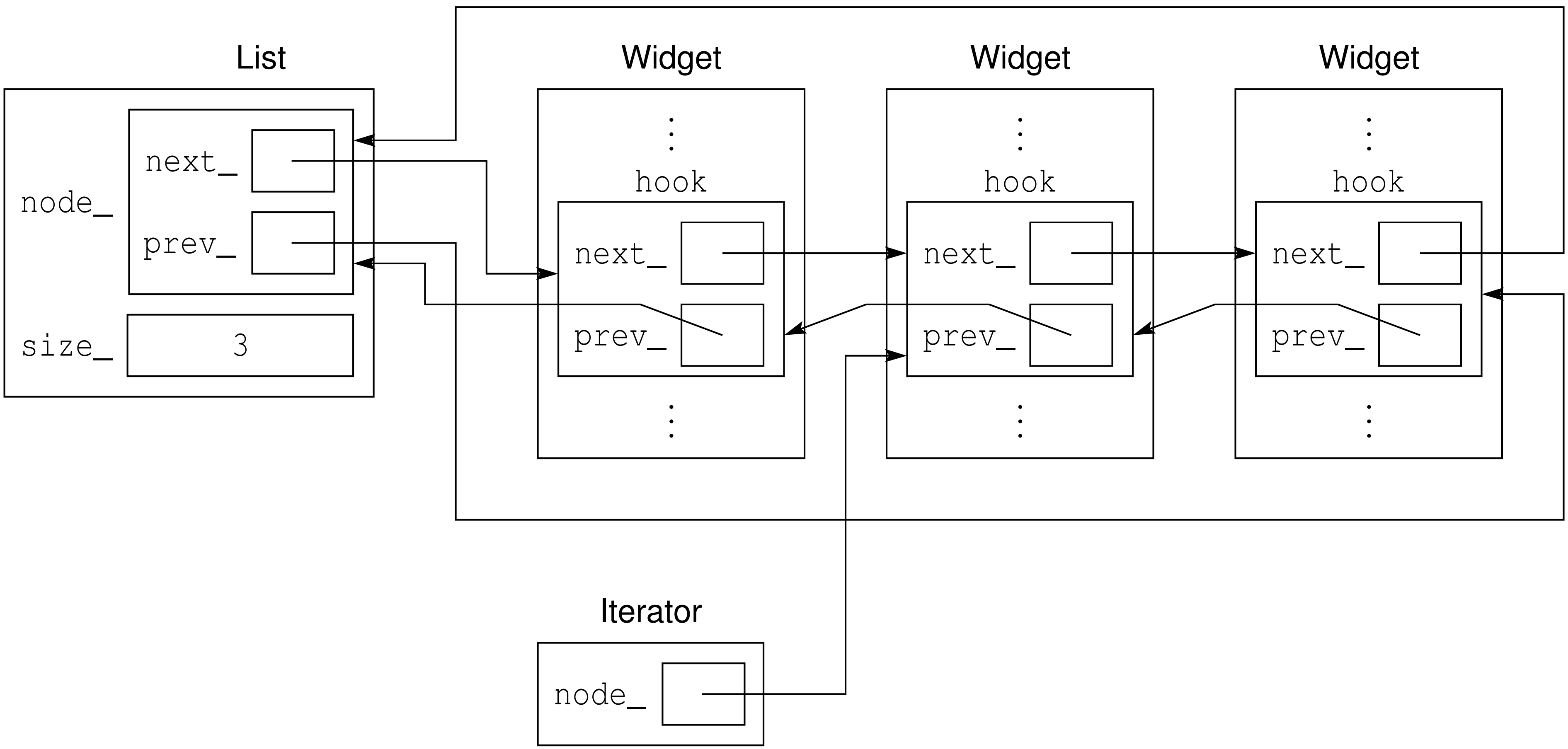
Intrusive Doubly-Linked List With Sentinel Node: Code

```
// list node class
struct list_hook {
    // ...
    list_hook* next_; // pointer to next node in list
    list_hook* prev_; // pointer to previous node in list
};

// list traits class (no data members)
template <class T, list_hook<T> T::* P> class list_traits {
    // functions for mapping between object and node pointers
};

// list iterator class (C determines if const_iterator)
template <class T, list_hook<T> T::* P, bool C> class list_iterator :
  list_traits<T, P>{
    // ...
    list_hook* node_; // pointer to node of referenced element
};

// list
template <class T, list_hook<T> T::* P> class list :
  list_traits<T, P> {
    // ...
    list_hook node_; // sentinel node
    std::size_t size_; // number of elements in list
};
```

Intrusive Doubly-Linked List With Sentinel Node: Code (Continued)

```
// list node with user data
struct Widget {
    // ...
    list_hook hook; // public
    // ...
};

// type of list of Widget objects
using Widget_list = list<Widget, &Widget::hook>;
```

Intrusive Doubly-Linked List With Sentinel Node: Diagram



Remarks on Intrusive Doubly-Linked List With Sentinel Node

- circular list avoids many special cases in implementation of list class (since circular list never empty and list has no beginning or end)
- node and value types are distinct (i.e., node pointers are of type `list_hook`, not `T`)
- storage cost of iterator is one pointer
- implementation requires non-portable construct to determine value pointer from node pointer
- determining value pointer from node pointer cannot work in all cases (in particular, if value type uses virtual inheritance)
- limitations on what types can be placed in container
- another variation on this intrusive list approach can be obtained by using inheritance to add required list state to user's type (instead of adding by data members), which has some advantages

Examples of Intrusive Containers

- as of C++17, all container classes in standard library are nonintrusive
- Boost library has good selection of intrusive containers, which includes (amongst others):
  - `boost::intrusive::slist` (intrusive singly-linked list)
  - `boost::intrusive::list` (intrusive doubly-linked list)
  - `boost::intrusive::set` (intrusive set/map)
  - `boost::intrusive::multiset` (intrusive multiset/multimap)
  - `boost::intrusive::unordered_set` (intrusive unordered set/map)
  - `boost::intrusive::unordered_multiset` (intrusive unordered multiset/multimap)
  - `boost::intrusive_ptr` (intrusive reference-counted smart pointer)

Section 6.3.8

Miscellany

Memory Management for Containers

- for reasons of efficiency or functionality (or even correctness), often necessary to:
  - separate memory allocation from construction
  - separate memory deallocation from destruction
- operator new can be used to perform only memory allocation (without construction)
- placement new can be used to perform only construction (without memory allocation)
- operator delete can be used to perform only memory deallocation (without destruction)
- direct invocation of destructor can be used to perform only destruction (without memory deallocation)
- allocator type provides interface that decouples allocation/deallocation and construction/destruction
- numerous convenience functions provided by standard library for dealing with uninitialized storage

Section 6.3.9

References

References I

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.*
   MIT Press, Cambridge, MA, USA, 3rd edition, 2009.
2. N. Dale. *C++ Plus Data Structures.*
   Jones and Bartlett, Sudbury, MA, USA, 3rd edition, 2003.
3. M. A. Weiss. *Data Structures and Algorithm Analysis in C++.*
   Pearson, Boston, MA, USA, 4th edition, 2014.
4. F. M. Carrano and J. J. Prichard. *Data Abstraction and Problem Solving With C++: Walls and Mirrors.*
   Addison Wesley, 3rd edition, 2001.
5. A. Drozdek. *Data Structures and Algorithms in C++.*
   Brooks/Cole, Pacific Grove, CA, USA, 2nd edition, 2001.

6. G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information.
Proc. of the USSR Academy of Sciences, 146:263–266, 1962.
In Russian.

7. R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms.
Acta Informatica, 1(4):290–306, 1972.

8. Boost C++ Libraries Web Site, `http://www.boost.org`.

9. M. Austern, The Standard Librarian: Defining Iterators and Const Iterators, Dr. Dobb's Journal, Jan. 2001, Available online at `http://www.drdobbs.com/the-standard-librarian-defining-iterato/184401331`.

10. R. Mattethat, Implementing Splay Trees in C++, Dr. Dobb's Journal, Sept. 2005, Available online at `http://www.drdobbs.com/cpp/implementing-splay-trees-in-c/184402007`.

Section 6.4

Finite-Precision Arithmetic

Code Example

- What do each of the following functions output when executed?

```
void func1() {
    double x = 0.1;
    double y = 0.3;
    double z = 0.4;
    if (x + y == z) {
        std::cout << "true\n";
    } else {
        std::cout << "false\n";
    }
}

void func2() {
    double x = 1e50;
    double y = -1e50;
    double z = 1.0;
    if (x + y + z == z + y + x) {
        std::cout << "true\n";
    } else {
        std::cout << "false\n";
    }
}

void func3() {
    for (double x = 0.0; x != 1.0; x += 0.1) {
        std::cout << "hello\n";
    }
}
```

Number Representations Using Different Radixes

- Note: All numbers are base 10, unless explicitly indicated otherwise.
- What is the representation of $\frac{1}{3}$ in base 3?
  $\frac{1}{3} = 0.\overline{3} = 0.1_3$
- What is the representation of $\frac{1}{10}$ in base 2?
  $\frac{1}{10} = 0.1 = 0.0\overline{0011}_2$
- A number may have a representation with a finite number of non-zero digits in one particular number base but not in another.
- Therefore, when a value must be represented with a limited number of significant digits, the number base matters (i.e., affects the approximation error).
- For example, in base 2, $\frac{1}{10}$ cannot be represented exactly using only a finite number of significant digits.
  $0.00011_2 = 0.09375$
  $0.000110011_2 = 0.099609375$
  $\ldots$

Finite-Precision Number Representations

- finite-precision number representation only capable of representing small fixed number of digits
- due to limited number of digits, many values cannot be represented exactly
- in cases that desired value cannot be represented exactly, choose nearest representable value (i.e., round to nearest representable value)
- finite-precision representations can suffer from error due to roundoff, underflow, and overflow
- two general classes of finite-precision representations:
  1. fixed-point representations
  2. floating-point representations

Fixed-Point Number Representations

- **fixed-point representation**: radix point remains fixed at same position in number
- if radix point fixed to right of least significant digit position, integer format results

Integer Format	$a_{n-1}\ a_{n-2} \cdots\ a_1\ a_0\ .$

- if radix point fixed to left of most significant digit position, purely fractional format results

Fractional Format	$.\ a_{n-1}\ a_{n-2} \cdots\ a_1\ a_0$

- fixed-point representations ***quite limited in range*** of values that can be represented
- numbers that vary greatly in magnitude cannot be represented easily using fixed-point representations
- one solution to range problem would be for programmer to maintain scaling factor for each fixed-point number, but this is clumsy and error prone

- **floating-point representation**: radix point is not fixed at particular position within number; instead radix point allowed to move and scaling factor automatically maintained to track position of radix point
- in general, floating-point value represents number x of form

$$x = sr^e,$$

- s is signed integer with fixed number of digits, and called **significand**
- e is signed integer with fixed number of digits, and called **exponent**
- r is integer satisfying $r \geq 2$, and called **radix**
- in practice, r typically 2
- for fixed r, representation of particular x not unique if no constraints placed on s and e (e.g., $5 \cdot 10^0 = 0.5 \cdot 10^1 = 0.05 \cdot 10^2$)

- to maximize number of significant digits in significand, s and e usually chosen such that first nonzero digit in significand is to immediate left of radix point (i.e., $1 \leq |s| < r$); number in this form called **normalized**; otherwise called **denormalized**
- other definitions of normalized/denormalized sometimes used but above one consistent with IEEE 754 standard
- Example:

$$
\begin{aligned}
0.75 &= 0.11_2 = 1.1_2 \cdot 2^{-1} \\
1.25 &= 1.01_2 = 1.01_2 \cdot 2^{0} \\
-0.5 &= -0.1_2 = -1.0_2 \cdot 2^{-1}
\end{aligned}
$$

IEEE 754 Standard (IEEE Std. 754-1985)

- most widely used standard for (binary) floating-point arithmetic
- specifies four floating-point formats: single, double, single extended, and double extended
- single and double formats called basic formats
- radix 2
- three integer parameters determine values representable in given format:
  - number p of significand bits (i.e., precision)
  - maximum exponent E_{max}
  - minimum exponent E_{min}
- parameters for four formats are as follows:

Parameter	Single	Single Extended	Double	Double Extended
p	24	≥ 32	53	≥ 64
E_{max}	127	> 1023	1023	≥ 16383
E_{min}	-126	≤ -1022	-1022	≤ -16382
Exponent bias	127	unspecified	1023	unspecified

- with each format, numbers of following form can be represented

$$(-1)^s 2^E (b_0.b_1 b_2 \cdots b_{p-1})$$

  where $s \in \{0,1\}$, E is integer satisfying $E_{\text{min}} \leq E \leq E_{\text{max}}$, and $b_i \in \{0,1\}$
- in addition, can represent four special values: $+\infty$, $-\infty$, signaling NaN, and quiet NaN
- NaNs produced by:
  - operations with at least one NaN operand
  - operations yielding indeterminate forms, such as $0/0$, $(\pm\infty)/(\pm\infty)$, $0 \cdot (\pm\infty)$, $(\pm\infty) \cdot 0$, $(+\infty) + (-\infty)$, and $(-\infty) + (\infty)$
  - real operations that yield complex results, such as square root of negative number, logarithm of negative number, inverse sine/cosine of number that lies outside $[-1,1]$

IEEE 754 Basic Formats

- always represent number in normalized form whenever possible; in such cases, $b_0 = 1$ and b_0 need not be stored explicitly as part of significand
- bit patterns with reserved exponent values (i.e., exponent values that lie outside the range $E_{\mathsf{min}} \le E \le E_{\mathsf{max}}$) used to represent ± 0, $\pm\infty$, denormalized numbers, and NaNs
- each of (basic) formats consist of three fields:
  - a sign bit, s
  - a biased exponent, $e = E +$ bias
  - a fraction, $f = .b_1 b_2 \cdots b_{p-1}$
- only difference between formats is size of biased exponent and fraction fields
- value represented by basic format number related to its sign, exponent, and fraction field, but relationship is complicated by the presence of zeros, infinities, and NaNs
- "strange" combination of biased and sign-magnitude formats used to encode floating-point value chosen so that nonnegative floating-point values ordered in same way as integers, allowing integer comparison to compare floating-point numbers

IEEE 754 Basic Formats (Continued)

- single format:

1	8	23
s	e	f
	MSB LSB	MSB LSB

- double format:

1	11	52
s	e	f
	MSB LSB	MSB LSB

- summary of encodings:

Case	Exponent	Fraction	Value
Normal	$E_{\text{min}} \le E \le E_{\text{max}}$	—	$(-1)^s 2^E (1+f)$
Denormal	$E = E_{\text{min}} - 1$	$f \neq 0$	$(-1)^s 2^{E_{\text{min}}} f$
Zero	$E = E_{\text{min}} - 1$	$f = 0$	$(-1)^s 0$
Infinity	$E = E_{\text{max}} + 1$	$f = 0$	$(-1)^s \infty$
NaN	$E = E_{\text{max}} + 1$	$f \neq 0$	NaN

- How would the number 5.25_{10} be represented in single format?
- $5.25_{10} = 101.01_2 \cdot 2^0 = 1.0101_2 \cdot 2^2$
- Therefore, $s = 0$, $e = 2_{10} + 127_{10} = 129_{10} = 10000001_2$, and $f = 0101000 \cdots 0$, resulting in the word:

0	10000001	01010000000000000000000
s	e	f

- How would the number -9.125_{10} be represented in double format?
- $-9.125_{10} = -1001.001_2 \cdot 2^0 = -1.001001_2 \cdot 2^3$
- Therefore, $s = 1$, $e = 3_{10} + 1023_{10} = 1026_{10} = 10000000010_2$, and $f = 001001000 \cdots 0$, resulting in the word:

1	10000000010	0010010000000000000000000000000000000000000000000000
s	e	f

Finite-Precision Arithmetic

- Understand the impact of using finite-precision arithmetic.
- Do not make invalid assumptions about the set of values that can be represented by a particular fixed-point or floating-point type.
- Integer arithmetic can ***overflow***. Be careful to avoid overflow.
- Floating-point arithmetic can ***overflow and underflow***.
- Perhaps, more importantly, however, floating-point arithmetic has ***roundoff error***. If you are not deeply troubled by the presence of roundoff error, you should be as it can cause major problems in many situations.

References I

1. D. Goldberg. What every computer scientist should know about floating-point arithmetic.
   ACM Computing Surveys, 23(1):5–48, Mar. 1991
2. IEEE Std. 754-1985 — IEEE standard for binary floating-point arithmetic, 1985
3. IEEE Std. 754-2008 — IEEE standard for floating-point arithmetic, 2008

Talks I

1. John Farrier, Demystifying Floating Point, CppCon, Bellevue, WA, USA, Sept. 24, 2015. Available online at `https://youtu.be/k12BJGSc2Nc`.

Section 6.5

Interval Arithmetic

Interval Arithmetic

- interval arithmetic is technique for placing bounds on error in numerical computation
- often values provided as input to numerical computation not known exactly, rather only known to within certain tolerance
- uncertainty may be due to measurement error or other factors
- consider numerous measured quantities that are provided as input to some numerical computation
- since measured quantity never known exactly (as measurement always introduces uncertainty), more natural to represent quantity by range
- therefore, would be convenient to have form of arithmetic that operates on values that correspond to ranges
- this is essentially what interval arithmetic does
- interval arithmetic represents each value as range of possibilities and defines set of rules for performing arithmetic on these ranges

Applications of Interval Arithmetic

- rounding error analysis in numerical algorithms
- filtered robust geometric predicates
- robustly finding intersection of curves and surfaces
- more robust root finding in ray tracing
- computing optimal solutions to geometric matching problems under bounded error
- finding polygonal approximations of implicit curves
- computer-assisted mathematical proofs

Real Interval Arithmetic

- in real interval arithmetic, each value is represented as real interval:

$$[a_1, a_2] = \{x \in \mathbb{R} \mid a_1 \leq x \leq a_2\}$$

- addition, subtraction, and multiplication defined as:

$$\begin{aligned} A + B &= \{a + b \mid a \in A \wedge b \in B\} \\ A - B &= \{a - b \mid a \in A \wedge b \in B\} \\ A \cdot B &= \{a \cdot b \mid a \in A \wedge b \in B\} \end{aligned}$$

- assuming division by interval containing 0 is not allowed, division defined as:

$$A/B = \{a/b \mid a \in A \wedge b \in B\}$$

Addition and Subtraction

- addition:

$$A+B=[a_1,a_2]+[b_1,b_2]=[a_1+b_1,a_2+b_2]$$

- negation:

$$-B=-[b_1,b_2]=[-b_2,-b_1]$$

- formula for negation follows from fact that:
  - $x \geq b_1 \Rightarrow -x \leq -b_1$ and
  - $x \leq b_2 \Rightarrow -x \geq -b_2$
- subtraction:

$$A-B=[a_1,a_2]-[b_1,b_2]=[a_1-b_2,a_2-b_1]$$

- formula for subtraction follows from combining addition and negation

- multiplication:

$$A \cdot B = [a_1, a_2] \cdot [b_1, b_2] = [\min\{a_1 b_1, a_1 b_2, a_2 b_1, a_2 b_2\}, \max\{a_1 b_1, a_1 b_2, a_2 b_1, a_2 b_2\}]$$

(e.g., $[a_1, a_2] \cdot [b_1, b_2] = [a_1 b_1, a_2 b_2]$ if $0 \leq a_1 \leq a_2$ and $0 \leq b_1 \leq b_2$)

- reciprocal (assuming division by interval containing 0 not allowed):

$$1/B = 1/[b_1, b_2] = [1/b_2, 1/b_1]$$

- formula for reciprocal follows from fact that, since $0 \notin [b_1, b_2]$, $x \in [b_1, b_2], b_1, b_2$ all have same sign (implying $b_1 x > 0$ and $b_2 x > 0$) and consequently:
  - $x \geq b_1 \Rightarrow \frac{x}{b_1 x} \geq \frac{b_1}{b_1 x} \Rightarrow 1/b_1 \geq 1/x \Rightarrow 1/x \leq 1/b_1$
  - $x \leq b_2 \Rightarrow \frac{x}{b_2 x} \leq \frac{b_2}{b_2 x} \Rightarrow 1/b_2 \leq 1/x \Rightarrow 1/x \geq 1/b_2$
- division (assuming division by interval containing 0 not allowed):

$$A/B = [a_1, a_2]/[b_1, b_2] = [\min\{a_1/b_1, a_1/b_2, a_2/b_1, a_2/b_2\}, \max\{a_1/b_1, a_1/b_2, a_2/b_1, a_2/b_2\}]$$

- formula for division follows from fact that division is simply multiplication by reciprocal

Allowing Division By Interval Containing Zero

- consider implications of allowing division by interval containing zero
- reciprocal, if $0 \in [b_1, b_2]$:

$$1/B = 1/[b_1, b_2] = \begin{cases} (-\infty, 1/b_1] & b_1 \neq 0, b_2 = 0 \\ [1/b_2, +\infty) & b_1 = 0, b_2 \neq 0 \\ (-\infty, 1/b_1] \cup [1/b_2, +\infty) & b_1 \neq 0, b_2 \neq 0 \\ \emptyset & b_1 = b_2 = 0 \end{cases}$$

- thus, if division by interval containing 0 is allowed, result cannot always be represented by interval of form

$$[a_1, a_2] = \{x \in \mathbb{R} \mid a_1 \leq x \leq a_2\}$$

- in particular, arithmetic can yield result that corresponds to:
  - interval unbounded at one end
  - empty set
  - union of two separate intervals

Allowing Division By Interval Containing Zero (Continued)

- to accommodate division by interval containing zero, represent sets of following forms:

$$
\begin{gathered}
[a_1, a_2] = \{x \in \mathbb{R} \mid a_1 \leq x \leq a_2\} \\
[a_1, +\infty) = \{x \in \mathbb{R} \mid x \geq a_1\} \\
(-\infty, a_2] = \{x \in \mathbb{R} \mid x \leq a_2\} \\
(-\infty, +\infty) \\
\emptyset
\end{gathered}
$$

- for sake of simplicity, result of form $(-\infty, \beta_1] \cup [\beta_2, +\infty)$ (where $\beta_1 < \beta_2$) is mapped to $(-\infty, +\infty)$

Floating-Point Interval Arithmetic

- in case of floating-point interval arithmetic, interval bounds are floating-point values
- that is, represent intervals of following form, where F is set of machine-representable real numbers:

$$[a_1, a_2] = \{x \in F \mid a_1 \leq x \leq a_2\}$$

- since floating-point value can only represent finite number of real numbers, some real numbers cannot be represented exactly
- when arithmetic operation performed, result must always be rounded to machine-representable value
- processor typically allows for control over how rounding performed by supporting several rounding modes, such as:
  - round to nearest
  - round towards zero
  - round upwards (i.e., towards $+\infty$)
  - round downwards (i.e., towards $-\infty$)

Floating-Point Interval Arithmetic (Continued)

- must ensure that rounding does not cause interval to no longer bracket result that would be obtained by (exact) real interval arithmetic
- need to select shortest interval that contains result that would be obtained from (exact) real interval arithmetic
- lower bound of result must be computed with rounding downwards
- upper bound of result must be computed with rounding upwards
- using rounding in this way ensures that resulting interval will bracket idealized (exact real) interval

Floating-Point Interval Arithmetic Operations

$$A + B = [a_1, a_2] + [b_1, b_2] = [a_1 \overset{+}{\bigtriangledown} b_1, a_2 \overset{+}{\bigtriangleup} b_2]$$

$$A - B = [a_1, a_2] - [b_1, b_2] = [a_1 \overset{-}{\bigtriangledown} b_2, a_2 \overset{-}{\bigtriangleup} b_1]$$

$$A \cdot B = [a_1, a_2] \cdot [b_1, b_2]$$

	$b_2 \leq 0$	$b_1 < 0 < b_2$	$b_1 \geq 0$
$a_2 \leq 0$	$[a_2 \overset{*}{\bigtriangledown} b_2, a_1 \overset{*}{\bigtriangleup} b_1]$	$[a_1 \overset{*}{\bigtriangledown} b_2, a_1 \overset{*}{\bigtriangleup} b_1]$	$[a_1 \overset{*}{\bigtriangledown} b_2, a_2 \overset{*}{\bigtriangleup} b_1]$
$a_1 < 0 < a_2$	$[a_2 \overset{*}{\bigtriangledown} b_1, a_1 \overset{*}{\bigtriangleup} b_1]$	$[\min\{a_1 \overset{*}{\bigtriangledown} b_2, a_2 \overset{*}{\bigtriangledown} b_1\},$ $\max\{a_1 \overset{*}{\bigtriangleup} b_1, a_2 \overset{*}{\bigtriangleup} b_2]\}$	$[a_1 \overset{*}{\bigtriangledown} b_2, a_2 \overset{*}{\bigtriangleup} b_2]$
$a_1 \geq 0$	$[a_2 \overset{*}{\bigtriangledown} b_1, a_1 \overset{*}{\bigtriangleup} b_2]$	$[a_2 \overset{*}{\bigtriangledown} b_1, a_2 \overset{*}{\bigtriangleup} b_2]$	$[a_1 \overset{*}{\bigtriangledown} b_1, a_2 \overset{*}{\bigtriangleup} b_2]$

$$A / B = [a_1, a_2] / [b_1, b_2] \text{ where } 0 \notin [b_1, b_2]$$

	$b_2 < 0$	$b_1 > 0$
$a_2 \leq 0$	$[a_2 \overset{/}{\bigtriangledown} b_1, a_1 \overset{/}{\bigtriangleup} b_2]$	$[a_1 \overset{/}{\bigtriangledown} b_1, a_2 \overset{/}{\bigtriangleup} b_2]$
$a_1 < 0 < a_2$	$[a_2 \overset{/}{\bigtriangledown} b_2, a_1 \overset{/}{\bigtriangleup} b_2]$	$[a_1 \overset{/}{\bigtriangledown} b_1, a_2 \overset{/}{\bigtriangleup} b_1]$
$a_1 \geq 0$	$[a_2 \overset{/}{\bigtriangledown} b_2, a_1 \overset{/}{\bigtriangleup} b_1]$	$[a_1 \overset{/}{\bigtriangledown} b_2, a_2 \overset{/}{\bigtriangleup} b_1]$

Allowing Division by Intervals Containing Zero

- to accommodate division by intervals containing zero, represent intervals of form

$$
\begin{gathered}
[a_1, a_2] = \{x \in \mathbb{R} \mid a_1 \leq x \leq a_2\} \\
[a_1, +\infty) = \{x \in \mathbb{R} \mid x \geq a_1\} \\
(-\infty, a_2] = \{x \in \mathbb{R} \mid x \leq a_2\} \\
(-\infty, +\infty) \\
\emptyset
\end{gathered}
$$

- arithmetic operations as defined on subsequent slides
- if any operand is $\emptyset$, result of operation is also $\emptyset$

$$A + B$$

	$(-\infty, b_2]$	$[b_1, b_2]$	$[b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, a_2]$	$(-\infty, a_2 \mathbin{\overset{+}{\bigtriangleup}} b_2]$	$(-\infty, a_2 \mathbin{\overset{+}{\bigtriangleup}} b_2]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, a_2]$	$(-\infty, a_2 \mathbin{\overset{+}{\bigtriangleup}} b_2]$	$[a_1 \mathbin{\overset{+}{\bigtriangledown}} b_1, a_2 \mathbin{\overset{+}{\bigtriangleup}} b_2]$	$[a_1 \mathbin{\overset{+}{\bigtriangledown}} b_1, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty)$	$(-\infty, +\infty)$	$[a_1 \mathbin{\overset{+}{\bigtriangledown}} b_1, +\infty)$	$[a_1 \mathbin{\overset{+}{\bigtriangledown}} b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

$$A - B$$

	$(-\infty, b_2]$	$[b_1, b_2]$	$[b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, a_2]$	$(-\infty, +\infty)$	$(-\infty, a_2 \mathbin{\overset{-}{\bigtriangleup}} b_1]$	$(-\infty, a_2 \mathbin{\overset{-}{\bigtriangleup}} b_1]$	$(-\infty, +\infty)$
$[a_1, a_2]$	$[a_1 \mathbin{\overset{-}{\bigtriangledown}} b_2, +\infty)$	$[a_1 \mathbin{\overset{-}{\bigtriangledown}} b_2, a_2 \mathbin{\overset{-}{\bigtriangleup}} b_1]$	$(-\infty, a_2 \mathbin{\overset{-}{\bigtriangleup}} b_1]$	$(-\infty, +\infty)$
$[a_1, +\infty)$	$[a_1 \mathbin{\overset{-}{\bigtriangledown}} b_2, +\infty)$	$[a_1 \mathbin{\overset{-}{\bigtriangledown}} b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Multiplication

$A \cdot B$

	$[b_1, b_2]$ $b_2 \leq 0$	$[b_1, b_2]$ $b_1 < 0 < b_2$	$[b_1, b_2]$ $b_1 \geq 0$	$[0, 0]$
$[a_1, a_2]$ $a_2 \leq 0$	$[a_2 \overset{*}{\triangledown} b_2, a_1 \overset{*}{\vartriangle} b_1]$	$[a_1 \overset{*}{\triangledown} b_2, a_1 \overset{*}{\vartriangle} b_1]$	$[a_1 \overset{*}{\triangledown} b_2, a_2 \overset{*}{\vartriangle} b_1]$	$[0, 0]$
$[a_1, a_2]$ $a_1 < 0 < a_2$	$[a_2 \overset{*}{\triangledown} b_1, a_1 \overset{*}{\vartriangle} b_1]$	$[\min\{a_1 \overset{*}{\triangledown} b_2, a_2 \overset{*}{\triangledown} b_1\},$ $\max\{a_1 \overset{*}{\vartriangle} b_1, a_2 \overset{*}{\vartriangle} b_2\}]$	$[a_1 \overset{*}{\triangledown} b_2, a_2 \overset{*}{\vartriangle} b_2]$	$[0, 0]$
$[a_1, a_2]$ $a_2 \geq 0$	$[a_2 \overset{*}{\triangledown} b_1, a_1 \overset{*}{\vartriangle} b_2]$	$[a_2 \overset{*}{\triangledown} b_1, a_2 \overset{*}{\vartriangle} b_2]$	$[a_1 \overset{*}{\triangledown} b_1, a_2 \overset{*}{\vartriangle} b_2]$	$[0, 0]$
$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$
$(-\infty, a_2]$ $a_2 \leq 0$	$[a_2 \overset{*}{\triangledown} b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2 \overset{*}{\vartriangle} b_1]$	$[0, 0]$
$(-\infty, a_2]$ $a_2 \geq 0$	$[a_2 \overset{*}{\triangledown} b_1, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2 \overset{*}{\vartriangle} b_2]$	$[0, 0]$
$[a_1, +\infty)$ $a_1 \leq 0$	$(-\infty, a_1 \overset{*}{\vartriangle} b_1]$	$(-\infty, +\infty)$	$[a_1 \overset{*}{\triangledown} b_2, +\infty)$	$[0, 0]$
$[a_1, +\infty)$ $a_1 \geq 0$	$(-\infty, a_1 \overset{*}{\vartriangle} b_2]$	$(-\infty, +\infty)$	$[a_1 \overset{*}{\triangledown} b_1, +\infty)$	$[0, 0]$
$(-\infty, \infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[0, 0]$

$A \cdot B$

	$(-\infty, b_2]$ $b_2 \leq 0$	$(-\infty, b_2]$ $b_2 \geq 0$	$[b_1, +\infty)$ $b_1 \leq 0$	$[b_1, +\infty)$ $b_1 \geq 0$	$(-\infty, +\infty)$
$[a_1, a_2]$ $a_2 \leq 0$	$[a_2 \overset{*}{\bigtriangledown} b_2, +\infty)$	$[a_1 \overset{*}{\bigtriangledown} b_2, +\infty)$	$(-\infty, a_1 \overset{*}{\bigtriangleup} b_1]$	$(-\infty, a_2 \overset{*}{\bigtriangleup} b_1]$	$(-\infty, +\infty)$
$[a_1, a_2]$ $a_1 < 0 < a_2$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, a_2]$ $a_2 \geq 0$	$(-\infty, a_1 \overset{*}{\bigtriangleup} b_2]$	$(-\infty, a_2 \overset{*}{\bigtriangleup} b_2]$	$[a_2 \overset{*}{\bigtriangledown} b_1, +\infty)$	$[a_1 \overset{*}{\bigtriangledown} b_1, +\infty)$	$(-\infty, +\infty)$
$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$
$(-\infty, a_2]$ $a_2 \leq 0$	$[a_2 \overset{*}{\bigtriangledown} b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2 \overset{*}{\bigtriangleup} b_1]$	$(-\infty, +\infty)$
$(-\infty, a_2]$ $a_2 \geq 0$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty)$ $a_1 \leq 0$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty)$ $a_1 \geq 0$	$(-\infty, a_1 \overset{*}{\bigtriangleup} b_2]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[a_1 \overset{*}{\bigtriangledown} b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, \infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Division

A/B, $0 \notin B$

	$[b_1, b_2]$ $b_2 < 0$	$[b_1, b_2]$ $b_1 > 0$	$(-\infty, b_2]$ $b_2 < 0$	$[b_1, +\infty)$ $b_1 > 0$
$[a_1, a_2]$ $a_2 \leq 0$	$[a_2 \overset{/}{\bigtriangledown} b_1, a_1 \overset{/}{\bigtriangleup} b_2]$	$[a_2 \overset{/}{\bigtriangledown} b_1, a_2 \overset{/}{\bigtriangleup} b_2]$	$[0, a_1 \overset{/}{\bigtriangleup} b_2]$	$[a_1 \overset{/}{\bigtriangledown} b_1, 0]$
$[a_1, a_2]$ $a_1 < 0 < a_2$	$[a_2 \overset{/}{\bigtriangledown} b_2, a_1 \overset{/}{\bigtriangleup} b_2]$	$[a_1 \overset{/}{\bigtriangledown} b_1, a_2 \overset{/}{\bigtriangleup} b_1]$	$[a_2 \overset{/}{\bigtriangledown} b_2, a_1 \overset{/}{\bigtriangleup} b_2]$	$[a_1 \overset{/}{\bigtriangledown} b_1, a_2 \overset{/}{\bigtriangleup} b_1]$
$[a_1, a_2]$ $a_1 \geq 0$	$[a_2 \overset{/}{\bigtriangledown} b_2, a_1 \overset{/}{\bigtriangleup} b_1]$	$[a_1 \overset{/}{\bigtriangledown} b_2, a_2 \overset{/}{\bigtriangleup} b_1]$	$[a_2 \overset{/}{\bigtriangledown} b_2, 0]$	$[0, a_2 \overset{/}{\bigtriangleup} b_1]$
$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$
$(-\infty, a_2]$ $a_2 \leq 0$	$[a_2 \overset{/}{\bigtriangledown} b_1, +\infty)$	$(-\infty, a_2 \overset{/}{\bigtriangleup} b_2]$	$[0, +\infty)$	$(-\infty, 0]$
$(-\infty, a_2]$ $a_2 \geq 0$	$[a_2 \overset{/}{\bigtriangledown} b_2, +\infty)$	$(-\infty, a_2 \overset{/}{\bigtriangleup} b_1]$	$[a_2 \overset{/}{\bigtriangledown} b_2, +\infty)$	$(-\infty, a_2 \overset{/}{\bigtriangleup} b_1]$
$[a_1, +\infty)$ $a_1 \leq 0$	$(-\infty, a_1 \overset{/}{\bigtriangleup} b_2]$	$[a_1 \overset{/}{\bigtriangledown} b_1, +\infty)$	$(-\infty, a_1 \overset{/}{\bigtriangleup} b_2]$	$[a_1 \overset{/}{\bigtriangledown} b_1, +\infty)$
$[a_1, +\infty)$ $a_1 \geq 0$	$(-\infty, a_1 \overset{/}{\bigtriangleup} b_1]$	$(a_1 \overset{/}{\bigtriangledown} b_2, +\infty)$	$(-\infty, 0]$	$[0, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

$A/B,\, 0 \in B$

	$[0,0]$	$[b_1,b_2]$ $b_1 < b_2 = 0$	$[b_1,b_2]$ $0 = b_1 < b_2$	$(-\infty,b_2]$ $b_2 = 0$	$[b_1,+\infty)$ $b_1 = 0$	$(-\infty,+\infty)$
$[a_1,a_2]$ $a_2 < 0$	$\emptyset$	$[a_2 \mathbin{\triangledown\!\!\!/} b_1, +\infty)$	$(-\infty, a_2 \mathbin{\triangle\!\!\!/} b_2]$	$[0,+\infty)$	$(-\infty,0]$	$(-\infty,+\infty)$
$[a_1,a_2]$ $a_1 \leq 0 \leq a_2$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$
$[a_1,a_2]$ $a_1 > 0$	$\emptyset$	$(-\infty, a_1 \mathbin{\triangle\!\!\!/} b_1]$	$[a_1 \mathbin{\triangledown\!\!\!/} b_2, +\infty)$	$(-\infty,0]$	$[0,+\infty)$	$(-\infty,+\infty)$
$(-\infty,a_2]$ $a_2 < 0$	$\emptyset$	$[a_2 \mathbin{\triangledown\!\!\!/} b_1, +\infty)$	$(-\infty, a_2 \mathbin{\triangle\!\!\!/} b_2]$	$[0,+\infty)$	$(-\infty,0]$	$(-\infty,+\infty)$
$(-\infty,a_2]$ $a_2 > 0$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$
$[a_1,+\infty)$ $a_1 < 0$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$
$(a_1,+\infty)$ $a_1 > 0$	$\emptyset$	$(-\infty, a_1 \mathbin{\triangle\!\!\!/} b_1]$	$[a_1 \mathbin{\triangledown\!\!\!/} b_2, +\infty)$	$(-\infty,0]$	$[0,+\infty)$	$(-\infty,+\infty)$
$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$

Comparisons

- definition of comparison operations introduces some complications
- many ways in which comparison operations might be defined
- for comparison operator $\circ$ (i.e., equality, inequality, less than, greater than, less than or equal, greater than or equal), one possible way to define $[a_1, a_2] \circ [b_1, b_2]$ would be as follows:
  - yields true if $x \circ y$ is satisfied for all $x \in [a_1, a_2]$ and all $y \in [b_1, b_2]$,
  - yields false if $x \circ y$ is violated for all $x \in [a_1, a_2]$ and all $y \in [b_1, b_2]$,
  - yields indeterminate (or throws exception) otherwise
- for example, with preceding definition:
  - $[0, 1] \leq [1, 2]$ would be true
  - $[0, 1] \leq [-2, -1]$ would be false
  - $[0, 2] \leq [1, 3]$ would be indeterminate
  - $[0, 1] = [0, 0]$ would be indeterminate
- above definition of comparison operations is particularly useful in number applications

Setting and Querying Rounding Mode

- header file `cfenv` contains various information relevant to floating-point environment
- defines macros (which expand to nonnegative integral constants) for following rounding modes:
  - `FE_TOWARDZERO`: round towards zero
  - `FE_TONEAREST`: round to nearest representable value
  - `FE_UPWARD`: round towards positive infinity
  - `FE_DOWNWARD`: round towards negative infinity
- current rounding mode can be set with `std::fesetround`
- `int fesetround(int round)`
  - attempts to set current rounding mode to `round`
  - returns 0 upon success non-zero value otherwise
- current rounding mode can be queried with `std::fegetround`
- `int fegetround()`
  - returns value of current rounding mode
- floating-point environment access and modification only meaningful when `#pragma STDC FENV_ACCESS` is supported and set to `ON`

Impact of Current Rounding Mode

- current rounding mode ***affects***:
  - results of floating-point arithmetic operations outside of constant expressions
  - results of standard library mathematical functions (e.g., `sin`, `cos`, `tan`, `exp`, `log`, and `sqrt`)
  - floating-point to floating-point implicit conversion and casts
  - string conversions (e.g., `strtod`)
  - library rounding functions `nearbyint`, `rint`, and `lrint`
- current rounding mode ***does not affect***:
  - floating-point to integer implicit conversions and casts (which are always towards zero)
  - results of floating-point arithmetic operations in constant expressions (which are always to nearest)
  - library functions `round`, `lround`, `ceil`, `floor`, and `trunc`
- behavior of many things affected by current rounding mode
- since some algorithms may rely on use of particular rounding mode, one must be careful to always restore previous rounding mode

Rounding Mode Example

```
#include <iostream>
#include <cmath>
#include <cfenv>
#include <limits>

#pragma STDC FENV_ACCESS ON

int main() {
    std::cout.precision(std::numeric_limits<double>::max_digits10);
    int old_mode = std::fegetround();
    int modes[] = {FE_TONEAREST, FE_TOWARDZERO, FE_UPWARD, FE_DOWNWARD};
    for (auto mode : modes) {
        if (std::fesetround(mode)) {abort();}
        std::cout << std::sqrt(2.0) << '\n';
    }
    if (std::fesetround(old_mode)) {abort();}
    std::cout << std::sqrt(2.0) << '\n';
}

/* Example output:
1.4142135623730951
1.4142135623730951
1.4142135623730952
1.4142135623730951
1.4142135623730951
*/
```

Section 6.5.1

Applications in Geometry Processing

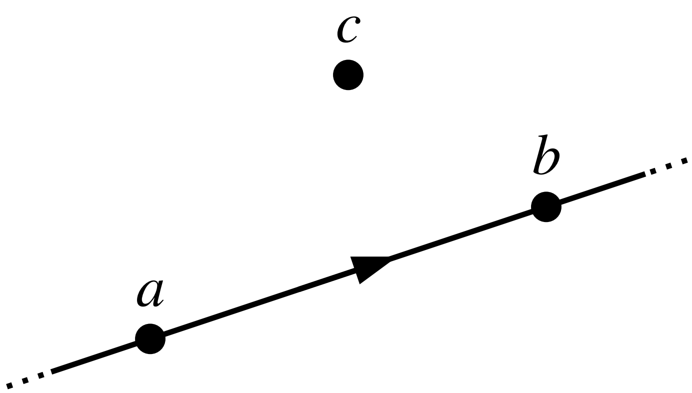
Geometric Predicates

- interval arithmetic frequently employed in geometry processing
- one application of interval arithmetic is for efficient implementation of exact geometric predicates
- geometric predicate tests for one of small number of possibilities involving geometric objects such as points, lines, and planes
- some basic geometric predicates include tests for such things as:
  - on which side of oriented line point located (i.e., 2-dimensional orientation test)
  - on which side of oriented plane point located (i.e., 3-dimensional orientation test)
  - on which side of circle point located (i.e., in-circle test)
  - on which side of sphere point located (i.e., in-sphere test)
- geometric predicates like those above essential in many geometric algorithms
- exact predicate is one that must always yield correct result

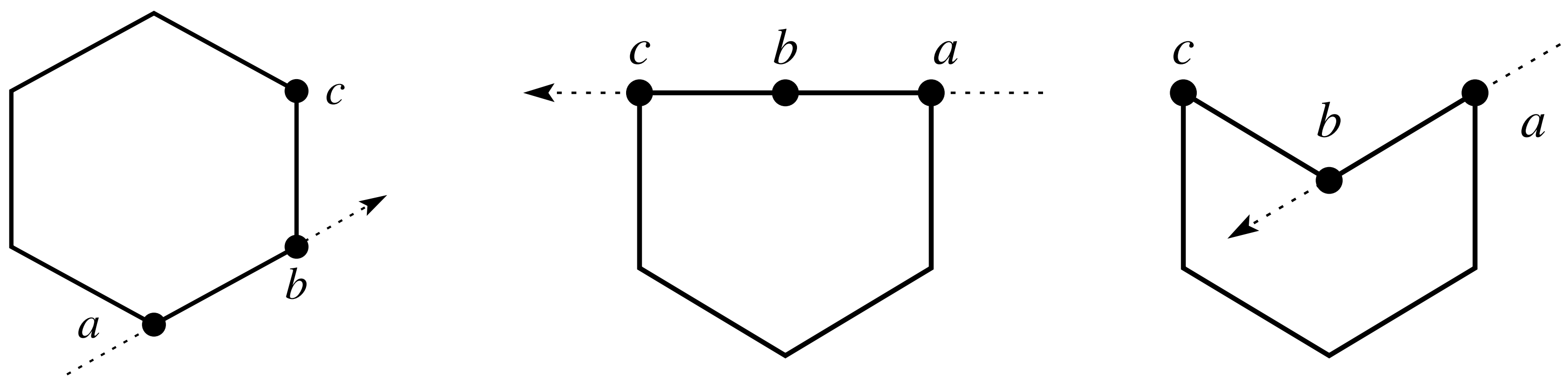
Filtered Geometric Predicates

- determining result of geometric predicate involves arithmetic computation
- if arithmetic used for computation not exact, predicate may yield incorrect result
- vast majority of algorithms cannot tolerate incorrect results from predicates
- unfortunately, using exact arithmetic extremely costly
- use interval arithmetic to quickly determine bound on numerical results of interest
- if bound obtained from interval arithmetic sufficient to make determination of predicate result, high cost of using exact arithmetic avoided
- only if bound insufficient, recompute result using exact arithmetic
- in practice, interval arithmetic often sufficient to determine predicate result, leading to great increase in efficiency

Two-Dimensional Orientation Test

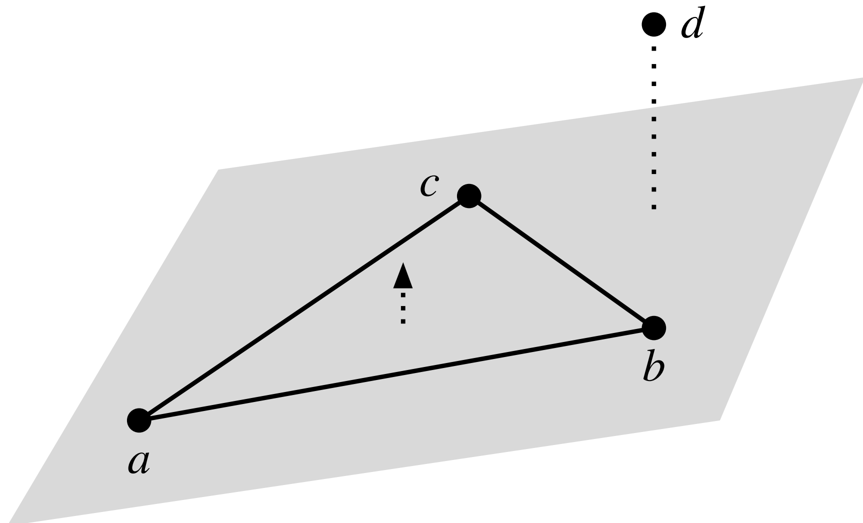


- given three points $a = (a_x, a_y)$, $b = (b_x, b_y)$, and $c = (c_x, c_y)$ in $\mathbb{R}^2$, determine to which side of directed line through a and b point c lies
- can be determined from sign of determinant of 2×2 matrix
- $\operatorname{orient2d}(a, b, c) = \det \begin{bmatrix} a_x - c_x & b_x - c_x \\ a_y - c_y & b_y - c_y \end{bmatrix}$
- if $\operatorname{orient2d}(a, b, c)$ is positive, negative, or zero, then c is to left of, to right of, or collinear with directed line (through a and b), respectively



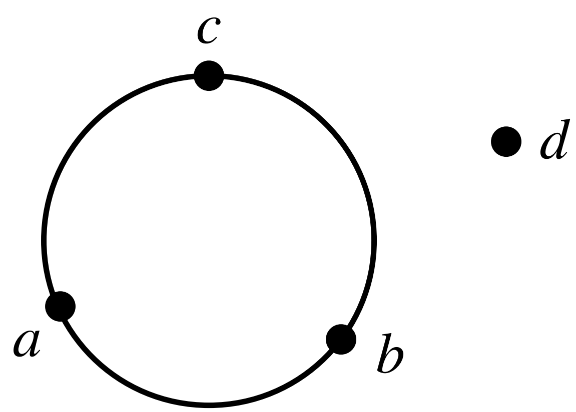
- let a, b, and c be three consecutive vertices of polygon in counterclockwise (CCW) order
- polygon is ***strictly convex*** if and only if, for every choice of a, b, c, c is to left of directed line through ab (i.e., $\mathrm{orient2d}(a, b, c) > 0$)
- polygon is ***convex*** if and only if, for every choice of a, b, and c, c is to left of or collinear with directed line through ab (i.e., $\mathrm{orient2d}(a, b, c) \geq 0$)

Three-Dimensional Orientation Test



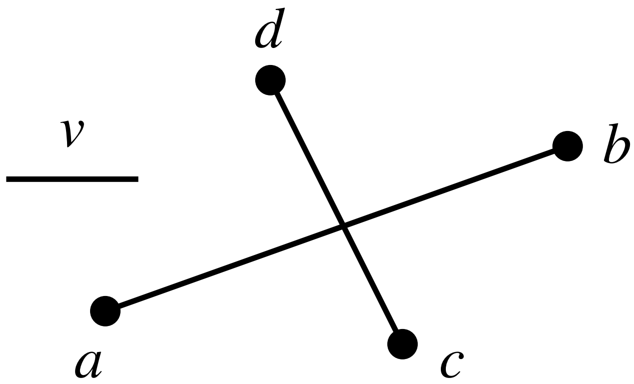
- given four points $a = (a_x, a_y, a_z)$, $b = (b_x, b_y, b_z)$, $c = (c_x, c_y, c_z)$, and $d = (d_x, d_y, d_z)$ in $\mathbb{R}^3$, determine to which side of oriented plane through a, b, and c point d lies
- can be determined from sign of determinant of 3×3 matrix
- $\operatorname{orient3d}(a, b, c, d) = \det \begin{bmatrix} a_x - d_x & b_x - d_x & c_x - d_x \\ a_y - d_y & b_y - d_y & c_y - d_y \\ a_z - d_z & b_z - d_z & c_z - d_z \end{bmatrix}$
- if $\operatorname{orient3d}(a, b, c, d)$ is positive, negative, or zero, then d lies below, above, or is coplanar with oriented plane (through a, b, and c), respectively

In-Circle Test



- given four points $a = (a_x, a_y)$, $b = (b_x, b_y)$, $c = (c_x, c_y)$, and $d = (d_x, d_y)$ in $\mathbb{R}^2$, determine whether d is inside, outside, or on the circle through a, b, and c
- can be determined from sign of determinant of 3×3 matrix
- project points a, b, c, and d upwards (in third dimension) onto paraboloid $f(x,y) = x^2 + y^2$ to obtain four points (in $\mathbb{R}^3$) a', b', c', and d'; then perform 3-dimensional orientation test on resulting four points
- $\operatorname{inCircle}(a,b,c,d) = \operatorname{orient3d}(a',b',c',d')$, where $a' = (a_x, a_y, a_x^2 + a_y^2)$, $b' = (b_x, b_y, b_x^2 + b_y^2)$, $c' = (c_x, c_y, c_x^2 + c_y^2)$, and $d' = (d_x, d_y, d_x^2 + d_y^2)$
- if $\operatorname{inCircle}(a,b,c,d)$ is positive, negative, or zero, then d lies respectively inside, outside, or on the circle through a, b, and c

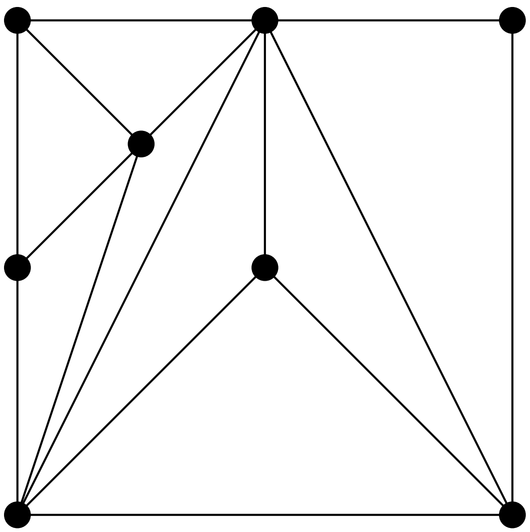
Preferred-Direction Test



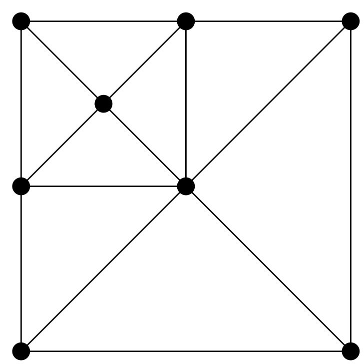
- given two line segments ab and cd and vector v, determine if, compared to orientation of cd, orientation of ab is more close, less close, or equally close to the orientation of v
- can be determined from result of computation involving dot products
- $\mathrm{prefDir}(a,b,c,d,v) = |d-c|^2\,((b-a)\cdot v)^2 - |b-a|^2\,((d-c)\cdot v)^2$
- if $\mathrm{prefDir}(a,b,c,d,v)$ is positive, negative, or zero, then compared to orientation of cd, orientation of ab is more close, less close, or equally close to orientation of v, respectively

Triangulations

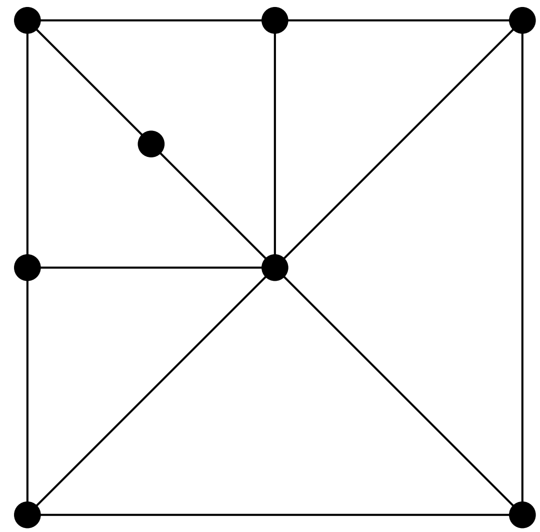
- A **triangulation** of a set V of vertices is a set T of triangles such that:
  - the union of the vertices of all triangles in T is V;
  - the interiors of any two triangles in T are disjoint; and
  - the union of the triangles in T is the convex hull of V.



Triangulation

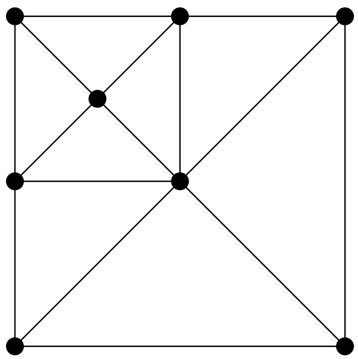


Triangulation

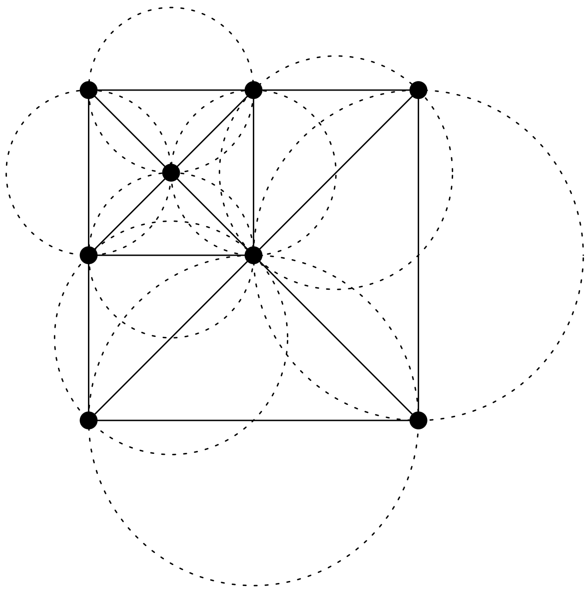


Invalid Triangulation

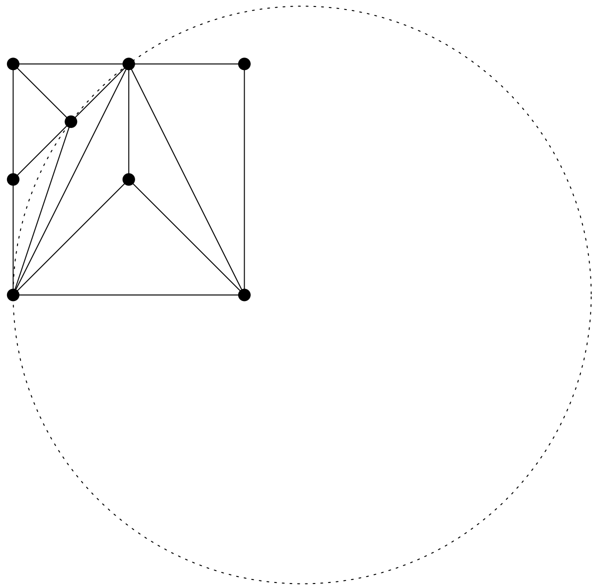
- A triangulation is said to be **Delaunay** if each triangle in the triangulation is such that the interior of its ***circumcircle contains no vertices***.



Delaunay Triangulation



Delaunay Triangulation Showing Circumcircles



Non-Delaunay Triangulation Showing Violation of Circumcircle Condition

Comments on Delaunay Triangulations

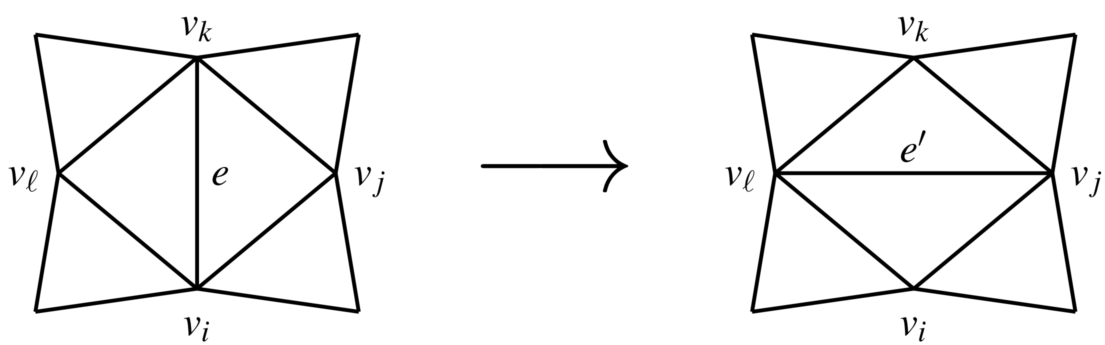
- Delaunay triangulation maximizes minimum interior angle of all triangles in triangulation
- avoids long-thin triangles to whatever extent is possible
- long-thin triangles often undesirable for interpolation purposes; can lead to large discretization error and large errors in derivatives
- Delaunay triangulation only guaranteed to be unique if no four points are cocircular
- when not unique, schemes exist for making unique choice from set of all possible Delaunay triangulations, such as one proposed in:

  C. Dyken and M. S. Floater. Preferred directions for resolving the non-uniqueness of Delaunay triangulations.
  Computational Geometry—Theory and Applications, 34:96–101, 2006.

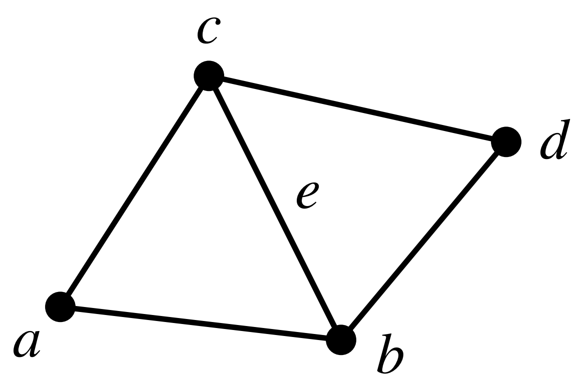
- dual graph of Delaunay triangulation is Voronoi diagram (circumcircle centers become vertices, original vertices become faces)

Edge Flips

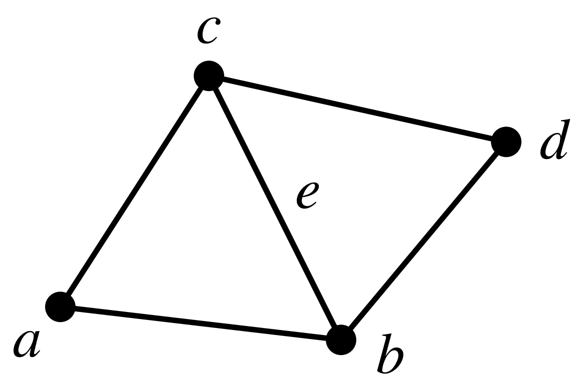
- edge e in triangulation said to be **flippable** if e has two incident faces (i.e., is not on triangulation boundary) and union of these two faces is strictly convex quadrilateral q.
- if e is flippable, valid triangulation obtained if e deleted from triangulation and replaced by other diagonal e' of quadrilateral q
- such transformation known as **edge flip**
- edge-flip example:



- number of different triangulations of n vertices upper bounded by $\binom{n}{2} = \frac{n^2-n}{2}$, which is $O(n^2)$
- all triangulations of set of vertices have same number of edges
- every triangulation reachable from every other triangulation by edge flips



- given flippable edge e in triangulation with incident faces abc and dcb (whose union is strictly convex quadrilateral), determine if e is locally Delaunay
- result of predicate can be determined using in-circle test
- define:
  $$\mathrm{localDelaunay}(a,b,c,d) = \begin{cases} 1 & \mathrm{inCircle}(a,b,c,d) \geq 0 \\ 0 & \mathrm{inCircle}(a,b,c,d) < 0 \end{cases}$$
- if $\mathrm{localDelaunay}(a,b,c,d) \neq 0$, edge e is locally Delaunay
- if every flippable edge in triangulation is locally Delaunay, triangulation is Delaunay



- given flippable edge e in triangulation with incident faces abc and dcb (whose union is strictly convex quadrilateral), determine if e is locally Delaunay with preferred directions given by vectors u and v (where u and v are nonzero and neither parallel nor orthogonal)
- result of predicate can be determined using in-circle and preferred-direction tests
- define:

$$\alpha(a,b,c,d,u,v) = \begin{cases} 1 & \operatorname{prefDir}(b,c,a,d,u) > 0 \\ 0 & \operatorname{prefDir}(b,c,a,d,u) < 0 \\ 1 & \operatorname{prefDir}(b,c,a,d,u) = 0 \text{ and } \operatorname{prefDir}(b,c,a,d,v) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Preferred-Directions Local-Delaunay Test (Continued)

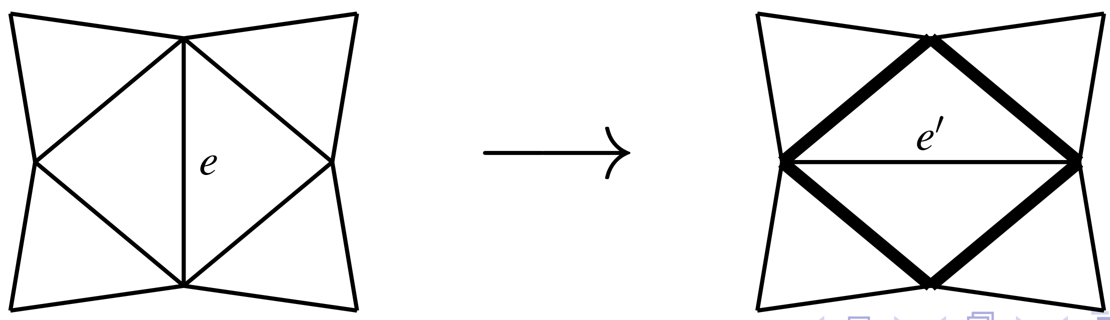
- define:
  $$\mathrm{localPrefDirDelaunay}(a,b,c,d,u,v) = \begin{cases} 1 & \mathrm{inCircle}(a,b,c,d) > 0 \\ 0 & \mathrm{inCircle}(a,b,c,d) < 0 \\ \alpha(a,b,c,d,u,v) & \text{otherwise} \end{cases}$$
- if (and only if) $\mathrm{localPrefDirDelaunay}(a,b,c,d,u,v) \neq 0$, edge e is locally Delaunay with preferred directions u and v
- if every flippable edge in triangulation is locally preferred-directions Delaunay, triangulation is preferred-directions Delaunay

Lawson Local Optimization Procedure (LOP)

- Lawson local optimization procedure (LOP) finds optimal triangulation of set of points via edge flips
- flippable edge said to be **optimal** if: 1) it is not flippable; or 2) it is flippable and satisfies some optimality criterion, such as the locally-Delaunay or preferred-directions locally-Delaunay condition
- edge said to be **suspect** if its optimality is currently uncertain
- initially, all flippable edges are marked as suspect
- while at least one suspect edge remains, perform following:
  - select suspect edge e
  - if edge e is optimal, mark e as not suspect; otherwise, flip e to obtain edge e', mark e' as not suspect, and mark any edges whose optimality might be affected by flip of e as suspect
- essentially, LOP simply keeps flipping (flippable) edges that are not optimal until all edges are optimal

Finding Delaunay Triangulations with Lawson LOP

- given any triangulation of set P of points, can compute Delaunay triangulation of P using Lawson LOP
- select optimality criterion as locally-Delaunay or preferred-directions locally-Delaunay condition
- when edge flipped, which edges can have their optimality affected?
- let e denote edge being flipped
- let q denote quadrilateral formed by union of two faces incident on e
- let e' denote edge obtained by applying edge flip to e
- edges that should be marked as suspect are all flippable edges belonging to q
- for example, if edge e' was produced by flipping edge e, would need to mark all edges drawn with thicker line (as shown below) as suspect



Section 6.5.2

References

References I

1. U. W. Kulisch. Complete interval arithmetic and its implementation on the computer.
   In A. Cuyt, W. Kramer, W. Luther, and P. Markstein, editors, *Numerical Validation in Current Hardware Architectures*, pages 7–26.
   Springer-Verlag, Berlin, Germany, 2009.

2. G. Bohlender, U. Kulisch, and R. Lohner. Definition of the arithmetic operations and comparison relations for an interval arithmetic standard, Nov. 2008.

3. IEEE Std. 1788-2015 — IEEE standard for interval arithmetic, 2015.

4. H. Bronnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry.
   Discrete Applied Mathematics, 109:25–47, 2001.

5. C. Dyken and M. S. Floater. Preferred directions for resolving the non-uniqueness of Delaunay triangulations. *Computational Geometry—Theory and Applications*, 34:96–101, 2006.

6. C. L. Lawson. Software for C^1 surface interpolation. In J. R. Rice, editor, *Mathematical Software III*, pages 161–194. Academic Press, New York, NY, USA, 1977.

7. B. Delaunay. Sur la sphere vide. *Bulletin of the Academy of Sciences of the USSR, Classe des Sciences Mathematiques et Naturelle*, 7(6):793–800, 1934.

8. J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363, 1997.

References III

9 S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry.
In *Proc. of Symposium on Computational Geometry*, pages 163–172, 1993.

10 V. Lefevre. Correctly rounded arbitrary-precision floating-point summation.
IEEE Trans. on Computers, 2017.
To appear.

Section 6.6

Cache-Efficient Code

The Memory Latency Problem

- over time, processors have continued to become faster
- speed improvements in memory, however, have not kept pace with processors
- compared to speed of processor, main memory is ***very slow***
- consequently, bottlenecks in algorithms can often be due to memory speed
- very substantial amount of complexity in modern processors devoted to reducing impact of memory latency
- particularly important feature for hiding memory latency is cache
- effective utilization of cache often critical to writing high-performance code

Section 6.6.1

Memory Hierarchy and Caches

Principle of Locality

- **locality of reference**: programs do not access all code or data uniformly
- two basic types of locality:
  1. temporal
  2. spatial
- **temporal locality**: tendency to reuse same information stored in memory within relatively small time interval (e.g., code in loops, top of stack)
- example (where accesses to `i` and `sum` have good temporal locality, due to their repeated use in loop):

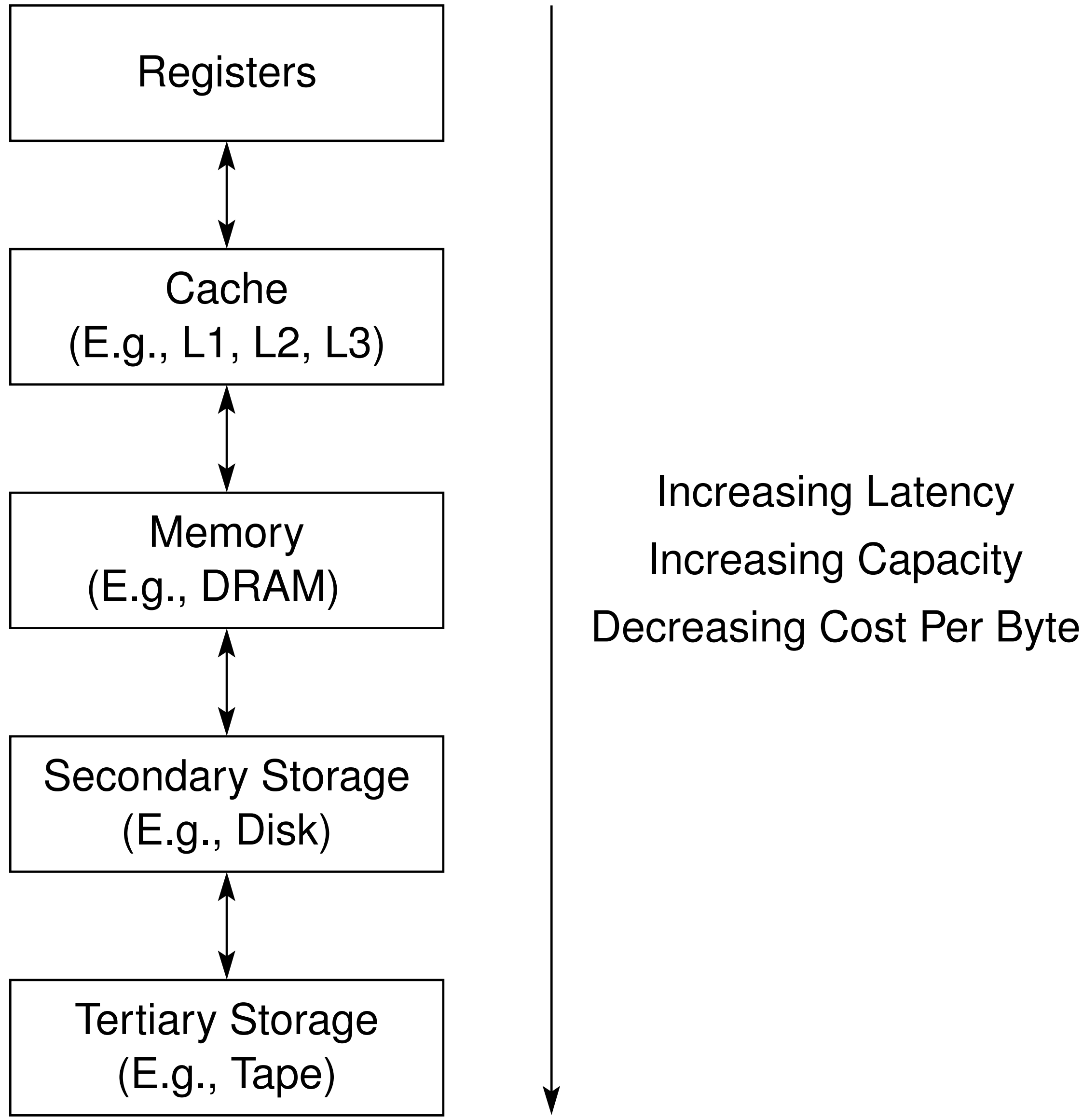
```
int func(int);
int sum = 0;
for (int i = 0; i < 10000; ++i) {sum += func(i);}
```

- **spatial locality**: tendency to use information stored in nearby locations in memory together (e.g., sequential code, neighbouring elements in array)
- example (where accesses to neighbouring elements of `a` have good spatial locality):

```
int a[1024];
// ...
a[42] = a[43] * a[44] + a[45];
```

- to exploit locality, memory hierarchy is employed

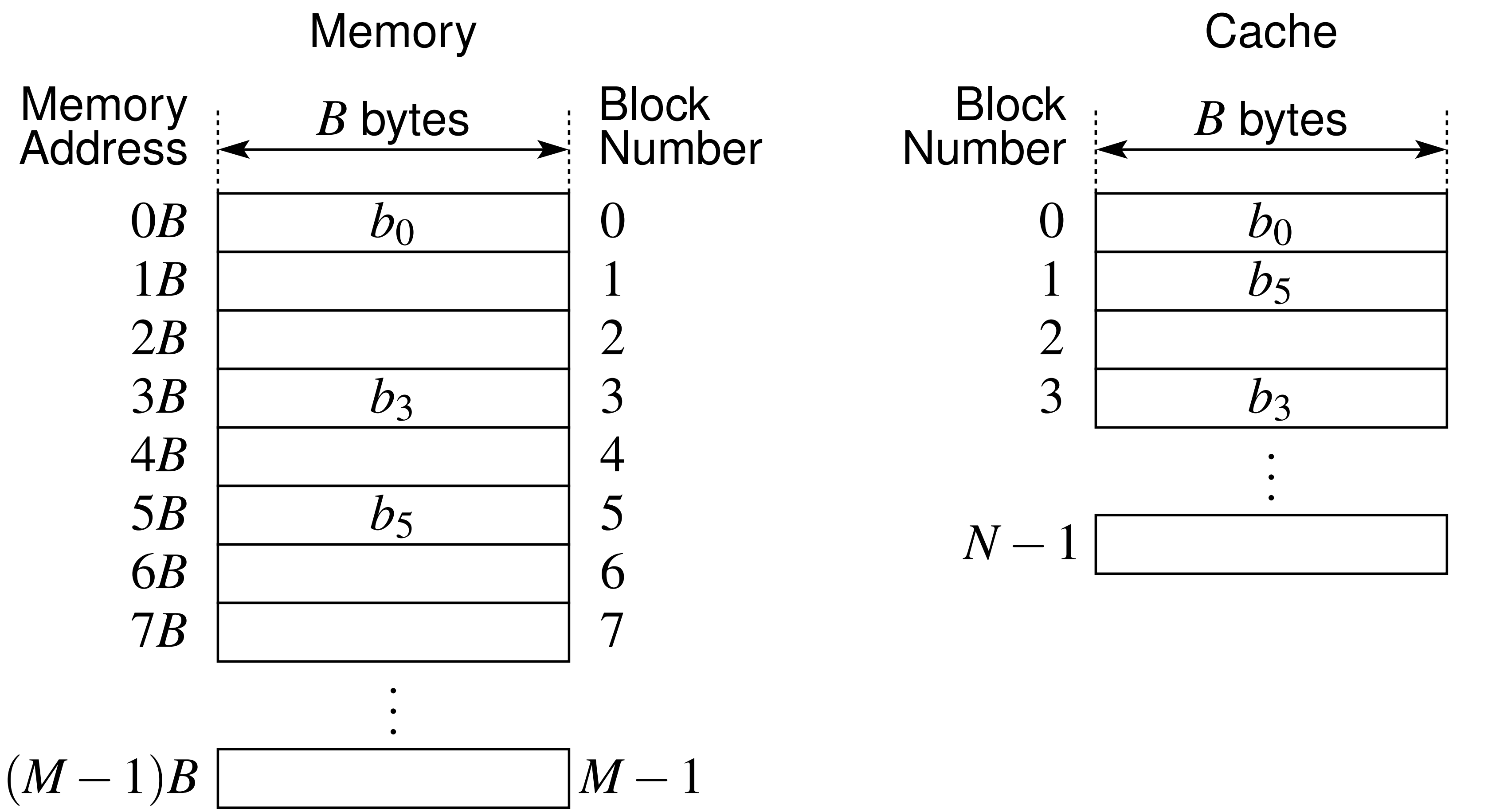
Memory Hierarchy



Caches

- **cache**: fast (but relatively small) memory
- **data cache** (a.k.a. D cache): cache that holds only data
- **instruction cache** (a.k.a., I cache): cache that holds only instructions
- **unified cache**: cache that holds both instructions and data
- **translation lookaside buffer (TLB)**: memory cache that stores recent translations of virtual to physical addresses
- may be several levels to cache hierarchy
- **level-1 (L1) cache** closest to processor, while **last-level (LL) cache** farthest
- when processor needs to read or write location, checks cache
- when data needed is available in cache, **cache hit** said to occur
- when data needed cannot be supplied by cache, **cache miss** said to occur
- cache may be local to single core or shared between multiple cores
- L1 cache usually on core and local to core, while higher-level caches often shared between some or all cores

Memory and Cache



- memory partitioned into blocks of B bytes (where B is typically power of two)
- memory comprised of M blocks for total memory size of BM bytes
- cache can hold N blocks for total cache size of BN bytes
- size of cache much less than size of memory (i.e., $BN \ll BM$)

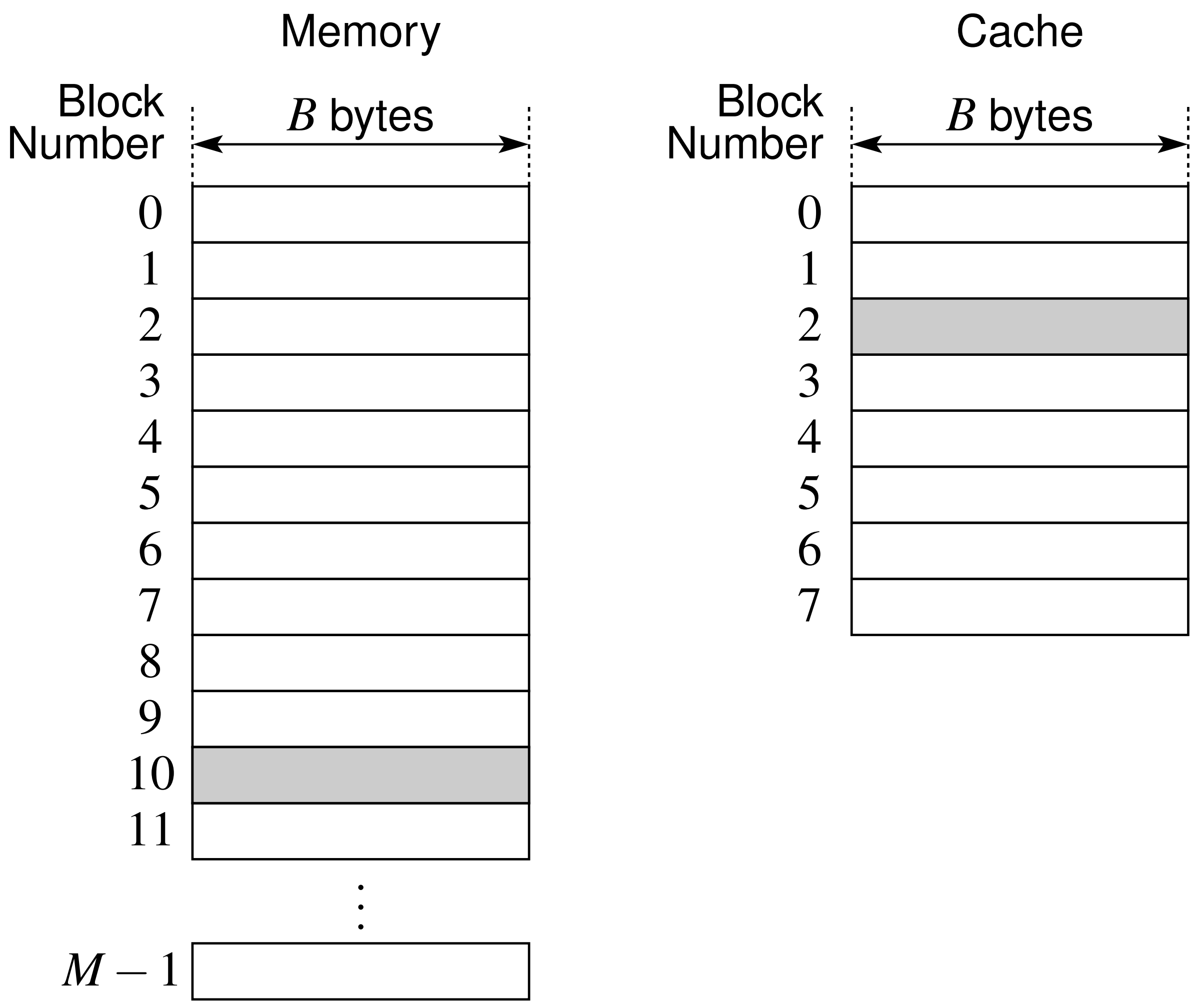
Block Placement

- **block placement policy**: strategy used to determine where block can be placed in cache
- three basic block placement polices:
  1. direct mapped
  2. set associative
  3. fully associative
- **direct mapped**: each block has only one place it can appear in cache
- typically, memory block i mapped to cache block $\mathrm{mod}(i,n)$, where n is number of blocks in cache
- **set associative**: block can be placed in restricted number of places in cache; block first mapped to group of blocks in cache called set, and then block can be placed anywhere within that set
- typically, memory block i can be placed in any cache block in set $\mathrm{mod}(i,S)$, where S is number of sets in cache
- if each set contains k blocks, called **k-way set associative**
- **fully associative**: block can be placed anywhere in cache

Block Placement (Continued)

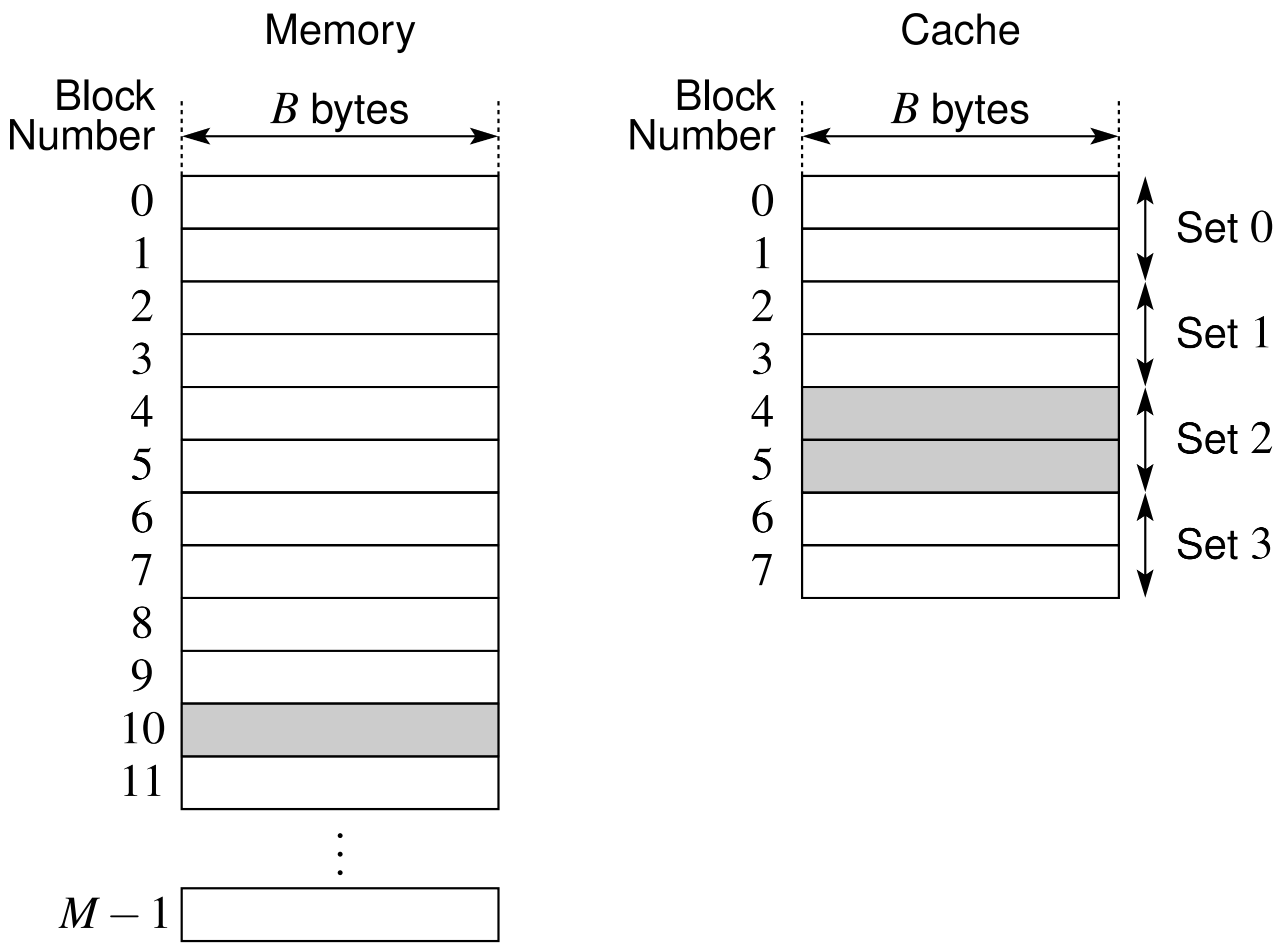
- strictly speaking, set associative includes direct mapped and fully associative as special cases
- direct mapped equivalent to 1-way set associative
- fully associative equivalent to N-way set associative, where N is total number of blocks in cache
- block placement policies typically employ expressions of form $\operatorname{mod}(n, m)$ where $m = 2^k$, since result is simply given by k least significant bits (LSBs) of n
- for example:
  - $\operatorname{mod}(10, 4) = \operatorname{mod}(1010_2, 2^2) = 10_2 = 2$
  - $\operatorname{mod}(42, 16) = \operatorname{mod}(101010_2, 2^4) = 1010_2 = 10$

Direct-Mapped Cache Example



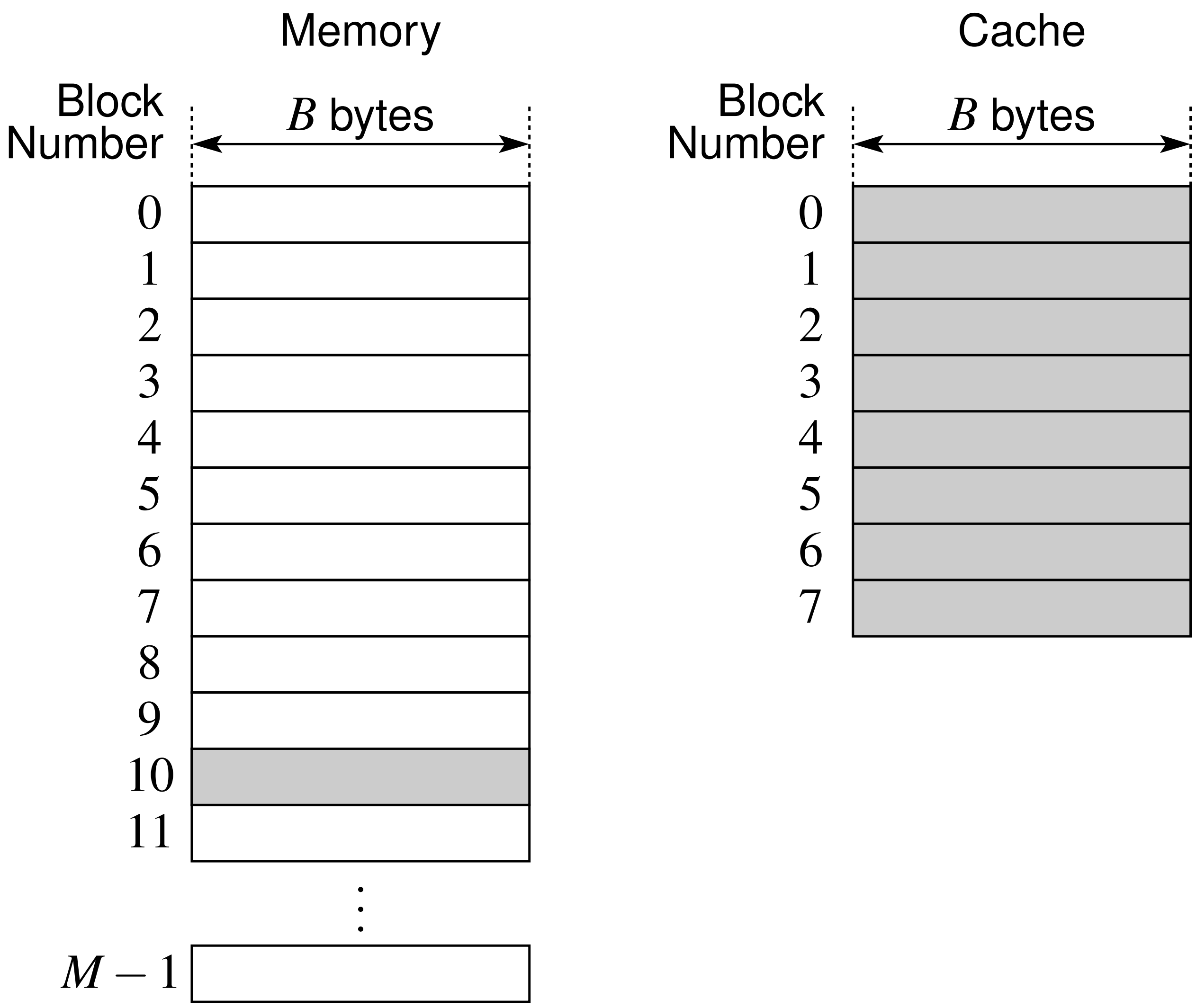
- memory block i can only be placed in cache block $\mathrm{mod}(i,N)$, where N is number of blocks in cache
- for example, if $N = 8$ (as above), memory block 10 can only be placed in cache block $\mathrm{mod}(10,8) = 2$ [recall: $\mathrm{mod}(1010_2, 2^3) = 010_2 = 2$]

K-Way Set-Associative Cache Example



- memory block i can be placed in any of K cache blocks in set $\mathrm{mod}(i, S)$, where S is number of sets
- for example, if $S = 4$ and $K = 2$ (as above), memory block 10 can be placed in any of cache blocks in set $\mathrm{mod}(10, 4) = 2$ [recall: $\mathrm{mod}(1010_2, 2^2) = 10_2 = 2$]

Fully-Associative Cache Example

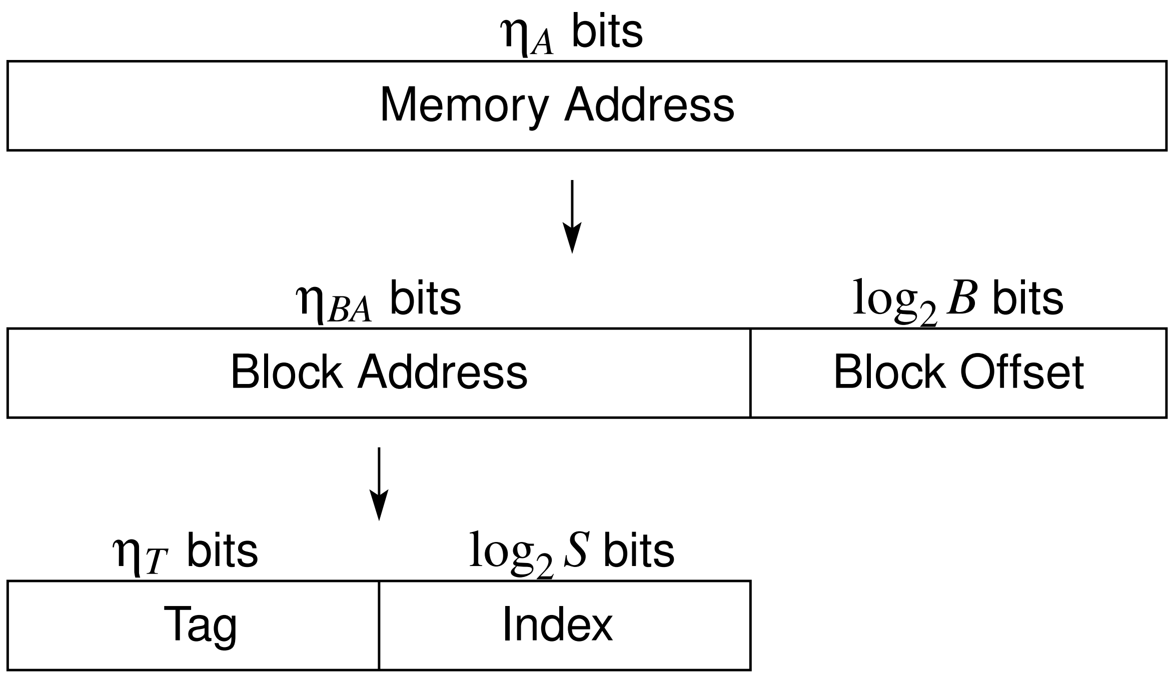


- any memory block can be placed in any cache block
- for example, memory block 10 could be placed in any cache block

Block Identification

- **block identification strategy**: method used to find block if in cache
- when address is referenced, need to determine if associated data in cache, and if it is, find it
- without loss of generality, we can consider case of K-way set associative cache
- memory block i can be mapped to any block in set $s = \operatorname{mod}(i, S)$, where S is number of sets in cache
- each cache entry is associated with one particular set in cache and contains:
  - valid bit to indicate if cache entry in use
  - tag to identify which block is in cache entry (if block is valid)
  - data for block (if block is valid)

Decomposition of Memory Address



- B is cache block size, N is number of blocks in cache, K is cache associativity, and S is number of sets (where $S = N/K$)
- memory address decomposed into block address and block offset
- block address then decomposed into tag and index
- in fully associative case (i.e., $S = 1$), index not present
- index s identifies set in which block i can be placed (i.e., $s = \operatorname{mod}(i, S)$)

Block Identification

Cache Entries for ith Set (for K-Way Set Associative Cache)

Valid	Tag	Data
$v_{i,0}$	$t_{i,0}$	$d_{i,0}$
$v_{i,1}$	$t_{i,1}$	$d_{i,1}$
$\vdots$	$\vdots$	$\vdots$
$v_{i,K-1}$	$t_{i,K-1}$	$d_{i,K-1}$

- need to determine if any entry matches tag and (if not fully associative) index
- first determine set in which block can be placed:
  - if not fully associative, determined by index
  - otherwise, cache only has one set
- then look in this set for matching tag
- if match found, cache hit; otherwise, cache miss

Block Replacement

- **block replacement policy**: strategy used to determine which block should be replaced (i.e., evicted) upon miss when no unused cache entry available
- in case of direct mapped cache, only one choice for block to replace so no freedom in choice of replacement policy
- in case of set-associative or fully-associative cache, have some choice in block to replace
- some commonly-used replacement policies include:
  1. random
  2. least recently used (LRU)
  3. first-in first-out (FIFO)
  4. approximate LRU
- **random**: block to be replaced is randomly chosen (often using pseudorandom number generator)
- **least-recently used (LRU)**: block that has not been used for longest time is replaced
- **first-in first-out (FIFO)**: block that has been in cache longest is replaced

Write Policy

- **write policy**: strategy used to handle writes to memory
- two aspects to write policy:
  1. cache-hit policy (i.e., how to handle cache hit)
  2. cache-miss policy (i.e., how to handle cache miss)
- two basic write-hit policies:
  1. **write through**: information written to both block in cache and block in lower-level memory
  2. **write back**: information written only to block in cache; modified cache block written to main memory only when replaced
- two basic write-miss policies:
  1. **write allocate** (a.k.a. fetch on write): write miss brings block into cache, followed by write-hit action
  2. **no write allocate** (a.k.a. write around): write miss only updates lower-level memory, leaving cache unchanged
- usually, write through used with no write allocate, and write back used with write allocate
- write through always combined with write buffer to avoid always having to wait for lower-level memory

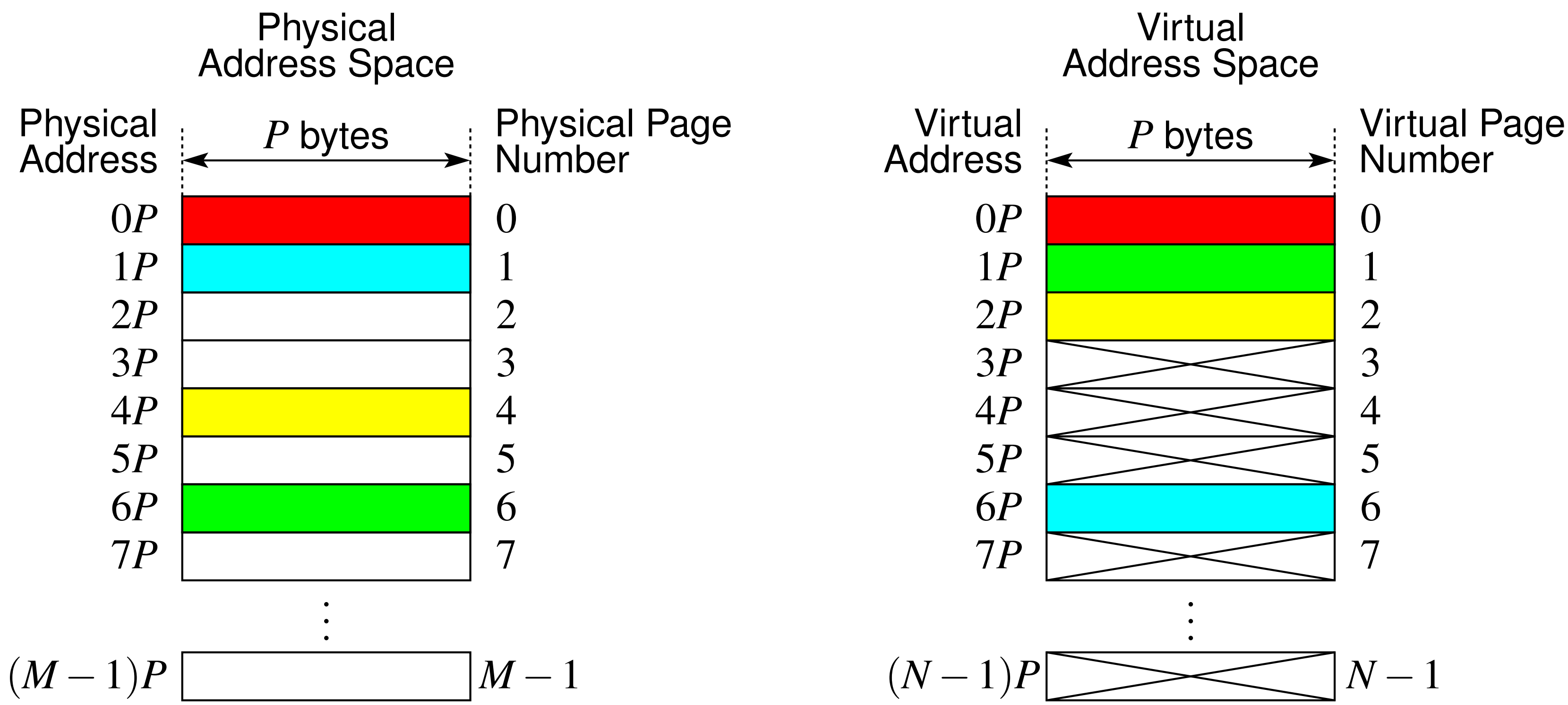
Cache Misses

- **compulsory miss** (a.k.a. cold miss): miss due to address being accessed for first time (impossible to avoid; misses even with infinite sized cache)
- **capacity miss**: miss due to cache not being large enough (i.e., program working set is much larger than cache capacity resulting in block being evicted from cache and later accessed again)
- **conflict miss**: miss due to limited associativity (i.e., miss that would have been avoided with fully associative cache); occurs when too many blocks mapped to same set resulting in memory locations being mapped to same cache entry
- **coherence miss**: miss due to cache flushes to keep multiple caches consistent (i.e., coherent) in multiprocessor system
- **true sharing miss**: coherence miss that is due to multiple threads sharing same data in cache block
- **false sharing miss**: coherence miss that is due to threads accessing different data that happens to reside in same cache block (i.e., cache block is shared between threads but not data within cache block)

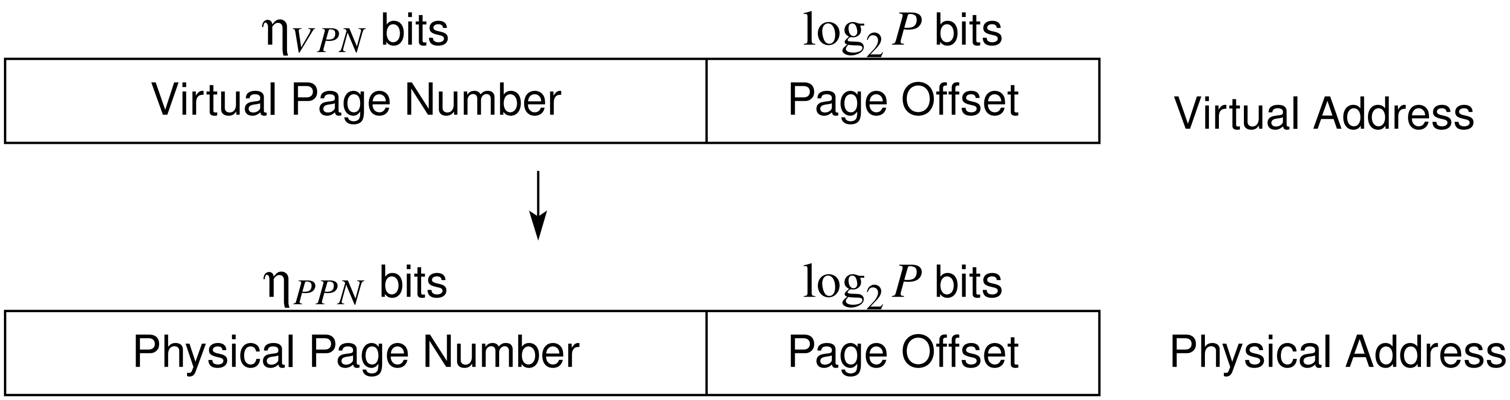
Virtual Memory

- virtual memory is memory management technique that maps addresses called virtual address into physical addresses in computer memory
- allows amount of memory used by system to exceed that which is physically available
- allows processes to share memory
- provides memory protection
- each process has its own virtual address space
- programs access memory using virtual addresses
- memory management unit (MMU) translates virtual addresses to physical addresses

Virtual Address Space



- memory partitioned into pages of size P bytes (where P is typically power of two)
- physical address space comprised of M pages
- virtual address space comprised of N pages
- virtual address space typically at least as large as physical address space (i.e., $PN \geq PM$)
- can arbitrarily map pages in virtual address space to physical pages



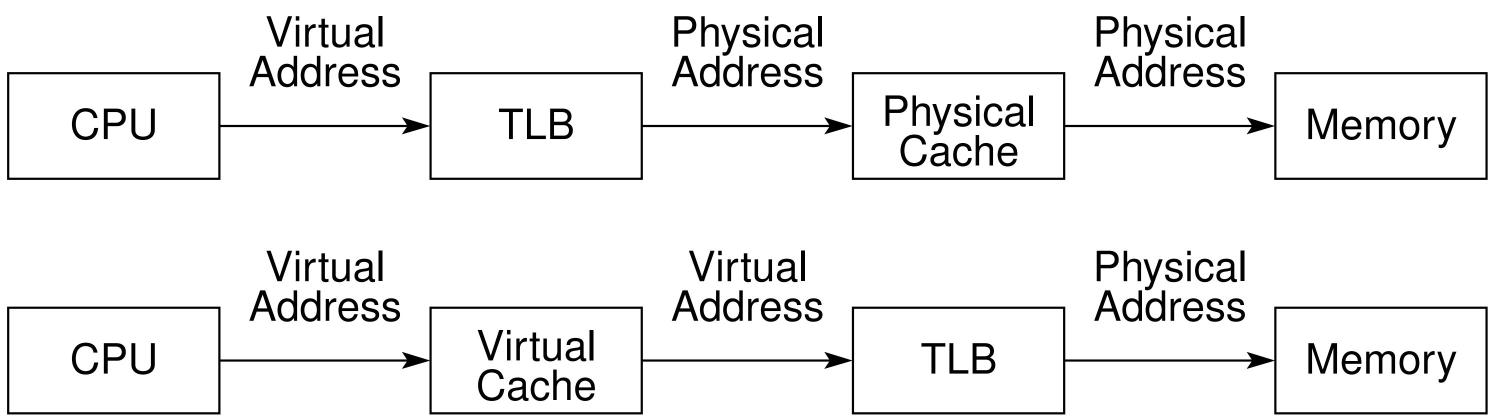
- P is page size
- virtual address and physical address both decomposed into page number and page offset
- address translation only changes page number part of address
- when virtual address translated to physical address, page offset does not change

Translation Lookaside Buffer (TLB)

- address translation is slow process
- to reduce translation time, use cache called translation lookaside buffer (TLB)
- TLB caches information for address-translation mappings

Virtual and Physical Caches

- if virtual memory employed, question arises as to whether memory caches should use virtual or physical addressing
- cache that employs physical addressing called **physical cache** (or physically-addressed cache)
- cache that employs virtual addressing called **virtual cache** (or virtually-addressed cache)
- key difference between use of virtual and physical cache is where address translation takes place:



- in case of accessing physical cache, always require address translation
- in case of accessing virtual cache, only need address translation on cache miss

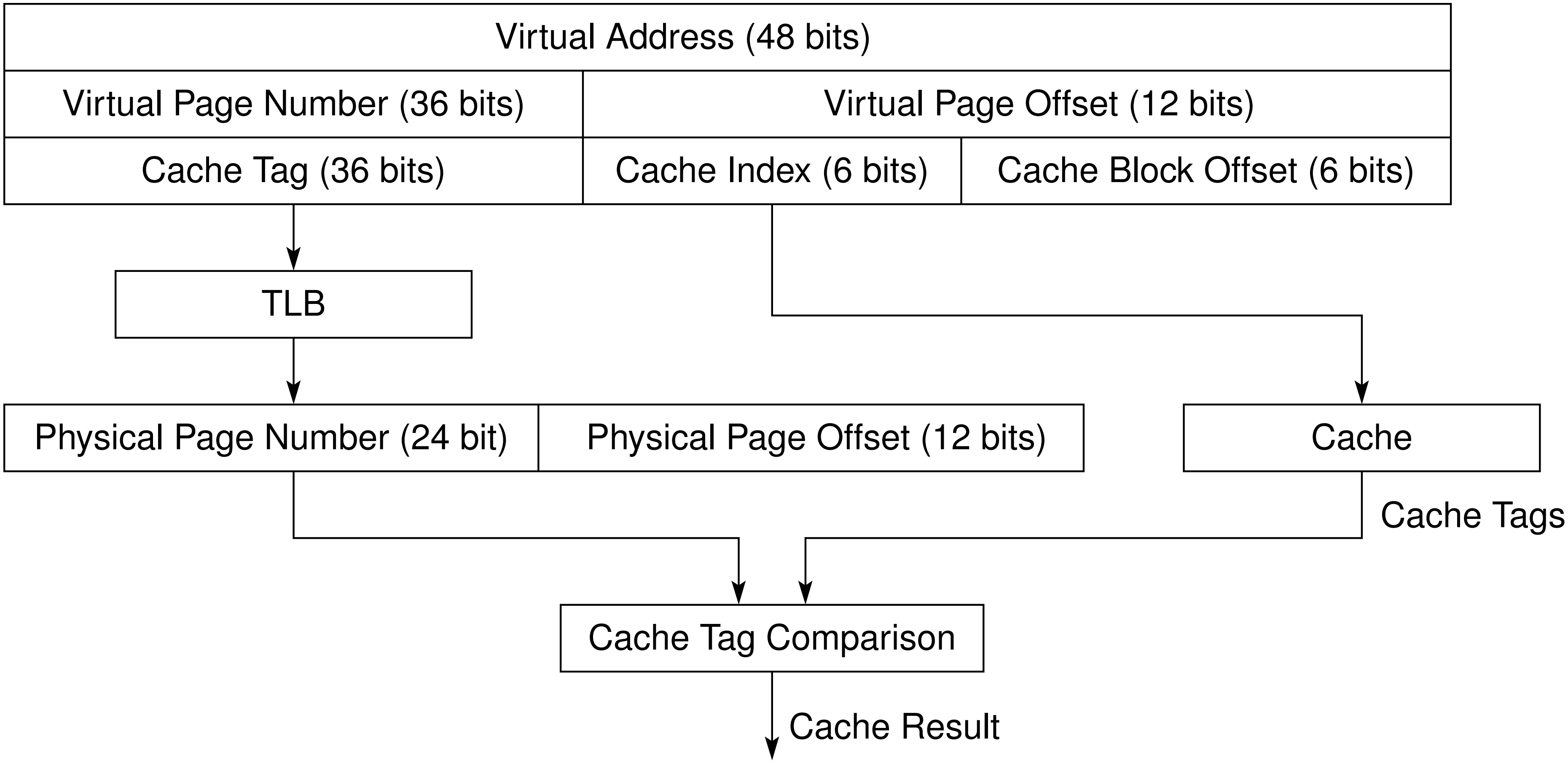
Virtual Versus Physical Caches

- virtual cache has advantage of eliminating address translation time for cache hit
- virtual cache has disadvantage of introducing numerous complications:
  - same virtual address (in different processes) can refer to distinct physical addresses (which is typically resolved by adding process ID to virtual address instead of flushing cache on each context switch)
  - two distinct virtual addresses can refer to same physical address, which is called aliasing (aliasing typically resolved, in case of direct-mapped cache, by restricting address mapping such that aliases map to same cache set)

Virtually-Indexed Physically-Tagged (VIPT) Caches

- cache accesses require tag and index
- in case of virtually-indexed physically-tagged cache, index derived from virtual address and tag derived from physical address
- virtually-indexed physically-tagged cache tries to achieve simplicity of physical cache with speed closer to that of virtual cache
- recall that page offset is unaffected by address translation
- use page offset part of virtual address (which is unaffected by address translation) to determine index for cache (i.e., select set in cache)
- doing this allow us to overlap reading of tags and performing address translation
- this approach faster, but imposes some restrictions on cache parameters
- in particular, number of sets in cache cannot exceed number of cache blocks per page (without additional complications)
- L1 cache often virtually indexed and physically tagged

VIPT Cache Example



- 48-bit virtual address, 36-bit physical address
- 64-byte cache block
- 4 KB page size
- L1 data cache: 32 KB, 8-way set associative, 64 entries per set

Cache Performance

- **hit rate**: fraction of memory access that hit in cache
- **miss rate**: fraction of memory access that miss in cache (1 - hit rate)
- **miss penalty**: time to replace block from lower level in memory hierarchy to cache
- **hit time**: time to access cache memory (including tag comparison)
- **average memory access time (AMAT)**:
  - AMAT = hit time + miss rate · miss penalty

Intel Core i7

- 64-bit processor, x86-64 instruction set
- 36-bit physical addresses and 48-bit virtual addresses
- three-level cache hierarchy; all levels use 64-byte block size; two-level TLB
- L1 cache:
  - I cache: 32 KB 4-way set associative; D cache: 32 KB 8-way set associative; per core, pseudo LRU replacement, virtually indexed and physically tagged
- L2 cache:
  - 256 KB, 8-way set associative, per core, pseudo-LRU replacement, physically indexed (and tagged)
- L3 cache:
  - 2 MB per core, 16-way set associative, pseudo-LRU replacement (with ordered selection algorithm), physically indexed (and tagged)
- first-level TLB:
  - I TLB: 128 entries, 4-way set associative, pseudo-LRU replacement; D TLB: 64 entries, 4-way set associative, pseudo-LRU replacement
- second-level TLB:
  - 512 entries, 4-way set associative, pseudo-LRU replacement, 4 KB page size

ARM Cortex A8

- 32-bit processor, ARM v7 instruction set
- 32-bit physical and virtual addresses
- two-level cache hierarchy; both levels use 64-byte block size
- L1 cache:
  - separate I and D caches; 16 KB or 32 KB 4-way set associative using way prediction and random replacement; virtually indexed and physically tagged
- optional L2 cache:
  - 8-way set associative, 128 KB to 1 MB; physically indexed and physically tagged
- TLB:
  - pair of TLBs (I and D), each of which fully associative with 32 entries and variable page size (4 KB, 16 KB, 64 KB, 1 MB, 16 MB); replacements done by round robin
  - TLB misses handled in hardware, which walks page table structure in memory

Section 6.6.2

Cache-Efficient Algorithms

Cache-Efficient Algorithms

- to effectively exploit cache, need to maximize locality
- various transformations can be applied to code in order to increase locality
- algorithm may be either cache aware or cache oblivious
- **cache aware**: has knowledge of memory hierarchy such as cache parameters (e.g., cache size, cache block size)
- **cache oblivious**: has no knowledge of particulars of memory hierarchy

Code Transformations to Improve Cache Efficiency

- numerous transformations can be applied to code in order to improve spatial and/or temporal locality
- **merging arrays**: improve spatial locality by using array of aggregate type instead of multiple arrays
- **loop interchange**: change nesting of loops to access data in order stored in memory
- **loop fusion**: combine two or more independent loops that have same looping and some variables overlap
- **blocking**: improve temporal locality by accessing blocks of data repeatedly

Array Merging Example

- before array merging:

```
constexpr int num_points = 32'768;
static double x[num_points]; // x coordinates
static double y[num_points]; // y coordinates
static double z[num_points]; // z coordinates
```

- after array merging:

```
constexpr int num_points = 32'768;
struct Point {
    double x; // x coordinate
    double y; // y coordinate
    double z; // z coordinate
};
static Point p[num_points];
```

- x, y, and z coordinates of particular point likely to be accessed together
- use array of aggregate type instead of three separate arrays in order to improve spatial locality and reduce potential conflicts

Loop Interchange Example

- before loop interchange:

```
constexpr int n = 2'048;
static double a[n][n];
// ...
for (int j = 0; j < n; ++j) {
    for (int i = 0; i < n; ++i) {
        a[i][j] *= 2.0;
    }
}
```

- after loop interchange:

```
constexpr int n = 2'048;
static double a[n][n];
// ...
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        a[i][j] *= 2.0;
    }
}
```

- interchange loops so that array elements accessed consecutively instead of with large stride in order to improve locality and reduce potential conflicts

Loop Fusion Example

- before loop fusion:

```
constexpr int n = 2'048;
static float a[n][n], b[n][n], c[n][n], d[n][n];
// ...
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j)
      {a[i][j] = b[i][j] * c[i][j];}
}
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j)
      {d[i][j] = a[i][j] + c[i][j];}
}
```

- after loop fusion:

```
constexpr int n = 2'048;
static float a[n][n], b[n][n], c[n][n], d[n][n];
// ...
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        a[i][j] = b[i][j] * c[i][j];
        d[i][j] = a[i][j] + c[i][j];
    }
}
```

- merge loops in order to improve temporal locality (due to reuse of `a[i][j]` and `c[i][j]` in each innermost loop iteration)

Blocking Example

- for square matrices A and B, compute matrix product AB and add result to matrix C
- before blocking:

```
template <class T, int N>
void naive_multiply(const T (&a)[N][N], const T (&b)[N][N],
  T (&c)[N][N]) {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            double s = 0;
            for (int k = 0; k < N; ++k) {
                s += a[i][k] * b[k][j];
            }
            c[i][j] += s;
        }
    }
}
```

- want to partition computation into blocks of size $B \times B$, where B chosen so that each block fits in cache

Blocking Example (Continued 1)

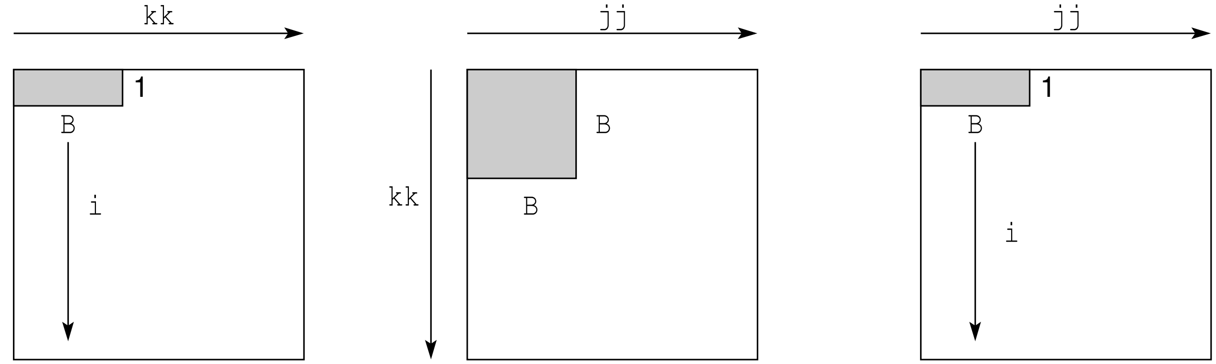
- after blocking (with blocking factor B):

```
template <int B, class T, int N>
void blocked_multiply(const T (&a)[N][N], const T (&b)[N][N],
  T (&c)[N][N]) {
    for (int kk = 0; kk < N; kk += B) {
        for (int jj = 0; jj < N; jj += B) {
            for (int i = 0; i < N; ++i) {
                for (int j = jj; j < std::min(jj + B, N); ++j) {
                    double s = 0;
                    for (int k = kk; k < std::min(kk + B, N); ++k) {
                        s += a[i][k] * b[k][j];
                    }
                    c[i][j] += s;
                }
            }
        }
    }
}
```

- performing computation using blocking significantly improves locality
- potentially many fewer cache misses
- unfortunately, code using blocking much less readable (i.e., more difficult to understand)

Blocking Example (Continued 2)

- graphical interpretation of blocked matrix multiply:



use 1 × B row sliver B times

use B × B block N times in succession

update successive elements of 1 × B row sliver

- key idea is that block of b brought into cache, fully utilized, then discarded
- innermost loop pair (i.e., for j and k) multiplies 1 × B sliver of a by B × B block of b and accumulates result in 1 × B sliver of c
- references to a have: good spatial locality, since elements accessed consecutively in loop for k; and good temporal locality, since each sliver accessed B times in succession in loop for j
- references to b have good temporal locality, since entire block accessed N times in succession in loop for i
- references to c have good spatial locality since each element of sliver written in succession in loop for j

Cache-Aware Versus Cache-Oblivious Algorithms

- cache-aware approaches require knowledge of memory hierarchy and caches (e.g., cache size and cache block size for each level of cache) in order to choose key tuning parameters
- often, such knowledge of memory hierarchy difficult to obtain in reliable manner
- furthermore, effective cache size may differ significantly from true cache size, if multiple threads using cache (which reduces effective cache size)
- if tuning parameters not well chosen, performance can potentially be very poor
- in contrast, cache oblivious approaches:
  - require no knowledge of memory hierarchy and caches
  - require no "magical" tuning parameters
  - effectively autotune
  - handle multilevel caches automatically
  - well accommodate multiprogrammed environments

Section 6.6.3

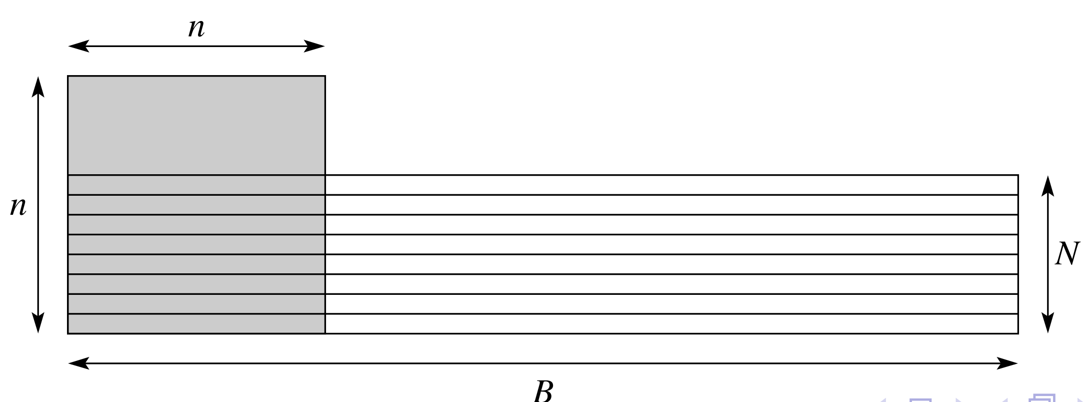
Cache-Oblivious Algorithms

Idealized Cache Model

- idealized cache model employs two-level memory hierarchy (i.e., cache and main memory)
- cache size is M bytes
- cache block size is B bytes
- number of cache blocks is $N = M/B$
- cache said to be **tall** if $N > c'B$ for some sufficiently large constant $c' \geq 1$ (or loosely speaking, number of cache blocks exceeds block size)
- assumptions of ideal cache model:
  - fully associative
  - optimal replacement policy (i.e., evict cache block whose next access will be furthest in future)
  - tall cache
- idealized model only crude approximation to real-world caches
- real-world caches usually not fully associative and never employ optimal replacement policy (which requires noncausal hardware)
- real-world caches, however, usually tend to be tall

Remarks on Tall Caches

- suppose that cache has size M with block size B and $N = M/B$ entries
- cache is said to be **tall** if $N > c'B$ for some sufficiently large constant $c' \geq 1$; otherwise, said to be **short**
- essentially, tall property ensures that N exceeds B by large enough margin that any (possibily non-contiguous) data of size D is guaranteed to fit in cache if $D \leq M$
- that is, if size of some data does not exceed cache size, then that data must fit in cache
- this is not the case for short caches
- for example, $n \times n$ block of elements inside larger array stored in row-major order with $n^2 < M$ will not necessarily fit in cache



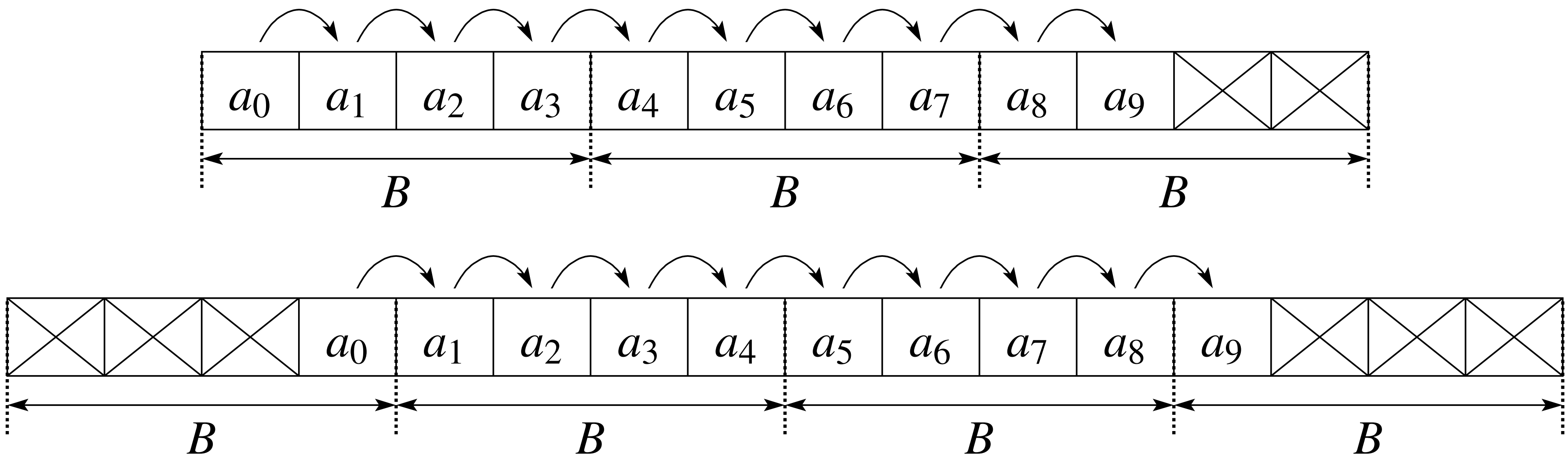
Remarks on Assumption of Optimal-Replacement Policy

- reasonable to question validity of assumption of optimal-replacement policy in idealized cache model
- Sleator and Tarjan (1985) have shown that amortized cost of LRU replacement policy within constant factor of optimal replacement policy
- suppose that algorithm that incurs Q cache misses on ideal cache of size M
- then, on fully-associative cache of size $2M$ that uses LRU replacement policy, at most $2Q$ cache misses
- therefore, to within constant factor, LRU replacement as good as optimal replacement
- implication is that for asymptotic analysis can assume optimal or LRU replacement as convenient
- in this sense, assumption of optimal-replacement policy is quite reasonable

Cache-Oblivious Algorithms

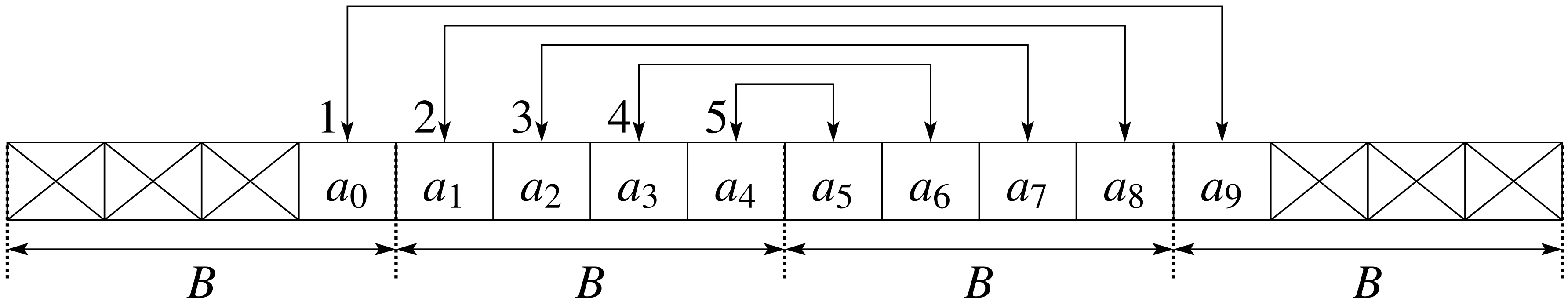
- when analyzing algorithms with respect to idealized cache model typically we are interested in
  - amount of work W (ordinary running time)
  - number of cache misses Q
- cache oblivious algorithms often based on divide and conquer

Scanning



- cache block holds B array elements
- consider scanning N elements of array in order (e.g., to compute sum or minimum/maximum)
- requires $\Theta(N)$ work (assuming work per element is $O(1)$)
- scanning N elements stored contiguously in memory incurs either $\lceil N/B \rceil + 1$ or $\lceil N/B \rceil$ cache misses (i.e., $\Theta(N/B)$ cache misses)
- may require one more than $\lceil N/B \rceil$ cache misses due to arbitrary alignment
- cache oblivious and optimal

Array Reversal



- cache block holds B array elements
- consider reversing elements of N-element array a
- use two parallel scans, one from each end of array, and each step swaps two corresponding elements
- for i in $0, 1, \ldots, \lfloor N/2 \rfloor - 1$, swap $a[i]$ and $a[N-1-i]$
- requires $\Theta(N)$ work
- incurs either $\lceil N/B \rceil + 1$ or $\lceil N/B \rceil$ cache misses, assuming at least two blocks fit in cache (i.e., $\Theta(N/B)$ cache misses)
- cache oblivious and optimal

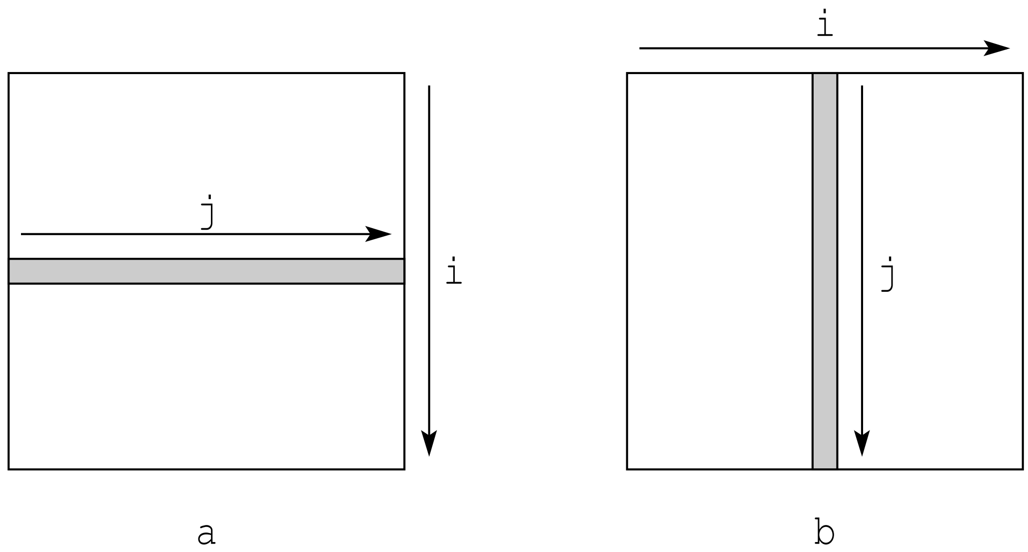
Naive Matrix Transposition

- naive matrix transpose code has following form:

```
template <class T, int M, int N>
void transpose(const T (&a)[M][N], T (&b)[N][M]) {
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < N; ++j) {
            b[j][i] = a[i][j];
        }
    }
}
```

- arrays stored in row-major order
- although data in `a` being accessed sequentially, data in `b` being accessed with large stride
- many unnecessary cache misses on accesses to `b` if number of rows in `b` sufficiently large

Naive Matrix Transposition: Performance



- requires $\Theta(mn)$ work (which is optimal)
- in innermost loop, accesses to `b` use potentially large stride
- strided access to `b` can potentially result in large number of cache misses
- if all blocks for entire column of `b` cannot be kept resident in cache simultaneously, every access to `b` will miss
- in this case, at most $\lceil mn/L \rceil + 1 + mn$ cache misses
- any matrix-transpose algorithm must access all mn elements of `a` and all mn elements of `b`, which incurs at most $2(\lceil mn/L \rceil + 1)$ cache misses
- naive matrix-transposition incurs mn more cache misses than this

Cache-Oblivious Matrix Transposition

- given $m \times n$ matrix A and $n \times m$ matrix B, place A^T into B
- based on divide and conquer strategy
- algorithm halves largest of dimensions m and n, and recurs
- if $m = \max\{m, n\}$ (i.e., number of rows in A largest):

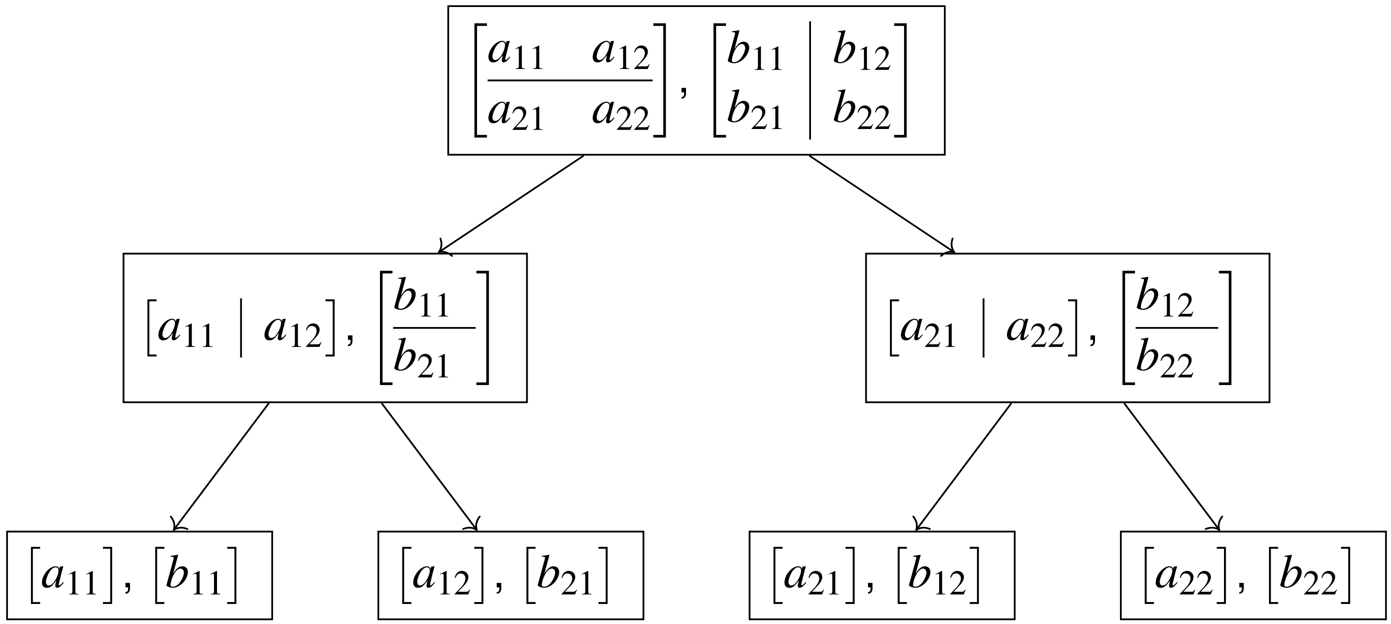
$$A = \begin{bmatrix} A_1 & A_2 \end{bmatrix}, \quad B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

- if $n = \max\{m, n\}$ (i.e., number of columns in A largest):

$$A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}, \quad B = \begin{bmatrix} B_1 & B_2 \end{bmatrix}$$

- conceptually, base case for recursion occurs when $m = n = 1$
- in practice, stop recursion earlier

Cache-Oblivious Matrix Transposition Example



Cache-Oblivious Matrix Transposition: Performance

- let L denote number of array elements per cache block
- for $m \times n$ matrix, cache-oblivious matrix-transposition algorithm:
  - requires $\Theta(mn)$ work
  - incurs $\Theta(1 + mn/L)$ cache misses
- any matrix-transposition algorithm must write to mn distinct elements, which occupy at least $\lceil mn/L \rceil = \Omega(1 + mn/L)$ cache lines
- therefore, cache-oblivious algorithm is asymptotically optimal

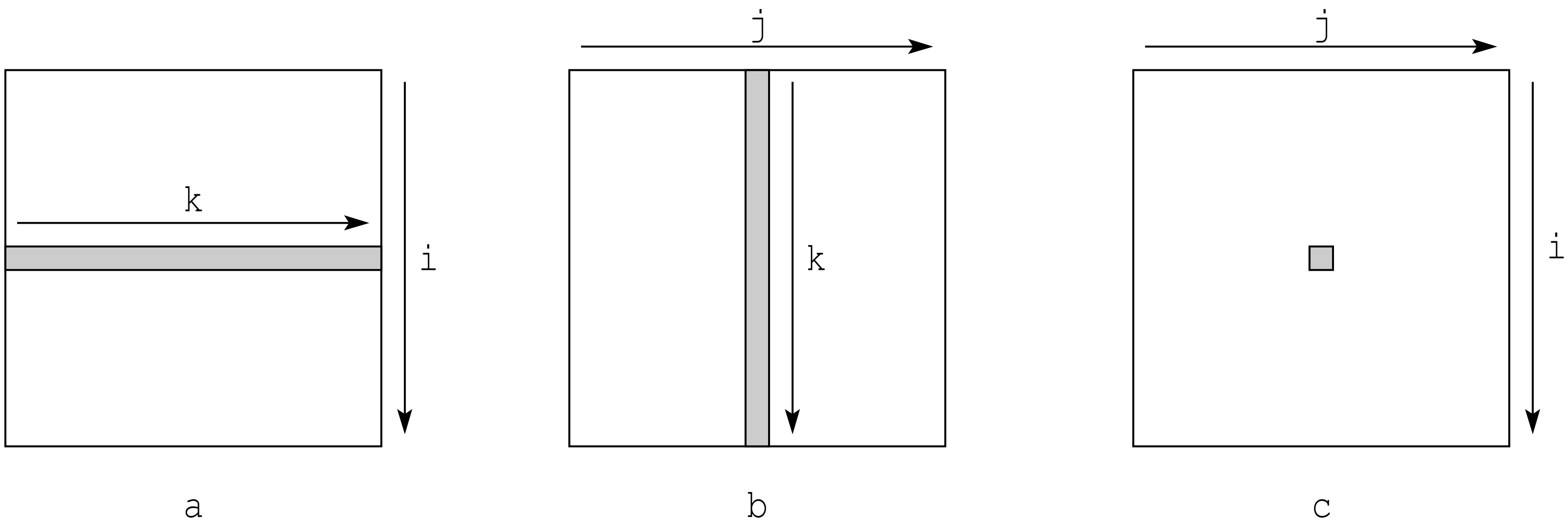
Naive Matrix Multiplication

- naive matrix multiply code has following form:

```
template <class T, int m, int n, int p>
void multiply(const T (&a)[m][n], const T (&b)[n][p],
  T (&c)[m][p]) {
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < p; ++j) {
            T sum = T(0);
            for (int k = 0; k < n; ++k) {
                sum += a[i][k] * b[k][j];
            }
            c[i][j] = sum;
        }
    }
}
```

- arrays stored in row-major order
- in innermost loop, `b` accessed with potentially large stride, which is problematic
- in second innermost loop, row of `a` is accessed `p` times in succession, which is problematic if row does not fit in cache
- many unnecessary cache misses likely to result in case of larger matrices

Naive Matrix Multiplication: Performance



- cache block holds B matrix elements
- innermost loop (in which `k` varies) computes dot product of `i`th row of `a` with `k`th column of `b` to yield (`i`, `j`)th element of `c`
- second innermost loop (over `j`) changes column of `b` to use in dot product with `i`th row of `a` (reusing `i`th row of `a` `p` times)
- requires $\Theta(mnp)$ work, which is $\Theta(n^3)$ in case of square matrices
- assuming that row of `a` and column of `b` do not fit in cache simultaneously, algorithm incurs $\Theta(mnp/B + n^2 p/B + mp/B)$ cache misses, which is $\Theta(n^3/B)$ in case of square matrices

Cache-Oblivious Matrix Multiplication

- given $m \times n$ matrix A and $n \times p$ matrix B, compute AB
- based on divide and conquer strategy
- algorithm halves largest of three dimensions m, n, and p, and recurs
- if $m = \max\{m, n, p\}$ (i.e., number of rows in A largest):

$$AB = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} B = \begin{bmatrix} A_1 B \\ A_2 B \end{bmatrix}$$

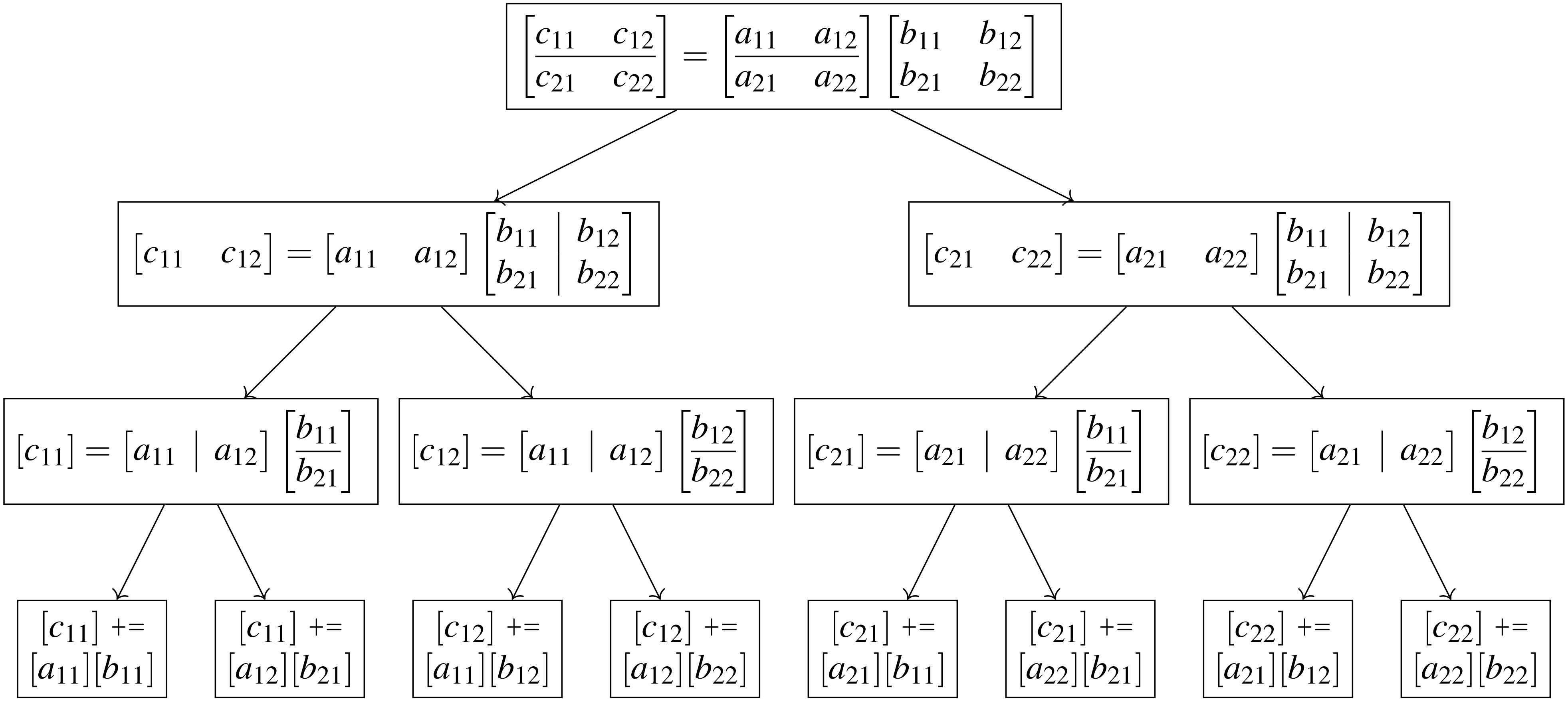
- if $n = \max\{m, n, p\}$ (i.e., number of columns in A and rows in B largest):

$$AB = \begin{bmatrix} A_1 & A_2 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} = A_1 B_1 + A_2 B_2$$

- if $p = \max\{m, n, p\}$ (i.e., number of columns in B largest):

$$AB = A \begin{bmatrix} B_1 & B_2 \end{bmatrix} = \begin{bmatrix} AB_1 & AB_2 \end{bmatrix}$$

- conceptually, base case for recursion occurs when $m = n = p = 1$, in which case two elements multiplied and added into result matrix
- in practice, however, stop recursion at higher level



$\left[\begin{array}{cc} c_{11} & c_{12} \\ \hline c_{21} & c_{22} \end{array}\right] = \left[\begin{array}{cc} a_{11} & a_{12} \\ \hline a_{21} & a_{22} \end{array}\right] \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$
$\begin{bmatrix} c_{11} & c_{12} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \end{bmatrix} \left[\begin{array}{c|c} b_{11} & b_{12} \\ b_{21} & b_{22} \end{array}\right]$
$\begin{bmatrix} c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{21} & a_{22} \end{bmatrix} \left[\begin{array}{c|c} b_{11} & b_{12} \\ b_{21} & b_{22} \end{array}\right]$
$[c_{11}] = \left[\begin{array}{c|c} a_{11} & a_{12} \end{array}\right] \left[\begin{array}{c} b_{11} \\ \hline b_{21} \end{array}\right]$
$[c_{12}] = \left[\begin{array}{c|c} a_{11} & a_{12} \end{array}\right] \left[\begin{array}{c} b_{12} \\ \hline b_{22} \end{array}\right]$
$[c_{21}] = \left[\begin{array}{c|c} a_{21} & a_{22} \end{array}\right] \left[\begin{array}{c} b_{11} \\ \hline b_{21} \end{array}\right]$
$[c_{22}] = \left[\begin{array}{c|c} a_{21} & a_{22} \end{array}\right] \left[\begin{array}{c} b_{12} \\ \hline b_{22} \end{array}\right]$
$[c_{11}] \mathrel{+}= [a_{11}][b_{11}]$
$[c_{11}] \mathrel{+}= [a_{12}][b_{21}]$
$[c_{12}] \mathrel{+}= [a_{11}][b_{12}]$
$[c_{12}] \mathrel{+}= [a_{12}][b_{22}]$
$[c_{21}] \mathrel{+}= [a_{21}][b_{11}]$
$[c_{21}] \mathrel{+}= [a_{22}][b_{21}]$
$[c_{22}] \mathrel{+}= [a_{21}][b_{12}]$
$[c_{22}] \mathrel{+}= [a_{22}][b_{22}]$

Cache-Oblivious Matrix Multiplication: Performance

- to mutiply $m \times n$ matrix by $n \times p$ matrix:
  - requires $\Theta(mnp)$ work
  - incurs $\Theta\left(m+n+p+\frac{1}{L}(mn+np+mp)+\frac{1}{BM^{1/2}}mnp\right)$ cache misses
- to multiply two square matrices (i.e., $m = n = p$):
  - requires $\Theta(n^3)$ work
  - incurs $\Theta\left(\frac{1}{BM^{1/2}}n^3\right)$ cache misses
- Hong and Kung (1981) have shown this to be optimal bound for cache misses for matrix multiplication
- therefore, cache-oblivious algorithm is optimal

Strassen's Algorithm for Matrix Multiplication

- given two $n \times n$ matrices A and B where n is power of two, compute $C = AB$
- approach based on divide and conquer
- partition A, B, and C into equally sized block matrices:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, \quad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, \quad C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

- define (using only 7 matrix multiplications instead of 8):

$$\begin{aligned} M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}), & M_2 &= (A_{2,1} + A_{2,2})B_{1,1}, \\ M_3 &= A_{1,1}(B_{1,2} - B_{2,2}), & M_4 &= A_{2,2}(B_{2,1} - B_{1,1}), \\ M_5 &= (A_{1,1} + A_{1,2})B_{2,2}, & M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}), \\ M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \end{aligned}$$

- can compute C as follows:

$$\begin{aligned} C_{1,1} &= M_1 + M_4 - M_5 + M_7, & C_{1,2} &= M_3 + M_5, \\ C_{2,1} &= M_2 + M_4, & C_{2,2} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

- Strassen's matrix multiplication algorithm optimal in cache-oblivious sense

Discrete Fourier Transform (DFT)

- discrete Fourier transform (DFT) of vector x of n complex numbers is vector y (of n complex numbers) given by

$$y(i) = \sum_{j=0}^{n-1} x(j)\omega_n^{-ij} \quad \text{where } \omega_n = e^{2\pi\sqrt{-1}/n}$$

- for any factorization $n = n_1 n_2$ of n, we have

$$y(i_1 + i_2 n_1) = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} x(j_1 n_2 + j_2] \omega_{n_1}^{-i_1 j_1} \right) \omega_n^{-i_1 j_2} \right] \omega_{n_2}^{-i_2 j_2}$$

- in preceding equation, inner and outer summations are DFTs
- operationally, computation specified in above equation can be performed by:
  1. computing n_2 DFTs of size n_1 (i.e., inner summmation)
  2. multiplying result by factors $w_n^{-i_1 j_2}$ (called twiddle factors)
  3. computing n_1 DFTs of size n_2 (i.e., outer summmation)

Cache-Oblivious Fast Fourier Transform (FFT)

- “six-step” variant of Cooley-Tukey FFT algorithm
- want to compute (one-dimensional) FFT of n element array x, where n is composite and preferably power of two
- FFT is computed in place (i.e., output in x)
- algorithm consists of following steps (in order):
  1. factor n as $n = n_1 n_2$, where n_1 is as close to $\sqrt{n}$ as possible
  2. treat input vector x as row-major $n_1 \times n_2$ matrix A, and use cache-oblivious transpose algorithm to transpose A to auxiliary array B and then copy B back to A
  3. inner summation corresponds to DFT of n_2 rows of transposed matrix; compute n_2 DFTs of size n_1 recursively
  4. multiply A by twiddle factors
  5. transpose A in place so that inputs to next stage placed in contiguous locations
  6. compute n_1 DFTs of rows of matrix recursively
  7. transpose A in place to yield output array x with elements in correct order

Cache-Oblivious FFT: Performance

- can be proven by induction that algorithm requires $O(n \log_2 n)$ work
- cache block holds L elements of array
- Z cache size in units of array element size
- can be shown that algorithm incurs $O(1 + (n/L)(1 + \log_Z n))$ cache misses
- preceding cache miss result asymptotically optimal for Cooley-Tukey algorithm, matching lower bound by Hong and Kung when n is exact power of two

Section 6.6.4

References

References I

1. H. Prokop. Cache-oblivious algorithms.
   Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, June 1999.
2. E. D. Demaine. Cache-oblivious algorithms and data structures.
   In *Lecture Notes from the EEF Summer School on Massive Data Sets*, BRICS, University of Aarhus, Denmark, June 2002.
3. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran.
   Cache-oblivious algorithms.
   ACM Transactions on Algorithms, 8(1):4:1–4:22, Jan. 2012.
4. J. Hennessy and D. A. Patterson. *Computer Architecture — A Quantitative Approach*.
   Morgan-Kaufmann, San Francisco, CA, USA, 4th edition edition, 2007.

5. D. H. Bailey. FFTs in external or hierarchical memory.
Journal of Supercomputing, 4:23–35, 1990.

6. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, II: Hierarchical multilevel memories.
Algorithmica, 12:148–169, 1994.

7. P. Kumar. Cache-oblivious algorithms.
In *Algorithms for Memory Hierarchies*, pages 192–212. Springer Verlag, 2003.

8. V. Strassen. Gaussian elimination is not optimal.
Numerische Mathematik, 13(4):354–356, Aug. 1969.

9. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules.
Communications of the ACM, 28(2):202–208, Feb. 1985.

[10] J.-W. Hong and H. T. Kung. I/O complexity: the red-blue pebbling game. In *Proc. of ACM Symposium on Theory of Computing*, pages 326–333, Milwaukee, WI, USA, 1981.

[11] B. Jacob and T. Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4):60–75, July 1998.

[12] S. Chatterjee and S. Sen. Cache-efficient matrix transposition. In *Proc. of IEEE International Symposium on High-Performance Computer Architecture*, Toulouse, France, 2000.

[13] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

Section 6.7

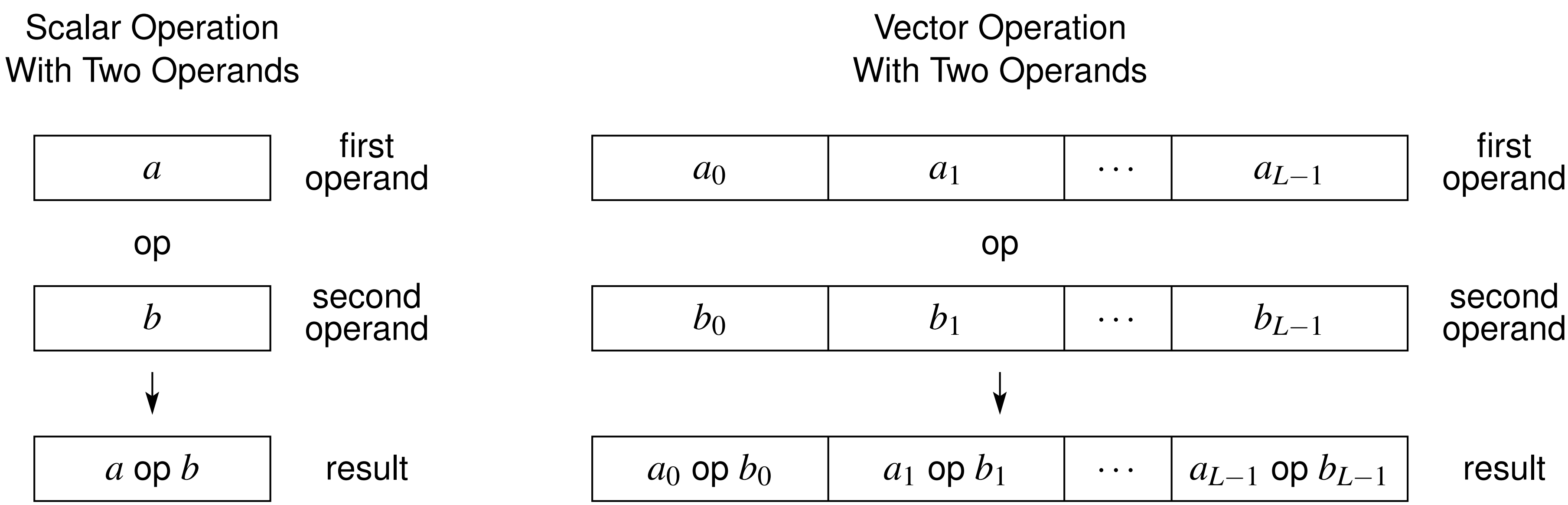
Vectorization

Section 6.7.1

Vector Processing

Vector Processing

- vector processor has instruction set that can operate on one-dimensional arrays of data called vectors
- vector processing has its roots in early supercomputers
- approach has been refined significantly over the years
- attempts to exploit data-level parallelism
- most modern processors provide some level of vector processing functionality



- each operand of scalar instruction is single value
- each operand of vector instruction is set of L values known as **vector**
- L called **vector length**
- same operation applied to each of L elements of vector
- operation might, for example, be: load/store, arithmetic operation, logical operation, comparison, conversion operation, or shuffle operation

Vector-Memory and Vector-Register Architectures

- two basic approaches to vector processing:
  1. vector-memory architecture
  2. vector-register architecture
- vector-memory architecture:
  - for all vector operations, operands fetched from main memory and results written back to main memory
  - includes early vector machines through mid 1980s
  - no longer used much (if at all) in modern processors
  - large startup time for vector operations
- vector-register architecture:
  - for all vector operations except loads and stores, operands read from and written to vector registers
  - used by most modern processors that support vector operations

Vector-Register Architectures

- vector register is collection of N elements of same type, where each element is M bits in size
- N called vector length
- vector register size NM typically 128 to 512
- advantages of vector processing:
  - potential speedup by factor of N
  - often more energy efficient relative to other approaches for increasing performance (such as wider superscalar or higher clock rate)
  - potentially smaller code size, since single instruction can perform multiple operations

Vector Extensions

- modern high-performance CPU architectures have specialized instructions to exploit parallelism in loops
- commonly referred as single-instruction multiple-data (SIMD) extensions
- operate on multiple elements of wide vector register simultaneously
- reduces runtime trip count of loop by vectorization factor
- requires sophisticated analysis and heuristics in order to make good decisions about vectorization safety and profitability
- widen each operation in loop from scalar type to vector type
- applies same operation in parallel to number of data items packed into large register (e.g., 64, 128, 256, 512 bits)
- particularly useful for algorithms with high degree of data-level parallelism, such as those often found in multimedia systems, graphics, and image/video/audio processing

Intel x86/x86-64 Streaming SIMD Extensions (SSE)

- Streaming SIMD Extensions (SSE) is family of vector extensions to Intel x86/x86-64 instruction set architecture (namely, SSE, SSE2, SSE3, SSSE3, SSE4.1, and SSE 4.2)
- collectively, SSE family added:
  - in case of x86: 8 128-bit vector registers, known as XMM0 to XMM7
  - in case of x86-64: 16 128-bit vector registers, known as XMM0 through XMM15
- each vector register can be used to hold:
  - 16 8-bit bytes
  - 8 16-bit integers
  - 4 32-bit integers
  - 2 64-bit integers
  - 4 32-bit single-precision floating-point numbers
  - 2 64-bit double-precision floating-point numbers

Intel x86/x86-64 Advanced Vector Extensions (AVX)

- Advanced Vector Extensions (AVX) is family of vector extensions to Intel x86/x86-64 instruction set architecture (namely, AVX, AVX2, and AVX-512) that builds upon SSE
- AVX extends 16 vector registers of SSE from 128 to 256 bits
- renames vector registers as YMM0 to YMM7 for x86 and YMM0 to YMM15 for x86-64
- each 256-bit vector register can be used to hold:
  - 32 8-bit bytes
  - 16 16-bit integers
  - 8 32-bit integers
  - 4 64-bit integers
  - 8 32-bit single-precision floating-point numbers
  - 4 64-bit double-precision floating-point numbers
  - 8 32-bit single-precision floating-point numbers
  - 4 64-bit double-precision floating-point numbers
- AVX-512 extends vector registers to 512 bits

- NEON is vector extension to ARM Cortex-A series and Cortex-R52 processors
- 16 128-bit vector registers
- NEON instructions perform same operations in all lanes of vectors
- vector registers can hold:
  - 16 8-bit character
  - 8 16-bit integer
  - 4 32-bit integer
  - 2 64-bit integer
  - 8 16-bit floating-point (only in Armv8.2-A)
  - 4 32-bit floating-point
  - 2 64-bit floating-point (only in Armv8-A/R)

Checking for Processor Vector Support on Linux

- on Linux systems, information on processor can be found in `/proc/cpuinfo`
- level of processor support for vector operations can be determined by checking for various processor flags/features in this file
- on Intel x86/x86-64 systems, look for flags/features:
  - `mmx`, `sse`, `sse2`, `ssse3`, `sse4_1`, `sse4_2`, `avx`, `avx2`
- on ARM systems, look for flags/features:
  - `neon`

Section 6.7.2

Code Vectorization

Vectorization

- consider loop in function:

```
void axpy(float a, float* x, float y, int n) {
    for (int i = 0; i < n; ++i) {
        x[i] = a * x[i] + y;
    }
}
```

- loop vectorization: scalar computations in body of above loop could be grouped to allow use of vector operations
- consider code in basic block:

```
a = b + c * d;
e = f + g * h;
i = j + k * l;
m = n + o * p;
```

- basic-block vectorization: four statements in preceding code follow similar pattern and could be grouped together to allow vector operations to be used

Conceptualizing Loop Vectorization

- can think of loop vectorization in terms of loop unrolling
- consider following loop where, for simplicity, we assume `n` multiple of 4:

```
for (int i = 0; i < n; ++i) {c[i] = a[i] + b[i];}
```

- can partially unroll loop to obtain following, where each iteration of new loop corresponds to 4 iterations of original loop:

```
for (int i = 0; i < n; i += 4) {
    c[i + 0] = a[i + 0] + b[i + 0]; // iteration i
    c[i + 1] = a[i + 1] + b[i + 1]; // iteration i + 1
    c[i + 2] = a[i + 2] + b[i + 2]; // iteration i + 2
    c[i + 3] = a[i + 3] + b[i + 3]; // iteration i + 3
}
```

- code in body of new loop can be mapped to vector operations of length 4 on vector registers v0, v1, and v2:
  1. load a[i] to a[i + 3] into v0
  2. load b[i] to b[i + 3] into v1
  3. add v0 and v1, writing result into v2
  4. store v2 into c[i] to c[i + 3]
- using non-standard C++ syntax, vectorized loop can be expressed as:

```
for (int i = 0; i < n; i += 4)
  {c[i : i + 3] = a[i : i + 3] + b[i : i + 3];}
```

Approaches to Vectorization

- several approaches to vectorization can be taken:
  1. auto-vectorization
     - compiler automatically vectorizes code when deemed both safe and profitable
  2. auto-vectorization with compiler hints
     - annotations added to source code to guide auto-vectorization
  3. explicit directives
     - special directives added to source code to exercise control over vectorization (e.g., OpenMP, Cilk Plus)
  4. computation using vector data types
     - use special vector types provided by compiler
  5. compiler intrinsics
     - use special low-level functions provided by compiler
  6. inline assembly language
     - use SIMD instructions directly by using assembly language
- above approaches listed in order of decreasing ease of use and increasing degree of programmer control

Auto-Vectorization

- easiest way to vectorize code is to have compiler do this automatically
- called auto-vectorization
- most compilers have support for auto-vectorization
- advantages of auto-vectorization:
  - easy to use
  - less error prone (no bugs, unless compiler has bug)
  - sometimes compiler may be able to make better judgement as to whether vectorization would be beneficial
- compiler, however, must be very conservative when vectorizing code
- compiler cannot transform code in way that changes its behavior
- unfortunately, compiler often does not have sufficient knowledge of code behavior to perform vectorization well (or at all)

- GCC supports auto-vectorization
- GCC has two vectorizers:
  1. loop vectorizer
  2. basic-block vectorizer
- both vectorizers enabled by default for optimization level of at least 3 (where optimization level specified with `-O` option)
- GCC fully supports OpenMP 4.5 for C/C++ (but not Fortran) as of GCC 6.1 and fully supports OpenMP 4.0 as of GCC 4.9.1

GCC Compiler Options Related to Vectorization

- `-ftree-vectorize` and `-fno-tree-vectorize`
  - enable and disable all vectorization, respectively
- `-ftree-loop-vectorize` and `-fno-tree-loop-vectorize`
  - enable and disable loop vectorizer, respectively
- `-ftree-slp-vectorize` and `-fno-tree-slp-vectorize`
  - enable and disable basic-block vectorizer, respectively
- `-fopt-info-vec-optimized`
  - enable remarks that identify places in code where vectorization successfully applied
- `-fopt-info-vec-missed`
  - enable remarks that identify places in code where vectorization could not be applied
- `-march=native`
  - use instructions supported by local CPU
  - to see which flags are enabled with -march=native, use:

    ```
    g++ -march=native -Q --help=target
    ```

GCC Compiler Options Related to Vectorization (Continued)

- `-fopenmp`
  - enable OpenMP support (which requires GOMP library)
- `-fopenmp-simd`
  - enable OpenMP SIMD support (which does not require run-time library)
- `-S`
  - produce assembly language output only (instead of object code)
- `-fverbose-asm`
  - enable generation of more verbose assembly language output (e.g., compiler version and command-line options, source-code lines associated with assembly instructions, hints on which high-level expressions correspond to various assembly instruction operands)

Clang Compiler and Vectorization

- Clang supports auto-vectorization
- Clang has two vectorizers:
  1. loop vectorizer
  2. superword-level parallelism (SLP) vectorizer
- loop vectorizer widens instructions in loops to operate on multiple consecutive iterations (i.e., performs loop vectorization)
- SLP vectorizer combines similar independent scalar instructions into vector instructions
- both loop and SLP vectorizers enabled by default for optimization level of at least 1 (where optimization level specified by `-O` option)
- Clang supports all non-offloading features of OpenMP 4.5 as of Clang 3.9

Clang Compiler Options Related to Vectorization

- `-fvectorize` and `-fno-vectorize`
  - enable and disable loop vectorizer, respectively
- `-fslp-vectorize` and `-no-fslp-vectorize`
  - enable and disable SLP vectorizer, respectively
- `-fslp-vectorize-aggressive`
  - enable more aggressive vectorization in SLP vectorizer
- `-Rpass=loop-vectorize`
  - enable remarks that identify loops that were successfully vectorized
- `-Rpass-missed=loop-vectorize`
  - enable remarks that identify loops that failed vectorization and indicate if vectorization specified
- `-Rpass-analysis=loop-vectorize`
  - enable remarks that identify statements that caused vectorization to fail
- `-fopenmp`
  - enable OpenMP support (which requires OMP library)
- `-S`
  - produce assembly language output only (instead of object code)

Assessing Quality of Vectorized Code

- to assess quality of vectorized code generated by compiler, often very helpful to view assembly code generated by compiler
- quick inspection of assembly code can often give clear indication as to how well particular part of code was vectorized
- most compilers provide option to generate assembly source as compilation output (instead of object code)
- to assist in locating assembly source corresponding to particular part of C++/C source code (such as loop) can inject comments into assembly code using **`asm`**
- example:

```
float innerprod(float* a, float* b, int n) {
    float result = 0.0f;
    asm volatile ("# loop start");
    for (int i = 0; i < n; ++i) {result += a[i] * b[i];}
    asm volatile ("# loop end");
    return result;
}
```

Assessing Quality of Vectorized Code (Continued)

```
	.file	"inner_product_1.cpp"
	.text
	.globl _Z9innerprodPfS_i
	.type	_Z9innerprodPfS_i, @function
_Z9innerprodPfS_i:
.LFB0:
	.cfi_startproc
#APP
# 3 "inner_product_1.cpp" 1
	# loop start
# 0 "" 2
#NO_APP
	xorl	%eax, %eax
	vxorps %xmm0, %xmm0, %xmm0
.L3:
	cmpl	%eax, %edx
	jle .L2
	vmovss (%rdi,%rax,4), %xmm1
	vfmadd231ss (%rsi,%rax,4), %xmm1, %xmm0
	incq	%rax
	jmp .L3
.L2:
#APP
# 5 "inner_product_1.cpp" 1
	# loop end
# 0 "" 2
#NO_APP
	ret
	.cfi_endproc
```

Auto-Vectorization with Hints

- in order to allow compiler to perform auto-vectorization more effectively, can provide hints to compiler
- place annotations in code to provide compiler with additional information to guide vectorization
- annotations typically provide information that compiler could not reasonably deduce on its own but is important in making decisions regarding vectorization
- approach is relatively easy to use since compiler still does most of work
- must be careful to provide correct information to compiler, however; otherwise, compiler may generate incorrect code

Obstacles to Vectorization

- numerous obstacles to vectorization:
  - data dependencies
  - control-flow dependencies
  - aliasing
  - noncontiguous memory accesses
  - misaligned data
- by eliminating such obstacles, compiler can perform auto-vectorization more effectively

Data Dependencies and Vectorization

- vectorization changes order of computation compared to sequential case
- changing order of computation may yield different result
- cannot replace sequential loop with vectorized version if this would change result of computation
- need to consider independence of unrolled loop operations, which depends on vectorization factor
- three types of data dependencies:
  1. flow dependency (read after write)
  2. output dependency (write after write)
  3. antidependency (write after read)
- flow and output dependencies are of most concern for vectorization

- **flow dependency** (also called **read-after-write dependency**) is type of data dependency that occurs when variable is written in one iteration of loop and read in subsequent iteration
- **dependency distance** is difference in iteration number in which read and write of variable occur
- example of flow dependency with dependency distance of 1:

```
for (int i = 1; i < n; ++i)
  {a[i] = a[i - 1] + 1;}
```

- if dependency distance less than vectorization factor, vectorized loop cannot be guaranteed to yield same result as sequential version

Flow Dependence Example

- consider vectorization of following loop with vectorization factor of 4:

```
for (int i = 1; i < n; ++i)
  {a[i] = a[i - 1] + b[i];}
```

- loop exhibits flow dependence (i.e., read after write) on `a[i-1]` (dependence distance 1)
- loop in partially unrolled form (assuming number of iterations multiple of 4):

```
for (int i = 1; i < n; i += 4) {
    a[i + 0] = a[i - 1] + b[i + 0];
    a[i + 1] = a[i + 0] + b[i + 1];
    a[i + 2] = a[i + 1] + b[i + 2];
    a[i + 3] = a[i + 2] + b[i + 3];
}
```

- loop in vectorized form (assuming number of iterations multiple of 4):

```
for (int i = 1; i < n; i += 4)
  {a[i : i + 3] = a[i - 1 : i + 2] + b[i : i + 3];}
```

- vectorized loop will not always produce same results as sequential loop (due to flow dependence with dependence distance 1)
- therefore, with vectorization factor of 4, loop not legal to vectorize

Flow Dependence Example: Sequential Loop

- suppose that:

```
constexpr int n = 5;
int a_data[n] = {-1, -2, -3, -4, -5};
int b_data[n] = {0, 1, 2, 3, 4};
int* a = a_data;
int* b = b_data;
```

- sequential loop:

```
for (int i = 1; i < n; ++i) {
    a[i] = a[i - 1] + b[i]
}
```

- computation for loop iteration:

i	a[i - 1]	b[i]	a[i]
1	-1	1	0
2	0	2	2
3	2	3	5
4	5	4	9

- upon loop termination, array pointed to by `a` contains:

```
{-1, 0, 2, 5, 9}
```

Flow Dependence Example: Vectorized Loop

- again, suppose that:

```
constexpr int n = 5;
int a_data[n] = {-1, -2, -3, -4, -5};
int b_data[n] = {0, 1, 2, 3, 4};
int* a = a_data;
int* b = b_data;
```

- vectorized loop:

```
for (int i = 1; i < n; i += 4) {
    a[i : i + 3] = a[i - 1 : i + 2] + b[i : i + 3];
}
```

- computation for loop iteration:

i	a[i - 1 : i + 2]	b[i : i + 3]	a[i : i + 3]
1	{-1, -2, -3, -4}	{1, 2, 3, 4}	{0, 0, 0, 0}

- upon loop termination, array pointed to by `a` contains:

```
{-1, 0, 0, 0, 0}
```

Flow Dependence Example

- consider vectorizing following loop using vectorization factor of 4:

```
for (int i = 5; i < n; ++i)
  {a[i] = a[i - 5] + b[i];}
```

- loop exhibits flow dependence (i.e., read after write) on `a[i-5]` (dependence distance 5)
- loop in partially unrolled form (assuming number of iterations multiple of 4):

```
for (int i = 5; i < n; i += 4) {
    a[i + 0] = a[i - 5] + b[i + 0];
    a[i + 1] = a[i - 4] + b[i + 1];
    a[i + 2] = a[i - 3] + b[i + 2];
    a[i + 3] = a[i - 2] + b[i + 3];
}
```

- loop in vectorized form (assuming number of iterations multiple of 4):

```
for (int i = 5; i < n; i += 4)
  {a[i : i + 3] = a[i - 5 : i - 2] + b[i : i + 3];}
```

- vectorized loop will always yield same result as sequential loop since no flow dependence occurs within single iteration of vectorized loop
- with vectorization factor of 4, loop legal to vectorize

Output Dependencies

- **output dependency** (also called **write-after-write dependency**) is type of data dependency that occurs when same variable is written in more than one iteration
- example of output dependency:

```
for (int i = 0; i < n; ++i)
  {a[i % 2] = b[i] + c[i];}
```

- generally unsafe to perform vectorization of loops with output dependencies

Control-Flow Dependencies and Vectorization

- control-flow dependencies can lead to different operations for elements in vector
- consider loop in following function:

```
void func(float* a, float* b, int n) {
    for (int i = 0; i < n; ++i) {
        a[i] = (a[i] > 1.0) ? a[i] / b[i] : a[i];
    }
}
```

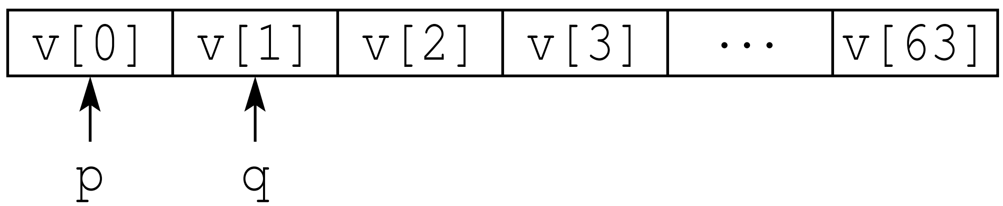
- code has control-flow dependence on `a[i]` (code behavior depends on condition `a[i] > 1.0`)
- good compiler might be able to vectorize above function
- when control-flow dependencies become more complex, however, vectorization extremely difficult or impossible to perform
- therefore, control-flow dependencies are best avoided

Aliasing

- when same memory location can be accessed through different names, **aliasing** said to occur
- example of aliasing:
  - code:

```
float v[64];
float* p = &v[0];
float* q = &v[1];
// p and q can be used to access same memory
// e.g., p[1] and q[0] refer to same object
```

  - memory layout:



- aliasing often limits ability of compiler to perform optimization
- in effect, aliasing can introduce new data dependencies that would not otherwise exist
- failing to take aliasing into account could lead to illegal optimizations (i.e., optimizations that change code behavior)

Aliasing and Optimization: An Example

- consider code:

```
void func(int* a, int* b, int* c) {
    *a = 42;
    *b = 0;
    *c = *a;
}
```

- at first glance, might seem that code can be optimized to yield:

```
void func(int* a, int* b, int* c) {
    *a = 42;
    *b = 0;
    *c = 42;
}
```

- above optimized code is incorrect, since `a` might equal `b`, in which case `*c` should be assigned `0`, not `42`

Aliasing and Vectorization: An Example

- consider code:

```
void add(float* a, float* b, float* c) {
    for (int i = 0; i < 1024; ++i) {
        a[i] = b[i] + c[i];
    }
}
```

- if only this code visible to compiler, simply vectorizing loop in this function is not legal
- `a` could be aliased to `b` or `c` (i.e., storage pointed to by `a`, `b`, and `c` could overlap)
- in this case, sequential and parallel execution of loop would yield different results
- best compiler could do might be to:
  - generate two different versions of code for loop, one without vectorization for aliasing case and one with vectorization for case of no aliasing
  - emit runtime aliasing check that decides which version of code for loop to execute
- this solution less than ideal as it incurs cost of runtime check and results in increased code size

The `__restrict__` Keyword

- sometimes highly beneficial to have means to indicate to compiler that aliasing cannot occur (so that compiler can better optimize code)
- although not part of C++ standard, some compilers support special keyword for this purpose; for example:
  - GCC and Clang support `__restrict__` keyword
  - MSVC supports `__restrict` keyword
- keyword can be applied to pointer or reference
- during execution of block in which restricted pointer/reference `p` is declared, if some object that is accessible through `p` (directly or indirectly) is modified by any means, then all access to that object in that block must occur through `p` (directly or indirectly)
- important only to use `__restrict__` if certain that no aliasing can occur; otherwise, code behavior likely to be incorrect
- example:

```
void func(int* __restrict__ p, int* __restrict__ q) {
    // compiler can safely assume that any data modified though p
    // will only be accessed through p; and similarly for q
    // thus, data pointed to by p and q cannot overlap
    // ... (code modifies data pointed to by p and q)
}
```

Noncontiguous Memory Accesses

- vector load/store operation typically reads/writes contiguous block of memory (that is appropriately aligned)
- noncontiguous data typically needs multiple instructions to be read/written
- example of code with noncontiguous memory accesses:

```
// in loop, array elements accesses with stride 2
for (int i = 0; i < n; i += 2) {
    c[i] = a[i] + b[i];
}
```

- sometimes noncontiguous memory access problem can be addressed by choosing different layout for data in memory (e.g., struct of arrays instead of array of structs)
- other times, problem may be resolvable by restructuring code to perform computations in different order

- for reasons of performance, vector load and store operations often impose restrictions on data alignment
- typically, target address for vector load or store of n-byte register needs to be aligned on n-byte boundary
- for some architectures, such alignment is strict requirement (i.e., code will not work if data misaligned)
- for other architectures, such alignment is not strictly required, but substantial performance penalty may be incurred in case of misaligned data
- for this reason, important to align data appropriately whenever possible
- also, to allow compiler to vectorize in most effective manner possible, important to let compiler know when data is appropriately aligned

- sometimes not possible or practical to avoid misaligned data
- in such cases, can still partially vectorize
- peel first few iterations of loop where data is misaligned and process data using scalar operations
- peel last few iterations (as necessary) where insufficient data to fill vector register and process data using scalar operations
- use vector operations for remainder of iterations
- compared to case of properly aligned data that is multiple of vector size, above approach likely to be slower and have larger code size
- alternatively, could add padding before and/or after data to ensure data with padding is appropriately aligned and multiple of vector length, but this approach often not practical

Controlling Alignment of Data

- for non-heap allocation, can use **alignas** qualifier to control alignment of object
- for heap allocation, can use `std::aligned_alloc` to allocate memory with particular alignment
- `std::free` can be used to free memory allocated by `std::aligned_alloc`
- example:

```
#include <cassert>
#include <cstdlib>
#include <cstdint>

int main() {
    alignas(4096) static char buffer[65536];
    static_assert(alignof(buffer) == 4096);
    float* fp = static_cast<float*>(
      std::aligned_alloc(4096, sizeof(float)));
    if (!fp) {return 1;}
    assert(!(reinterpret_cast<intptr_t>(fp) % 4096));
    std::free(fp);
}
```

Informing Compiler of Data Alignment

- to facilitate more effective vectorization by compiler, important to be able to indicate data alignment in code
- unfortunately, C++ standard does not provide mechanism for doing this
- some compilers (such as GCC and Clang) support intrinsic function called `__builtin_assume_aligned` that can be used to indicate alignment
- `__builtin_assume_aligned` declared as:

```
void* __builtin_assume_aligned(const void *p, size_t align, ...);
```

- this function simply returns its first argument `p` and allows compiler to assume that returned pointer is at least `align` bytes aligned (when invoked with two arguments)
- example:

```
void func(float* a, float* b, int n) {
    // *a and *b can be assumed aligned to 64-byte boundary
    a = static_cast<float*>(__builtin_assume_aligned(a, 64));
    b = static_cast<float*>(__builtin_assume_aligned(b, 64));
    for (int i = 0; i < n; ++i) {/* ... */}
}
```

- in case of compilers that do not support `__builtin_assume_aligned`, another approach would need to be found

Profitability of Vectorization

- vectorization can often provide significant speedup (in some cases linear with vectorization factor), but costs need to be considered
- vector loop bodies can be larger than their scalar forms, as more complex operations may be needed, increasing code size
- vector loop may have increased startup costs to prepare for vectorized execution
- if aliasing is potential problem, require overhead of runtime aliasing check
- vector instructions may take more cycles

Vectorization Example (Version 1)

- source code:

```
#include <cstddef>

template <std::size_t n, class T>
void add(const T (&a)[n], T (&b)[n]) {
    for (int i = 0; i < n; ++i) {
        b[i] += a[i];
    }
}
```

- since a and b may be aliased, compiler must generate code that correcly handles aliased case (as well as non-aliased case)
- often, will generate code that tests for aliasing at run time and uses result to decide between code for aliased case or non aliased case
- since compiler does not know alignment of a and b, must generate code that handles any valid alignment

- source code:

```
#include <cstddef>

template <std::size_t n, class T>
void add(const T (&__restrict__ a)[n],
  T (&__restrict__ b)[n]) {
    for (int i = 0; i < n; ++i) {
        b[i] += a[i];
    }
}
```

- compiler can assume no aliasing (due to use of `__restrict__`)
- since compiler does not know alignment of `a` and `b`, must generate code that handles any valid alignment

Vectorization Example (Version 3)

- source code:

```
#include <cstddef>

template <std::size_t n, std::size_t align, class T>
void add(const T (& __restrict__ a)[n],
  T (& __restrict__ b)[n]) {
    const T* ap = static_cast<const T*>(
      __builtin_assume_aligned(&a, align));
    T* bp = static_cast<T*>(
      __builtin_assume_aligned(&b, align));
    for (int i = 0; i < n; ++i) {
        bp[i] += ap[i];
    }
}
```

- compiler can assume no aliasing (due to use of `__restrict__`) and `align`-byte alignment (due to use of `__builtin_assume_aligned`)
- code generated for vectorized loop in case of `add<65536, 16 * alignof(float), float>`:

```
.L2:
    vmovaps (%rsi,%rax), %ymm0
    vaddps (%rdi,%rax), %ymm0, %ymm0
    vmovaps %ymm0, (%rsi,%rax)
    addq   $32, %rax
    cmpq   $262144, %rax
    jne .L2
```

Vectorization Example

- when using `add` function, must be careful to ensure that assumptions about data alignment are not violated
- source code:

```
#include <cstddef>
#include <iostream>
#include <algorithm>
#include <numeric>
#include "example4_util.hpp"

int main() {
    constexpr std::size_t n = 65536;
    constexpr std::size_t align = 16 * alignof(float);
    alignas(align) static float a[n];
    alignas(align) static float b[n];
    std::iota(&a[0], &a[n], 1);
    std::fill(&b[0], &b[n], -1);
    add<n, align>(a, b);
    for (auto i : b) {std::cout << i << '\n';}
}
```

- if code does not ensure correct alignment of data, code will not work correctly (and probably will result in crash)

Basic Requirements for Vectorizable Loops

- countable: number of loop iterations known at run time upon entry to loop (e.g., implies no conditional termination of loop)
- straight-line code (i.e., no control flow); no switch statements; if statements only allowable when can be implemented as masked assignments
- must be innermost loop if nested
- no function calls, except some basic math functions (such as `std::pow`, `std::sqrt`, and `std::sin`) and some inline functions

OpenMP SIMD Constructs

- OpenMP is industry standard API for parallel computing
- supports C++, C, and Fortran
- OpenMP 4.0 added constructs for expressing SIMD data-level parallelism
- although OpenMP offers large amount of functionality, we only focus on SIMD-related functionality here
- use pragmas to control vectorization
- `simd` pragma allows explicit control of vectorization of for loops
- `declare simd` pragma instructs compiler to generate vectorized version of function (which can be used to vectorize loops containing function calls)

OpenMP `simd` Pragma

- vectorized loop can be achieved with OpenMP `simd` pragma
- syntax:

```
#pragma omp simd [clause...]
/* for statement in canonical form */
```

- `simd` pragma must be immediately followed by for loop in canonical form
- optional clauses may be specified to affect behavior of pragma (i.e., `safelen`, `linear`, `aligned`, `private`, `lastprivate`, `reduction`, and `collapse`)
- amongst other things, canonical form of for loop implies:
  - induction variable has integer, pointer, or random-access iterator type
  - limited test and increment/decrement for induction varaible
  - iteration count known before execution of loop
- can target inner or outer loops
- loop must be suitable for vectorization (e.g., no data-dependence problems)
- example:

```
#pragma omp simd
for (int i = 0; i < n; ++i) {c[i] = a[i] + b[i];}
```

OpenMP `declare simd` Pragma

- can generate vectorized versions of functions with `declare simd` pragma
- syntax:

```
#pragma omp declare simd [clause...]
/* function declaration/definition */
```

- optional clauses may be specified to affect behavior of pragma (i.e., `simdlen`, `linear`, `aligned`, `uniform`, `inbranch`, and `notinbranch`)
- example:

```
#pragma omp declare simd
float foo(float a, float b, float c) {
    return a * b + c;
}
```

OpenMP SIMD-Related Pragma Clauses

- `safelen(`*length*`)`
  - specifies *length* as maximum number of iterations that can be run concurrently in safe manner (i.e., without data-dependence problems)
- `collapse(`*n*`)`
  - specifies how many (nested) loops to associate with loop construct (i.e., how many nested loops to combine)
- `simdlen(`*length*`)`
  - specifies *length* as prefered length of vector registers used
- `aligned(`*argument-list*[:*alignment*]`)`
  - specifies items in *argument-list* as having given alignment (e.g., *alignment*)
- `uniform(`*argument-list*`)`
  - indicates each argument in *argument-list* has constant value between iterations of given loop (i.e., constant value across all SIMD lanes)
- `inbranch`
  - specifies that function will always be called from inside conditional statement of SIMD loop
- `notinbranch`
  - specifies that function will never be called from inside conditional statement of SIMD loop

OpenMP SIMD-Related Pragma Clauses (Continued)

- `linear`(*list*[:*linear-step*])
  - specifies that, for every iteration of original scalar loop, each variable in *list* is incremented by particular step *step* (i.e., variable is incremented by *step* times vector length for vectorized loop)
- `private`(*list*)
  - declares variables in *list* to be private to each iteration
- `lastprivate`(*list*)
  - declares variables in *list* to be private to each iteration, and last value is copied out from last iteration instance
- `reduction`(*operator*:*list*)
  - specifies variables in *list* are reduction variables for operator *operator*

Example: Vectorized Loop

```
#include <cstddef>
#include <iostream>
#include <numeric>

template <std::size_t align, std::size_t n, class T>
[[ gnu::noinline ]]
void multiply(const T (&a)[n], const T (&b)[n], T (&c)[n]) {
    #pragma omp simd aligned(a, b, c : align)
    for (int i = 0; i < n; ++i) {
        c[i] = a[i] * b[i];
    }
}

int main() {
    constexpr std::size_t n = 65536;
    constexpr std::size_t align = 16 * alignof(float);
    alignas(align) static float a[n];
    alignas(align) static float b[n];
    alignas(align) static float c[n];
    std::iota(a, &a[n], 0);
    std::iota(b, &b[n], 0);
    multiply<align>(a, b, c);
    for (auto x : c) {
        std::cout << x << '\n';
    }
}
```

Example: Vectorized Loop and Function

```
#include <cstddef>
#include <iostream>
#include <numeric>

#pragma omp declare simd notinbranch
float func(float a, float b) {
    return a * a + b * b;
}

int main() {
    constexpr std::size_t n = 65536;
    constexpr std::size_t align = 16 * alignof(float);
    alignas(align) static float a[n];
    alignas(align) static float b[n];
    alignas(align) static float c[n];
    std::iota(a, &a[n], 0);
    std::iota(b, &b[n], 0);
    #pragma omp simd aligned(a, b, c : align)
    for (int i = 0; i < n; ++i) {
        c[i] = func(a[i], b[i]);
    }
    for (auto x : c) {
        std::cout << x << '\n';
    }
}
```

Section 6.7.3

References

1. Pablo Halpern, Introduction to Vector Parallelism, CppCon, Bellevue, WA, USA, Sept. 21, 2016. Available online at `https://youtu.be/h6Q-5Q2N5ck`.
2. Georg Zitzlsberger, C++ SIMD parallelism with Intel Cilk Plus and OpenMP 4.0, Meeting C++, Berlin, Germany, Dec. 5–6, 2014. Available online at `https://youtu.be/6oKRL7jz2LY`.

References I

1. OpenMP web site, `http://www.openmp.org`.
2. Clang OpenMP page, `http://openmp.llvm.org`.
3. Cilk Plus web site, `http://www.cilkplus.org`.
4. A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):25–38, Jan. 1997.

Section 6.8

Documentation for Software Development

Documentation for Software Development

- documentation plays essential role in software development process
- many benefits to formalizing in writing various aspects of software at different points in development process
- consider two types of documents:
  1. software requirements specification
  2. software design description
- software requirements specification (SRS): describes what software should do (from external viewpoint)
- software design description (SDD): describes how software works internally

Software Requirements Specification (SRS)

- establishes agreement between consumer and contractors on what software is expected to do as well as what it is not expected to do
- can be thought of as contract between customer and contractor
- functionality: what does software do? (what problem does it solve?)
- external interfaces: how does software interact with external agents, such as humans, hardware, and software (e.g., command-line interface, graphical user interface, application program interface)
- performance: speed, availability, response time, recovery time of various functions
- attributes: considerations regarding reliability, availability, maintainability, portability, security
- design constraints imposed on implementation: implementation language, resource limits, operating environments
- assumptions upon which requirements are based

SRS (Continued)

- distinguish classes of requirements:
  - essential: software will be unacceptable unless requirement met
  - conditional: would enhance software if requirement met, but not unacceptable if requirement not met
  - optional: class of functionality that may or may not be worthwhile
- should not leave details of software requirements to be determined
- only focus on what the software needs to do, not how done (i.e., should not describe any design or implementation details)
- typical use cases
- constraints imposed on software:
  - time constraints
  - memory constraints
- software limitations:
  - restrictions on input data
  - allowable ranges for parameters of methods
  - dependencies on other software (e.g., other programs needed to function)

External Interfaces

- external interfaces: how software interacts with external agents, such as humans, hardware, and software
- command line interface (CLI) (for program)
  - options (e.g., required versus optional, default settings)
  - standard input, output, error
  - exit status
- graphical user interface (GUI) (for program)
  - window layout
  - user interaction (e.g., mouse/keyboard actions)
- application program interface (API) (for library)
  - constants
  - types, classes/methods
  - functions
  - namespaces
- format of all data used by software

Benefits of SRS

- establishes basis for agreement between customer and contractors
- reduces development effort by thoroughly considering all requirements before starting design
- provides basis for estimating costs and schedules
- provides baseline for validation and verification
- facilitates transfer of software product to new users or machines
- serves as basis for enhancement

SRS Example: Sorting Program

- single program that performs sorting
- given records as input, program sorts records and outputs records in sorted order
- record data format (for input and output):
  - records delimited by single newline character
  - each record consists of one or more fields, separated by one or more whitespace characters
- restrictions/constraints:
  - may assume sufficient memory to buffer all records
  - software must work without any modification to source code on any platform with C++ compiler compliant with C++11 standard
- records read from standard input
- sorted records written to standard output
- any error/warning messages written to standard error
- sorts records using nth field in record as key
- can sort in ascending or descending order
- sort key may be numeric or string

SRS Example: Sorting Program (Continued)

- command line interface:

```
sort [-r] [-k $n] [-n]
```

Option	Description
`-k $n`	Sort using nth field in record; if not specified, n defaults to 1.
`-n`	Treat key as real number (instead of string) for sorting purposes; if not specified, key treated as string.
`-r`	Sort in descending (instead of ascending) order; if not specified, defaults to ascending order.

- give examples illustrating expected use cases

Software Design Description (SDD)

- high-level design: overview of entire system, identifying all its components at some level of abstraction (i.e., overall software architecture)
- detailed design (a.k.a. low-level design): full details of system and its components (e.g., types, functions, APIs, pseudocode, etc.)
- describes high-level and detailed design of software
- some context regarding functionality provided by software
- how design is recursively structured into constituent parts and role of those parts
- types and interfaces (e.g., classes and public members)
- data structures used to represent information to be processed
- internal interfaces (and external interfaces not described in SRS)
- interaction amongst entities
- algorithms

SDD (Continued)

- describe overall structure of software
- carefully consider choice of data structures used to represent information being processed, as choice will almost always have performance implications
- specify any data formats used internally by software
- provide pseudocode for key parts of software
- state any potentially limiting assumptions made

Benefits of SDD

- encourages better planning by forcing design ideas to be more carefully considered and organized
- allows greater scrutiny of design
- captures important design decisions, such as rationale for particular design choices
- allows newcomers to development team to become acquainted with software more easily
- provides point of reference to be used throughout project
- promotes reuse of code (since well documented code more likely to be reused)
- facilitates better software testing (since certain types of testing benefit from understanding of software design)

SDD Example: Sorting Program

- `Key` alias for type that represents sort key (alias for `std::string`)
- `Compare` functor class for comparing `Key` objects
- `Dataset` class represents collection of all records
- specify all class interfaces (i.e., public members)
- `Dataset` class provides:
  - constructor that creates dataset by reading all records from input stream
  - function to output all records in sorted order to output stream
- `Dataset` class to use `std::multimap<Key, std::string, Compare>`
- allows n records to be sorted in $O(n \log n)$ time [n insertions, each requiring $O(\log n)$ time]
- handling n records requires $O(n)$ memory
- only uses C++ standard library

Requirements/Design Document for Degree Project

- document is combination of SRS and SDD with some added information about testing strategies
- briefly introduce problem being addressed by software
- describe each program and library to be developed
- identify parts of any external software (e.g., programs or libraries) that will be used
- describe user interface (e.g., CLI, GUI) for each program
- fully specify all data formats used
- describe overall structure of each program and library
- identify all key data structures and algorithms to be used
- provide pseudocode for key parts of the software
- state any potentially limiting assumptions made by software
- indicate how programs and library code will be tested
- offer any other information that may be helpful (since above list is not exhaustive)
- provide sufficient detail for other people to understand how software is to be structured and how it will be implemented and tested

References

1. IEEE Std. 1016-2009 — IEEE standard for information technology — systems design — software design descriptions, July 2009.
2. IEEE Std. 830-1998 — IEEE recommended practice for software requirements specifications, Oct. 1998.

Part 7

Debugging and Testing Tools

Section 7.1

Debuggers

Source-Level Debuggers

- unfortunately, software does not always work as intended due to errors in code (i.e., bugs)
- how does one go about fixing bugs in time-efficient manner?
- source-level debugger is essential tool
- single stepping: step through execution of code, one source-code line at a time
- breakpoints: pause execution at particular points in code
- watchpoints: pause execution when the value of variable is changed
- print values of variables

GNU Debugger (GDB)

- GNU Debugger (GDB) is powerful source-level debugger
- home page: `http://www.gnu.org/software/gdb`
- available on most platforms (e.g., Unix, Microsoft Windows)
- most popular source-level debugger on Unix systems
- allows one to see what is happening inside program as it executes or what a program was doing at the moment it crashed
- has all of the standard functionality of a source-level debugger (e.g., breakpoints, watchpoints, single-stepping)
- `gdb` command
- command-line usage:

  `gdb` [*options*] *executable*

`help`

Print help information.

`quit`

Exit debugger.

`run` [*arglist*]

Start the program (with *arglist* if specified).

`print` *expr*

Display the value of the expression *expr*.

`bt`

Display a stack backtrace.

`list`

Type the source code lines in the vicinity of where the program is currently stopped.

gdb Commands (Continued)

`break` *function*

: Set a breakpoint at the function *function*.

`watch` *expr*

: Set a watchpoint for the expression *expr*.

`c`

: Continue running the program (e.g., after stopping at a breakpoint).

`next`

: Execute the next program line, stepping over any function calls in the line.

`step`

: Execute the next program line, stepping into any function calls in the line.

GNU Data Display Debugger (DDD)

- graphical front-end to command-line debuggers such as GDB
- has some fancy graphical data display functionality
- all `gdb` commands available in text window, but can use graphical interface to enter commands as well
- home page: `http://www.gnu.org/software/ddd`
- `ddd` command

Section 7.2

Code Sanitizers

Code Sanitizers

- **code sanitizer**: tool for automatically performing variety of run-time checks on code
- typically requires compiler to instrument code
- may also need library for run-time support
- several code sanitizers supported by Clang and/or GCC
  - address sanitizer
  - thread sanitizer
  - memory sanitizer
  - undefined-behavior sanitizer
  - leak sanitizer
- sanitizers easy to use
- can easily catch many bugs
- overhead of code sanitizer typically much less than that of other competing approaches for detecting similar types of bugs

Address Sanitizer (ASan)

- Address Sanitizer (ASan) can be used to detect numerous errors related to memory addressing, such as:
  - out of bounds accesses to heap, stack, and globals
  - heap use after free
  - stack use after return
  - stack use after scope
  - double or invalid free
  - memory leaks
  - initialization order problems
- supported by both Clang and GCC
- compiler instruments all loads/stores and inserts redzones around stack and global variables
- run-time library provides malloc replacement (with redzone and quarantine functionality) and bookkeeping for error messages
- typically introduces about 2 times slowdown
- about 1.5 to 3 times memory overhead

Using Address Sanitizer

- need to enable address sanitizer at compile and link time using `-fsanitize=address` option for Clang and GCC
- environment variable `ASAN_OPTIONS` can be set to whitespace-separated list of options to control some sanitizer behavior at run time
- some sanitizer options include:
  - `strip_path_prefix`
  - `verbosity`
  - `detect_leaks`
  - `allocator_may_return_null`
  - `check_initialization_order`
  - `detect_stack_use_after_return`
  - `new_delete_type_mismatch`
  - `exitcode`
- to enable checking for initialization order problems, use `ASAN_OPTIONS="check_initialization_order=1"`

Out-of-Bounds Access to Globals

global_buffer_overflow.cpp

```
#include <iostream>
int a[4] = {1, 2, 3, 4};
int main() {
    for (int i = 0; i <= 4; ++i) {
        std::cout << a[i] << '\n';
    }
}
```

program output (truncated):

```
=================================================================
==3359==ERROR: AddressSanitizer: global-buffer-overflow on address 0
    x0000006020d0 at pc 0x000000400d31 bp 0x7ffeb78d2350 sp 0
    x7ffeb78d2348
READ of size 4 at 0x0000006020d0 thread T0
   #0 0x400d30 in main global_buffer_overflow.cpp:5
   #1 0x7f83da8d4fdf in __libc_start_main (/lib64/libc.so.6+0x1ffdf)
   #2 0x400bf8 (global_buffer_overflow+0x400bf8)
0x0000006020d0 is located 0 bytes to the right of global variable 'a'
    defined in 'global_buffer_overflow.cpp:2:5' (0x6020c0) of size 16
SUMMARY: AddressSanitizer: global-buffer-overflow
    global_buffer_overflow.cpp:5 in main
Shadow bytes around the buggy address:
```

Out-of-Bounds Access to Stack

stack_buffer_overflow.cpp

```
#include <iostream>
int main() {
    int a[4] = {1, 2, 3, 4};
    for (int i = 0; i <= 4; ++i)
      {std::cout << a[i] << '\n';}
}
```

program output (truncated):

```
=================================================================
==3364==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffc3e811cf0
     at pc 0x000000400e53 bp 0x7ffc3e811c70 sp 0x7ffc3e811c68
READ of size 4 at 0x7ffc3e811cf0 thread T0
    #0 0x400e52 in main stack_buffer_overflow.cpp:5
    #1 0x7f10c1c7afdf in __libc_start_main (/lib64/libc.so.6+0x1ffdf)
    #2 0x400c48 (stack_buffer_overflow+0x400c48)
Address 0x7ffc3e811cf0 is located in stack of thread T0 at offset 112 in frame
    #0 0x400d06 in main stack_buffer_overflow.cpp:2
  This frame has 2 object(s):
    [32, 33) '__c'
    [96, 112) 'a' <== Memory access at offset 112 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind
     mechanism or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow stack_buffer_overflow.cpp:5 in
     main
Shadow bytes around the buggy address:
```

Out-of-Bounds Access to Heap

heap_buffer_overflow.cpp

```
#include <iostream>
#include <cstring>
int main() {
    char* p = new char[5];
    std::strcpy(p, "Hello");
    std::cout << p << '\n';
    delete[] p;
}
```

program output (truncated):

```
=================================================================
==3360==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000015
     at pc 0x7f7497932399 bp 0x7ffd8defc240 sp 0x7ffd8defb9f0
WRITE of size 6 at 0x602000000015 thread T0
    #0 0x7f7497932398 in __interceptor_memcpy ../../../../src/libsanitizer/asan/
        asan_interceptors.cc:456
    #1 0x400dd4 in main heap_buffer_overflow.cpp:5
    #2 0x7f7496c7dfdf in __libc_start_main (/lib64/libc.so.6+0x1ffdf)
    #3 0x400ca8 (heap_buffer_overflow+0x400ca8)
0x602000000015 is located 0 bytes to the right of 5-byte region [0x602000000010,0
     x602000000015)
allocated by thread T0 here:
    #0 0x7f7497997170 in operator new[](unsigned long) ../../../../src/
        libsanitizer/asan/asan_new_delete.cc:82
    #1 0x400dbf in main heap_buffer_overflow.cpp:4
SUMMARY: AddressSanitizer: heap-buffer-overflow ../../../../src/libsanitizer/asan
     /asan_interceptors.cc:456 in __interceptor_memcpy
Shadow bytes around the buggy address:
```

Use After Free

use_after_free.cpp

```
int main() {
    int* p = new int[16];
    delete[] p;
    *p = 42;
}
```

program output (truncated):

```
=================================================================
==3366==ERROR: AddressSanitizer: heap-use-after-free on address 0x606000000020 at
      pc 0x000000400836 bp 0x7ffc752b5c20 sp 0x7ffc752b5c18
WRITE of size 4 at 0x606000000020 thread T0
    #0 0x400835 in main use_after_free.cpp:4
    #1 0x7f0b6dab5fdf in __libc_start_main (/lib64/libc.so.6+0x1ffdf)
    #2 0x400738 (use_after_free+0x400738)
0x606000000020 is located 0 bytes inside of 64-byte region [0x606000000020,0
     x606000000060)
freed by thread T0 here:
    #0 0x7f0b6e7cfe70 in operator delete[](void*) ../../../../src/libsanitizer/
        asan/asan_new_delete.cc:128
    #1 0x400801 in main use_after_free.cpp:3
previously allocated by thread T0 here:
    #0 0x7f0b6e7cf170 in operator new[](unsigned long) ../../../../src/
        libsanitizer/asan/asan_new_delete.cc:82
    #1 0x4007f1 in main use_after_free.cpp:2
SUMMARY: AddressSanitizer: heap-use-after-free use_after_free.cpp:4 in main
Shadow bytes around the buggy address:
```

Stack Use After Return

stack_use_after_return.cpp

```cpp
int* g = nullptr;
void foobar() {int i = 42; g = &i;}
int main() {
    foobar();
    return *g;
}
```

program output (truncated) (with `ASAN_OPTIONS=detect_stack_use_after_return=1`):

```
=================================================================
==3365==ERROR: AddressSanitizer: stack-use-after-return on address 0x7f74e7500020
      at pc 0x000000400a88 bp 0x7ffdbd534e20 sp 0x7ffdbd534e18
READ of size 4 at 0x7f74e7500020 thread T0
    #0 0x400a87 in main stack_use_after_return.cpp:5
    #1 0x7f74eb1dffdf in __libc_start_main (/lib64/libc.so.6+0x1ffdf)
    #2 0x4008b8 (stack_use_after_return+0x4008b8)
Address 0x7f74e7500020 is located in stack of thread T0 at offset 32 in frame
    #0 0x400976 in foobar() stack_use_after_return.cpp:2
  This frame has 1 object(s):
    [32, 36) 'i' <== Memory access at offset 32 is inside this variable
HINT: this may be a false positive if your program uses some custom stack unwind
     mechanism or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-use-after-return stack_use_after_return.cpp:5 in
      main
Shadow bytes around the buggy address:
```

Stack Use After Scope

use_after_scope.cpp

```
#include <iostream>
int main() {
    int* p;
    {int x = 0; p = &x;}
    std::cout << *p << '\n';
}
```

program output (truncated):

```
=================================================================
==3367==ERROR: AddressSanitizer: stack-use-after-scope on address 0x7ffe1d6a6c40
     at pc 0x000000400b6b bp 0x7ffe1d6a6c10 sp 0x7ffe1d6a6c08
READ of size 4 at 0x7ffe1d6a6c40 thread T0
    #0 0x400b6a in main use_after_scope.cpp:5
    #1 0x7fea2e596fdf in __libc_start_main (/lib64/libc.so.6+0x1ffdf)
    #2 0x400a58 (use_after_scope+0x400a58)
Address 0x7ffe1d6a6c40 is located in stack of thread T0 at offset 32 in frame
    #0 0x400b16 in main use_after_scope.cpp:2
  This frame has 1 object(s):
    [32, 36) 'x' <== Memory access at offset 32 is inside this variable
HINT: this may be a false positive if your program uses some custom stack unwind
     mechanism or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-use-after-scope use_after_scope.cpp:5 in main
Shadow bytes around the buggy address:
```

Double Free

double_free.cpp

```
int main() {
    int* p = new int[16];
    delete[] p;
    delete[] p;
}
```

program output (truncated):

```
=================================================================
==3358==ERROR: AddressSanitizer: attempting double-free on 0x606000000020 in
     thread T0:
    #0 0x7fdc05ed8e70 in operator delete[](void*) ../../../../src/libsanitizer/
        asan/asan_new_delete.cc:128
    #1 0x4007b9 in main double_free.cpp:4
    #2 0x7fdc051befdf in __libc_start_main (/lib64/libc.so.6+0x1ffdf)
    #3 0x4006e8 (double_free+0x4006e8)
0x606000000020 is located 0 bytes inside of 64-byte region [0x606000000020,0
     x606000000060)
freed by thread T0 here:
    #0 0x7fdc05ed8e70 in operator delete[](void*) ../../../../src/libsanitizer/
        asan/asan_new_delete.cc:128
    #1 0x4007b1 in main double_free.cpp:3
previously allocated by thread T0 here:
    #0 0x7fdc05ed8170 in operator new[](unsigned long) ../../../../src/
        libsanitizer/asan/asan_new_delete.cc:82
    #1 0x4007a1 in main double_free.cpp:2
SUMMARY: AddressSanitizer: double-free ../../../../src/libsanitizer/asan/
     asan_new_delete.cc:128 in operator delete[](void*)
```

Memory Leaks

memory_leak.cpp

```
#include <iostream>
#include <cstring>
int main() {
    char* p = new char[1024];
    std::strcpy(p, "Hello, World!\n");
    std::cout << p;
}
```

program output (truncated):

```
Hello, World!
=================================================================
==3362==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 1024 byte(s) in 1 object(s) allocated from:
    #0 0x7f7413651170 in operator new[](unsigned long) ../../../../src/
        libsanitizer/asan/asan_new_delete.cc:82
    #1 0x400b51 in main memory_leak.cpp:4
SUMMARY: AddressSanitizer: 1024 byte(s) leaked in 1 allocation(s).
```

Initialization Order Problems

init_order_main.cpp

```
#include <iostream>
extern int B;
int A = B;
int main()
  {std::cout << A << '\n';}
```

init_order_other.cpp

```
#include <cstdlib>
int B = std::atoi("42");
```

program output (truncated) (with `ASAN_OPTIONS=check_initialization_order=1`):

```
=================================================================
==3361==ERROR: AddressSanitizer: initialization-order-fiasco on address 0
     x000000602440 at pc 0x000000400f14 bp 0x7fff92151540 sp 0x7fff92151538
READ of size 4 at 0x000000602440 thread T0
    #0 0x400f13 in __static_initialization_and_destruction_0 init_order_main.cpp
        :3
    #1 0x400f13 in _GLOBAL__sub_I_A init_order_main.cpp:5
    #2 0x40103c in __libc_csu_init (init_order+0x40103c)
    #3 0x7f933e2e7f6e in __libc_start_main (/lib64/libc.so.6+0x1ff6e)
    #4 0x400c98 (init_order+0x400c98)
0x000000602440 is located 0 bytes inside of global variable 'B' defined in '
     init_order_other.cpp:2:5' (0x602440) of size 4
  registered at:
    #0 0x7f933ef5b7c8 in __asan_register_globals ../../../../src/libsanitizer/
        asan/asan_globals.cc:317
    #1 0x400fe9 in _GLOBAL__sub_I_00099_1_B (init_order+0x400fe9)
SUMMARY: AddressSanitizer: initialization-order-fiasco init_order_main.cpp:3 in
     __static_initialization_and_destruction_0
Shadow bytes around the buggy address:
```

Thread Sanitizer (TSan)

- Thread Sanitizer (TSan) detects data races and deadlocks
- supported by Clang and GCC
- compiler instruments code to intercept all loads/stores
- run-time library provides malloc replacement, intercepts all synchronization, and handles loads/stores
- does not instrument prebuilt libraries and inline assembly
- about 4 to 10 times slower
- about 5 to 8 times more memory
- only supported on 64-bit Linux

Using Thread Sanitizer

- need to enable sanitizer at compile and link time using `-fsanitize=thread` option for Clang and GCC
- environment variable `TSAN_OPTIONS` can be set to whitespace-separated list of options to control some sanitizer behavior at run time
- some sanitizer options include:
  - `strip_path_prefix`
  - `verbosity`
  - `report_bugs`
  - `history_size`
  - `suppressions`
  - `exitcode`
- for example, to set per-thread history size value to 7, use `TSAN_OPTIONS="history_size=7"`
- at least some versions of TSan do not detect potential deadlock if it actually happens (although arguably if deadlock happens, probably it will be noticed)

Data Race

data_race.cpp

```
#include <thread>
int x = 0;
int main() {
    std::thread t([&]{x = 42;});
    x = 43;
    t.join();
}
```

program output (truncated):

```
==================
WARNING: ThreadSanitizer: data race (pid=10305)
  Write of size 4 at 0x000001553848 by thread T1:
    #0 main::$_0::operator()() const data_race.cpp:4:22 (data_race+0x4bfc81)
    #1 void std::_Bind_simple<main::$_0 ()>::_M_invoke<>(std::_Index_tuple<>) /usr/lib/gcc/x86_64-redhat-
        linux/4.9.2/../../../../include/c++/4.9.2/functional:1699:18 (data_race+0x4bfbf8)
    #2 std::_Bind_simple<main::$_0 ()>::operator()() /usr/lib/gcc/x86_64-redhat-linux/4.9.2/../../../../
        include/c++/4.9.2/functional:1688:16 (data_race+0x4bfb98)
    #3 std::thread::_Impl<std::_Bind_simple<main::$_0 ()> >::_M_run() /usr/lib/gcc/x86_64-redhat-linux
        /4.9.2/../../../../include/c++/4.9.2/thread:115:13 (data_race+0x4bf94c)
    #4 execute_native_thread_routine_compat /gcc-7.1.0/build/x86_64-pc-linux-gnu/libstdc++-v3/src/c
        ++11/../../../../../src/libstdc++-v3/src/c++11/thread.cc:110 (libstdc++.so.6+0xba46f)
  Previous write of size 4 at 0x000001553848 by main thread:
    #0 main data_race.cpp:5:4 (data_race+0x4be2ce)
  Location is global 'x' of size 4 at 0x000001553848 (data_race+0x000001553848)
  Thread T1 (tid=10310, running) created by main thread at:
    #0 pthread_create /llvm-clang-4.0.0/src/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:897 (
        data_race+0x44f89b)
    #1 __gthread_create /gcc-7.1.0/build/x86_64-pc-linux-gnu/libstdc++-v3/include/x86_64-pc-linux-gnu/bits/
        gthr-default.h:662 (libstdc++.so.6+0xba5b2)
    #2 std::thread::_M_start_thread(std::shared_ptr<std::thread::_Impl_base>, void (*)()) /gcc-7.1.0/build/
        x86_64-pc-linux-gnu/libstdc++-v3/src/c++11/../../../../../src/libstdc++-v3/src/c++11/thread.cc:191
        (libstdc++.so.6+0xba5b2)
    #3 main data_race.cpp:4:14 (data_race+0x4be2bd)
SUMMARY: ThreadSanitizer: data race data_race.cpp:4:22 in main::$_0::operator()() const
```

deadlock.cpp

```
#include <iostream>
#include <thread>
#include <mutex>
std::mutex m0;
std::mutex m1;
void func1(int n) {
    for (auto i = n; i > 0; --i) {
        std::lock_guard<std::mutex> l0(m0);
        std::lock_guard<std::mutex> l1(m1);
        std::cout << "a\n";
    }
}
void func2(int n) {
    for (auto i = n; i > 0; --i) {
        std::lock_guard<std::mutex> l1(m1);
        std::lock_guard<std::mutex> l0(m0);
        std::cout << "b\n";
    }
}
int main() {
    std::thread t1([]{func1(1);});
    std::thread t2([]{func2(1);});
    t1.join(); t2.join();
}
```

Deadlock (Continued)

program output (truncated):

```
==================
WARNING: ThreadSanitizer: lock-order-inversion (potential deadlock) (pid=3207)
  Cycle in lock order graph: M9 (0x0000006022e0) => M10 (0x0000006022a0) => M9
  Mutex M10 acquired here while holding mutex M9 in thread T1:
    #0 pthread_mutex_lock ../../../../src/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc
           :3608 (libtsan.so.0+0x000000038e0f)
    #1 __gthread_mutex_lock /usr/local/sde-2.15.0/packages/gcc-7.1.0/include/c++/7.1.0/x86_64-pc-linux-gnu/
           bits/gthr-default.h:748 (deadlock+0x000000401511)
    #2 std::mutex::lock() /usr/local/sde-2.15.0/packages/gcc-7.1.0/include/c++/7.1.0/bits/std_mutex.h:103 (
           deadlock+0x000000401511)
    #3 std::lock_guard<std::mutex>::lock_guard(std::mutex&) /usr/local/sde-2.15.0/packages/gcc-7.1.0/include/
           c++/7.1.0/bits/std_mutex.h:162 (deadlock+0x000000401511)
    #4 func1(int) deadlock.cpp:9 (deadlock+0x000000401511)
[text deleted]
    Hint: use TSAN_OPTIONS=second_deadlock_stack=1 to get more informative warning message
  Mutex M9 acquired here while holding mutex M10 in thread T2:
    #0 pthread_mutex_lock ../../../../src/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc
           :3608 (libtsan.so.0+0x000000038e0f)
    #1 __gthread_mutex_lock /usr/local/sde-2.15.0/packages/gcc-7.1.0/include/c++/7.1.0/x86_64-pc-linux-gnu/
           bits/gthr-default.h:748 (deadlock+0x000000401451)
    #2 std::mutex::lock() /usr/local/sde-2.15.0/packages/gcc-7.1.0/include/c++/7.1.0/bits/std_mutex.h:103 (
           deadlock+0x000000401451)
    #3 std::lock_guard<std::mutex>::lock_guard(std::mutex&) /usr/local/sde-2.15.0/packages/gcc-7.1.0/include/
           c++/7.1.0/bits/std_mutex.h:162 (deadlock+0x000000401451)
    #4 func2(int) deadlock.cpp:16 (deadlock+0x000000401451)
[text deleted]
SUMMARY: ThreadSanitizer: lock-order-inversion (potential deadlock) /usr/local/sde-2.15.0/packages/gcc-7.1.0/
       include/c++/7.1.0/x86_64-pc-linux-gnu/bits/gthr-default.h:748 in __gthread_mutex_lock
==================
a
b
ThreadSanitizer: reported 1 warnings
```

Memory Sanitizer (MSan)

- Memory Sanitizer (MSan) detects reads from uninitialized memory
- in contrast, ASan cannot detect uninitialized reads
- currently, MSan only supported by Clang (not GCC)
- compiler instruments all loads/stores
- uses bit to bit shadow mapping
- if not all code instrumented (so that not all stores are observed), false positives can result
- about 3 to 6 times slowdown
- about 2 to 3 times memory overhead

Using Memory Sanitizer

- need to enable sanitizer at compile and link time using `-fsanitize=memory` option for Clang
- environment variable `MSAN_OPTIONS` can be set to whitespace-separated list of options to control some sanitizer behavior at run time
- some sanitizer options include:
  - `strip_path_prefix`
  - `verbosity`
- for example, to set verbosity level to 2, use `MSAN_OPTIONS="verbosity=2"`

uninitialized_1.cpp

```
int main(int argc, char** argv) {
    int x[2];
    x[0] = 1;
    if (x[argc % 2]) {
        return 1;
    }
}
```

program output (truncated):

```
==22595==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x4a46c3 in main uninitialized_1.cpp:4:6
    #1 0x7f5d3908ffdf in __libc_start_main (/lib64/libc.so.6+0x1ffdf)
    #2 0x41a77e in _start (uninitialized_1+0x41a77e)
SUMMARY: MemorySanitizer: use-of-uninitialized-value uninitialized_1.
     cpp:4:6 in main
Exiting
```

Undefined-Behavior Sanitizer (UBSan)

- Undefined-Behavior Sanitizer (UBSan) detects code that results in various types of undefined behavior
- some types of problems detected include:
  - using misaligned or null pointer
  - signed integer overflow
  - conversion to, from, or between floating-point types which would overflow destination
  - reaching end of value-returning function with returning value
  - out of bounds array indexing where array bound can be statically determined
- compiler instruments code with extra checks
- supported by Clang and GCC
- slowdown varies between 0% and 50%

Using Undefined-Behavior Sanitizer

- need to enable sanitizer at compile and link time using `-fsanitize=undefined` option for Clang and GCC
- environment variable `UBSAN_OPTIONS` can be set to whitespace-separated list of options to control some sanitizer behavior at run time
- some sanitizer options include:
  - `suppressions`
  - `strip_path_prefix`
  - `verbosity`

signed_integer_overflow.cpp

```
#include <iostream>
#include <limits>
int main() {
    int x = std::numeric_limits<int>::max();
    int y = x + 1;
    std::cout << y << '\n';
}
```

program output:

```
signed_integer_overflow.cpp:5:14: runtime error: signed integer
    overflow: 2147483647 + 1 cannot be represented in type 'int'
-2147483648
```

invalid_shift.cpp

```
#include <iostream>
int main() {
    int x = 32678;
    int y = 1 << x;
    std::cout << y << '\n';
}
```

program output:

```
invalid_shift.cpp:4:12: runtime error: shift exponent 32678 is too
    large for 32-bit type 'int'
0
```

Leak Sanitizer (LSan)

- Leak Sanitizer (LSan) detects memory leaks
- supported by Clang and GCC
- adds almost no performance overhead until end of program, at which point extra leak-detection checks performed
- need to enable sanitizer at compile and link time using `-fsanitize=leak` option for Clang and GCC (or by using ASan, which includes LSan functionality)
- environment variable `LSAN_OPTIONS` can be set to whitespace-separated list of options to control some sanitizer behavior at run time
- some sanitizer options include:
  - `strip_path_prefix`
  - `verbosity`

heap_buffer_overflow.cpp

```
#include <iostream>
#include <cstring>
int main() {
    char* p = new char[1024];
    std::strcpy(p, "Hello, World!\n");
    std::cout << p;
}
```

program output:

```
Hello, World!
=================================================================
==10786==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 1024 byte(s) in 1 object(s) allocated from:
    #0 0x7faa5e0a7436 in operator new[](unsigned long) ../../../../src/
        libsanitizer/lsan/lsan_interceptors.cc:164
    #1 0x400894 in main memory_leak.cpp:4
SUMMARY: LeakSanitizer: 1024 byte(s) leaked in 1 allocation(s).
```

Section 7.2.1

References

Talks I

1. Kostya Serebryany. Sanitize Your C++ Code, CppCon, 2014. Available online at `https://youtu.be/V2_80g0eOMc`.
2. Kostya Serebryany, Beyond Sanitizers, CppCon, 2015. Available online at `https://youtu.be/qTkYDA0En6U`.

Section 7.3

Clang-Tidy

Clang-Tidy

- Clang-Tidy is static analysis tool for C/C++, which is part of Clang
- supports many checks, which consider such things as:
  - correctness
  - efficiency
  - readability
  - modern style
- can automatically fix code in many cases
- by default, only small subset of checks enabled
- probably not advisable to enable all checks, since many benign warnings may result, obscuring warnings that indicate serious problems
- web site: `https://clang.llvm.org/extra/clang-tidy`

The `clang-tidy` Command

- to generate compile-commands file (i.e., `compile_commands.json`), add following option to `cmake` command:
  `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON`
- command line has following form:

```
clang-tidy [options] [$source_file]...
```

- some options include:

Option	Description
`-checks=`*string*	specify check to include/exclude
`-p` *build_path*	set build path to *build_path*
`-version`	print version information and exit
`-help`	print help information and exit
`-list-checks`	list all enabled checks and exit
`-fix`	apply suggested fixes
`-fix-errors`	apply suggested fixes even if compilation errors found
`-warnings-as-errors=`*string*	treat specified warnings as errors

Some Supported Checks

- uninitialized arguments
- dereferencing null pointers
- division by zero
- address of stack memory that escape function
- undefined result of binary operator
- uninitialized array subscript
- assigning uninitialized values
- uninitialized branch condition
- blocks that capture uninitialized values
- uninitialized value being returned from function
- value-returning function that does not return value
- new-delete mismatch
- dead code (e.g., dead stores)
- use of unsafe functions (e.g., `getpw`, `gets`, `mktemp`, `strcpy`, `strcat`)
- use of inferior random number generating functions (e.g., `drand48`)
- some specific examples on subsequent slides

Division By Zero

divide_by_zero.cpp

```
int func(int x) {
    if (!x) {
        return 1024 / x;
    } else {
        return x;
    }
}
```

clang-tidy **output:**

```
divide_by_zero.cpp:3:15: warning: Division by zero [clang-analyzer-core
    .DivideZero]
              return 1024 / x;
                         ^

divide_by_zero.cpp:2:6: note: Assuming 'x' is 0
       if (!x) {
           ^

divide_by_zero.cpp:2:2: note: Taking true branch
       if (!x) {
       ^

divide_by_zero.cpp:3:15: note: Division by zero
              return 1024 / x;
                         ^
```

New-Delete Mismatch

new_delete_mismatch.cpp

```
int main() {
    char* p = new char[1024];
    delete p;
}
```

clang-tidy **output:**

```
new_delete_mismatch.cpp:3:2: warning: 'delete' applied to a pointer that was
     allocated with 'new[]'; did you mean 'delete[]'? [clang-diagnostic-
     mismatched-new-delete]
       delete p;
       ^
             []
new_delete_mismatch.cpp:2:12: note: allocated with 'new[]' here
       char* p = new char[1024];
                 ^
new_delete_mismatch.cpp:3:2: warning: Memory allocated by 'new[]' should be
     deallocated by 'delete[]', not 'delete' [clang-analyzer-unix.
     MismatchedDeallocator]
       delete p;
       ^
new_delete_mismatch.cpp:2:12: note: Memory is allocated
       char* p = new char[1024];
                 ^
new_delete_mismatch.cpp:3:2: note: Memory allocated by 'new[]' should be
     deallocated by 'delete[]', not 'delete'
       delete p;
       ^
```

Missing Return Statement

no_return.cpp

```
int func(int x) {
    if (x >= 0) {
        return 1;
    }
}
```

clang-tidy **output:**

```
no_return.cpp:5:1: warning: control may reach end of non-void function
    [clang-diagnostic-return-type]
}
^
```

Stack Address Escapes Function

stack_address_escape.cpp

```
int* p;

void test() {
    int x = 42;
    p = &x;
}
```

clang-tidy **output:**

```
stack_address_escape.cpp:6:1: warning: Address of stack memory
     associated with local variable 'x' is still referred to by the
     global variable 'p' upon returning to the caller. This will be a
     dangling reference [clang-analyzer-core.StackAddressEscape]
}
^
stack_address_escape.cpp:6:1: note: Address of stack memory associated
     with local variable 'x' is still referred to by the global variable
      'p' upon returning to the caller. This will be a dangling
     reference
}
^
```

Undefined Operand

undefined_operand.cpp

```
int test() {
    int x;
    return x + 1;
}
```

clang-tidy **output:**

```
undefined_operand.cpp:3:11: warning: The left operand of '+' is a
    garbage value [clang-analyzer-core.UndefinedBinaryOperatorResult]
        return x + 1;
                 ^
undefined_operand.cpp:2:2: note: 'x' declared without an initial value
        int x;
        ^
undefined_operand.cpp:3:11: note: The left operand of '+' is a garbage
    value
        return x + 1;
                 ^
```

Section 7.3.1

References

1. Daniel Jasper, Keep Your Code Sane With Clang Tidy, Meeting C++, Berlin, Germany, Dec. 4–5, 2015. Available online at `https://youtu.be/nzCLcfH3pb0`.

Section 7.4

Valgrind

Valgrind

- can detect many memory management and threading bugs
- can profile programs in detail
- home page: `http://www.valgrind.org`
- `valgrind` command
- `valkyrie` command (GUI for Memcheck and Helgrind tools in Valgrind)

Section 7.4.1

References

References I

1. P. Floyd. Valgrind part 1 — introduction. *Overload*, 108:14–15, Apr. 2012.
2. P. Floyd. Valgrind part 2 — basic memcheck. *Overload*, 109:24–28, June 2012.
3. P. Floyd. Valgrind part 3 — advanced memcheck. *Overload*, 110:4–7, Aug. 2012.
4. P. Floyd. Valgrind part 4 — cachegrind and callgrind. *Overload*, 111:4–7, Oct. 2012.
5. P. Floyd. Valgrind part 5 — massif. *Overload*, 112:20–24, Dec. 2012.

Section 7.5

Gcov and LLVM Cov

Gcov

- Gcov is code coverage analysis tool
- can obtain basic statistics on how many times each line of code executes
- can be used as guide for improving efficiency of code or to discover untested parts of programs
- GCov is part of GCC
- can be used in conjunction with compiler from GCC or Clang
- in order to generate data for GCov, program being run must be properly instrumented
- compiler instruments code to: count number of times each line of code executes and write this information to data files upon program termination
- when program run, coverage data files generated
- coverage data files can then be processed and displayed with Gcov
- web site: `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

LLVM Cov

- LLVM provides program called `llvm-cov` with functionality similar to GCov
- `gcov` subcommand of `llvm-cov` has very similar interface as `gcov`
- consqeuently, we only will consider case of Gcov in detail herein
- web site: `http://llvm.org/docs/CommandGuide/llvm-cov.html`

Using Gcov

- build each program for which code coverage information is desired
- must compile and link with `--coverage` option with GCC or Clang
- one block-graph description (`.gcno` extension) file generated for each object file produced during compilation and placed in same directory as corresponding object file
- run each program for which coverage information is desired one or more times
- when program exits, one count (`.gcda` extension) file generated for each object file associated with program
- if output file does not yet exist, file created
- if output file already exists, statistics are added to those already there
- that is, statistics maintained in data files are cumulative

Using Gcov (Continued)

- run `gcov` program to format data for display
- `gcov` generates sourcefile.gcov (or transformed version of this name) for sourcefile.cpp
- optimization and inline functions can cause strange behaviors in coverage statistics
- for example, optimization can cause multiple lines of code to be merged together, which will lead to unusual results for affected lines

Example: Source Code

example_main.cpp

```
#include <iostream>
#include <cstdlib>
double signum(double x);
int main(int argc, char** argv) {
    if (argc < 2) {
        return 1;
    }
    double x = std::atof(argv[1]);
    std::cout << signum(x) << '\n';
}
```

example_signum.cpp

```
double signum(double x) {
    if (x > 0) {
        return 1.0;
    } else if (x < 0) {
        return -1.0;
    } else {
        return 0.0;
    }
}
```

Example: Build and Run Program and Run Gcov

- build program `example` using GCC, ensuring that `--coverage` option is used for both compiling and linking; for example, using command sequence like:

```
g++ -c --coverage example_main.cpp
g++ -c --coverage example_signum.cpp
g++ -o example example_main.o example_signum.o --coverage
```

- run `example` program twice as follows:

```
example
example 1.0
```

- since `example` program run twice, statistics are accumulated from both runs of program
- run Gcov; for example, using command like:

```
gcov example_main.o example_signum.o
```

- view resulting .gcov files

Example: Gcov Output

example_main.cpp.gcov

```
        -:    0:Source:example_main.cpp
        -:    0:Programs:2
        -:    1:#include <iostream>
        -:    2:#include <cstdlib>
        -:    3:double signum(double x);
        2:    4:int main(int argc, char** argv) {
        2:    5:    if (argc < 2) {
        1:    6:        return 1;
        -:    7:    }
        1:    8:    double x = std::atof(argv[1]);
        1:    9:    std::cout << signum(x) << '\n';
        7:   10:}
```

example_signum.cpp.gcov

```
        -:    0:Source:example_signum.cpp
        -:    0:Programs:2
        1:    1:double signum(double x) {
        1:    2:    if (x > 0) {
        1:    3:        return 1.0;
    #####:    4:    } else if (x < 0) {
    #####:    5:        return -1.0;
        -:    6:    } else {
    #####:    7:        return 0.0;
        -:    8:    }
        -:    9:}
```

Lcov

- Lcov is graphical front end for Gcov
- collects Gcov data from multiple source files and creates HTML pages containing source code annotated with coverage information
- also provides overview pages for easy navigation
- Lcov supports statement, function, and branch coverage measurement
- web site: `http://ltp.sourceforge.net/coverage/lcov.php`

Using Lcov

- build project with GCC or Clang and ensure that `--coverage` option is used for compiling and linking as in earlier Gcov example
- run program to collect coverage data as in earlier Gcov example
- process coverage data with Lcov; for example, using command like:
```
lcov --capture --directory . --output-file coverage.info
```
- generate HTML output using `genhtml`; for example, using command like:
```
genhtml coverage.info --output-directory output
```
- view in browser; for example, using command like:
```
firefox output/index.html
```

Example: Lcov Output

LCOV - code coverage report

Current view:	top level - gcov - example_main.cpp (source / functions)		Hit	Total	Coverage
Test:	coverage.info	Lines:	6	6	100.0 %
Date:	2017-07-30	Functions:	3	3	100.0 %

```
Line data    Source code
           : #include <iostream>
           : #include <cstdlib>
           : double signum(double x);
         2 : int main(int argc, char** argv) {
         2 :         if (argc < 2) {
         1 :                 return 1;
           :         }
         1 :         double x = std::atof(argv[1]);
         1 :         std::cout << signum(x) << '\n';
         7 : }
```

Generated by: LCOV version 1.10

Example: Lcov Output (Continued)

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - gcov - example_signum.cpp (source / functions)				
Test:	coverage.info	Lines:	3	6	50.0 %
Date:	2017-07-30	Functions:	1	1	100.0 %

```
         Line data    Source code
  1             1 : double signum(double x) {
  2             1 :         if (x > 0) {
  3             1 :                 return 1.0;
  4             0 :         } else if (x < 0) {
  5             0 :                 return -1.0;
  6               :         } else {
  7             0 :                 return 0.0;
  8               :         }
  9               : }
```

Generated by: LCOV version 1.10

Section 7.5.1

References

References I

1. B. J. Gough, An Introduction to GCC, Network Theory Limited, UK, 2004.
2. GNU Compiler Collection (GCC), `https://gcc.gnu.org`.
3. LLVM Clang, `https://clang.llvm.org`.

Section 7.6

Catch2

- Catch2 (originally known as Catch) is multiparadigm test framework for C++
- Catch2 stands for “C++ automated test cases in a header”
- primarily distributed as single header library
- open source; released under Boost Software License
- written by Phil Nash
- official Git repository: `http://github.com/catchorg/Catch2`
- Google group: `http://groups.google.com/group/catch-forum`

```
#include <stdexcept>
#include <numeric>

class counter {
public:
    using count_type = std::size_t;
    static constexpr count_type max_count()
      {return std::numeric_limits<count_type>::max();}
    counter(count_type count = 0) : count_(count) {}
    count_type get_count() const {return count_;}
    void increment() {
        if (count_ == max_count())
          {throw std::overflow_error("counter overflow");}
        ++count_;
    }
private:
    count_type count_;
};
```

Counter Class Example: Test Code

```
#define CATCH_CONFIG_MAIN
#include <catch/catch.hpp>
#include "counter.hpp"

TEST_CASE("constructor", "[counter]") {
    counter x;
    CHECK(x.get_count() == 0);
    counter y(1);
    CHECK(y.get_count() == 1);
}

TEST_CASE("maximum count", "[counter]") {
    CHECK(counter::max_count() == std::numeric_limits<
      counter::count_type>::max());
}

TEST_CASE("increment (no overflow)", "[counter]") {
    counter x(0);
    REQUIRE(x.get_count() == 0);
    x.increment();
    CHECK(x.get_count() == 1);
}

TEST_CASE("increment (overflow)", "[counter]") {
    counter x(counter::max_count());
    CHECK_THROWS_AS(x.increment(), std::overflow_error);
}
```

Approximate Comparison Example

```
#define CATCH_CONFIG_MAIN
#include <catch/catch.hpp>

TEST_CASE("addition") {
    float x = 0.0f;
    for (int i = 0; i < 10; ++i) {
        x += 0.1f;
    }
    CHECK(x == 1.0f);
      // may fail due to roundoff error
    CHECK(x == Approx(1.0f));
      // should pass
}
```

Talks I

1. Phil Nash, Modern C++ Testing with Catch2, Meeting C++, Berlin, Germany, Nov. 9, 2017. Available online at `https://youtu.be/3tIE6X5FjDE`.
2. Phil Nash, Modern C++ Testing with Catch2, C++ Edinburgh, Edinburgh, UK, Aug. 14, 2017. Available online at `https://youtu.be/grC0S6ZK59U`.
3. Phil Nash, Test Driven C++ with Catch, CppCon, Bellevue, WA, USA, Sept. 22, 2015. Available online at `https://youtu.be/gdzP3pAC6UI`.
4. Phil Nash, Testdriven C++ with Catch, Meeting C++, Berlin, Germany, Dec. 5–6, 2014. Available online at `https://youtu.be/C2LcIp56i-8`.

Part 8

Performance Analysis Tools

Section 8.1

Perf

Linux Kernel Perf Event Interface

- Linux kernel provides Perf Event (i.e., `perf_event`) interface for performance monitoring
- `perf_event_open` system call returns file descriptor that can then be used to collect performance information
- collection of performance data started and stopped with `ioctl` system call
- performance data accessed either via `read` or `mmap` system call
- Perf Event interface used by numerous performance analysis tools and libraries on Linux systems (e.g., Perf and PAPI)
- supports many profiling/tracing features, including:
  - CPU performance monitoring counters
  - statically defined tracepoints
  - user and kernel dynamic tracepoints
- good documentation on Perf Event interface is scarce

Perf

- open-source profiling tool
- can collect aggregated counts of events during code execution
- can perform event-driven sample-based profiling
- uses Perf Event interface of Linux kernel
- noninvasive (i.e., no code instrumentation required)
- low overhead (i.e., code runs close to native speed)
- sample-based profiling can collect stack traces in addition to instruction pointer
- does not provide call counts for functions
- web site: `https://perf.wiki.kernel.org`

Events

- hardware event:
  - event measurable by performance monitoring unit (PMU) of processor
  - examples: CPU cycles (`cycles`) and cache misses (`cache-misses`)
- hardware cache event:
  - event measurable by PMU of processor
  - examples: L1 data cache load misses (`L1-dcache-load-misses`) and data translation-lookaside-buffer load misses (`dTLB-load-misses`)
- software event:
  - low-level events based on kernel counters
  - examples: CPU clock (`cpu-clock`) and page fault (`page-faults`)
- kernel tracepoint event:
  - predefined static instrumentation points in kernel code where trace information can be collected
  - examples: entering `open` system call (`syscalls:sys_enter_open`) and context switch (`sched:sched_switch`)
- probe event:
  - user-defined events dynamically inserted into kernel
  - created using uprobes or kprobes

Some Events

Event	Description
`cache-misses`	cache misses
`cache-references`	cache accesses
`cycles`	CPU cycles
`cpu-clock`	CPU wall-time clock
`instructions`	CPU instructions
`cs`	context switches
`faults`	page faults

- stack trace is list of stack frames

Event-Based Sampling

- with event-based sampling, sampling process driven by one or more types of events
- sample is taken upon occurrence of every nth event, where n is either:
  - directly specified by user; or
  - dynamically chosen by kernel in order to (approximately) meet average sampling rate specified by user
- default sampling event is cycles with average sampling rate that depends on Perf version (typically 1000 Hz to 4000 Hz)
- cycles event does not necessarily have constant relationship with time, due to CPU frequency scaling
- each sample captures:
  - timestamp
  - CPU number, process ID (PID), and thread ID (TID)
  - instruction pointer
  - stack trace (optional)
- can perform sampling:
  - system wide, per processor, per program, or per thread

- event specifier consists of event name, optionally followed by colon and then one of more event modifiers
- list of event modifiers as follows:

Modifier	Description
`u`	user-space counting
`k`	kernel counting
`h`	hypervisor counting
`i`	non-idle counting
`G`	guest counting (in KVM guests)
`H`	host counting (not in KVM guests)
`p`	preciseness level (i.e., amount of skid)
`S`	read sample value
`D`	pin event to PMU

Event Specifiers (Continued)

- number n of `p`'s in modifier influences preciseness of event measurement as follows:

n	Description
0	can have arbitrary skid
1	must have constant skid
2	requested (but not required) to have zero skid
3	must have zero skid

- if zero skid required but not supported, error will be generated
- some examples of event specifiers are as follows:

Event Specifier	Meaning
`cycles:u`	clock cycles in user space
`cache-misses:u`	cache misses in user space
`cache-misses:k`	cache misses in kernel
`cache-misses:uppp`	cache misses in user space with zero skid

Hardware Event Skid

- measurements involving hardware counters typically employ interrupts
- when hardware counter for event overflows, interrupt occurs
- when overflow interrupt occurs, takes CPU some amount of time to stop processor and pointpoint exactly which instruction was active at time of overflow
- due to this delay, can often be offset in execution flow between instruction claimed to be active at time of overflow and instruction that actually was active
- this offset known as **skid**
- in some cases, for example, skid could result in caller function event being recorded in callee function
- due to skid, some care must be taken when interpreting profiling results

The `perf` Program

- functionality of Perf software provided by `perf` program
- command line interface has following form:

```
perf [options] command [args]
```

- some common commands include:

Command	Description
`list`	list all symbolic event types
`stat`	run command and gather performance count statistics
`record`	run command and record its profile into Perf data file
`report`	read Perf data (created by Perf record) and display profile
`script`	read Perf data (created by Perf record) and display trace output
`annotate`	read Perf data (created by Perf record) and display annotated code

- some common options include:

Option	Description
`--help`	print help information and exit
`--version`	print version information and exit

Perf List Command

- list all symbolic event types
- command line interface has following form:

```
perf list [event_type]
```

- event types include:

Event Type	Description
`hw`	hardware
`sw`	software
`cache`	cache
`tracepoint`	tracepoint
`pmu`	PMU
glob_expr	any event matching glob expression *glob_expr*

- only lists event types available to invoking user
- some events only available to root user

Perf List Example

```
$ perf list
List of pre-defined events (to be used in -e):
  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  bus-cycles                                         [Hardware event]
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  ref-cycles                                         [Hardware event]
[text deleted]
  alignment-faults                                   [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
[text deleted]
  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-loads                                    [Hardware cache event]
  L1-dcache-prefetch-misses                          [Hardware cache event]
  L1-dcache-store-misses                             [Hardware cache event]
  L1-dcache-stores                                   [Hardware cache event]
  L1-icache-load-misses                              [Hardware cache event]
[text deleted]
  cache-misses OR cpu/cache-misses/                  [Kernel PMU event]
  cache-references OR cpu/cache-references/          [Kernel PMU event]
  cpu-cycles OR cpu/cpu-cycles/                      [Kernel PMU event]
  instructions OR cpu/instructions/                  [Kernel PMU event]
  mem-loads OR cpu/mem-loads/                        [Kernel PMU event]
  mem-stores OR cpu/mem-stores/                      [Kernel PMU event]
[text deleted]
```

Perf Stat Command

- run command and gather performance count statistics
- command line interface has following form:

  ```
  perf stat [options] command [args]
  ```

- some common options include:

Option	Description
`-e` *event*	specify event for which to gather statistics
`-p` *pid*	consider events on existing process ID
`-t` *tid*	consider events on existing thread ID
`-a`	consider all processors (i.e., system wide)
`-r` *n*	repeat command n times and print averages and standard deviations
`-C` *cpu*	consider only CPUs specified by *cpu*

Perf Stat Example

```
$ perf stat dd if=/dev/urandom of=/dev/null bs=1K count=32K status=none
 Performance counter stats for
 'dd if=/dev/urandom of=/dev/null bs=1K count=32K status=none':

       1727.055828      task-clock (msec)         #    0.999 CPUs utilized
                 1      context-switches          #    0.001 K/sec
                13      cpu-migrations            #    0.008 K/sec
                60      page-faults               #    0.035 K/sec
     5,805,261,702      cycles                    #    3.361 GHz
     2,115,865,103      stalled-cycles-frontend   #   36.45% frontend
                                                      cycles idle
   <not supported>      stalled-cycles-backend
    12,108,757,065      instructions              #    2.09  insns per cycle
                                                  #    0.17  stalled cycles
                                                       per insn
       254,471,634      branches                  #  147.344 M/sec
           257,282      branch-misses             #    0.10% of all branches

       1.728232622 seconds time elapsed
```

Perf Record Command

- run command and record its profile into Perf data file
- command line interface has following form:

```
perf record [options] command [args]
```

- some common options include:

Option	Description
`-e` *event*	specify event name
`-a`	collect data from all processors
`-p` *pid*	collect data from existing process ID *pid*
`-t` *tid*	collect data from existing thread ID *tid*
`-C` *cpu*	collect data from CPUs *cpu*
`-c` *count*	set event count between samples to *count*
`-o` *file*	set output file to *file*
`-F` *freq*	set sampling frequency to approximately *freq*
`-g`	enable call graph (i.e., stack trace) recording

- output file defaults to `perf.data`
- by default, uses `cycles` event with sampling frequency set to version-dependent value (typically, 1000 Hz to 4000 Hz)

Perf Record Example

```
$ perf record -g -F 99 -o perf.data dd if=/dev/urandom of=/dev/null \
  bs=1K count=3200K status=none
[ perf record: Woken up 9 times to write data ]
[ perf record: Captured and wrote 2.121 MB perf.data (16246 samples) ]
$ ls
perf.data
```

Perf Report Command

- read Perf data (created by Perf record) and display profile
- command line interface has following form:

```
perf report [options]
```

- some common options include:

Option	Description
`-i` *file*	set input file to *file*
`-v`	increase verbosity level
`-n`	show number of samples for each symbol
`-C` *cpu*	only show events for CPU *cpu*
`--pid` *pid*	only show events for process ID *pid*
`--tid` *tid*	only show events for thread ID *tid*
`-d` *dsos*	only consider symbols in DSO/object files *dsos*
`-S` *syms*	only consider symbols *syms*
`-s` *key*	sort data by key *key* (such as PID)
`--stdio`	use stdio interface
`-U`	only display entries that resolve to symbol
`-D`	dump raw trace data

- input file defaults to `perf.data`

```
$ perf record -g -e cycles:u -F 13000 -o perf.data ./array_sum
1
1
$ perf report -i perf.data -d array_sum --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
# dso: array_sum
# Samples: 1K of event 'cycles:u'
# Event count (approx.): 158559166
#
# Children      Self  Command    Symbol
# ........  ........  .........  ................
#
    79.94%    79.94%  array_sum  [.] naive_sum
            |
            ---naive_sum
               main
               __libc_start_main
               0x48e258d4c544155

     7.56%     7.56%  array_sum  [.] improved_sum
            |
            ---improved_sum
               main
               __libc_start_main
               0x48e258d4c544155



#
# (For a higher level overview, try: perf report --sort comm,dso)
#
```

Perf Script Command

- read Perf data (created by Perf record) and display trace output
- command line interface has following form:

  `perf script [options]`

- some common options include:

Option	Description
`-i` *file*	set input file to *file*
`--pid` *pid*	only show events for process ID *pid*
`--tid` *tid*	only show events for thread ID *tid*
`-C` *cpu*	only show events for CPU *cpu*

- input file defaults to `perf.data`

Perf Script Example

```
$ perf record -g -e cycles:u -F 13000 -o perf.data ./array_sum
1
1
$ perf script -i perf.data
array_sum 15602 2408817.214222:          1 cycles:u:
                 cf0 _start (/usr/lib64/ld-2.20.so)

array_sum 15602 2408817.214230:          1 cycles:u:
                 cf0 _start (/usr/lib64/ld-2.20.so)

array_sum 15602 2408817.214234:          2 cycles:u:
                 cf0 _start (/usr/lib64/ld-2.20.so)

array_sum 15602 2408817.214237:          7 cycles:u:
                 cf0 _start (/usr/lib64/ld-2.20.so)

array_sum 15602 2408817.214241:         25 cycles:u:
                 cf0 _start (/usr/lib64/ld-2.20.so)

array_sum 15602 2408817.214245:         88 cycles:u:
                 cf0 _start (/usr/lib64/ld-2.20.so)

array_sum 15602 2408817.214248:        308 cycles:u:
                 cf0 _start (/usr/lib64/ld-2.20.so)

array_sum 15602 2408817.214253:       1081 cycles:u:
    ffffffff8179bef0 page_fault ([kernel.kallsyms])
                 cf0 _start (/usr/lib64/ld-2.20.so)

array_sum 15602 2408817.214270:       3147 cycles:u:
                4980 _dl_start (/usr/lib64/ld-2.20.so)

array_sum 15602 2408817.214274:       4536 cycles:u:
                4b8f _dl_start (/usr/lib64/ld-2.20.so)
                 cf8 _dl_start_user (/usr/lib64/ld-2.20.so)
[text deleted]
```

Perf Annotate Command

- read Perf data (created by Perf record) and display annotated code
- command line interface has following form:

```
perf annotate [options]
```

- some common options include:

Option	Description
`-i` *file*	set input file to *file*
`-s` *sym*	annotate symbol *sym*
`-d` *dsos*	only consider symbols in DSO/object files *dsos*
`-v`	increase verbosity level
`-l`	print matching source lines
`-P`	do not shorten displayed pathnames
`-k` *file*	set vmlinux pathname to *file*
`--stdio`	use stdio interface
`--no-source`	disable displaying of source code

- input file defaults to `perf.data`

Perf Annotate Example

```
$ perf record -g -e cycles:u -F 13000 -o perf.data ./array_sum
[text deleted]
$ perf annotate -i perf.data -s naive_sum -l --stdio
[text deleted]
          :   double naive_sum(const double a[][N]) {
    0.00 :     400807:       push   %rbp
    0.00 :     400808:       mov    %rsp,%rbp
    0.00 :     40080b:       lea    0x4000(%rdi),%rcx
          :          double sum = 0.0;
    0.00 :     400812:       pxor   %xmm0,%xmm0
    0.00 :     400816:       lea    0x2000000(%rdi),%rdx
    0.00 :     40081d:       mov    %rdi,%rax
          :          for (int j = 0; j < N; ++j) {
          :                 for (int i = 0; i < M; ++i) {
          :                        sum += a[i][j];
    0.00 :     400820:       addsd  (%rax),%xmm0
 array_sum.cpp:11  100.00 :   400824:       add    $0x4000,%rax
[text deleted]
          :   double naive_sum(const double a[][N]) {
          :          double sum = 0.0;
          :          for (int j = 0; j < N; ++j) {
          :                 for (int i = 0; i < M; ++i) {
    0.00 :     40082a:       cmp    %rdx,%rax
    0.00 :     40082d:       jne    400820 <naive_sum(double const (*) [2048])+0x19>
    0.00 :     40082f:       add    $0x8,%rdi
[text deleted]
    0.00 :     400833:       cmp    %rcx,%rdi
    0.00 :     400836:       jne    400816 <naive_sum(double const (*) [2048])+0xf>
[text deleted]
          :                 }
          :          }
          :          return sum;
          :   }
    0.00 :     400838:       pop    %rbp
    0.00 :     400839:       retq
```

Example: Source Code

```
#include <iostream>
#include <algorithm>

constexpr int M = 4096;
constexpr int N = 4096;

[[gnu::noinline]]
double naive_sum(const double a[][N]) {
    double sum = 0.0;
    for (int j = 0; j < N; ++j) {
        for (int i = 0; i < M; ++i) {
            sum += a[i][j];
        }
    }
    return sum;
}

[[gnu::noinline]]
double improved_sum(const double a[][N]) {
    double sum = 0.0;
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < N; ++j) {
            sum += a[i][j];
        }
    }
    return sum;
}
```

Example: Source Code (Continued)

```
int main() {
    for (int i = 0; i < 16; ++i) {
        static double a[M][N];
        static double b[M][N];
        std::fill_n(&a[0][0], M * N, 1.0 / (M * N));
        std::fill_n(&b[0][0], M * N, 1.0 / (M * N));
        std::cout << naive_sum(a) << ' ';
        std::cout << improved_sum(b) << '\n';
    }
}
```

Profile of Cycles

```
# To display the perf.data header info, please use --header/--header-only options.
#
# dso: array_sum
# Samples: 16K of event 'cycles:u'
# Event count (approx.): 14049539983
#
# Children      Self  Command    Symbol
# ........  ........  .........  .....................
#
    99.97%     0.00%  array_sum  [.] __libc_start_main
            |
            ---__libc_start_main
               0x46e258d4c544155

    99.97%    10.92%  array_sum  [.] main
            |
            ---main
               __libc_start_main
               0x46e258d4c544155

    82.97%    82.97%  array_sum  [.] naive_sum
            |
            ---naive_sum
               main
               __libc_start_main
               0x46e258d4c544155

     5.90%     5.90%  array_sum  [.] improved_sum
            |
            ---improved_sum
               main
               __libc_start_main
               0x46e258d4c544155

[text deleted]
```

Cycles for naive_sum

```
         :   0000000000400807 <naive_sum(double const (*) [4096])>:
         :   _Z9naive_sumPA4096_Kd():
[text deleted]
         :   [[gnu::noinline]]
         :   double naive_sum(const double a[][N]) {
    0.00 :     400807:        push   %rbp
    0.00 :     400808:        mov    %rsp,%rbp
    0.00 :     40080b:        lea    0x8000(%rdi),%rcx
         :           double sum = 0.0;
    0.00 :     400812:        pxor   %xmm0,%xmm0
    0.00 :     400816:        lea    0x8000000(%rdi),%rdx
    0.00 :     40081d:        mov    %rdi,%rax
         :           for (int j = 0; j < N; ++j) {
         :                   for (int i = 0; i < M; ++i) {
         :                           sum += a[i][j];
    0.00 :     400820:        addsd  (%rax),%xmm0
 array_sum.cpp:12   99.93 :   400824:        add    $0x8000,%rax
[text deleted]
         :           for (int j = 0; j < N; ++j) {
         :                   for (int i = 0; i < M; ++i) {
    0.07 :     40082a:        cmp    %rdx,%rax
    0.00 :     40082d:        jne    400820 <naive_sum(double const (*) [4096])+0x19>
    0.00 :     40082f:        add    $0x8,%rdi
[text deleted]
         :   [[gnu::noinline]]
         :   double naive_sum(const double a[][N]) {
         :           double sum = 0.0;
         :           for (int j = 0; j < N; ++j) {
    0.00 :     400833:        cmp    %rcx,%rdi
    0.00 :     400836:        jne    400816 <naive_sum(double const (*) [4096])+0xf>
         :                   for (int i = 0; i < M; ++i) {
         :                           sum += a[i][j];
         :                   }
         :           }
         :           return sum;
         :   }
    0.00 :     400838:        pop    %rbp
    0.00 :     400839:        retq
```

Cycles for improved_sum

```
           :   000000000040083a <improved_sum(double const (*) [4096])>:
           :   _Z12improved_sumPA4096_Kd():
[text deleted]
           :   [[gnu::noinline]]
           :   double improved_sum(const double a[][N]) {
      0.00 :     40083a:        push   %rbp
      0.00 :     40083b:        mov    %rsp,%rbp
      0.00 :     40083e:        lea    0x8000000(%rdi),%rdx
           :           double sum = 0.0;
      0.00 :     400845:        pxor   %xmm0,%xmm0
      0.00 :     400849:        lea    0x8000(%rdi),%rax
           :           for (int i = 0; i < M; ++i) {
           :                   for (int j = 0; j < N; ++j) {
           :                           sum += a[i][j];
      0.00 :     400850:        addsd  (%rdi),%xmm0
 array_sum.cpp:23    99.70 :    400854:        add    $0x8,%rdi
[text deleted]
           :           for (int i = 0; i < M; ++i) {
           :                   for (int j = 0; j < N; ++j) {
      0.30 :     400858:        cmp    %rax,%rdi
      0.00 :     40085b:        jne    400850 <improved_sum(double const (*) [4096])+0x16>
           :   }
           :
           :   [[gnu::noinline]]
           :   double improved_sum(const double a[][N]) {
           :           double sum = 0.0;
           :           for (int i = 0; i < M; ++i) {
      0.00 :     40085d:        cmp    %rdx,%rdi
      0.00 :     400860:        jne    400849 <improved_sum(double const (*) [4096])+0xf>
           :                   for (int j = 0; j < N; ++j) {
           :                           sum += a[i][j];
           :                   }
           :           }
           :           return sum;
           :   }
      0.00 :     400862:        pop    %rbp
      0.00 :     400863:        retq
```

Profile of Cache Misses

```
# To display the perf.data header info, please use --header/--header-only options.
#
# dso: array_sum
# Samples: 25K of event 'cache-misses:u'
# Event count (approx.): 256620000
#
# Children      Self  Command    Symbol
# ........  ........  .........  .....................
#
    99.99%     0.00%  array_sum  [.] __libc_start_main
            |
            ---__libc_start_main
               0x46e258d4c544155

    99.99%     3.67%  array_sum  [.] main
            |
            ---main
               __libc_start_main
               0x46e258d4c544155

    93.74%    93.73%  array_sum  [.] naive_sum
            |
            ---naive_sum
               main
               __libc_start_main
               0x46e258d4c544155

     2.58%     2.58%  array_sum  [.] improved_sum
            |
            ---improved_sum
               main
               __libc_start_main
               0x46e258d4c544155

[text deleted]
```

Cache Misses for naive_sum

```
          :   0000000000400807 <naive_sum(double const (*) [4096])>:
          :   _Z9naive_sumPA4096_Kd():
[text deleted]
          :   [[gnu::noinline]]
          :   double naive_sum(const double a[][N]) {
    0.00  :     400807:       push   %rbp
    0.00  :     400808:       mov    %rsp,%rbp
    0.00  :     40080b:       lea    0x8000(%rdi),%rcx
          :           double sum = 0.0;
    0.00  :     400812:       pxor   %xmm0,%xmm0
    0.00  :     400816:       lea    0x8000000(%rdi),%rdx
    0.00  :     40081d:       mov    %rdi,%rax
          :           for (int j = 0; j < N; ++j) {
          :                   for (int i = 0; i < M; ++i) {
          :                           sum += a[i][j];
    0.00  :     400820:       addsd  (%rax),%xmm0
 array_sum.cpp:12    99.93 :    400824:       add    $0x8000,%rax
[text deleted]
          :           for (int j = 0; j < N; ++j) {
          :                   for (int i = 0; i < M; ++i) {
    0.07  :     40082a:       cmp    %rdx,%rax
    0.00  :     40082d:       jne    400820 <naive_sum(double const (*) [4096])+0x19>
    0.00  :     40082f:       add    $0x8,%rdi
[text deleted]
          :   [[gnu::noinline]]
          :   double naive_sum(const double a[][N]) {
          :           double sum = 0.0;
          :           for (int j = 0; j < N; ++j) {
    0.00  :     400833:       cmp    %rcx,%rdi
    0.00  :     400836:       jne    400816 <naive_sum(double const (*) [4096])+0xf>
          :                   for (int i = 0; i < M; ++i) {
          :                           sum += a[i][j];
          :                   }
          :           }
          :           return sum;
          :   }
    0.00  :     400838:       pop    %rbp
    0.00  :     400839:       retq
```

Cache Misses for improved_sum

```
         :   000000000040083a <improved_sum(double const (*) [4096])>:
         :   _Z12improved_sumPA4096_Kd():
[text deleted]
         :   [[gnu::noinline]]
         :   double improved_sum(const double a[][N]) {
    0.00 :     40083a:       push   %rbp
    0.00 :     40083b:       mov    %rsp,%rbp
    0.00 :     40083e:       lea    0x8000000(%rdi),%rdx
         :           double sum = 0.0;
    0.00 :     400845:       pxor   %xmm0,%xmm0
    0.00 :     400849:       lea    0x8000(%rdi),%rax
         :           for (int i = 0; i < M; ++i) {
         :                   for (int j = 0; j < N; ++j) {
         :                           sum += a[i][j];
    0.00 :     400850:       addsd  (%rdi),%xmm0
 array_sum.cpp:23   99.70 :   400854:       add    $0x8,%rdi
[text deleted]
         :           for (int i = 0; i < M; ++i) {
         :                   for (int j = 0; j < N; ++j) {
    0.30 :     400858:       cmp    %rax,%rdi
    0.00 :     40085b:       jne    400850 <improved_sum(double const (*) [4096])+0x16>
         :   }
         :
         :   [[gnu::noinline]]
         :   double improved_sum(const double a[][N]) {
         :           double sum = 0.0;
         :           for (int i = 0; i < M; ++i) {
    0.00 :     40085d:       cmp    %rdx,%rdi
    0.00 :     400860:       jne    400849 <improved_sum(double const (*) [4096])+0xf>
         :                   for (int j = 0; j < N; ++j) {
         :                           sum += a[i][j];
         :                   }
         :           }
         :           return sum;
         :   }
    0.00 :     400862:       pop    %rbp
    0.00 :     400863:       retq
```

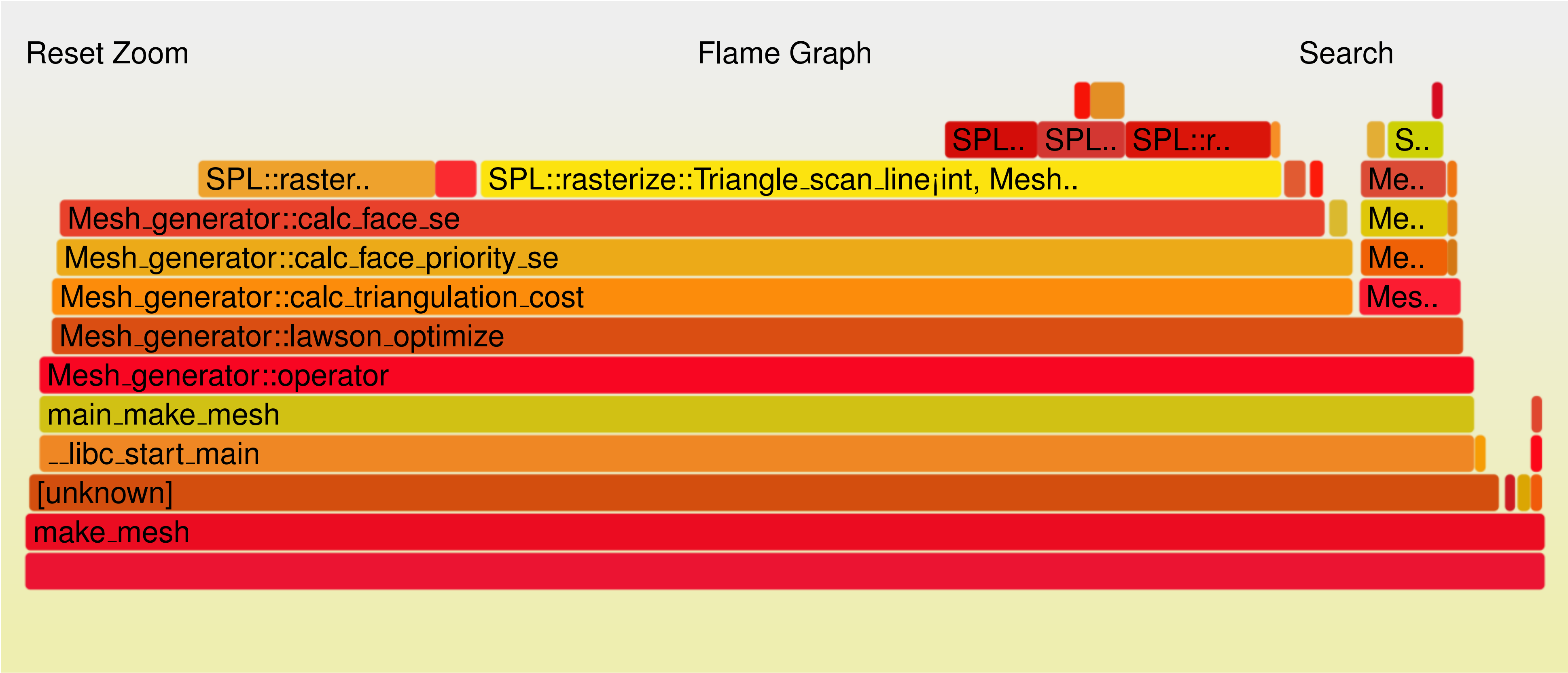
Additional Remarks

- avoid sampling in lockstep with periodic behavior exhibited by programs (e.g., caused by timeouts or loops)
- since programmers often choose timeout (and other timing related) values to be "nice" numbers, such as integer multiples of 0.01 s, may be beneficial to choose sampling frequency of 99 Hz instead of 100 Hz or 999 Hz instead of 1000 Hz
- sample-based profiling only provides meaningful results if sufficient number of samples collected
- can use `taskset` command to pin process for particular CPU
- might want to force single-threaded program to run on fixed CPU so that migration does not impact measurements (e.g., due to cacheing effects)

Flame Graphs

- flame graph provides way to visualize collection of stack traces
- useful for visualizing output of profiler that collects stack traces using sampling (e.g., Perf)
- stack trace represented as column of boxes, with each box corresponding to function in stack trace
- function executing at time of stack trace shown at top of column
- vertical direction corresponds to stack depth
- horizontal direction spans stack trace collection (does not represent time)
- left to right ordering has no special meaning
- when identical function boxes horizontally adjacent, merged
- width of each function box shows frequency with which function present in part of stack trace ancestry
- functions with wider boxes more frequent in stack traces than those with narrower boxes

Flame Graph Example



Generating Flame Graphs

- can generate flamegraphs from Perf data by using software available from
  - `https://github.com/brendangregg/FlameGraph`
- need to use `stackcollapse-perf.pl` and `flamegraph.pl` programs
- convert Perf data from binary to text format via Perf script command; for example:

  ```
  perf script -i perf.data > tmp.perf
  ```
- fold stack samples into single lines via `stackcollapse-perf.pl command`; for example:

  ```
  stackcollapse-perf.pl tmp.perf > tmp.folded
  ```
- generate flame graph in SVG format via `flamegraph.pl` command; for example:

  ```
  flamegraph.pl tmp.folded > flamegraph.svg
  ```

Section 8.1.1

References

Talks I

1. Brendan Gregg, Linux Profiling at Netflix, Southern California Linux Expo (SCaLE), Los Angeles, CA, USA, Feb. 27, 2015. Available online at `https://youtu.be/_Ik8oiQvWgo`.
2. Mans Rullgard, Performance Analysis Using the perf Suite, Embedded Linux Conference, March 2015, San Jose, CA, USA. Available online at `https://youtu.be/kWnx6eOGVYo`.
3. Chandler Carruth, Tuning C++: Benchmarks, and CPUs, and Compilers! Oh My!, CppCon, Bellevue, WA, USA, Sept. 24, 2015. Available online at `https://youtu.be/nXaxk27zwlk`.
4. Brendan Gregg, Blazing Performance with Flame Graphs, Large Installation System Administration Conference (LISA), Washington, DC, USA, Nov. 2013. Available online at `https://youtu.be/nZfNehCzGdw`.

References I

1. B. Gregg The Flame Graph, ACM Queue, March 2016, pages 1–28.

Section 8.2

Performance API (PAPI) Software

Motivation

- often easy to identify in general terms which parts of code are slow
- sometimes more difficult to pinpoint precise reason why code is slow (i.e., what is precise cause of bottleneck)
- often need to consider factors such as:
  - cache behavior
  - memory and resource contention
  - floating-point efficiency
  - branch behavior
- often, processor itself in best position to provide information related to above factors

Hardware Performance Counters

- hardware performance counters are specialized registers used to measure various aspects of processor performance
- hardware counters can provide insight into:
  - timing
  - cache behaviors (e.g., cache misses and cache coherence protocol events)
  - branch behaviors (e.g., incorrect branch predictions)
  - pipeline behavior (e.g., stalls)
  - memory and resource access patterns
  - floating-point efficiency
  - instructions per cycle
- hardware counter information can be obtained with:
  - subroutine or basic block resolution
  - process or thread attribution
- provide low-level information that often cannot be obtained easily through other means
- useful for performance analysis and tuning (e.g., identifying bottlenecks in code)
- use of hardware performance counters has no or little overhead

Performance API (PAPI) Software

- Performance API (PAPI) software provides portable and efficient API for accessing hardware performance counters found on modern processors
- more generally allows monitoring of system information on range of components, such as CPUs, network interface cards, and power monitors
- consists of library and several utility programs
- open source
- written in C
- supports most mainstream Unix-based operating systems (e.g., Linux, OS X, and other Unix variants); older versions support Microsoft Windows
- supports most modern processors (e.g., Intel and AMD 32- and 64-bit x86, ARM, MIPS, Intel Itanium II, UltraSparc I, II, and III, and IBM Power 4, 5, 6, and 7)
- web site: `http://icl.utk.edu/papi`

Events

- event is simply some action that can be counted
- **native event**: event that is specified in platform-dependent manner and directly corresponds to particular hardware counter
- which native events are available will depend on underlying hardware
- **preset event**: event that is specified in platform-independent manner, which is then mapped to appropriate native event(s) (e.g., `PAPI_TOT_INS`)
- **derived event**: preset event derived from multiple native events
- if hardware does not directly support counting of particular event, event count can sometimes be computed by using combination of native events
- for example, `PAPI_L1_TCM` might be derived from L1 data misses plus L1 instruction misses
- preset events usually available for most processors, where derived events used in cases where no corresponding native event exists

Events (Continued)

- which events supported and which combinations of supported events can be used together depends on hardware
- hardware will typically have some upper limit on number of events that can be monitored simultaneously
- some events often cannot be used with others (even if upper limit on number of events not exceeded)
- `papi_avail` or `papi_native_avail` utility (discussed later) can be used to determine number of hardware counters available
- `papi_avail` utility (discussed later) can be used to determine which preset events are supported
- `papi_native_avail` utility (discussed later) can be used to determine which native events are supported
- `papi_event_chooser` utility (discussed later) can be used to determine which events can be used with which other events

PAPI

- must include header file `papi.h`
- library initialized with function `PAPI_library_init`
- depending on which functions used, may need to explicitly initialize library

PAPI High-Level Interface

- calls low-level API
- easier to use than low-level API
- usually enough for more basic measurements
- for preset events only
- high-level interface functions will initialize library if needed (so `PAPI_library_init` need not be explicitly called)

Functions in PAPI High-Level Interface

Function	Description
`PAPI_accum_counters`	add current counts to array and reset counters
`PAPI_flips`	get floating-point instruction rate and real and processor time
`PAPI_flops`	get floating-point operation rate and real and processor time
`PAPI_ipc`	get instructions per cycle and real and processor time
`PAPI_num_counters`	get number of hardware counters available on system
`PAPI_read_counters`	copy current counts to array and reset counters
`PAPI_start_counters`	start counting hardware events
`PAPI_stop_counters`	stop counters and return current counts

Some Commonly-Used Preset Events

Instruction Mix

Name	Description
PAPI_LD_INS	number of load instructions
PAPI_SR_INS	number of store instructions
PAPI_LST_INS	number of load/store instructions
PAPI_BR_INS	number of branch instructions
PAPI_INT_INS	number of integer instructions
PAPI_FP_INS	number of floating-point instructions
PAPI_VEC_INS	number of vector/SIMD instructions
PAPI_VEC_SP	number of single-precision vector/SIMD instructions
PAPI_VEC_DP	number of double-precision vector/SIMD instructions
PAPI_TOT_INS	number of instructions in total

Some Commonly-Used Preset Events (Continued 1)

Clock Cycles

Name	Description
`PAPI_TOT_CYC`	total number of clock cycles

FLOPS

Name	Description
`PAPI_FP_OPS`	number of floating-point operations
`PAPI_SP_OPS`	number of floating-point operations executed, optimized to count scaled single-precision vector operations
`PAPI_DP_OPS`	number of floating-point operations executed, optimized to count scaled double-precision vector operations

Translation Lookaside Buffer (TLB)

Name	Description
`PAPI_TLB_DM`	number of data TLB misses
`PAPI_TLB_IM`	number of instruction TLB misses
`PAPI_TLB_TL`	number of TLB misses (in total)

Some Commonly-Used Preset Events (Continued 2)

L1 Cache Behavior

Name	Description
`PAPI_L1_DCA`	number of L1 data cache accesses
`PAPI_L1_DCH`	number of L1 data cache hits
`PAPI_L1_DCM`	number of L1 data cache misses
`PAPI_L1_DCR`	number of L1 data cache reads
`PAPI_L1_DCW`	number of L1 data cache writes
`PAPI_L1_ICA`	number of L1 instruction cache accesses
`PAPI_L1_ICH`	number of L1 instruction cache hits
`PAPI_L1_ICM`	number of L1 instruction cache misses
`PAPI_L1_ICR`	number of L1 instruction cache reads
`PAPI_L1_ICW`	number of L1 instruction cache writes
`PAPI_L1_LDM`	number of L1 load misses
`PAPI_L1_STM`	number of L1 store misses
`PAPI_L1_TCA`	number of L1 cache accesses (in total)
`PAPI_L1_TCH`	number of L1 cache hits (in total)
`PAPI_L1_TCM`	number of L1 cache misses (in total)
`PAPI_L1_TCR`	number of L1 cache reads (in total)
`PAPI_L1_TCW`	number of L1 cache writes (in total)

L2 and L3 Cache Behavior

Name	Description
PAPI_L2_LDM	number of L2 load misses
PAPI_L2_STM	number of L2 store misses
PAPI_L2_TCA	number of L2 cache accesses (in total)
PAPI_L2_TCH	number of L2 cache hits (in total)
PAPI_L2_TCM	number of L2 cache misses (in total)
PAPI_L2_TCR	number of L2 cache reads (in total)
PAPI_L2_TCW	number of L2 cache writes (in total)
PAPI_L3_LDM	number of L3 load misses
PAPI_L3_STM	number of L3 store misses
PAPI_L3_TCA	number of L3 cache accesses (in total)
PAPI_L3_TCH	number of L3 cache hits (in total)
PAPI_L3_TCM	number of L3 cache misses (in total)
PAPI_L3_TCR	number of L3 cache reads (in total)
PAPI_L3_TCW	number of L3 cache writes (in total)

Event Usage Examples

- most frequently used events are often those related to cache behavior
- instructions per cycle could be computed from events:
  - PAPI_TOT_CYC and PAPI_TOT_INS
- L1 cache data miss rate could be computed from events:
  - PAPI_L1_DCM and PAPI_L1_DCA; or
  - PAPI_L1_DCM and PAPI_L1_DCH; or
  - PAPI_L1_DCM, PAPI_LD_INS, and PAPI_SR_INS
- L2 cache (total) miss rate could be computed from events:
  - PAPI_L2_TCM and PAPI_L2_TCA; or
  - PAPI_L2_TCM and PAPI_L2_TCH; or
  - PAPI_L2_TCM, PAPI_LD_INS, and PAPI_SR_INS

Code Example Using PAPI High-Level Interface

```
#include <iostream>
#include <papi.h>

void do_work() {for (volatile auto i = 1'000'000; i > 0; --i) {}}

int main() {
    constexpr int num_events = 2;
    int events[num_events] = {PAPI_TOT_INS, PAPI_TOT_CYC};
    long long values[num_events];
    if (PAPI_start_counters(events, num_events) != PAPI_OK)
      {std::cerr << "cannot start counters\n"; return 1;}
    do_work();
    if (PAPI_stop_counters(values, num_events) != PAPI_OK)
      {std::cerr << "cannot stop counters\n"; return 1;}
    for (auto i : values) {std::cout << i << '\n';}
}
```

PAPI Low-Level Interface

- several dozen functions available in low-level API
- provides increased efficiency and functionality
- can obtain more detailed information about hardware
- low-level interface works with event sets
- **event set**: set of events to be monitored
- some low-level API functions listed on next slide
- low-level interface functions do not initialize library (so `PAPI_library_init` must be called explicitly)

Some Functions in PAPI Low-Level Interface

Function	Description
`PAPI_library_init`	initialize PAPI library
`PAPI_shutdown`	cleanup PAPI library
`PAPI_create_eventset`	create event set
`PAPI_destroy_eventset`	destroys empty event set
`PAPI_cleanup_eventset`	removes all events from event set
`PAPI_add_event`	add preset or native hardware event to event set
`PAPI_add_events`	add multiple preset or native hardware events to event set
`PAPI_start`	start counting hardware events in event set
`PAPI_read`	read hardware counters from event set
`PAPI_reset`	reset hardware event counts in event set
`PAPI_accum`	adds hardware counters from event set to elements in array and resets counters
`PAPI_stop`	stop counting hardware events in event set

Code Example Using PAPI Low-Level Interface

```
#include <iostream>
#include <papi.h>

void do_work() {for (volatile auto i = 1'000'000; i > 0; --i) {}}

int main() {
    constexpr int num_events = 2;
    int event_set = PAPI_NULL;
    int events[num_events] = {PAPI_TOT_INS, PAPI_TOT_CYC};
    long long values[num_events];
    if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT)
      {std::cerr << "cannot initialize\n"; return 1;}
    if (PAPI_create_eventset(&event_set) != PAPI_OK)
      {std::cerr << "cannot create event set\n"; return 1;}
    if (PAPI_add_events(event_set, events, num_events) != PAPI_OK)
      {std::cerr << "cannot add events\n"; return 1;}
    if (PAPI_start(event_set) != PAPI_OK)
      {std::cerr << "cannot start\n"; return 1;}
    do_work();
    if (PAPI_stop(event_set, values) != PAPI_OK)
      {std::cerr << "cannot stop\n"; return 1;}
    if (PAPI_cleanup_eventset(event_set) != PAPI_OK)
      {std::cerr << "cannot cleanup event set\n"; return 1;}
    if (PAPI_destroy_eventset(&event_set) != PAPI_OK)
      {std::cerr << "cannot destroy event set\n"; return 1;}
    for (auto i : values) {std::cout << i << '\n';}
}
```

PAPI Utilities

Name	Description
papi_avail	provides availability and detail information for PAPI pre-set events
papi_clockres	measures and reports clock latency and resolution for PAPI timers
papi_cost	computes execution time costs for basic PAPI operations
papi_command_line	executes PAPI preset or native events from command line
papi_decode	provides availability and detail information for PAPI pre-set events
papi_event_chooser	given list of named events, lists other events that can be counted with them
papi_mem_info	provides information on memory architecture of current processor
papi_native_avail	provides detailed information for PAPI native events
papi_version	provides version information for PAPI

Example `papi_avail` Output

```
Available events and hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 5.3.2.0
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel(R) Core(TM) i7-3820QM CPU @ 2.70GHz (58)
CPU Revision             : 9.000000
CPUID Info               : Family: 6  Model: 58  Stepping: 9
CPU Max Megahertz        : 3700
CPU Min Megahertz        : 1200
Hdw Threads per core     : 2
Cores per Socket         : 4
Sockets                  : 1
NUMA Nodes               : 1
CPUs per Node            : 8
Total CPUs               : 8
Running in a VM          : no
Number Hardware Counters : 11
Max Multiplex Counters   : 64
--------------------------------------------------------------------------------

    Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
PAPI_L3_DCM  0x80000004  No    No   Level 3 data cache misses
PAPI_L3_ICM  0x80000005  No    No   Level 3 instruction cache misses
PAPI_L1_TCM  0x80000006  No    Yes  Level 1 cache misses
PAPI_L2_TCM  0x80000007  Yes   No   Level 2 cache misses
PAPI_L3_TCM  0x80000008  Yes   No   Level 3 cache misses
[99 lines deleted]
-------------------------------------------------------------------------
Of 108 possible events, 43 are available, of which 14 are derived.

avail.c                                     PASSED
```

Example papi_mem_info Output

```
Memory Cache and TLB Hierarchy Information.
------------------------------------------------------------------------
TLB Information.
  There may be multiple descriptors for each level of TLB
  if multiple page sizes are supported.

L1 Data TLB:
  Page Size:              4 KB
  Number of Entries:     64
  Associativity:          4

[other TLB information deleted]

Cache Information.

L1 Data Cache:
  Total size:            32 KB
  Line size:             64 B
  Number of Lines:      512
  Associativity:          8

L1 Instruction Cache:
  Total size:            32 KB
  Line size:             64 B
  Number of Lines:      512
  Associativity:          8

L2 Unified Cache:
  Total size:           256 KB
  Line size:             64 B
  Number of Lines:     4096
  Associativity:          8

[information for L3 Unified Cache deleted]

mem_info.c                              PASSED
```

Example papi_native_avail Output

```
Available native events and hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 5.3.2.0
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel(R) Core(TM) i7-3820QM CPU @ 2.70GHz (58)
CPU Revision             : 9.000000
CPUID Info               : Family: 6  Model: 58  Stepping: 9
CPU Max Megahertz        : 3700
CPU Min Megahertz        : 1200
Hdw Threads per core     : 2
Cores per Socket         : 4
Sockets                  : 1
NUMA Nodes               : 1
CPUs per Node            : 8
Total CPUs               : 8
Running in a VM          : no
Number Hardware Counters : 11
Max Multiplex Counters   : 64
--------------------------------------------------------------------------------

===============================================================================
 Native Events in Component: perf_event
===============================================================================
[lines deleted]
--------------------------------------------------------------------------------
| perf::L1-DCACHE-LOADS                                                        |
|           L1 cache load accesses                                             |
--------------------------------------------------------------------------------
[lines deleted]
===============================================================================
 Native Events in Component: coretemp
===============================================================================
| coretemp:::hwmon0:temp1_input                                                |
|           degrees C, acpitz module, label ?                                  |
--------------------------------------------------------------------------------
[lines deleted]

Total events reported: 322
native_avail.c                          PASSED
```

Section 8.2.1

References

1. B. Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, July 2002.
2. P. Mucci, Performance Monitoring with PAPI, Dr. Dobb's Journal, June 2005. Available online at `http://www.drdobbs.com/tools/performance-monitoring-with-papi/184406109`.

Section 8.3

Gprof

Gprof

- open-source tool for code-execution profiling
- can be used to collect statistics from program run, including:
  - amount of time spent in each function
  - how many times each function called
  - callers and callees of each function (i.e., call graph information)
- based on compiler instrumentation of code and sampling
- works with GCC and Clang compilers
- instrumentation added to code gathers function call information used to generate call graphs and function call counts
- timing of code execution accomplished by statistical sampling at run time
- program counter probed at regular intervals by interrupting program
- typical sampling period 100 or 1000 samples/second

Comments on Gprof

- since sampling is statistical process, timing measurements not exact (i.e., only statistical approximation)
- if *too few samples taken* (e.g., in case of short-running program), timing measurements very inaccurate
- overhead caused by instrumentation can be quite high (about 30% to 260%)
- *overhead distorts timing measurements* (e.g., instrumentation added to code changes code timing) so timing of code with and without profiling can potentially be quite different
- may not correctly handle multi-threaded applications
- cannot profile shared libraries
- cannot measure time spent in kernel mode (e.g., system calls); only user-space code profiled
- has difficulties with call graphs containing non-trivial cycles (e.g., mutual recursion)

The gprof Command

- command line interface has following form:

```
gprof [options] [executable_file] [profile_file...]
```

- `executable_file` defaults to `a.out`
- `profile_file` defaults to `gmon.out`
- some common options include:

Option	Description
`-b`	omit explanations of meaning of all fields in output
`-I` *dirs*	add directories *dirs* to search path for source files
`-p`	show flat profile
`-q`	show call graph
`-h`	print help information and exit
`-s`	summarize information in profile data files and write to `gmon.sum`

Using Gprof

- compile and link program with `-pg` option; for example:

```
g++ -c -pg -g -O example_1.cpp
g++ -c -pg -g -O example_2.cpp
g++ -pg -g -O -o example example_1.o example_2.o
```

- run program which will produce profiling data file `gmon.out`; for example:

```
example
```

- run `gprof` to analyze profiling data; for example:

```
gprof example
```

- several gmon files can be combined with `gprof -s` to accumulate data over several runs of program
- `gprof2dot` can be used to convert call graph to graphical form

Gprof Output

- output can be generated in following forms:
  - flat profile
  - call graph
- flat profile reports:
  - how much of total execution time spent in each function
  - how many times each function called
  - output sorted by percentage
- call graph reports:
  - for each function, which functions called it, which other functions it called, and how many times
  - estimate of how much time spent in subroutines of each function
- flat profile useful to identify most expensive functions
- call graph useful to identify places where function calls could be eliminated

Example: Source Code

```
#include <algorithm>

constexpr int M = 1024;
constexpr int N = 1024;
constexpr int P = 1024;

// c += a * b
void naive_matmul(const double a[][N], const double b[][P],
  double c[][P]) {
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < N; ++j) {
            for (int k = 0; k < P; ++k)
              {c[i][j] += a[i][k] * b[k][j];}
        }
    }
}

// c += a * b
void improved_matmul(const double a[][N], const double b[][P],
  double c[][P]) {
    for (int i = 0; i < M; ++i) {
        for (int k = 0; k < P; ++k) {
            for (int j = 0; j < N; ++j)
              {c[i][j] += a[i][k] * b[k][j];}
        }
    }
}
```

Example: Source Code (Continued)

```
int main(int argc, char** argv) {
    static double a[M][N];
    static double b[N][P];
    static double c0[M][P];
    static double c1[M][P];
    std::fill_n(&a[0][0], M * N, 1.0);
    std::fill_n(&b[0][0], N * P, 1.0);
    std::fill_n(&c0[0][0], M * P, 0.0);
    naive_matmul(a, b, c0);
    std::fill_n(&a[0][0], M * N, 1.0);
    std::fill_n(&b[0][0], N * P, 1.0);
    std::fill_n(&c1[0][0], M * P, 0.0);
    improved_matmul(a, b, c1);
}
```

Flat Profile Information

- each row in table corresponds to function
- columns in table have following meanings:
  - % time: percentage of total running time of program used by this function
  - cumulative seconds: running sum of number of seconds accounted for by this function and those listed above it
  - self seconds: number of seconds accounted for by this function alone (i.e., excluding descendants)
  - calls: number of times this function was invoked if function is profiled, blank otherwise
  - self ms/call: average number of milliseconds spent in this function per call (excluding descendants) if function is profiled, blank otherwise
  - total ms/call: average number of milliseconds spent in this function and its descendants per call if function is profiled, blank otherwise
  - name: name of function
- entries in table sorted first by self seconds and then by function name

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 89.48      7.55     7.55        1     7.55     7.55  naive_matmul(
                                                        double const (*) [1024],
                                                        double const (*) [1024],
                                                        double (*) [1024])
 11.23      8.50     0.95        1     0.95     0.95  improved_matmul(
                                                        double const (*) [1024],
                                                        double const (*) [1024],
                                                        double (*) [1024])
  0.12      8.51     0.01                             main
  0.00      8.51     0.00        1     0.00     0.00
                              _GLOBAL__sub_I__Z12naive_matmulPA1024_KdS1_PA1024_d
```

Call Graph Information

- table describes call graph of program
- one multi-line entry in table per function, with each entry containing information for function and its callers and callees
- line with index number in left margin lists current function
- lines above current function list its callers
- lines below current function list its callees
- for current function, fields have following meanings:
  - index: unique integer index for this function
  - % time: percentage of total time spent in this function and its children
  - self: total amount of time spent in this function
  - children: total amount of time propagated into this function by its children
  - called: number of times this function called nonrecursively (possibly followed by “+” and number of recursive calls)
  - name: name of this function (with index printed after it)

Call Graph Information (Continued)

- for each parent of current function, fields have following meanings:
  - self: amount of time propagated directly from function into this parent
  - children: amount of time propagated from function's children into this parent
  - called: number of times parent called function, followed by "`/`", followed by total number of times function called
  - name: name of this parent (with its index printed after name)
- if parents of current function cannot be determined, "`<spontaneous>`" is printed in name field
- for each child of current function, fields have following meanings:
  - self: amount of time propagated directly from child to current function
  - children: amount of time propagated from child's chidlren to current function
  - called: number of times current function called child, followed by "`/`", followed by total number of times child called
  - name: name of function (followed by its index)

Example: Call Graph

```
                Call graph


granularity: each sample hit covers 2 byte(s) for 0.12% of 8.51 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.01    8.50                 main [1]
                7.55    0.00       1/1           naive_matmul(double const (*) [1024],
                                                   double const (*) [1024],
                                                   double (*) [1024]) [2]
                0.95    0.00       1/1           improved_matmul(double const (*) [1024],
                                                   double const (*) [1024],
                                                   double (*) [1024]) [3]
-----------------------------------------------
                7.55    0.00       1/1           main [1]
[2]     88.7    7.55    0.00       1         naive_matmul(double const (*) [1024],
                                               double const (*) [1024],
                                               double (*) [1024]) [2]
-----------------------------------------------
                0.95    0.00       1/1           main [1]
[3]     11.1    0.95    0.00       1         improved_matmul(double const (*) [1024],
                                               double const (*) [1024],
                                               double (*) [1024]) [3]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [16]
[10]     0.0    0.00    0.00       1
                               _GLOBAL__sub_I__Z12naive_matmulPA1024_KdS1_PA1024_d [10]
-----------------------------------------------

Index by function name

  [10] _GLOBAL__sub_I__Z12naive_matmulPA1024_KdS1_PA1024_d (matmul_array.cpp) [3]
    improved_matmul(double const (*) [1024], double const (*) [1024], double (*) [1024])
   [2] naive_matmul(double const (*) [1024], double const (*) [1024], double (*) [1024])
   [1] main
```

Section 8.3.1

References

1. S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler.
ACM SIGPLAN Notices, 17(6):120–126, June 1982.

2. S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler.
ACM SIGPLAN Notices, 39(4):49–57, Apr. 2004.

Section 8.4

Valgrind/Callgrind

Valgrind

- suite of simulation-based debugging and profiling tools
- open source
- simulates CPU in software
- web site: `http://valgrind.org`

Callgrind

- collects function call graph information and measures number of instructions executed and cache behavior for program
- does not measure execution time per se; but provides sufficient information to make clock cycle estimates (as is done in KCachegrind)
- can be used to determine cache hit/miss counts and miss rate on program wide, per function, and per source-code line basis
- simulates L1 instruction/data cache and L2 cache
- parameters for each cache can be specified (i.e., size, associativity, and line size) but default to values taken from machine's cache
- simplistic cache model only approximates real cache
- handles code in shared libraries
- typically 15 to 100 times slower (depending on whether cache and branch simulation enabled)
- does not fully support IEEE 754 (floating-point arithmetic standard)
- consequently, code that uses floating-point arithmetic in particular ways may not behave correctly

Support for Floating-Point Arithmetic in Valgrind

- unfortunately, Valgrind does not fully support IEEE 754 floating-point standard (at time of this writing, at least)
- lacks support for floating-point exceptions
- lacks full support for floating-point rounding modes (e.g., some instructions always use round to nearest)
- consequently, code that relies on control over rounding mode will not behave correctly (e.g., code using interval arithmetic)
- code using CGAL library, for example, is often problematic, due to heavy use of interval arithmetic (which needs rounding-mode control) for efficient exact geometric predicates
- does not support extended floating-point formats used by some architectures
- consequently, can sometimes obtain less accurate results from floating-point arithmetic (which in extreme cases might cause numerical instability)

The `valgrind` Command with Callgrind Tool

- command line interface has following form:

```
valgrind [options] program [program_options]
```

- to use Callgrind tool, must specify `--tool=callgrind` option
- some tool-independent options include:

Option	Description
`--help`	print help information and exit
`--log-file=`*file*	set file for log information to *file*

- some Callgrind-specific options include:

Option	Description
`--callgrind-out-file=`*file*	sets output file to *file*; defaults to `callgrind.out-$pid` where `$pid` is process ID
`--cache-sim=`*b*	specifies if information on cache use should be collected, where *b* is `yes` or `no`
`--branch-sim=`*b*	specifies if branching information should be collected, where *b* is `yes` or `no`

The `callgrind_annotate` Command

- command line interface has following form:

  `callgrind_annotate [options] $callgrind_out_file`

- some options include:

Option	Description
`--help`	print help information and exit
`--auto=`b	specifies if all source files should be annotated, where b is `yes` or `no`

Using Callgrind

- build code as one would normally (no compile-time instrumentation is needed); for example:

```
g++ -g -O -o array_sum array_sum.cpp
```

- run program using `valgrind` with Callgrind tool; for example:

```
valgrind --tool=callgrind --cache-sim=yes \
  --log-file=callgrind.log \
  --callgrind-out-file=callgrind.out \
  ./array_sum
```

- display results with `callgrind_annotate`; for example:

```
callgrind_annotate --auto=yes callgrind.out
```

- alternatively, display results in graphical form with tool like KCachegrind (discussed later)

Example: Source Code

```
#include <iostream>
#include <algorithm>

constexpr int M = 2048;
constexpr int N = 2048;

double naive_sum(const double a[][N]) {
    double sum = 0.0;
    for (int j = 0; j < N; ++j) {
        for (int i = 0; i < M; ++i)
          {sum += a[i][j];}
    }
    return sum;
}

double improved_sum(const double a[][N]) {
    double sum = 0.0;
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < N; ++j)
          {sum += a[i][j];}
    }
    return sum;
}
```

Example: Source Code (Continued)

```
int main() {
    static double a[M][N];
    std::fill_n(&a[0][0], M * N, 1.0 / (M * N));
    std::cout << naive_sum(a) << '\n';
    static double b[M][N];
    std::fill_n(&b[0][0], M * N, 1.0 / (M * N));
    std::cout << improved_sum(b) << '\n';
}
```

Example: Callgrind

```
$ valgrind --tool=callgrind --cache-sim=yes --branch-sim=yes --log-file=callgrind.log --
      callgrind-out-file=callgrind.out ./array_sum
$ cat callgrind.log
==23469== Callgrind, a call-graph generating cache profiler
==23469== Copyright (C) 2002-2013, and GNU GPL'd, by Josef Weidendorfer et al.
==23469== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==23469== Command: ./array_sum
==23469== Parent PID: 23449
==23469==
--23469-- warning: L3 cache found, using its data for the LL simulation.
==23469== For interactive control, run 'callgrind_control -h'.
==23469==
==23469== Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw Bc Bcm Bi Bim
==23469== Collected : 70339139 9142838 8663373 1601 4738282 1051026 1585 4728422 1050172
      17247597 30398 4923 423
==23469==
==23469== I   refs:      70,339,139
==23469== I1  misses:         1,601
==23469== LLi misses:         1,585
==23469== I1  miss rate:        0.0%
==23469== LLi miss rate:        0.0%
==23469==
==23469== D   refs:      17,806,211  ( 9,142,838 rd + 8,663,373 wr)
==23469== D1  misses:     5,789,308  ( 4,738,282 rd + 1,051,026 wr)
==23469== LLd misses:     5,778,594  ( 4,728,422 rd + 1,050,172 wr)
==23469== D1  miss rate:       32.5% (      51.8%   +      12.1%  )
==23469== LLd miss rate:       32.4% (      51.7%   +      12.1%  )
==23469==
==23469== LL refs:        5,790,909  ( 4,739,883 rd + 1,051,026 wr)
==23469== LL misses:      5,780,179  ( 4,730,007 rd + 1,050,172 wr)
==23469== LL miss rate:         6.5% (       5.9%   +      12.1%  )
==23469==
==23469== Branches:      17,252,520  (17,247,597 cond +     4,923 ind)
==23469== Mispredicts:       30,821  (    30,398 cond +       423 ind)
==23469== Mispred rate:         0.1% (       0.1%     +       8.5%   )
```

Example: `callgrind_annotate`

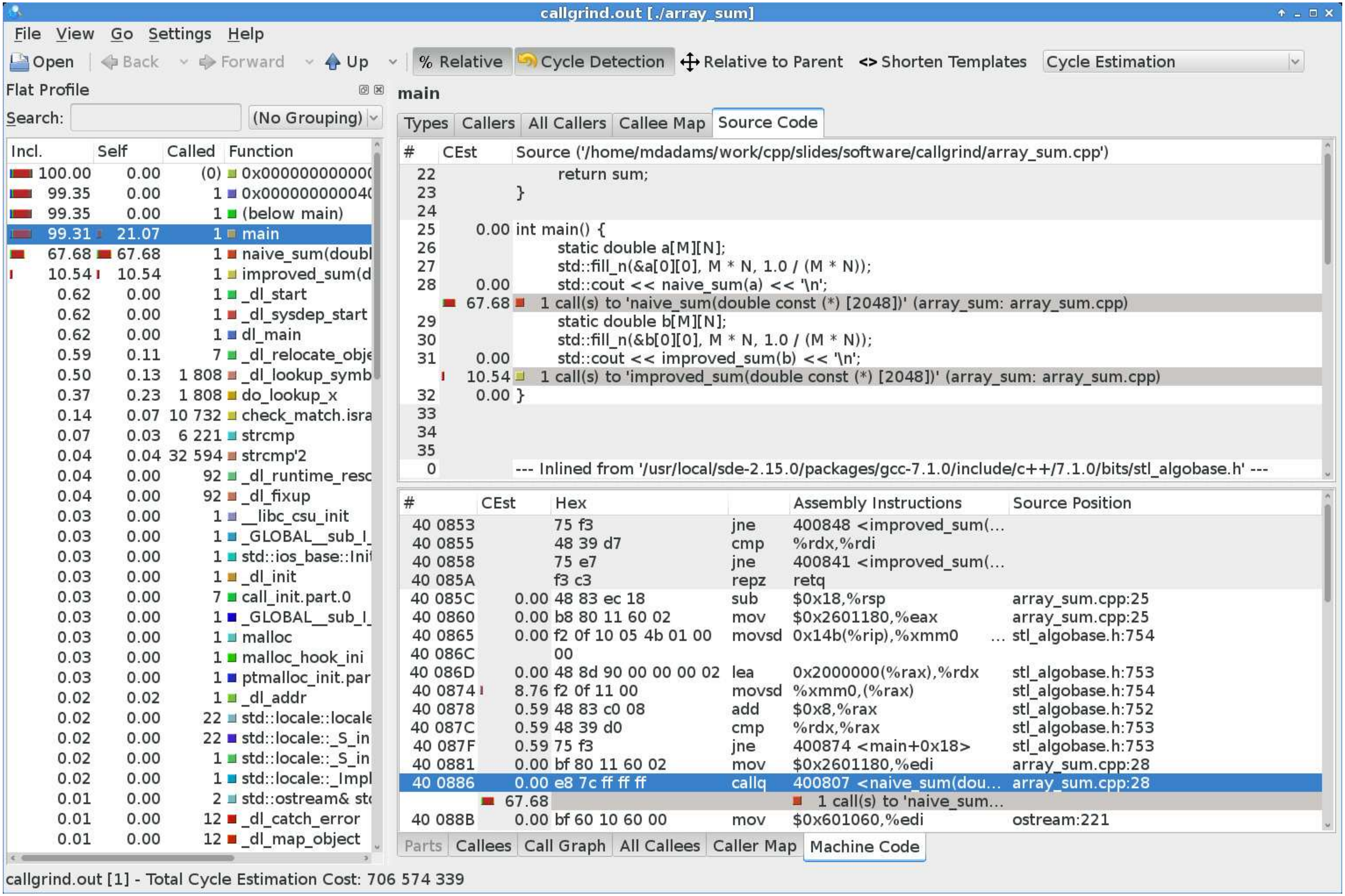
```
$ callgrind_annotate callgrind.out array_sum.cpp

[text deleted]
--------------------------------------------------------------------------------
-- User-annotated source: array_sum.cpp
--------------------------------------------------------------------------------
       Ir        Dr Dw I1mr      D1mr D1mw ILmr      DLmr DLmw        Bc   Bcm Bi Bim

        .         .  .    .         .    .    .         .    .         .     .  .   .  #include <iostream>
        .         .  .    .         .    .    .         .    .         .     .  .   .  #include <algorithm>
        .         .  .    .         .    .    .         .    .         .     .  .   .
        .         .  .    .         .    .    .         .    .         .     .  .   .  constexpr int M = 2048;
        .         .  .    .         .    .    .         .    .         .     .  .   .  constexpr int N = 2048;
        .         .  .    .         .    .    .         .    .         .     .  .   .
        1         0  0    1         0    0    1         .    .         .     .  .   .  double naive_sum(const double a[][N]) {
    4,097         .  .    .         .    .    .         .    .         .     .  .   .   double sum = 0.0;
    4,096         0  0    0         0    0    0         0    0     2,048     3  .   .   for (int j = 0; j < N; ++j) {
8,390,656         0  0    0         0    0    0         0    0 4,194,304 2,063  .   .       for (int i = 0; i < M; ++i)
8,388,608 4,194,304  0    0 4,194,304    0    0 4,194,304    .         .     .  .   .         {sum += a[i][j];}
        .         .  .    .         .    .    .         .    .         .     .  .   .   }
        .         .  .    .         .    .    .         .    .         .     .  .   .   return sum;
        1         1  0    0         1    0    0         1    .         .     .  .   .  }
        .         .  .    .         .    .    .         .    .         .     .  .   .
        1         .  .    .         .    .    .         .    .         .     .  .   .  double improved_sum(const double a[][N]) {
    2,049         .  .    .         .    .    .         .    .         .     .  .   .   double sum = 0.0;
    4,096         0  0    0         0    0    0         0    0     2,048     3  .   .   for (int i = 0; i < M; ++i) {
8,388,608         0  0    0         0    0    0         0    0 4,194,304 2,058  .   .       for (int j = 0; j < N; ++j)
8,388,608 4,194,304  0    0   524,288    0    0   524,288    .         .     .  .   .         {sum += a[i][j];}
        .         .  .    .         .    .    .         .    .         .     .  .   .   }
        .         .  .    .         .    .    .         .    .         .     .  .   .   return sum;
        1         1  0    0         1    0    0         1    .         .     .  .   .  }
        .         .  .    .         .    .    .         .    .         .     .  .   .
        2         0  0    1         0    0    1         .    .         .     .  .   .  int main() {
        .         .  .    .         .    .    .         .    .         .     .  .   .   static double a[M][N];
        .         .  .    .         .    .    .         .    .         .     .  .   .   std::fill_n(&a[0][0], M * N, 1.0 / (M * N));
        2         0  1    0         0    1    0         0    1         .     .  .   .   std::cout << naive_sum(a) << '\n';
16,787,459 4,194,305  0    1 4,194,305    0    1 4,194,305    0 4,196,352 2,066  .   .  => array_sum.cpp:naive_sum(double const (*) [2048]) (1x)
        .         .  .    .         .    .    .         .    .         .     .  .   .   static double b[M][N];
        .         .  .    .         .    .    .         .    .         .     .  .   .   std::fill_n(&b[0][0], M * N, 1.0 / (M * N));
        2         0  1    0         0    1    0         0    1         .     .  .   .   std::cout << improved_sum(b) << '\n';
16,783,363 4,194,305  0    0   524,289    0    0   524,289    0 4,196,352 2,061  .   .  => array_sum.cpp:improved_sum(double const (*) [2048]) (1x)
        6         2  0    1         2    0    1         1    .         .     .  .   .  }

--------------------------------------------------------------------------------
Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw Bc Bcm Bi Bim
--------------------------------------------------------------------------------
48 92  0    0  100    0    0  100    0 49  14  0   0  percentage of events annotated
```

- open-source call-graph profiling-data visualization tool
- part of K Desktop Environment (KDE)
- supports Callgrind data
- allows graphical visualization of:
  - call-graph relationships between functions (e.g., callers and callees)
  - function costs/counts
  - annotated source/assembly with costs/counts
- allows much easier intepretation of Callgrind data (relative to `callgrind_annotate`)
- to allow annotation of assembly, add `--dump-instr=yes` option to `valgrind` command for Callgrind
- use command of form:

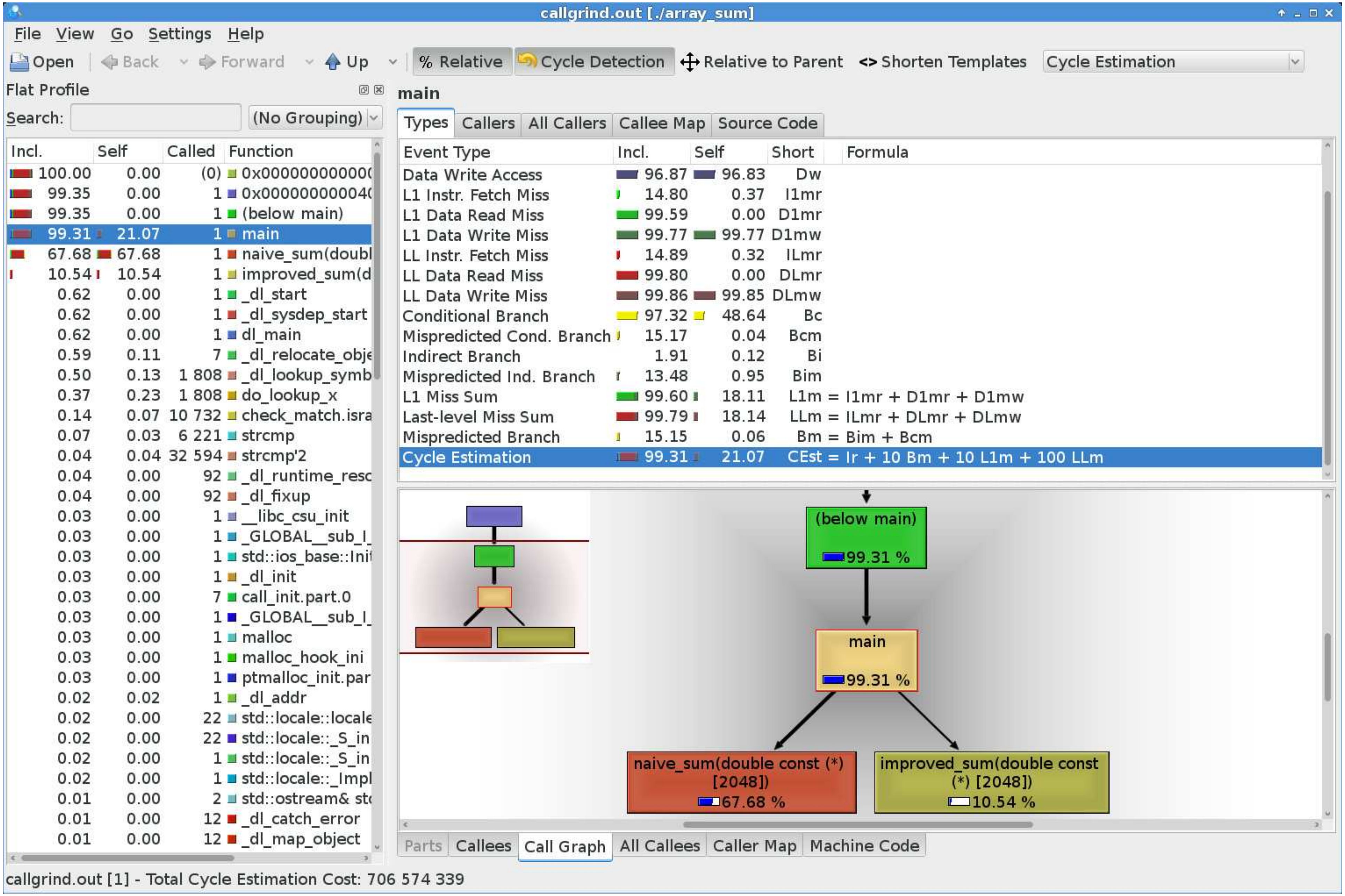
```
kcachegrind $callgrind_out_file
```

- web site: `https://kcachegrind.github.io`

KCachegrind Example: Source/Assembly Annotation



KCachegrind Example



Section 8.4.1

References

References I

1. P. Floyd. Valgrind part 4 — cachegrind and callgrind. *Overload*, 111:4–7, Oct. 2012.

Part 9

Build Tools

Section 9.1

Build Tools

Build Tools

- Build tools are programs that automate the creation of executable programs, libraries, and other artifacts from source code.
- Build tools also typically provide some basic facilities for testing and packaging the artifacts generated by the build process.
- Building software requires careful tracking of:
  - what items need to be built, and
  - the dependencies between these items.
- Dependency tracking is necessary to:
  - determine the order in which items must be built, and
  - minimize the number of items that need to be re-built when a change is made to the code.
- In the case of very small projects, it may be feasible to perform the build process manually.
- For larger projects, however, the build process is far too complex to manage by hand, and build tools are therefore needed.

Examples of Build Tools

- Some examples of build tools include:
  - CMake (a cross-platform tool)
  - GNU Build System (also known as Autotools) (for Unix)
  - Make (for Unix)
  - MSBuild (for Microsoft Visual Studio under Microsoft Windows)
  - Xcodebuild (for Apple Xcode)

Section 9.2

Make

Make

- `make` command
- controls generation of executables and/or other non-source files from program's source files
- extremely popular tool for automating build process
- available on many platforms (e.g., Unix, Microsoft Windows, Mac OS X); used extensively on Unix systems
- very flexible
- can handle building multiple programs consisting of hundreds of source files or single program consisting of only one source file
- can be used to build almost anything (i.e., need not be a program)
- for example, all materials for this course typeset using LaTeX (e.g., coursepack, slides, handouts, exams), and make utility used to compile LaTeX source code into PDF documents
- one of most popular implementations of make is GNU Make
- GNU Make web site: `http://www.gnu.org/software/make`

- target is something that can be built, typically (but not necessarily) file such as executable file or object file
- `make` command driven by data file called makefile
- makefile usually named `Makefile` or `makefile`
- command-line usage:

  `make` [*options*] [*targets*]
- *targets*: zero or more targets to be built
- *options*: zero or more options
- by default, looks for makefile called `makefile` and then `Makefile`
- if no targets are specified, will build first target specified in makefile
- only builds files that are out of date
- most common command-line options include:

`-n`	show commands that would be executed but do not actually execute them
`-f` *makefile*	use makefile *makefile*

- comment starts at hash character (i.e., "#") and continues until end of line; example:

```
# This comment continues until the end of the line.
```

- supports variables
- some important variables used by built-in rules:

Name	Description
CXX	C++ compiler command
CXXFLAGS	C++ compiler options
LDFLAGS	linker options

- to assign value to variable, use equal sign; example:

```
CXX = g++
```

- to substitute value of variable, use dollar sign followed by variable name in parentheses; example:

```
$(CXX)
```

- makefile specifies targets and rules for building targets
- each rule in makefile has following form:

$$\begin{array}{l} \textit{targets} \; : \; \textit{prerequisites} \\ \sqcup\!\sqcup\, \textit{commands} \\ \sqcup\!\sqcup\, \ldots \end{array}$$

- indentation shown above must be with tab character and not spaces
- *targets*: list of one or more targets
- *prerequisites*: files on which targets depend (i.e., files used to produce targets)
- *commands*: actions that must be carried out to produce target from its prerequisites

- normally, each target associated with file of same name (and building target will create this file)
- phony target: target that is not associated with any file
- to identify target as phony make it prerequisite of special target called "**.PHONY**"; example (specify `all` as phony target):

```
.PHONY: all
```

- some special built-in variables that can be used in rules:

Name	Description
`$@`	target
`$<`	name of first prerequisite
`$^`	names of all of prerequisites separated by spaces

```
CXX = g++           # The C++ compiler command.
CXXFLAGS = -g -O # The C++ compiler options.
LDFLAGS =           # The linker options (if any).

# The all target builds all of the programs handled by
# the makefile.
# This target has the dependency chain:
#    all -> hello -> hello.o -> hello.cpp
all: hello

# The clean target removes all of the executable files
# and object files produced by the build process.
clean:
␣␣␣␣rm -f hello *.o

# The hello target builds the hello executable.
hello: hello.o
␣␣␣␣$(CXX) $(CXXFLAGS) -o $@ $^ $(LDFLAGS)

# Indicate that the all and clean targets do not
# correspond to actual files.
.PHONY: all clean

# The following rule is effectively built into make and
# therefore need not be explicitly specified:
# hello.o: hello.cpp
# ␣$(CXX) $(CXXFLAGS) -c $<
```

Commentary on Makefile for `hello` Program

- `all` target: builds all of the programs handled by the makefile (e.g., `hello`)
- `clean` target: removes all of the executable files and object files produced by build process (e.g., `hello`, `hello.o`)
- although `all` and `clean` have no special meaning to make, very common practice to provide targets with these particular names in all makefiles
- `hello` target: compiles and links the hello program
- chain of dependencies for `all` target:

  `all` → `hello` → `hello.o` → `hello.cpp`
- `all` and `clean` examples of phony targets

Section 9.2.1

References

References

1. S. I. Feldman. Make — a program for maintaining computer programs. *Software: Practice and Experience*, 9(4):255–265, Apr. 1979.

Section 9.3

CMake

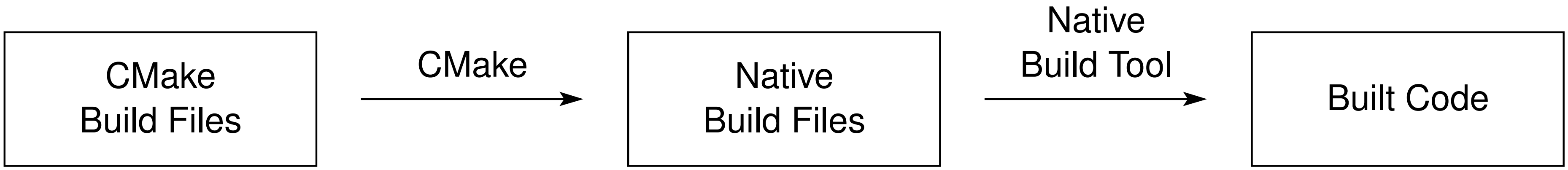
CMake

- CMake is open-source cross-platform family of tools designed to build, test, and package software
- controls software build process (e.g., compiling and linking) using simple platform- and compiler-independent configuration files
- used in conjunction with native build environments
- generates files appropriate for whatever build environment being used
- supports native build environments such as Unix Make, Apple Xcode, and Microsoft Visual Studio
- automatically generates dependency information for source files
- supports parallel builds
- created by Kitware (`http://www.kitware.com`)
- web site: `https://cmake.org`

Users of CMake

- CMake has very large user base and is employed in many open-source and commercial projects
- some users of CMake include:
  - Blender (`https://www.blender.org`)
  - Clang (`http://clang.llvm.org`)
  - Computational Geometry Algorithms Library (CGAL) (`http://www.cgal.org`)
  - JasPer Image Processing/Coding Tool Kit (`http://www.ece.uvic.ca/~mdadams/jasper`)
  - K Desktop Environment (KDE) (`https://www.kde.org`)
  - MySQL (`https://www.mysql.com`)
  - Netflix (`https://www.netflix.com`)
  - OpenCV (`http://opencv.org`)
  - Qt (`https://www.qt.io`)
  - Second Life (`http://secondlife.com`)

Build Process



- **CMake build files**: files used by CMake to describe build process for software project (i.e., `CMakeLists.txt` and other build files it references)
- **native build tool**: program used to build code for particular software development environment being employed (e.g., `make` for Unix, `MSBuild` for Microsoft Visual Studio, and `xcodebuild` for Apple Xcode)
- **native build files**: files used by native build tool to control build process (e.g., makefiles for Unix, project/solution files for Microsoft Visual Studio, and project files for Apple Xcode)
- build process consists of two steps:
  1. CMake used, with CMake build files as input, to produce native build files
  2. native build tool invoked to build code using native build files generated by CMake
- strictly speaking, CMake does not itself build code, but rather produces build files that can be used by native build tool to build code

- **source directory**: top-level directory of source tree for code to be built
- **binary directory**: directory under which all files generated by build process will be placed
- source directory must contain `CMakeLists.txt` file which is used to describe build process
- **cache**: file where CMake stores values of variables used for configuration of build process (i.e., `CMakeCache.txt` in binary directory)
- **build-system generator**: entity within CMake that produces native build files (i.e., build files targetting particular native build tool)
- CMake provides numerous generators (e.g., generators for Unix Make, Apple Xcode, and Microsoft Visual Studio)
- **build configuration**: description of build to be performed with particular set of parameters (e.g., optimized or debug version)
- some generators support multiple configurations using single build
- for generators that support only single configuration, need to specify which configuration to build

In-Source Versus Out-of-Source Builds

- **in-source build**: when binary directory chosen to be inside source tree (e.g., same as source directory)
- **out-of-source build**: when binary directory chosen to be outside source tree
- when out-of-source build used, contents of source directory not modified in any way by build process
- in contrast, when in-source build used, build process can generate many new files under source directory
- out-of-source builds have numerous advantages over in-source builds; in particular, out-of-source builds:
  - avoid cluttering source tree with many files generated by build process, which can cause numerous difficulties (e.g., interacting poorly with version control systems)
  - facilitate easy removal of all files generated by build process without risk of accidently removing source files
  - allow for multiple builds from single source tree (e.g., debug and release builds)
- for above reasons, in-source builds should generally be avoided

- To generate build files for a native build tool, use a command of the form:

```
cmake [options] [$source_directory]
```

- The binary directory is assumed to be the current directory.
- The source directory `$source_dir` defaults to the current directory (resulting in an in-source build).
- Some options include:

Option	Description
`-D` *var=val*	Set the CMake variable *var* to value *val*.
`-G` *gen*	Set the build-system generator to *gen*.
`--version`	Print name/version banner and exit.
`--help`	Print usage information and exit.
`--debug-output`	Enable debugging output.
`--trace`	Enable tracing output.

The cmake Command (Continued 1)

- Some supported generators include:

Name	Description
`Unix␣Makefiles`	makefiles for Unix Make
`Xcode`	project files for Apple Xcode
`Visual␣Studio␣12␣2013`	project files for Microsoft Visual Studio 12

- The makefile generator produces a top-level makefile with standard targets (e.g., all, clean, and install) as well as targets corresponding to each CMake target in the project.
- Some environment variables used by `cmake` include:

Option	Description
`CC`	The command for compiling C source.
`CXX`	The command for compiling C++ source.

- Although undocumented (and ***not officially supported***), the source and binary directories can both be specified on the command line by using the options "`-H`*source_dir*" and "`-B`*binary_dir*". (There cannot be a space between the option letter and its corrresponding argument.)

- To invoke the native build tool in a platform-independent manner for the build files in the binary directory `$binary_directory`, use a command of the form:

```
cmake --build $binary_directory [$options]
```

- Some options include:

Option	Description
`--target` *target*	Build the target *target* instead of the default targets.
`--config` *config*	For a multi-configuration generator, select the build configuration *config*. For a single-configuration generator, this option is ignored.
`--clean-first`	Build the "clean" target first.
`--`	Pass the remaining options to the native build tool.

Hello World Example

- source directory `$SOURCE_DIR` contains two files:

  `CMakeLists.txt` and `hello.cpp`

- commands to build with binary directory `$BINARY_DIR`:

```
cd $BINARY_DIR
cmake $SOURCE_DIR
cmake --build .
```

`$SOURCE_DIR/hello.cpp`

```cpp
#include <iostream>
int main() {std::cout << "Hello, World!\n";}
```

`$SOURCE_DIR/CMakeLists.txt`

```cmake
# Specify minimum required version of CMake.
cmake_minimum_required(VERSION 3.10 FATAL_ERROR)

# Specify project and identify languages used.
project(hello LANGUAGES CXX)

# Add program target called hello.
add_executable(hello hello.cpp)
```

Section 9.3.1

CMakeLists Files

Projects, Targets, and Build Configurations

- **project**: collection of source code to be built using CMake
- **target**: something to be built by build process, such as executable or library
- target typically associated with one or more source files
- target has numerous properties (e.g., compiler flags and linker flags)
- target names cannot contain whitespace
- by default, following build configurations are supported:

Name	Description
`Debug`	basic debugging code/information enabled
`Release`	basic optimization enabled
`RelWithDebInfo`	optimized build with debugging code/information enabled as well
`MinSizeRel`	smallest (but not necessarily fastest) code

Comments and Commands

- comment starts with hash character (i.e., "#") and continues until end of line
- file consists of sequence of commands
- command consists of following (in order):
  1. command name
  2. opening parenthesis
  3. whitespace-separated arguments
  4. closing parenthesis
- command example:

```
cmake_minimum_required(VERSION 3.1)
```

- command names are case insensitive
- anything in double quotes treated as single argument; for example, as in:

```
message("Hello World")
```

- backslash character can be used to escape character such as double quote; for example, as in:

```
message("\${X} is not a variable expansion")
```

Variables

- variable name is sequence of one or more letters, digits, and underscore characters that does not begin with digit (e.g., `MATH_LIBRARY`, `i`)
- variable names are case sensitive
- value of variable can be treated as string or list of strings
- value of variable `X` is accessed as `${X}`
- boolean tests are case insensitive
- all of following considered false: `OFF`, `0`, `NO`, `FALSE`, `NOTFOUND`, `*-NOTFOUND`, `IGNORE`
- variable can be internal or cache
- cache variable persists across separate invocations of `cmake` while internal variable does not
- internal variable take precedence over cache varaible

Modules

- module is file containing re-usable piece of CMakeLists code
- normally use "`.cmake`" file name extension
- most modules can be classified into one of following categories:
  - find
  - system introspection
  - utility
- find module:
  - determines location of software elements such as header files and libraries
  - often module name starts with prefix "`Find`"
  - examples: `FindBoost` and `FindOpenGL`
- system introspection module:
  - provides information about target system or compiler (e.g., size of various types, availability of header files, compiler version)
  - often module name starts with prefix "`Test`" or "`Check`"
  - examples: `CheckCXXSourceCompiles` and `CheckIncludeFile`
- utility module:
  - provides additional functions/macros for convenience
  - example: `ExternalProject`

Modules (Continued 1)

- module can be accessed via `include` command
- find module normally accessed via `find_package` command (instead of directly using `include` command)

Commonly-Used Variables

- Source and binary directories:
  - CMAKE_BINARY_DIR. The full path to the top-level directory of the current CMake build tree. For an in-source build, this is the same as CMAKE_SOURCE_DIR.
  - CMAKE_SOURCE_DIR. The full path to the top-level directory of the current CMake source tree. For an in-source build, this is the same as CMAKE_BINARY_DIR.
  - CMAKE_CURRENT_SOURCE_DIR. The full path to the source directory that is currently being processed by cmake.
  - CMAKE_CURRENT_BINARY_DIR. The full path to the binary directory that is currently being processed by cmake.
- Build type:
  - CMAKE_BUILD_TYPE. In the case of single-configuration generators, specifies the build type (e.g., Release, Debug, RelWithDebInfo, MinSizeRel). In the case of multi-configuration generators, unused.
  - BUILD_SHARED_LIBS. Specifies if all libraries created should default to shared (instead of static).
  - BUILD_TESTING. Specifies if testing is enabled (when the CTest module is used).

Commonly-Used Variables (Continued 1)

- C++ compiler:
  - CMAKE_CXX_COMPILER_ID. The C++ compiler in use (e.g., Clang, GNU, Intel, MSVC).
  - CMAKE_CXX_STANDARD. Used to initialize the CXX_STANDARD property on all targets, which selects version of C++ standard (e.g., 98, 11, and 14).
  - CMAKE_CXX_STANDARD_REQUIRED. Used to initialize the CXX_STANDARD_REQUIRED property of all targets. This property determines whether the specified version of C++ standard is required.
  - CMAKE_CXX_COMPILER. The compiler command used for C++ source code.
  - CMAKE_CXX_FLAGS. The compiler flags for compiling C++ source code.
  - CMAKE_CXX_FLAGS_DEBUG. The compiler flags for compiling C++ source code for a debug build.
  - CMAKE_CXX_FLAGS_RELEASE. The compiler flags for compiling C++ source code for a release build.
  - CMAKE_CXX_FLAGS_RELWITHDEBINFO. The compiler flags for compiling C++ source code for a release build with debug flags.
  - CMAKE_CXX_FLAGS_MINSIZEREL. The compiler flags for compiling C++ source code for a release build with minimum code size.

- Linker:
  - CMAKE_EXE_LINKER_FLAGS. The linker flags used to create executables. This variable also has configuration-specific variants, such as CMAKE_EXE_LINKER_FLAGS_RELEASE.
  - CMAKE_SHARED_LINKER_FLAGS. The linker flags used to create shared libraries. This variable also has configuration-specific variants, such as CMAKE_SHARED_LINKER_FLAGS_RELEASE.
  - CMAKE_STATIC_LINKER_FLAGS. The linker flags used to create static libraries. This variable also has configuration-specific variants, such as CMAKE_STATIC_LINKER_FLAGS_RELEASE.
- Target OS:
  - CMAKE_SYSTEM_NAME. The name of the target system's OS (e.g., Linux, Windows, Darwin).
  - UNIX. Specifies if the target system's OS is UNIX (or UNIX-like).
  - APPLE. Specifies if the target system's OS is Mac OS X.
  - WIN32. Specifies if the target system's OS is Microsoft Windows (32- or 64-bit).

Commonly-Used Variables (Continued 3)

- Makefile builds:
  - `CMAKE_VERBOSE_MAKEFILE`. Enable verbose output from Makefile builds.
  - `CMAKE_RULE_MESSAGES`. Specify if a progress message should be reported by each makefile rule.
- Other:
  - `CMAKE_MODULE_PATH`. The list of directories to search for CMake modules. (This is used by commands like `include` and `find_package`.)
  - `CMAKE_PREFIX_PATH`. The list of directories specifying installation prefixes to be searched by the `find_package`, `find_program`, `find_library`, and `find_file` commands.
  - `CMAKE_PROJECT_NAME`. The name of the current project.
  - `CMAKE_CURRENT_LIST_DIR`. The directory of the listfile currently being processed. (The values of `CMAKE_CURRENT_SOURCE_DIR` and `CMAKE_CURRENT_LIST_DIR` can differ, for example, when a listfile outside the current source directory is included.)

Commonly-Used Commands

- Initialization:
  - `cmake_minimum_required`. Set the minimum required version of Cmake for a project.
  - `cmake_policy`. Manage CMake policy settings. (This is used to select between old and new behaviors in CMake.)
  - `project`. Set a name, version, and enable languages for the entire project. (If no languages specified, defaults to C and C++.)
  - `option`. Provide an option that the user can (optionally) select.
- Adding targets:
  - `add_executable`. Add a program target.
  - `add_library`. Add a library target.
  - `add_test`. Add a test target. (This is used in conjunction with the module `CTest`.)
  - `add_custom_target`. Add a target with no output file that is always out of date.
  - `add_custom_command`. Add a custom build rule to the generated build system.

Commonly-Used Commands (Continued 1)

- Setting properties for a specific target:
  - `target_compile_definitions`. Add compile definitions to a target.
  - `target_compile_options`. Add compile options to a target
  - `target_include_directories`. Add include directories to a target.
  - `target_link_libraries`. Add libraries to the list of libraries to be used for linking a target. (May be used multiple times for the same target.)
  - `set_target_properties`. Set properties for a target. (Some properies include: `OUTPUT_NAME`, `SOVERSION`, and `VERSION`.)

Commonly-Used Commands (Continued 2)

- Setting properties for all targets:
  - `add_compile_options`. Adds options to the compilation of source files in the current directory and below. (This command should precede an `add_executable` or `add_library` command.)
  - `add_definitions`. Adds `-D` define flags to the compilation of source files in the current directory and below.
  - `include_directories`. Add directories to the list of include directories used for compiling programs.
  - `link_libraries`. Add libraries to the list of libraries used for linking programs. (This command appends to the list, each time it is invoked.)
  - `link_directories`. Specify directories in which the linker is to look for libraries.
- Processing other files or directories:
  - `add_subdirectory`. Add a subdirectory to the build.
  - `include`. Load and run CMake code from a file or module.

Commonly-Used Commands (Continued 3)

- Querying external packages and programs:
  - `find_package`. Load settings for an external software package (e.g., Doxygen, Threads, Boost, OpenGL, GLEW, GLUT, CGAL, PkgConfig).
  - `find_library`. Find an external library.
  - `find_program`. Find an external program.
- Assignment, control flow, functions, and macros:
  - `set`. Set a CMake, cache, or environment variable to a given value.
  - `if`, `elseif`, `else`, and `endif`. Conditionally execute a group of commands.
  - `foreach` and `endforeach`. Evaluate a group of commands for each value in a list.
  - `while` and `endwhile`. Evaluate a group of commands while a condition is true.
  - `function` and `endfunction`. Record a function for later invocation as a command.
  - `macro` and `endmacro`. Record a macro for later invocation as a command.

Commonly-Used Commands (Continued 4)

- String and list processing:
  - `list`. Perform operations on lists.
  - `string`. Perform operations on strings.
- Other:
  - `message`. Display a message to the user.
  - `configure_file`. Copy a file to another location and modify its contents.
  - `install`. Specify rules to run at install time (e.g., rules to install programs, libraries, and header files).
  - `math`. Evaluate mathematical expressions.
  - `file`. Manipulate files.
  - `enable_language`. Enable a language.

Commonly-Used Modules

- `CheckIncludeFiles` module, which provides:
  - `check_include_files`. Check if the specified files can be included.
- `CheckCXXSourceCompiles` module, which provides:
  - `check_cxx_source_compiles`. Check if the specified C++ source code compiles and links to produce an executable.
- `CheckFunctionExists` module, which provides:
  - `check_function_exists`. Check if the specified C function is provided by libraries on the system.
- `CTest` module:
  - Configure a project for testing with CTest/CDash.
- `CPack` module:
  - Configure a project to use CPack to build binary and source package installers.
- `PkgConfig` module, which requires `pkg-config` tool to be available and provides:
  - `pkg_search_module`. Finds a package via `pkg-config`.

Commonly-Used Modules (Continued 1)

- `ExternalProject` module, which provides:
  - `externalproject_add`. Create custom targets to build projects in external trees.
- `GNUInstallDirs` module:
  - Define GNU standard installation directories (e.g., `CMAKE_INSTALL_INCLUDEDIR`, `CMAKE_INSTALL_LIBDIR`, and `CMAKE_INSTALL_MANDIR`).
- `GenerateExportHeader` module, which provides:
  - `generate_export_header`. Generate a header file containing export macros to be used for a shared library.
- `CMakePackageConfigHelpers` module, which provides:
  - `configure_package_config_file`. Create a package configuration file for installing a project or library. (This should be used instead of `configure_file`.)
  - `write_basic_package_version_file`. Write a package version file.

Some Find and Pkg-Config Modules

- Boost
  - `https://cmake.org/cmake/help/v3.10/module/FindBoost.html`
  - variables: `Boost_FOUND`, `Boost_INCLUDE_DIRS`, `Boost_LIBRARY_DIRS`, `Boost_LIBRARIES`
  - imported targets: `Boost::boost`, `Boost::`*component*
- CGAL (Computational Geometry Algorithms Library)
  - variables: `CGAL_INCLUDE_DIRS`, `CGAL_LIBRARY`, `GMP_LIBRARIES`
- Doxygen
  - `https://cmake.org/cmake/help/v3.10/module/FindDoxygen.html`
  - variables: `DOXYGEN_FOUND`, `DOXYGEN_EXECUTABLE`
  - imported targets: `Doxygen::doxygen`, `Doxygen::dot`
- GLEW (OpenGL Extension Wrangler Library)
  - `https://cmake.org/cmake/help/v3.10/module/FindGLEW.html`
  - variables: `GLEW_FOUND`, `GLEW_INCLUDE_DIRS`, `GLEW_LIBRARIES`
  - imported targets: `GLEW::GLEW`
- GLFW (OpenGL Helper Library) [pkg-config module]
  - variables: `GLFW_FOUND`, `GLFW_INCLUDE_DIRS`, `GLFW_LIBRARIES`

Some Find and Pkg-Config Modules (Continued 1)

- GLUT (OpenGL Utility Toolkit)
  - `https://cmake.org/cmake/help/v3.10/module/FindGLUT.html`
  - variables: `GLUT_FOUND`, `GLUT_INCLUDE_DIR`, `GLUT_LIBRARIES`
  - imported targets: `GLUT::GLUT`
- OpenGL (Open Graphics Library)
  - `https://cmake.org/cmake/help/v3.10/module/FindOpenGL.html`
  - variables: `OPENGL_FOUND`, `OPENGL_INCLUDE_DIR`, `OPENGL_LIBRARIES`
  - imported targets: `OpenGL::GL`, `OpenGL::GLU`, `OpenGL::GLX`
- SPL (Signal/Geometry Processing Library)
  - variables: `SPL_FOUND`, `SPL_INCLUDE_DIRS`, `SPL_LIBRARY_DIRS`, `SPL_LIBRARIES`, `SNDFILE_INCLUDE_DIRS`, `SNDFILE_LIBRARIES`
- Threads
  - `https://cmake.org/cmake/help/v3.10/module/FindThreads.html`
  - variables: `CMAKE_THREAD_LIBS_INIT`
  - imported targets: `Threads::Threads`

Using Per-Target Versus Global Settings

- can set compiler options, compiler definitions, include directories, and link libraries in two ways:
  1. per target (e.g., using `target_compile_options`, `target_compile_definitions`, `target_include_directories`, and `target_link_libraries`)
  2. globally (e.g., using `add_compile_options`, `add_definitions`, `include_directories`, and `link_libraries`)
- per-target approach allows properties to be specified with finer granularity than global approach
- finer-granularity control over properties often necessary, especially when building more complex projects
- if executable targets in project do not all use same set of libraries, global specification of include directories and link libraries can introduce artificial dependencies on some libraries
- per-target specification of link libraries allows automatic propagation of library dependencies when hierarchies of libraries used (which, for example, may avoid need to link against same library multiple times)

Section 9.3.2

Examples

Hello World Example Revisited

hello.cpp

```cpp
#include <iostream>

int main() {std::cout << "Hello, World!\n";}
```

CMakeLists.txt

```cmake
# Specify minimum required version of CMake.
cmake_minimum_required(VERSION 3.10 FATAL_ERROR)

# Specify project and identify languages used.
project(hello LANGUAGES CXX)

# Print message indicating detected OS.
if (UNIX)
    set(platform "Unix")
elseif (WIN32)
    set(platform "Microsoft Windows")
else()
    set(platform "Unknown")
endif()
message("OS is ${platform}")

# Add program target called hello.
add_executable(hello hello.cpp)
```

Test Example

- want to build and test hello-world program
- code written in C++
- files in project:

```
CMakeLists.txt
hello.cpp
test_wrapper.in
run_test
```

- project has:
  - executable target `hello`
  - test target `run_test`

Test Example: Source Code (Including Some Scripts)

hello.cpp

```
#include <iostream>

int main() {std::cout << "Hello, World!\n";}
```

test_wrapper.in (with execute permission set)

```
#! /bin/sh
# Initialize the environment for the command being invoked.
export CMAKE_SOURCE_DIR="@CMAKE_SOURCE_DIR@"
export CMAKE_BINARY_DIR="@CMAKE_BINARY_DIR@"
"$@"
```

run_test

```
#! /bin/sh
# Test if the hello program produces the desired output.
($CMAKE_BINARY_DIR/hello | grep "^Hello, World!$") || \
  exit 1
```

CMakeLists.txt

```
# Specify minimum required version of CMake.
cmake_minimum_required(VERSION 3.10 FATAL_ERROR)

# Specify project and identify languages used.
project(hello LANGUAGES CXX)

# Include the CTest module for testing.
include(CTest)

# Find the Bourne shell.
find_program(sh SH_COMMAND)

# Add program target called hello.
add_executable(hello hello.cpp)

# Create a wrapper script that initializes the environment
# for any test scripts.
configure_file(${CMAKE_SOURCE_DIR}/test_wrapper.in
  ${CMAKE_BINARY_DIR}/test_wrapper @ONLY)

# Add a test that invokes run_test via a wrapper script.
add_test(run_test ${SH_COMMAND}
  ${CMAKE_BINARY_DIR}/test_wrapper
  ${CMAKE_SOURCE_DIR}/run_test)
```

Boost Log Example

- want to build simple program using Boost Log
- code written in C++
- uses Log component of Boost library
- files in project:

```
CMakeLists.txt
main.cpp
```

- project has:
  - executable target `my_app`

Boost Log Example: Source Code

main.cpp

```
#include <boost/log/trivial.hpp>

int main() {
    BOOST_LOG_TRIVIAL(warning)
      << "A warning severity message";
    BOOST_LOG_TRIVIAL(error)
      << "An error severity message";
    BOOST_LOG_TRIVIAL(fatal)
      << "A fatal severity message";
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10 FATAL_ERROR)
project(boost_example LANGUAGES CXX)

# Find the required libraries (i.e., POSIX threads and Boost).
set(Boost_USE_MULTITHREADED ON)
find_package(Threads REQUIRED)
find_package(Boost 1.54.0 REQUIRED COMPONENTS log)

# Define a program target called my_app.
add_executable(my_app main.cpp)

# Set the includes, defines, and libraries for the my_app target.
target_include_directories(my_app PUBLIC ${Boost_INCLUDE_DIRS})
target_compile_definitions(my_app PUBLIC "-DBOOST_LOG_DYN_LINK")
target_link_libraries(my_app ${Boost_LIBRARIES}
  ${CMAKE_THREAD_LIBS_INIT})
```

Boost Log Example: CMakeLists File [With Imported Targets]

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10 FATAL_ERROR)
project(boost_example LANGUAGES CXX)

# Find the required libraries (i.e., POSIX threads and Boost).
set(Boost_USE_MULTITHREADED ON)
find_package(Threads REQUIRED)
find_package(Boost 1.54.0 REQUIRED COMPONENTS log)

# Define a program target called my_app.
add_executable(my_app main.cpp)

# Set the defines, includes, and libraries for the my_app target.
target_compile_definitions(my_app PUBLIC "-DBOOST_LOG_DYN_LINK")
target_link_libraries(my_app Boost::log Threads::Threads)
```

OpenGL/GLFW Example

- want to build simple OpenGL/GLFW application
- code written in C++
- uses OpenGL and GLFW libraries (as well as GLEW library)
- files in project:

```
CMakeLists.txt
trivial.cpp
```

- project has:
  - executable target `trivial`

trivial.cpp

```
#include <cstdlib>
#include <GLFW/glfw3.h>

void display(GLFWwindow* window) {
    glfwMakeContextCurrent(window);
    glClearColor(0.0, 1.0, 1.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glfwSwapBuffers(window);
}

int main(int argc, char** argv) {
    if (!glfwInit()) {return EXIT_FAILURE;}
    glfwSwapInterval(1);
    GLFWwindow* window = glfwCreateWindow(512, 512, argv[0],
      nullptr, nullptr);
    if (!window) {
        glfwTerminate();
        return EXIT_FAILURE;
    }
    glfwSetWindowRefreshCallback(window, display);
    while (!glfwWindowShouldClose(window))
      {glfwWaitEvents();}
    glfwTerminate();
    return EXIT_SUCCESS;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10 FATAL_ERROR)
project(opengl_example LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED TRUE)

# Find the required libraries (i.e., OpenGL, GLEW, and GLFW).
find_package(OpenGL REQUIRED)
find_package(GLEW REQUIRED)
find_package(PkgConfig REQUIRED)
pkg_search_module(GLFW REQUIRED glfw3)

# Define a program target called trivial.
add_executable(trivial trivial.cpp)

# Set the includes and libraries for the trivial target.
target_include_directories(trivial PUBLIC ${GLFW_INCLUDE_DIRS}
  ${GLEW_INCLUDE_DIRS} ${OPENGL_INCLUDE_DIR})
target_link_libraries(trivial ${GLFW_LIBRARIES} ${GLEW_LIBRARIES}
  ${OPENGL_LIBRARIES})
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10 FATAL_ERROR)
project(opengl_example LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED TRUE)

# Find the required libraries (i.e., OpenGL, GLEW, and GLFW).
find_package(OpenGL REQUIRED)
find_package(GLEW REQUIRED)
find_package(PkgConfig REQUIRED)
pkg_search_module(GLFW REQUIRED glfw3)

# Define a program target called trivial.
add_executable(trivial trivial.cpp)

# Set the includes and libraries for the trivial target.
target_include_directories(trivial PUBLIC ${GLFW_INCLUDE_DIRS})
target_link_libraries(trivial ${GLFW_LIBRARIES} GLEW::GLEW
  OpenGL::GL)
```

OpenGL/GLUT Example

- want to build simple OpenGL/GLUT application
- code written in C++
- uses OpenGL and GLUT libraries
- files in project:

```
CMakeLists.txt
trivial.cpp
```

- project has:
  - executable target `trivial`

OpenGL/GLUT Example: Source Code

trivial.cpp

```
#include <GL/glut.h>

void display() {
    glClearColor(0.0, 1.0, 1.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glutSwapBuffers();
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(512, 512);
    glutCreateWindow(argv[0]);
    glutDisplayFunc(display);
    glutMainLoop();
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10 FATAL_ERROR)
project(opengl_example LANGUAGES CXX)

# Find the required libraries (i.e., OpenGL and GLUT).
find_package(OpenGL REQUIRED)
find_package(GLUT REQUIRED)

# Define a program target called trivial.
add_executable(trivial trivial.cpp)

# Set the includes and libraries for the trivial target.
target_include_directories(trivial PUBLIC ${GLUT_INCLUDE_DIR}
  ${OPENGL_INCLUDE_DIR})
target_link_libraries(trivial ${GLUT_LIBRARIES} ${OPENGL_LIBRARIES})
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10 FATAL_ERROR)
project(opengl_example LANGUAGES CXX)

# Find the required libraries (i.e., OpenGL and GLUT).
find_package(OpenGL REQUIRED)
find_package(GLUT REQUIRED)

# Define a program target called trivial.
add_executable(trivial trivial.cpp)

# Set the includes and libraries for the trivial target.
target_link_libraries(trivial GLUT::GLUT OpenGL::GL)
```

CGAL Example

- want to build simple CGAL application
- code written in C++
- uses CGAL library (as well as GMP library)
- files in project:

```
CMakeLists.txt
orient_test.cpp
```

- project has:
  - executable target `orient_test`

CGAL Example: Source Code

orient_test.cpp

```
#include <iostream>
#include <string>
#include <CGAL/Cartesian.h>
#include <CGAL/Filtered_kernel.h>

std::string toString(CGAL::Orientation orient) {
    switch (orient) {
    case CGAL::LEFT_TURN:
        return "left turn";
    case CGAL::RIGHT_TURN:
        return "right turn";
    case CGAL::COLLINEAR:
        return "collinear";
    }
}

int main(int argc, char** argv) {
    using Point = CGAL::Point_2<CGAL::Filtered_kernel<
      CGAL::Cartesian<double>>>;
    Point a, b, q;
    while (std::cin >> a >> b >> q) {
        auto orient = CGAL::orientation(a, b, q);
        std::cout << toString(orient) << '\n';
    }
}
```

CMakeLists.txt

```
# Specify minimum required version of CMake.
cmake_minimum_required(VERSION 3.10 FATAL_ERROR)

# Specify project and enable the C++ language.
project(cgal_demo LANGUAGES CXX)

# Find the required CGAL package.
find_package(CGAL REQUIRED)

# On some systems, GCC may need the -frounding-math option.
if (CMAKE_CXX_COMPILER_ID MATCHES GNU)
    add_compile_options("-frounding-math")
endif()

# Add a program target called orient_test.
add_executable(orient_test orient_test.cpp)

# Specify the includes and libraries for the orient_test target.
target_include_directories(orient_test PUBLIC ${CGAL_INCLUDE_DIRS})
target_link_libraries(orient_test ${CGAL_LIBRARY} ${GMP_LIBRARIES})
```

HG2G Example: Overview

- want to be able to build and install HG2G library and application that uses library
- code written in C++
- files in project:

```
CMakeLists.txt
app/CMakeLists.txt
app/answer.cpp
hg2g/CMakeLists.txt
hg2g/answer.cpp
hg2g/question.cpp
hg2g/include/hg2g/answer.hpp
hg2g/include/hg2g/config.hpp.in
```

- project has:
  - library target `hg2g`
  - executable target `answer`
  - option `HG2G_ZAPHOD` (which takes a boolean value)

HG2G Example: Library Source Code

hg2g/include/hg2g/config.hpp.in

```
#ifndef HG2G_CONFIG_H
#define HG2G_CONFIG_H
#define HG2G_VERSION "@HG2G_VERSION@"
#cmakedefine HG2G_ZAPHOD
#endif
```

hg2g/include/hg2g/answer.hpp

```
#include <string>
namespace hg2g {
std::string answer_to_ultimate_question();
std::string ultimate_question();
}
```

hg2g/answer.cpp

```
#include "hg2g/answer.hpp"
namespace hg2g {
std::string answer_to_ultimate_question() {return "42";}
}
```

hg2g/question.cpp

```
#include "hg2g/answer.hpp"
namespace hg2g {
std::string ultimate_question() {throw 42;}
}
```

HG2G Example: Application Source Code

app/answer.cpp

```
#include <iostream>
#include <hg2g/config.hpp>
#include <hg2g/answer.hpp>

int main() {
#ifdef HG2G_ZAPHOD
    std::cout << "HG2G_ZAPHOD is defined\n";
#endif
    std::cout << "According to version " << HG2G_VERSION <<
      " of the HG2G library:\n";
    std::cout <<
      "The answer to the ultimate question is " <<
      hg2g::answer_to_ultimate_question() << ".\n";
}
```

HG2G Example: CMakeLists Files

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10 FATAL_ERROR)
project(hg2g_example LANGUAGES CXX)
option(HG2G_ZAPHOD "Define HG2G_ZAPHOD" FALSE)

# Set the version number and name.
set(HG2G_VERSION_MAJOR 1)
set(HG2G_VERSION_MINOR 42)
set(HG2G_VERSION_PATCH 0)
string(CONCAT HG2G_VERSION "${HG2G_VERSION_MAJOR}"
  ".${HG2G_VERSION_MINOR}" ".${HG2G_VERSION_PATCH}")

# Process the subdirectories hg2g and app.
add_subdirectory(hg2g)
add_subdirectory(app)
```

app/CMakeLists.txt

```
# Add a program target called answer.
add_executable(answer answer.cpp)

# Link the answer program against the hg2g library.
target_link_libraries(answer hg2g)

# Install the answer program in the bin directory.
install(TARGETS answer DESTINATION bin)
```

hg2g/CMakeLists.txt

```
# Place the names of the header and source files into
# variables (for convenience).
set(hg2g_headers include/hg2g/answer.hpp
  "${CMAKE_CURRENT_BINARY_DIR}/include/hg2g/config.hpp")
set(hg2g_sources answer.cpp question.cpp)

# Add a library target called hg2g.
add_library(hg2g ${hg2g_sources} ${hg2g_headers})

# Specify the include directories for library.
target_include_directories(hg2g PUBLIC
  include
  "${CMAKE_CURRENT_BINARY_DIR}/include")

# Create a header file containing the config information.
configure_file(
  include/hg2g/config.hpp.in
  "${CMAKE_CURRENT_BINARY_DIR}/include/hg2g/config.hpp")

# Install the library in the lib directory.
install(TARGETS hg2g DESTINATION lib)

# Install the header files in the include/hg2g directory.
install(FILES ${hg2g_headers} DESTINATION include/hg2g)
```

External Project Example

hello, hg2g, and example␣100 are subdirectories containing CMake projects

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10 FATAL_ERROR)
# Specify the project and do not enable any languages.
project(examples LANGUAGES CXX)
# Include the module for external project functionality.
include(ExternalProject)
# Create a list of the subdirectories containing
# CMake projects to be built.
list(APPEND dirs hello hg2g "example 100")
# Add each project as an external project.
foreach(dir IN LISTS dirs)
    # Set target name to directory name with any
    # spaces changed to underscores.
    string(REPLACE " " "_" target "${dir}")
    # Add external project.
    externalproject_add("${target}"
      SOURCE_DIR "${CMAKE_SOURCE_DIR}/${dir}"
      BINARY_DIR "${CMAKE_BINARY_DIR}/${dir}"
      CMAKE_ARGS
      "-DCMAKE_CXX_COMPILER=${CMAKE_CXX_COMPILER}"
      INSTALL_COMMAND "")
endforeach()
```

- want to build LaTeX document (i.e., produce PDF document from LaTeX source)
- files in project:

```
CMakeLists.txt
main.tex
bib.bib
cmake_modules/UseLATEX.cmake
```

LaTeX Example: Source Code

main.tex

```
\documentclass{article}
\usepackage{graphicx}
\author{John Doe}
\title{Why I Like C++}
\begin{document}
\maketitle
\section{Why I Like C++}
What can I say?
C++~\cite{TCPL4} is a great language!\newline
\includegraphics[width=1in,height=1in,keepaspectratio]
  {cpp.png}
\bibliographystyle{plain}
\bibliography{bib}
\end{document}
```

bib.bib

```
@book{
    TCPL4,
    author = "B. Stroustrup",
    title = "The {C++} Programming Language",
    edition = "4th",
    publisher = "Addison Wesley",
    year = 2013
}
```

LaTeX Example: CMakeLists File

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10 FATAL_ERROR)

# Specify the project name and indicate that no languages
# should be enabled.
project(my_doc NONE)

# Add the cmake_modules directory to the module search path.
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH}
  ${CMAKE_SOURCE_DIR}/cmake_modules)

# Include the UseLATEX module.
include(UseLATEX)

# Specify the properties of the LaTeX document such as its
# constituent source files (e.g., LaTeX, BibTeX, images,
# figures, etc.)
add_latex_document(main.tex IMAGES cpp.png BIBFILES bib.bib)
```

cmake_modules/UseLATEX.cmake

This file is taken from `https://cmake.org/Wiki/images/8/80/UseLATEX.cmake`.

Section 9.3.3

References

References

1. Ken Martin and Bill Hoffman, Mastering CMake — A Cross-Platform Build System — CMake 3.1, Kitware, 2015. ISBN 978-1-930934-31-3.
2. CMake Tutorial (excerpt from the book "Mastering CMake"), `https://cmake.org/cmake-tutorial`.
3. CMake FAQ, `https://cmake.org/Wiki/CMake_FAQ`.
4. CMake Wiki, `https://cmake.org/Wiki/CMake`.
5. Kenneth Moreland, "UseLATEX.cmake: LaTeX Document Building Made Easy," Version 2.4.0. Available online at `https://cmake.org/Wiki/images/d/d7/UseLATEX.pdf`.
6. CMakeUserUseLATEX `https://cmake.org/Wiki/CMakeUserUseLATEX`
7. UseLATEX GitHub Site `https://github.com/kmorel/UseLATEX`

Talks

1. Bill Hoffman, Google Tech Talk — Building Science With CMake, October 8, 2015. Available online at `https://youtu.be/TqjtN8NGtl4`.

   A very basic introduction to CMake.

2. Daniel Pfeifer, Effective CMake, C++ Now, May 19, 2017, Aspen, CO, USA. Available online at `https://youtu.be/bsXLMQ6WgIk`.

Part 10

Version Control Systems

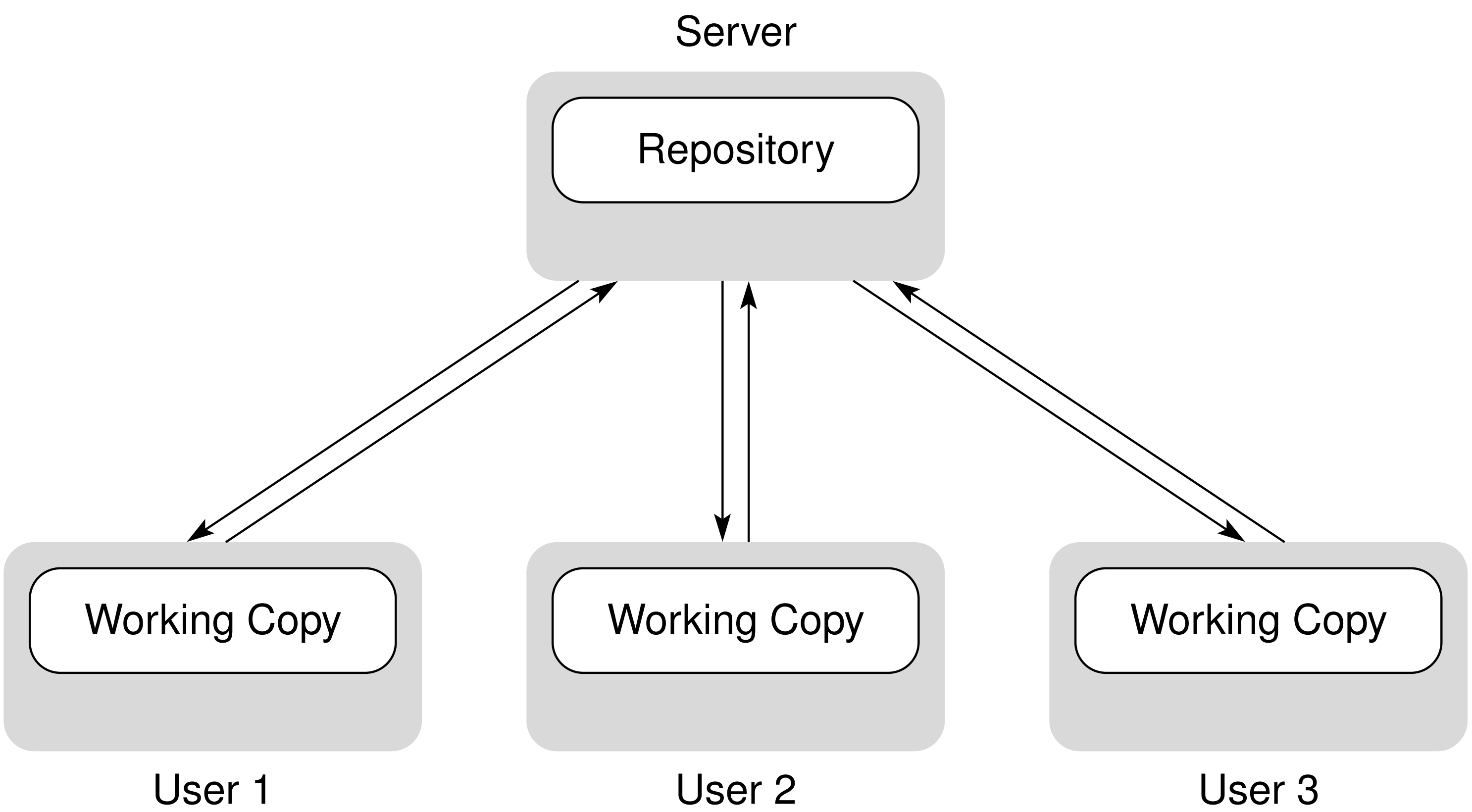
Section 10.1

Version Control Systems

Version Control Systems

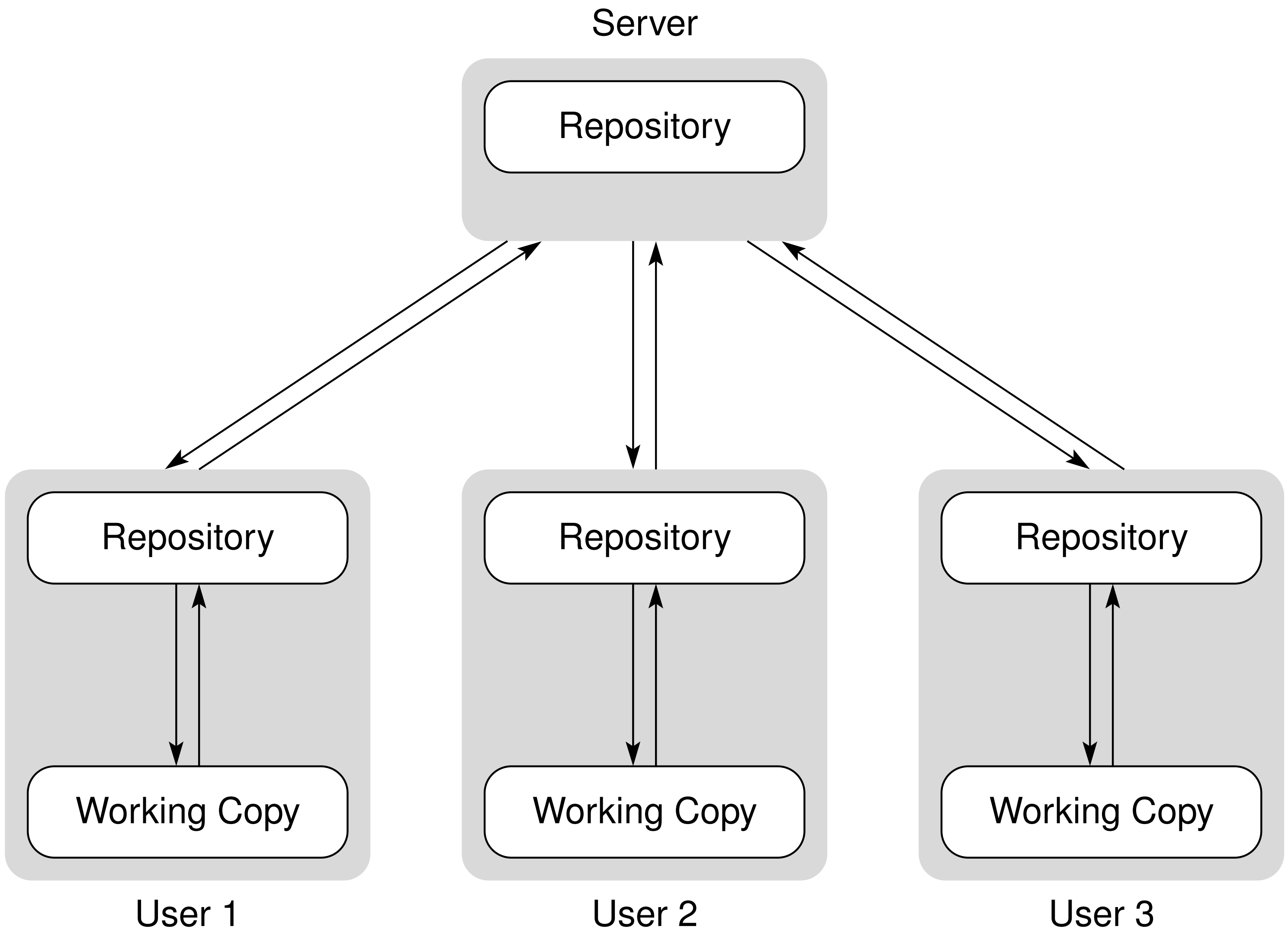
- **Version control** (also known as **revision control**) is the management of changes to programs, documents, and other collections of information.
- In practice, multiple versions of the same software will often be in existence at any given time.
- For the purposes of locating and fixing a bug, it is critically important to be able to access different versions of the software, since only certain versions of the software may have the bug.
- When concurrently developing multiple versions of some software, it is necessary to be able to keep track of what information belongs to which versions.
- Having developers manually maintain version information themselves is impractical, as this would be very error prone.
- Therefore, a version control system (VCS) is used to manage changes in a systematic manner.
- Some examples of VCSes include: Source Code Control System (SCCS), Revision Control System (RCS), Concurrent Versions System (CVS) Subversion, Mercurial, and Git.

Centralized Version Control



- repository resides only on server
- users do not have their own local copy of repository
- examples: CVS and Subversion

Distributed Version Control



- each user has their own local copy of repository
- examples: Git and Mercurial

Pros and Cons of Distributed Version Control

- advantages of distributed (over centralized) version control:
  - most operations (namely, ones that do not synchronize with other repositories) are local and extremely fast
  - all operations, except those that synchronize with other repositories, can be performed without network connection
  - more robust (e.g., data loss less likely due to replication of information across repositories, less reliance on network/server connectivity)
  - committing new changesets can be done locally without anyone else seeing them
  - easier to share changes with only, say, one or two people before showing changes to everyone
- disadvantages of distributed (relative to centralized) version control:
  - since repository is stored locally, more local disk space is required
  - if repository becomes large, downloading it can require considerable amount of time

Section 10.2

Git

- Git is open-source distributed (i.e., decentralized) version control system
- created by Linus Torvalds
- development started in 2005 with first release made later in same year
- designed to support projects varying in size from very small to very large with speed and efficiency
- can efficiently handle very large numbers of files
- can efficiently handle large numbers of parallel branches
- revision history of file modelled as directed acyclic graph (DAG)
- official web site: `https://git-scm.com`

Users of Git

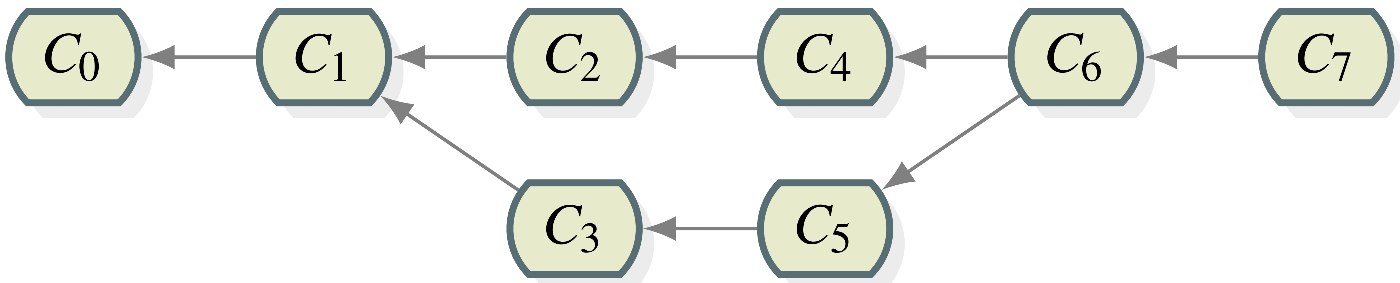
- Git has a very large user base and is employed heavily in industry
- some organizations using Git include:
  - Apple (`https://github.com/apple`)
  - eBay (`https://github.com/eBay`)
  - Facebook (`https://github.com/facebook`)
  - Google (`https://github.com/google`)
  - Intel (`https://github.com/intel`)
  - Microsoft (`https://github.com/Microsoft`)
  - NVIDIA (`https://github.com/NVIDIA`)
  - Twitter (`https://github.com/twitter`)
- some projects using Git include:
  - Linux Kernel (`https://github.com/torvalds/linux`)
  - Android (`https://android-review.googlesource.com`)
  - Qt (`http://code.qt.io`)
  - Gnome (`https://git.gnome.org`)
  - Eclipse (`https://git.eclipse.org`)
  - KDE (`https://github.com/KDE`)
  - FreeDesktop (`https://cgit.freedesktop.org`)

Repositories

- A **repository** is effectively a database that records the information for all of the versions of all of the files in the directory tree under version control.
- A **commit** is simply a record (i.e., snapshot) of all of the files that comprise a particular version in the repository.
- This record includes, for each file, the location of the file in the directory tree as well as the contents of the file.
- For each version of the directory tree, the repository has a corresponding commit (i.e., snapshot).

Revision History and Directed Acyclic Graphs

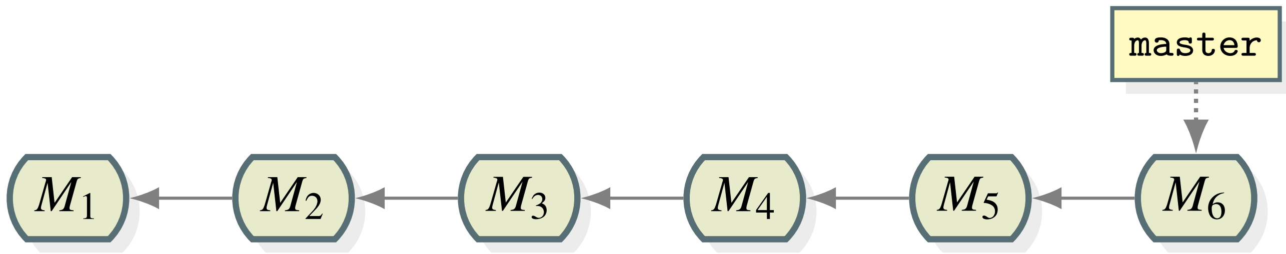
- The revision history can be represented as a directed acyclic graph (DAG).
- Each node in the graph corresponds to a commit in the repository.
- Each edge in the graph points to the immediately preceding commit in the revision history.
- Example of DAG:



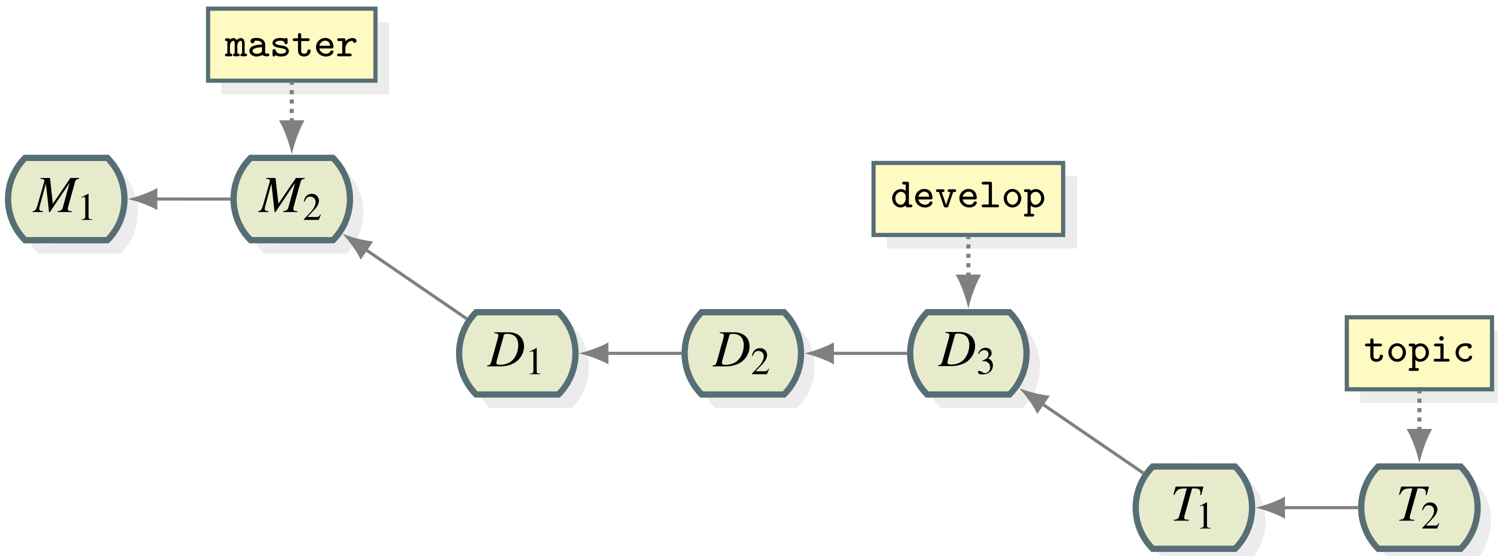
C_1 based on changing content from C_0;
C_2 based on changing content from C_1;
C_4 based on changing content from C_2;
C_6 based on ***merging*** content from C_4 and C_5;

$\ldots$

- single (master) branch:

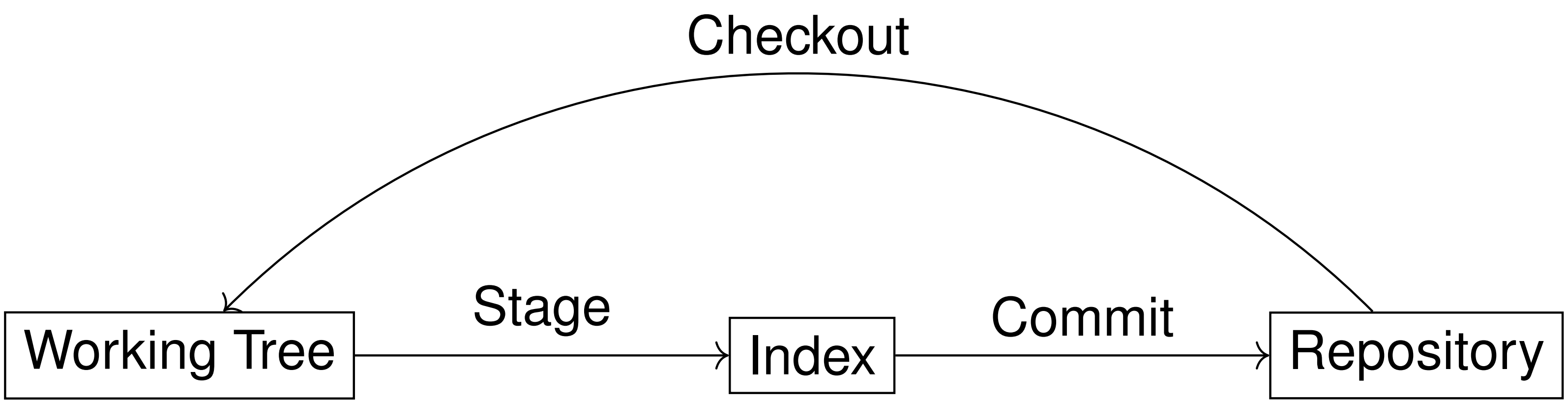


- master, development, and topic branches:



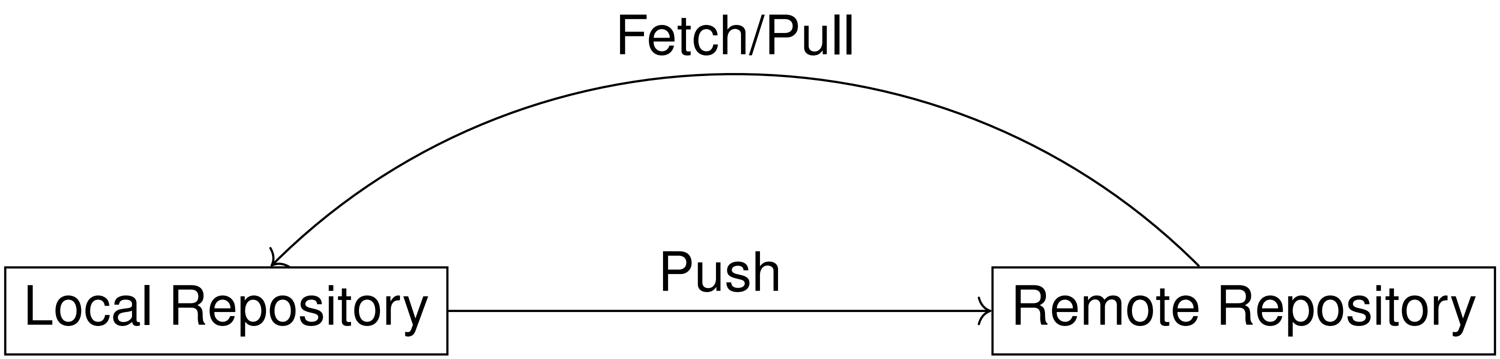
  - master branch: used for releases (highly stable, well tested)
  - development branch: used for development work (possibly unstable)
  - topic branch: used for highly experimental work

Local Picture



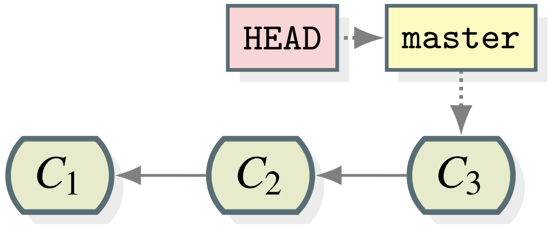
- three distinct types of data:
  - **working tree**: directory hierarchy containing files on which user is working
  - **index** (also known as **staging area**): place where changes that are tentatively marked to be committed are stored
  - **repository**: database used to store all versions of data and associated metadata
- three basic local operations on data:
  - **checking out**: populates working tree with particular version of data from repository
  - **staging**: applies changes in working tree to index
  - **committing**: applies changes in index to repository

Local and Remote Picture



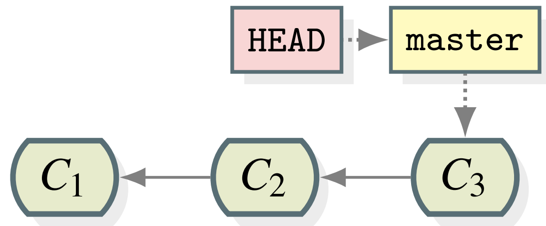
- **clone**: creates local repository that is copy of remote repository
- three basic operations for propagating changes between repositories:
  - **push**: propagate changes from local repository to remote repository
  - **fetch**: propagate changes from remote repository to local repository without merging changes
  - **pull**: propagate changes from remote repository to local repository and merge changes

- The name `HEAD` is a reference to the branch currently in use in the repository.
- Normally (i.e., except in the case of a detached `HEAD`), `HEAD` refers to the current branch.
- Consider a repository having a single `master` branch and three commits C_1, C_2, and C_3 (with C_3 being the most recent), where the current branch is `master`. This would appear as follows:



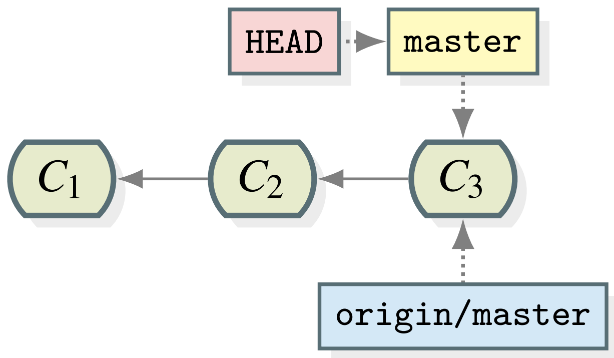
Remote-Tracking Branches

- Consider a remote repository whose commit history is as shown below, with a single branch `master`.



[remote repository]

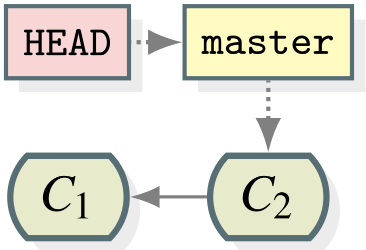
- Cloning the above repository will produce a new local repository whose commit history is as shown below, with a (local) branch `master` and a remote-tracking branch `origin/master`.



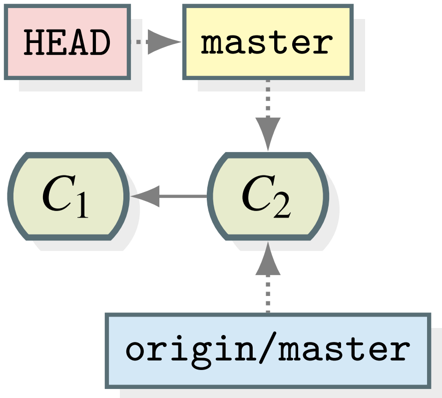
[local repository]

- A branch fetched from a remote repository is called a **remote-tracking branch**.
- A remote-tracking branch is a reference to a commit in the remote repository and is used for operations like pushing and fetching/pulling.

1. Consider the following remote repository with a single branch `master`:

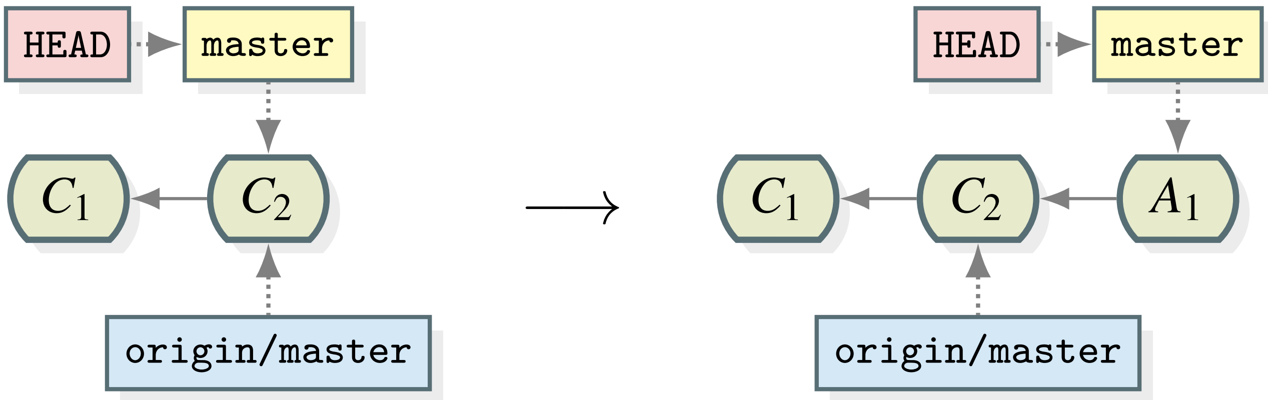


2. Cloning the remote repository yields a new local repository that is identical to the remote repository but with a remote-tracking branch `origin/master` added as follows:

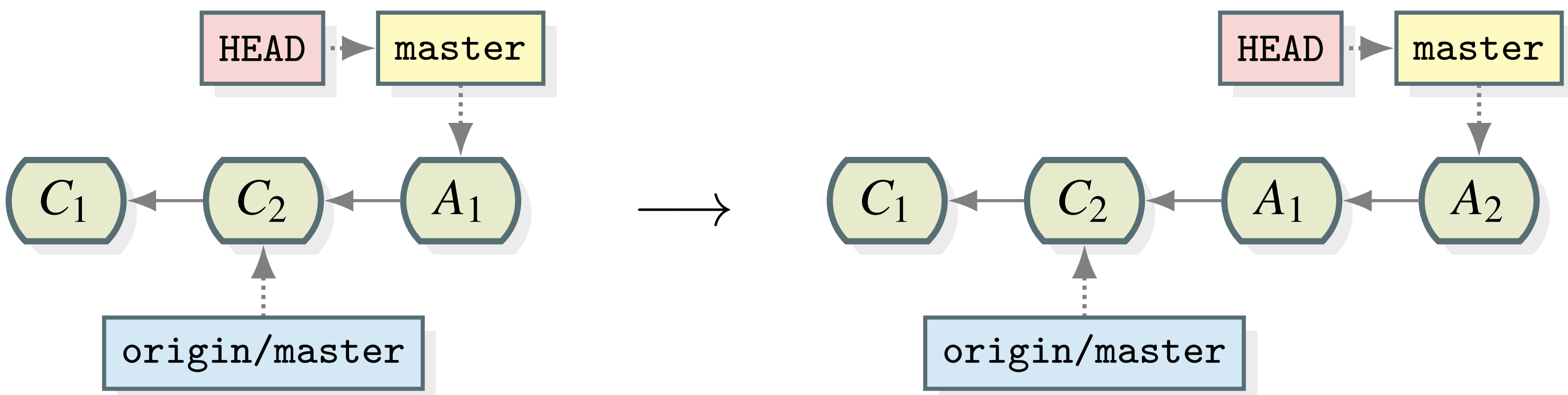


Commit History Example II

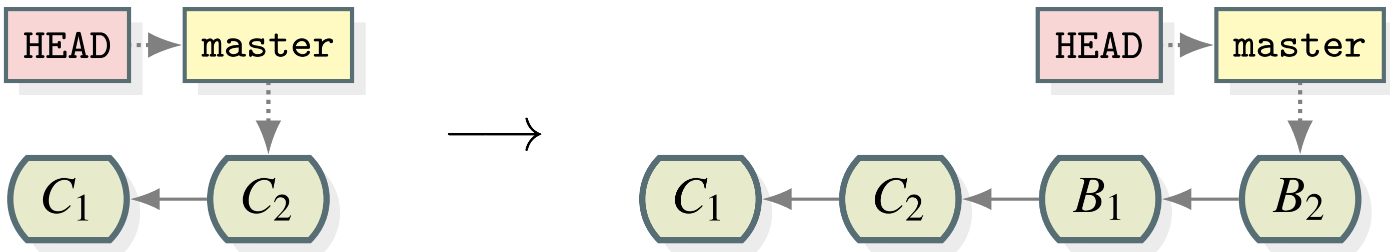
3. Committing change A_1 to the `master` branch of the local repository transforms the local repository as follows:



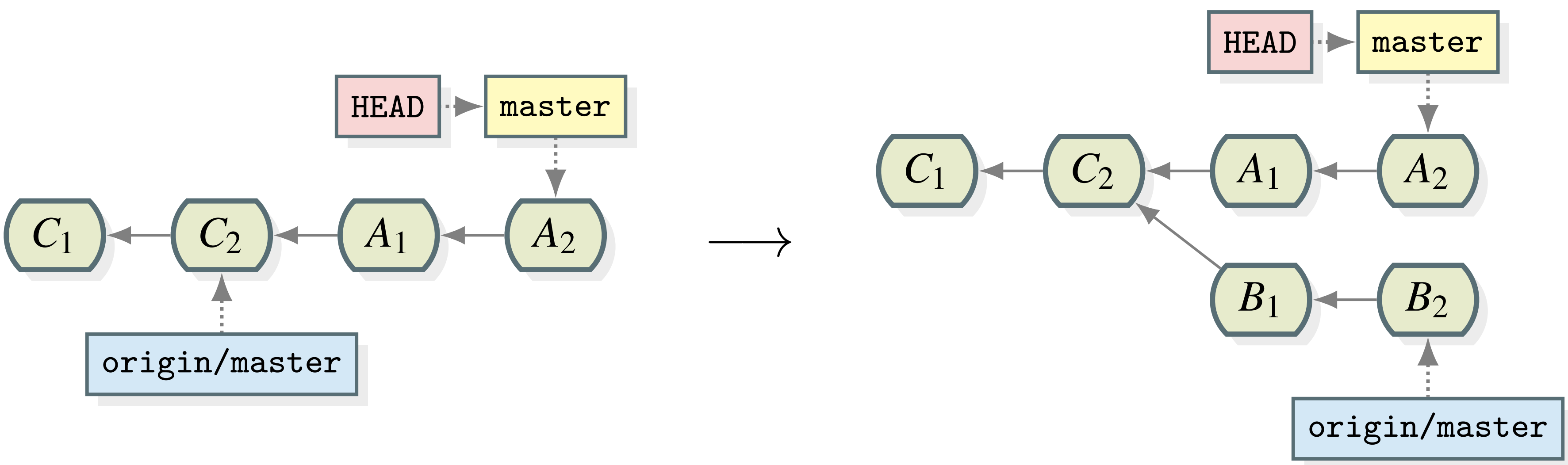
4. Committing change A_2 to the `master` branch of the local repository transforms the local repository as follows:



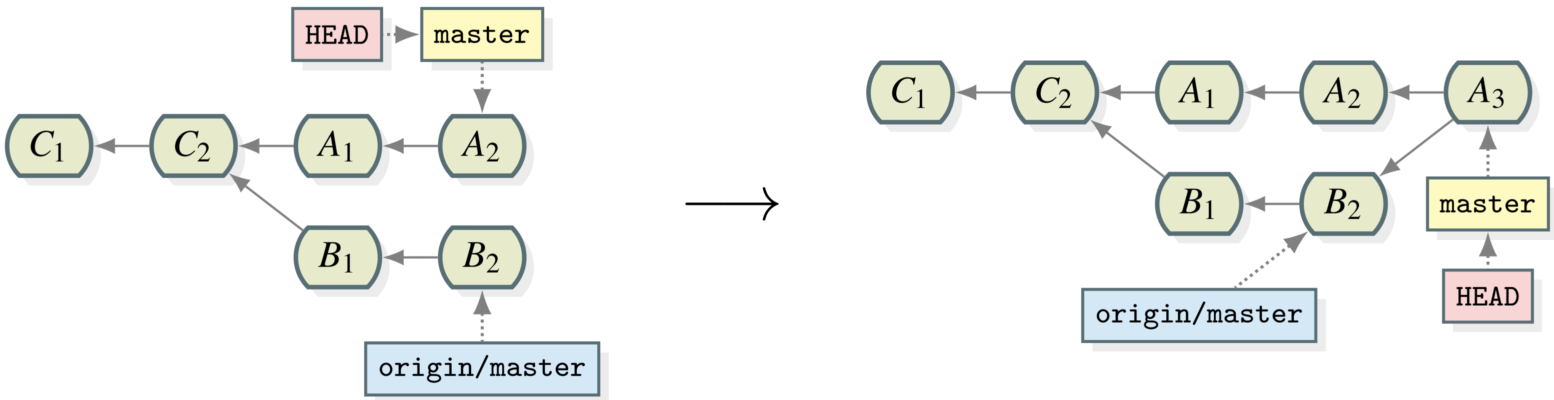
5. Another user committing changes B_1 and B_2 to the remote repository transforms the remote repository as follows:



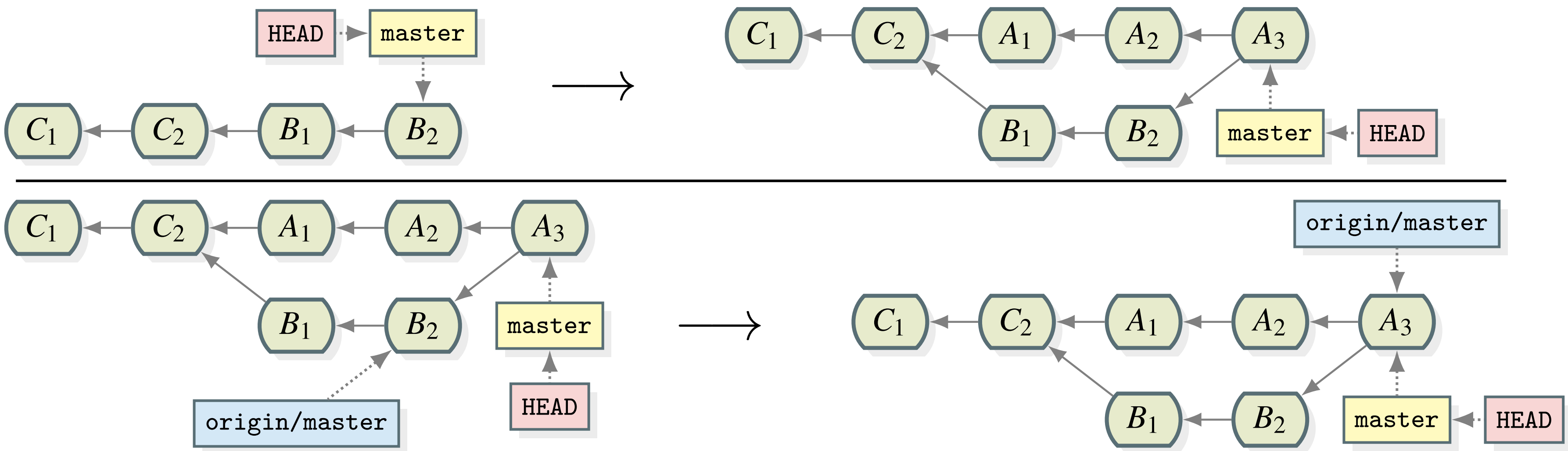
6. Fetching (to the local repository) from the remote repository transforms the local repository as follows:



7. Merging the `origin/master` branch into the `master` branch (in the local repository) transforms the local repository as follows:



8. Pushing (from the local repository) to the remote repository transforms the remote and local repositories, respectively, as follows:



Commit History Example: Commands to Setup Remote

```
TOP_DIR=`pwd`

cd $TOP_DIR
mkdir remote
cd remote
git init
printf "apple\n" >> fruits.txt
git add fruits.txt
git commit -m "Added file fruits.txt" # Commit C1
printf "banana\n" >> fruits.txt
git add fruits.txt
git commit -m "Added banana to fruits.txt" # Commit C2
git init --bare .git
mv .git $TOP_DIR/remote.git
cd $TOP_DIR
rm -rf remote
```

Commit History Example: Remaining Commands

```
cd $TOP_DIR
git clone remote.git local-1

cd $TOP_DIR/local-1
printf "grape\n" >> fruits.txt
git add fruits.txt
git commit -m "Added grape to fruits.txt" # Commit A1
printf "orange\n" >> fruits.txt
git add fruits.txt
git commit -m "Added orange to fruits.txt" # Commit A2

cd $TOP_DIR
git clone remote.git local-2
cd local-2
printf "red\n" >> colors.txt
git add colors.txt
git commit -m "Added file colors.txt" # Commit B1
printf "green\n" >> colors.txt
git add colors.txt
git commit -m "Added green to colors.txt" # Commit B2
git push

cd $TOP_DIR/local-1
git push # ERROR: local repository not up to date
git fetch
git merge -m "Merged changes." # Commit A3
git push
```

Git Configuration

- Git employs three levels of configuration settings, which in order of decreasing priority are as follows:
  1. local (i.e., per repository)
  2. global (i.e., per user)
  3. system (i.e., system wide)
- Configuring system settings may require administrator privileges.
- On Linux systems, the global settings are typically stored in the file `$HOME/.gitconfig`.

Git on One Slide

- Configure user information and clone the repository:

```
git config --global user.name "John Doe"
git config --global user.email jdoe@gmail.com
git clone $repository $directory
```

- Edit the working tree and stage changes as appropriate for the local repository:

```
git add $path_to_add
git mv $source_path $destination_path
git rm $file_to_remove
git rm -r $directory_to_remove
```

- Check what changes are staged and then commit these changes to the local repository:

```
git status
git commit
```

- Push changes from the local repository to the remote repository:

```
git push
```

- As needed, retrieve changes from the remote repository and merge them locally (e.g., if a push failed due to being out of date):

```
git pull
```

Section 10.2.1

Basic Commands

Determining the Version of Git

- To query the version of the Git software, type:

```
git --version
```

- The original release dates for a few versions of Git are as follows:

Version	Original Release Date
1.0	2005-12-21
1.7	2010-02-13
1.8	2012-10-21
1.9	2014-02-14
2.0	2014-05-28
2.3	2015-02-05
2.8	2016-03-28
2.12	2017-02-24
2.16	2018-01-17

Obtaining Help on the `git` Command

- To obtain general help for the `git` command, use a command of the form:

  ```
  git help [options]
  ```

- To obtain detailed information for the `git` command or guide `$item`, use a command of the form:

  ```
  git help [options] $item
  ```

- Some commonly-used options include:

Option	Description
`-a`	list all commands for which help available
`-g`	list all available help guides
`-w`	display in HTML format using a web browser
`-m`	display in man (i.e., manual page) format

- To obtain detailed help on the `commit` command with the information displayed in HTML format in a web browser, type:

  ```
  git help -w commit
  ```

- To list all of the help guides available, type:

  ```
  git help -g
  ```

Configuring Git

- To set the variable `$name` to the value `$value`, use a command of the form:

```
git config [options] $name $value
```

- To unset the variable `$name`, use a command of the form:

```
git config [options] --unset $name
```

- To list all of the current variables settings, use a command of the form:

```
git config [options] -l
```

- Some commonly-used options include:

Option	Description
`--system`	consider only the system-wide settings
`--global`	consider only the global (i.e., per-user) settings
`--local`	consider only the local (i.e., per-repository) settings

Some Commonly-Used Git Variables

Variable	Description
`core.askPass`	program for entering user name and password credentials
`core.editor`	program for editing
`core.pager`	program for paging output
`credential.helper`	external program to be called when a user name or password credential is needed
`user.name`	user's full name
`user.email`	user's email address
`web.browser`	program for browsing web

Configuring User Information

- To globally set the user name to “John Doe”, type:
  ```
  git config --global user.name "John Doe"
  ```
- To globally set the email address to “jdoe@gmail.com”, type:
  ```
  git config --global user.email jdoe@gmail.com
  ```
- To list all system, global, and local variables, type:
  ```
  git config -l
  ```
- To list only the global variables, type:
  ```
  git config --global -l
  ```
- To list only the local (i.e., per-repository) variables for the current repository, type:
  ```
  git config --local -l
  ```

Configuring User-Credential-Related Information

- To enable the global caching of user credentials for 1 hour (i.e., 3600 seconds), type:

```
git config --global \
  credential.helper 'cache --timeout=3600'
```

- To disable all caching of user credentials (i.e., at the system, global, and repository levels) and purge any cached values, type:

```
git config --unset credential.helper
git config --global --unset credential.helper
git config --system --unset credential.helper
git credential-cache exit
```

- To ensure that prompting for user credentials employs standard input/output (as opposed to, say, a pop-up window), type:

```
git config --unset core.askPass
git config --global --unset core.askPass
git config --system --unset core.askPass
unset GIT_ASKPASS
unset SSH_ASKPASS
```

Creating an Empty Repository

- To create an empty repository, use a command of the form:

```
git init [$directory]
```

- If `$directory` is not specified, it defaults to the current directory.
- The repository is created in the directory `$directory`.
- If `$directory` already contains a repository, the repository is re-initialized (in a non-destructive manner).
- All of the information used internally by Git to maintain the state of the repository is stored in a directory named `.git` at the top-level directory in the working tree.
- To create a new repository in the directory `hello` (which does not currently exist), type:

```
git init hello
```

Cloning a Repository

- To clone a repository `$repository`, use a command of the form:

```
git clone [options] $repository [$directory]
```

- If `$directory` is not specified, it defaults to a value derived from `$repository`.
- The cloned repository is placed in the directory `$directory`.
- The repository specifier `$repository` can be a URL (for a repository accessed through a network server) or a directory (for a repository accessed through the local file system).
- Some commonly-used options include:

Option	Description
`-b $branch`	after cloning, checkout the branch `$branch`

- To clone the repository associated with the URL `https://github.com/uvic-aurora/hello-world.git` to the directory `hello-world`, type:

```
git clone \
  https://github.com/uvic-aurora/hello-world.git \
  hello-world
```

Adding Files/Directories to the Index

- To add the files/directories `$path...` to the index (i.e., mark for committing later), use a command of the form::

```
git add $path...
```

- The contents of `$path...` ***at the time that the*** `git add` ***command is run*** are staged; subsequent changes to these contents are not automatically staged.
- When a directory is staged, all directories and files that are contained under it are staged (i.e., staging is recursive).
- To prevent certain files/directories in a directory from being staged, they can be listed in a `.gitignore` file in that directory.
- To add the files `README` and `LICENSE` to the index, type:

```
git add README LICENSE
```

- To add the directory `src` (and everything contained under it) to the index, type:

```
git add src
```

Removing Files/Directories from the Index

- To remove all changes from the index, type:

```
git reset
```

- To remove the files/directories `$path...` from the index, use a command of the form:

```
git reset $path...
```

- To undo the effects of the command "`git add README LICENSE`", type:

```
git reset README LICENSE
```

- To undo the effects of the command "`git add README`", type:

```
git reset README
```

Renaming Files

- To move the file/directory `$source` to `$destination`, use a command of the form:

```
git mv [options] $source $destination
```

- To move multiple files `$s_1`, `$s_2`, `...`, `$s_n` to the directory `$destination_directory`, use a command of the form:

```
git mv [options] $s_1 $s_2 ... $s_n \
  $destination_directory
```

- Some commonly-used options include:

Option	Description
`-f`	force moving even if the target exists

- To rename the file `README` to `README.old`, type:

```
git mv README README.old
```

- To move the files `hello.cpp` and `goodbye.cpp` to the directory `src`, type:

```
git mv hello.cpp goodbye.cpp src
```

Removing Files

- To remove the files/directories `$path...` from the working tree and the index, use a command of the form:

```
git rm [options] $path...
```

- Some commonly-used options include:

Option	Description
`-f`	override the up-to-date check
`-r`	if the given path is a directory, recursively remove files below it

- To remove the directory `src` and all files and directories beneath it from the working tree and index, type:

```
git rm -r src
```

- To remove the files `README` and `LICENSE` from the working tree and index, type:

```
git rm README LICENSE
```

Committing Changes

- To commit all staged changes, use a command of the form:

```
git commit [options]
```

- Some commonly-used options include:

Option	Description
`-a`	automatically stage any files that have been modified or deleted
`-m $message`	set message to `$message`

- To commit all staged changes with the message “Fixed overflow bug”, type:

```
git commit -m "Fixed overflow bug"
```

- To commit all staged changes with the message “Fixed overflow bug”, automatically staging any files that have been modified or deleted, type:

```
git commit -a -m "Fixed overflow bug"
```

Checking the Status of the Working Tree

- To display the status of the working tree, use a command of the form:

  ```
  git [options] status
  ```

- Some commonly-used options include:

Option	Description
`--long`	give the output in long format (default)
`--short`	give the output in short format

- The information displayed by this command includes:
  - paths (i.e., files and directories) that have differences between the index and the current HEAD commit, (i.e., what would be committed by running `git commit`)
  - paths that have differences between the working tree and index as well as paths that are not tracked by Git (i.e., what could by committed by running `git add` before `git commit`)
- To display the status of the working tree in long form, type:

  ```
  git status
  ```

Showing Commit Logs

- To show the commit history (which can be limited to a particular revision range `$revision_range` or files/directories `$path...`), use a command of the form:

```
git log [options] [$revision_range] [[--] $path...]
```

- Some commonly-used options include:

Option	Description
`--since $date`	select commits more recent than date `$date`
`--until $date`	select commits older than date `$date`
`-$n`	select last `$n$` commits
`-S $pattern`	select commits adding/removing string matching pattern `$pattern`
`--graph`	draw text-based graph of commit history
`--all`	consider all branches/remotes/tags
`--grep $pattern`	select only commits with message matching `$pattern`

Showing Commit Logs (Continued)

- To show the commit history for the file `README` since 2016-01-01, type:

```
git log --since 2016-01-01 README
```

- To show the commit history for all files between 2014-01-01 and 2014-12-31, type:

```
git log --since 2014-01-01 --until 2014-12-31
```

- To show the commit history for all files in all branches with a text-based graph, type:

```
git log --all --graph
```

- To show the commit history for all commits made since `v1.0` until and including `v2.0` (assuming that `v1.0` and `v2.0` exist), type:

```
git log v1.0..v2.0
```

Showing Changes

- To show changes between the working tree and the index (i.e., what could be staged but has not yet been) for files/directories `$path...` (which defaults to all files/directories), use a command of the form:

```
git diff [options] [$path...]
```

- To show changes between the index and the named commit `$commit` (which defaults to `HEAD`) for the files/directories `$path...` (which defaults to all files/directories), use a command of the form:

```
git diff [options] --cached [$commit] -- \
  [$path...]
```

- To show changes between the working tree and the named commit `$commit` (which defaults to `HEAD`) for the files/directories `$path...` (which defaults to all files/directories), use a command of the form:

```
git diff [options] [$commit] -- [$path...]
```

- To show changes between two arbitrary commits `$commit1` and `$commit2` for the files/directories `$path...`, use a command of the form:

```
git diff [options] $commit1 $commit2 -- \
  [$path...]
```

Showing Changes (Continued)

- Some commonly-used options include:

Option	Description
`-b`	ignore changes in amount of whitespace
`-w`	ignore all whitespace
`--ignore-blank-lines`	ignore blank lines

- To show all changes between the working tree and the index for all files/directories, type:

```
git diff
```

- To show all differences between the working tree and the index for the file `README`, ignoring changes in amount of whitespace, type:

```
git diff -b README
```

Finding Lines Matching a Pattern

- To find all lines of text in the files `$path...` (which defaults to all files) in the working tree that satisfy the condition specified by the `p_options`, use a command of the form:

```
git grep [options] [p_options] -- [$path...]
```

- Some commonly-used options include:

Option	Description
`-l`	print only names of files with matches
`-i`	ignore case
`--max-depth $depth`	descend at most `$depth` levels of directories
`-v`	select non-matching lines
`-F`	patterns are fixed strings
`-E`	patterns are extended POSIX regular expressions
`-e $pattern`	specify pattern `$pattern`
`--and`	logical and
`--or`	logical or
`--not`	logical not
`(`	for grouping logical operations
`)`	for grouping logical operations
`--cache`	search in the index instead of the working tree

Finding Lines Matching a Pattern (Continued)

- To search for the text "`hello`" in a case insensitive manner in all of the files in the working tree, type:

```
git grep -i -e hello
```

- To print only the names of the files that match the pattern specified in the preceding example, type:

```
git grep -i -e hello -l
```

- To find all of the files in the working tree with suffixes ".cpp" or ".hpp" that have lines containing either "`#include <vector>`" or "`#include <list>`", type:

```
git grep -e '#include <vector>' --or \
  -e '#include <list>' -- '*.cpp' '*.hpp'
```

- To perform the same search as the preceding example but in the index rather than the working tree, type:

```
git grep --cache -e '#include <vector>' --or \
  -e '#include <list>' -- '*.cpp' '*.hpp'
```

Removing Untracked Files and Directories

- To remove all untracked files in the working tree, use a command of the form:

  ```
  git clean [options]
  ```

- Some commonly-used options include:

Option	Description
`-d`	remove untracked directories in addition to untracked files
`-f`	force removal of files
`-i`	enable interactive mode
`-n`	show what would be done without actually doing anything
`-x`	do not use standard ignore rules (such as those specified in `.gitignore` files)

- To remove all untracked files and directories in the working tree excluding those ignored by Git, type:

  ```
  git clean -d -f
  ```

- To remove all untracked files and directories in the working tree including those ignored by Git, type:

  ```
  git clean -d -f -x
  ```

`.gitignore` Files

- A `.gitignore` file specifies which files and directories are intentionally untracked and should be ignored by Git.
- The purpose of a `.gitignore` file is to ensure that certain files not tracked by Git remain untracked.
- The `.gitignore` file lists patterns specifying files that should be ignored by Git.
- A "`!`" prefix negates a pattern.
- A leading slash matches the directory containing the `.gitignore` file. For example, `/hello.cpp` matches `hello.cpp` but not `some/subdirectory/hello.cpp`.
- The patterns in a `.gitignore` file apply to the directory containing the file as well as all directories below the file in the working tree.
- The patterns in a `.gitignore` file at a higher level in the tree are overridden by patterns in a `.gitignore` file at a lower level.
- A `.gitignore` file in the root directory of the working tree can be used to establish ignore defaults for the whole tree.

.gitignore File Example

```
# ignore all object files
*.o
# ignore all library files
*.a
# ignore foobaz only in this directory
/foobaz
# ignore foo only in directory example
/example/foo
```

.gitattributes Files

- A `.gitattributes` file is used to specify attributes for files/directories.
- For example, the determination of whether a file employs a binary or text format can be controlled via a `.gitattributes` file.
- An example of a `.gitattributes` file is as follows:

```
# Consider all PNM files to be binary.
*.pnm binary
```

- The settings in a `.gitattributes` file apply to the directory containing the file as well as all directories below the file in the working tree.
- The settings in a `.gitattributes` file at a higher level in the tree are overridden by settings in a `.gitattributes` file at a lower level.
- A `.gitattributes` file in the root directory of the working tree can be used to establish attribute defaults for the whole tree.

Tracking Empty Directories

- The current implementation of `git` does not allow empty directories to be tracked.
- The best workaround for this problem is to create a `.gitignore` file in the directory that ignores all files except the `.gitignore` file itself.
- Such a `.gitignore` file might look like the following:

```
# First, ignore everything in this directory.
*
# Now, override the preceding rule and force
# the .gitignore file not to be ignored.
!.gitignore
```

- This is not a perfect solution as it requires that the "empty" directory contain one file (namely, a `.gitignore` file) and this file be committed to the repository.

Section 10.2.2

Remote-Related Commands

Listing, Adding, and Removing Remotes

- To list the remotes, use a command of the form:

```
git remote [general_options]
```

- Some general options include:

Option	Description
`-v`	enable verbose mode

- To add the remote `$remote` with the associated URL `$url`, use a command of the form:

```
git remote add $remote $url
```

- To remove the remote `$remote`, use a command of the form:

```
git remote rm $remote
```

- To rename a remote from `$old` to `$new`, use a command of the form:

```
git remote rename $old $new
```

- To show detailed information on the remote `$remote`, use a command of the form:

```
git remote [general_options] show $remote
```

Fetching Changes from Another Repository

- To fetch changes from the remote `$remote` (which normally defaults to `origin`), use a command of the form:

  ```
  git fetch [$remote]
  ```

- A fetch operation gathers any commits from the target branch that do not exist in current branch, and stores them in the local repository.
- It is always safe to perform a fetch in sense that no conflicts can arise, since no merge is attempted.
- By fetching frequently, one can keep their local repository up to date without being forced to merge.

Pushing Changes to Another Repository

- To push changes to the branch `$branch` (which normally defaults to the current branch) of the remote `$remote` (which normally defaults to `origin`), use a command of the form:

```
git push [options] [$remote [$branch]]
```

- When pushing a new local branch to a remote, the `-u` option should be specified.
- To delete the branch `$branch` on the remote `$remote` only, use a command of the form:

```
git push --delete origin $branch
```

- The preceding command is useful if one wants to delete a branch that exists on the remote but not in the local repository.
- To delete the tag `$tag` on the remote `$remote` only, use a command of the form:

```
git push --delete origin $tag
```

- To push to the default remote and branch, type:

```
git push
```

Pulling Changes from Another Repository

- To pull changes from the branch `$branch` of the remote `$remote`, use a command of the form:

```
git pull [$remote [$branch]]
```

- To pull from the default remote and branch, type:

```
git pull
```

- A pull is approximately a fetch followed by merge.
- A pull automatically merges commits without letting them be reviewed first.
- For this reason, some people suggest that it is better to use fetch and merge separately instead of performing a pull.
- Also, the use of pull operations can, in some cases, result in unnecessary merge commits.

Merging Changes

- To merge changes from the branch `$branch` (which normally defaults to the upstream branch for the current branch) into the current branch, use a command of the form:

```
git merge [$branch]
```

- To merge from the default branch, type:

```
git merge
```

- Note that the merge direction is from the branch `$branch` into the current branch.
- It is advisable to ensure that any outstanding changes are committed before running `git merge` in order to reduce the likelihood of major difficulties in the case of a conflict.
- If a conflict arises, no commit will be performed and manual intervention is required to resolve the conflict.

Section 10.2.3

Branch-Related Commands

Listing, Creating, and Deleting Branches

- To list all of the branches, use a command of the form:

```
git branch [options]
```

- To create a branch `$branch`, use a command of the form:

```
git branch [options] $branch
```

- To delete the (local) branch `$branch`, use a command of the form:

```
git branch [options] -d $branch
```

- Some commonly-used options include:

Option	Description
`-a`	list both remote and local branches
`-r`	list or delete remote branches
`-v`	enable verbose mode for listing (use twice for extra verbose)

- When a new branch is created with `git branch`, this does not automatically checkout (i.e., switch to using) the new branch.

Checking Out a Branch

- To checkout (i.e., switch to) the branch `$branch`, use a command of the form:

  ```
  git checkout $branch
  ```

- Checking out a branch changes the files/directories of the working tree to match that branch.
- If you have local modifications to one or more files that are different between the current branch and the branch to which you are switching, the command refuses to switch branches in order to preserve your modifications in context.

Section 10.2.4

Tag-Related Commands

Listing, Creating, and Deleting Tags

- To list all tags, type:

```
git tag
```

- To tag a commit `$commit` (which defaults to `HEAD`) with the name `$name`, use a command of the form:

```
git tag [options] $name [$commit]
```

- To delete the (local) tags with names `$name...`, use a command of the form:

```
git tag -d $name...
```

- Some commonly-used options include:

Option	Description
`-a`	make an unsigned annotated tag

- To create an annotated tag `version-1.0` for the most recent commit on the `master` branch, type:

```
git tag -a version-1.0 master
```

- To delete the tag `version-1.0`, type:

```
git tag -d version-1.0
```

Pushing Tags

- To push a tag `$tag` to the remote `$remote`, use a command of the form:

```
git push $remote $tag
```

- To push the tag `v1.0` to the remote `origin`, type:

```
git push origin v1.0
```

Section 10.2.5

Miscellany

- can create exact duplicate of entire Git repository (including all tags and local branches) by using bare-clone and mirror-push operations
- to copy repository `$source_repo` to (already existing) remote repository `$destination_repo` (overwriting contents of repository), use command sequence:

```
# Create a bare clone of the repository.
git clone --bare $source_repo $bare_dir
# Mirror push to the destination repository.
git -C $bare_dir push --mirror $destination_repo
# Remove the temporary local bare repository.
rm -rf $bare_dir
```

- to copy repository `$source_repo` to local repository directory `$destination_dir`, simply perform bare clone operation using command:

```
git clone --bare $source_repo $destination_dir
```

Avoiding Repeated Passphrase Entry for SSH Authentication

- if SSH used to access repository, SSH passphrase often needs to be provided
- to avoid having to enter SSH passphrase every time it is needed, can use SSH Agent to cache passphrase and provide it as required
- to start SSH Agent and provide it with passphrase to cache for private key file `$key_file`, use command sequence:

```
# Start SSH Agent
eval `ssh-agent`
# Provide passphrase for particular key.
ssh-add $key_file
```

- on Unix systems, SSH key information typically stored in directory `$HOME/.ssh`

Additional Remarks

- A file is said to be **derived** if it is generated from one or more other files (e.g., an object file is derived from its corresponding source code file, a PDF or PostScript file is derived from its corresponding LaTeX source files).
- Do not place derived files under version control, as such files are completely redundant and can often lead to a significant increase in repository size.
- Do not place large unchanging datasets under version control, as this will greatly increase repository size without any tangible benefit (i.e., since the datasets are not changing, there will never be multiple versions of them to manage).
- Avoid placing sensitive information (e.g., passwords) under version control.
- Remember that deleting a file from a particular commit does not delete that file from the repository, since the file will still exist in other commits.

Git-Related Software

- Gitg. A GNOME GUI client for viewing Git repositories. `https://wiki.gnome.org/Apps/Gitg`.
- Meld. A visual diff and merge tool. `http://meldmerge.org`.
- Hub. A command-line wrapper for Git that facilitates easier use of GitHub. `https://hub.github.com`.

Section 10.2.6

References

1. Official Git Web Site. `https://git-scm.com`, August 2016.
   This web site has many excellent resources related to Git, including:
   1. Git Downloads: `https://git-scm.com/downloads`.
      This web page has the Git software for various platforms, including Linux, Mac OS X, and Windows.
   2. Git Book: Scott Chacon and Ben Straub, Pro Git, `http://git-scm.com/book`.
      This online book can also be downloaded in several formats (including PDF).
   3. Git Videos: `https://git-scm.com/videos`.
      This web page has several short videos on various aspects of Git.
2. Good Resources for Learning Git and GitHub, `https://help.github.com/articles/good-resources-for-learning-git-and-github` August 2016.
   This web page has a list of many excellent resources for learning both Git and GitHub.

References II

3. TryGit Tutorial, `https://try.github.com`, August 2016.
   This online Git tutorial allows the user to try Git in their web browser.

4. J. Loeliger. *Version Control with Git.*
   O'Reilly, Sebastopol, CA, USA, 2009.

Talks

1. Linus Torvalds, Google Tech Talk: Linus Torvalds on git — Git: Source code control the way it was meant to be!, May 2007. Available online at `https://youtu.be/4XpnKHJAok8`.

   Linus Torvalds shares his thoughts on Git, the source control management system he created.

2. Matthew McCullough, The Basics of Git and GitHub, July 2013. Available online at `https://youtu.be/U8GBXvdmHT4`.

   This is an excellent introduction to using Git.

3. Scott Chacon, Introduction to Git with Scott Chacon of GitHub, June 2011. Available online at `https://youtu.be/ZDR433b0HJY`.

   This is another popular introduction to using Git.

4. Matthew McCullough, Advanced Git: Graphs, Hashes, and Compression, Oh My!, Sept. 2012. Available online at `https://youtu.be/ig5E8CcdM9g`.

   This is a very good more advanced talk on Git.

Part 11

Miscellany

What Is Wrong With This Code?

foo.hpp

```
#ifndef foo_hpp
#define foo_hpp
namespace foo {
bool is_odd(int x) {return (x % 2) != 0;}
bool is_even(int x) {return (x % 2) == 0;}
}
#endif
```

main.cpp

```
#include <iostream>
#include "foo.hpp"
int main() {
    std::cout << foo::is_odd(42) << ' ' <<
      foo::is_even(42) << '\n';
}
```

other.cpp

```
#include "foo.hpp"
// ...
```

Solution: Functions Should Be Inline

foo.hpp

```
#ifndef foo_hpp
#define foo_hpp
namespace foo {
inline bool is_odd(int x) {return (x % 2) != 0;}
inline bool is_even(int x) {return (x % 2) == 0;}
}
#endif
```

main.cpp

```
#include <iostream>
#include "foo.hpp"
int main() {
    std::cout << foo::is_odd(42) << ' ' <<
      foo::is_even(42) << '\n';
}
```

other.cpp

```
#include "foo.hpp"
// ...
```

What Is Wrong With This Code?

foo.hpp

```
#ifndef foo_hpp
#define foo_hpp
namespace foo {
inline bool is_odd(int x);
inline bool is_even(int x);
}
#endif
```

foo.cpp

```
#include "foo.hpp"
namespace foo {
bool is_odd(int x) {return (x % 2) != 0;}
bool is_even(int x) {return (x % 2) == 0;}
}
```

app.cpp

```
#include <iostream>
#include "foo.hpp"
int main() {
    std::cout << foo::is_odd(42) << ' ' <<
      foo::is_even(42) << '\n';
}
```

Solution: Place Inline Function Definitions in Header File

foo.hpp

```
#ifndef foo_hpp
#define foo_hpp
namespace foo {
inline bool is_odd(int x) {return (x % 2) != 0;}
inline bool is_even(int x) {return (x % 2) == 0;}
}
#endif
```

app.cpp

```
#include <iostream>
#include "foo.hpp"
int main() {
    std::cout << foo::is_odd(42) << ' ' <<
      foo::is_even(42) << '\n';
}
```

What Is Wrong With This Code?

foo.hpp

```
#ifndef foo_hpp
#define foo_hpp
namespace foo {
template <typename T> T abs(const T& x);
}
#endif
```

foo.cpp

```
#include "foo.hpp"
namespace foo {
template <typename T> T abs(const T& x)
  {return (x < 0) ? (-x) : x;}
}
```

app.cpp

```
#include <iostream>
#include "foo.hpp"
int main() {
    std::cout << foo::abs(-42) << ' ' <<
      foo::abs(-3.14) << '\n';
}
```

First Solution: Explicit Template Instantiation

foo.hpp

```
#ifndef foo_hpp
#define foo_hpp
namespace foo {
template <typename T> T abs(const T& x);
}
#endif
```

foo.cpp

```
#include "foo.hpp"
namespace foo {
template <typename T> T abs(const T& x)
  {return (x < 0) ? (-x) : x;}
template int abs<int>(const int&);
template double abs<double>(const double&);
}
```

app.cpp

```
#include <iostream>
#include "foo.hpp"
int main() {
    std::cout << foo::abs(-42) << ' ' <<
      foo::abs(-3.14) << '\n';
}
```

foo.hpp

```
#ifndef foo_hpp
#define foo_hpp
namespace foo {
template <typename T> T abs(const T& x)
  {return (x < 0) ? (-x) : x;}
}
#endif
```

app.cpp

```
#include <iostream>
#include "foo.hpp"
int main() {
    std::cout << foo::abs(-42) << ' ' <<
      foo::abs(-3.14) << '\n';
}
```

Remarks on Headers Files and Function Declarations

- Every function (whether it be inline or non-inline, or template or non-template) must be ***declared*** before being used.
- Consequently, functions that are part of an interface should normally be declared in a ***header file*** so that users of the interface can obtain the declarations needed for the interface by simply including the header file.
- An inline function should always be ***defined*** before being used.
- Consequently, an inline function that is declared in a header file should normally also be ***defined*** in the file.
- A template function must be ***defined*** at its point of use in order for the template to be implicitly instantiated.
- Consequently, a template function that is declared in a header file should normally also be ***defined*** in the file.
- A function must not be defined more than once.
- Consequently, unless a function is inline or a template, it should not be defined in a header file, as this will result in multiple definitions if the header file is included by more than one source file.

What Is Wrong With This Code?

foo.hpp

```
#ifndef foo_hpp
#define foo_hpp
#include <cmath>
namespace foo {
double log(double x, double b);
}
#endif
```

foo.cpp

```
#include <cmath>
#include "foo.hpp"
namespace foo {
double log(double x, double b = 10.0)
  {return std::log(x) / std::log(b);}
}
```

app.cpp

```
#include <iostream>
#include "foo.hpp"
int main() {
    std::cout << foo::log(16.0, 2.0) << ' ' <<
      foo::log(10.0) << '\n';
}
```

Solution: Place Default Arguments in Header File

foo.hpp

```
#ifndef foo_hpp
#define foo_hpp
#include <cmath>
namespace foo {
double log(double x, double b = 10.0);
}
#endif
```

foo.cpp

```
#include <cmath>
#include "foo.hpp"
namespace foo {
double log(double x, double b)
  {return std::log(x) / std::log(b);}
}
```

app.cpp

```
#include <iostream>
#include "foo.hpp"
int main() {
    std::cout << foo::log(16.0, 2.0) << ' ' <<
      foo::log(10.0) << '\n';
}
```

Part 12

Additional Learning Resources

- Know what you do not know.
- Ask questions when you are uncertain about something and be sure that the person whom you ask is knowledgeable enough to give a ***correct*** answer.
- Know what information resources can be ***trusted***.
- Learn to use reference materials effectively (e.g., documentation on libraries, standards).

C++ References

- Some good references on various topics related to the C++ programming language, C++ standard library, and other C++ libraries (such as Boost) are listed on the slides that follow.
- Any information on C++ (e.g., books, tutorials, videos, seminars) from the following individuals (who are held in very high regard by the C++ community) is highly recommended:
  - Bjarne Stroustrup (the creator of C++)
  - Scott Meyers
  - Herb Sutter (Convener of ISO C++ standards committee for over 10 years)
  - Andrei Alexandrescu
  - Stephan Lavavej

C++ References I

1. ISO/IEC 14882:2017 — information technology — programming languages — C++, Dec. 2017.

   This is the definitive specification of the C++ language and standard library. This is an essential reference for any advanced programmer.

2. B. Stroustrup. *The C++ Programming Language.* Addison Wesley, 4th edition, 2013.

   This is the classic book on the C++ programming language and standard library, written by the creator of the language. This is one of the best references for first learning C++.

3. Standard C++ Foundation web site. `http://www.isocpp.org`, 2014.

   This is the web site of a non-profit organization whose purpose is to support the C++ software development community and promote the understanding and use of modern standard C++ on all compilers and platforms. This is an absolutely outstanding source of information on C++.

4. B. Stroustrup and H. Sutter (editors), C++ Core Guidelines, 2016, `http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`.

   This document provides a very detailed set of guidelines for writing good C++ code.

5. S. Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14.*
   O'Reilly Media, Cambridge, MA, USA, 2015.

   This book covers a list of 42 topics on how to better utilize the C++ language.

6. S. Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs.*
   Addison Wesley, Menlo Park, California, 1992.

   This book covers a list of 50 topics on how to better utilize the C++ language.

7. S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*.
   Addison Wesley, Menlo Park, California, 1996.
   This book covers a list of 35 topics on how to better utilize the C++ language. It builds on Meyers' earlier "Effective C++" book.

8. S. Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*.
   Addison Wesley, 2001.
   This book covers a list of 50 topics on how to better utilize the Standard Template Library (STL), an essential component of the C++ standard library.

9. N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*.
   Addison Wesley, Upper Saddle River, NJ, USA, 2nd edition, 2012.
   This is a very comprehensive book on the C++ standard library. This is arguably the best reference on the standard library (other than the C++ standard).

10 D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide.* Addison Wesley, 2002.

This is a very comprehensive book on template programming in C++. It is arguably one of the best books on templates in C++.

11 A. Williams. *C++ Concurrency in Action.* Manning Publications, Shelter Island, NY, USA, 2012.

This is a fairly comprehensive book on concurrency and multithreaded programming in C++. It is arguably the best book available for those who want to learn how to write multithreaded code using C++.

12 H. Sutter. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions.* Addison Wesley, 1999.

This book covers topics including (but not limited to): proper resource management, exception safety, RAII, and good class design.

13. H. Sutter. *More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions*. Addison Wesley, 2001.

    This book covers topics including (but not limited to): exception safety, effective object-oriented programming, and correct use of STL.

14. H. Sutter. *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions*. Addison Wesley, 2004.

    This book covers topics including (but not limited to): generic programming, optimization, resource management, and how to write modular code.

15. H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison Wesley, 2004.

    This book presents 101 best practices, idioms, and common pitfalls in C++ in order to allow the reader to become a more effective C++ programmer.

[16] A. Langer and K. Kreft. *Standard C++ IOStreams and Locales.* Addison Wesley, 2000.

This book provides a very detailed look at C++ I/O streams and locales.

[17] V. A. Punathambekar. How to interpret complex C/C++ declarations. `http://www.codeproject.com/Articles/7042/How-to-interpret-complex-C-C-declarations`, 2004.

This is a detailed tutorial on how to interpret complex C/C++ type declarations. This tutorial explains how type declarations are parsed in the language, which is essential for all programmers to understand clearly.

1. S. B. Lippman, J. Lajoie, and B. E. Moo. *C++ Primer*.
   Addison Wesley, Upper Saddle River, NJ, USA, 4th edition, 2005.
2. A. Koenig and B. E. Moo. *Accelerated C++: Practical Programming by Example*.
   Addison Wesley, Upper Saddle River, NJ, USA, 2000.
3. B. Eckel. *Thinking in C++—Volume 1: Introduction to Standard C++*.
   Prentice Hall, 2nd edition, 2000.
4. B. Eckel and C. Allison. *Thinking in C++—Volume 2: Practical Programming*.
   Prentice Hall, 1st edition, 2003.
5. B. Stroustrup. *Programming: Principles and Practice Using C++*.
   Addison Wesley, Upper Saddle River, NJ, USA, 2009.
   An introduction to programming using C++ by the creator of the language.

Other C++ References II

6. A. Alexandrescu. *Modern C++ Design.* Addison Wesley, Upper Saddle River, NJ, USA, 2001.

7. D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond.* Addison Wesley, Boston, MA, USA, 2004.

8. D. D. Gennaro. *Advanced C++ Metaprogramming.* CreateSpace Independent Publishing Platform, 2011.

9. Boost web site. `http://www.boost.org`, 2014. The web site for the Boost C++ libraries.

10. B. Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost.* Addison Wesley, Upper Saddle River, NJ, USA, 2005. An introduction to (some parts of) the Boost library.

11. B. Schaling. *The Boost C++ Libraries.*
XML Press, 2nd edition, 2014.
An introduction to the Boost library. Online version at `http://theboostcpplibraries.com`.

12. M. Kilpelainen. Overload resolution — selecting the function.
Overload, 66:22–25, Apr. 2005.
Available online at `http://accu.org/index.php/journals/268`.

Yet More C++ References I

1. Herb Sutter's Web Site: `http://herbsutter.com`
2. Herb Sutter's Guru of the Week: `http://www.gotw.ca/gotw/`
3. Bjarne Stroustrup's Web Site: `http://www.stroustrup.com`
4. ISO C++ Working Group web site: `http://www.open-std.org/jtc1/sc22/wg21/`
5. ISO C++ Standards Committee GitHub site: `https://github.com/cplusplus`
6. C++ FAQ: `http://www.parashift.com/c++-faq/`
7. Newsgroup comp.lang.c++.moderated: `https://groups.google.com/forum/#!forum/comp.lang.c++.moderated`
8. `http://en.cppreference.com`
9. `http://www.cplusplus.com`
10. Stackoverflow: `http://stackoverflow.com`

Yet More C++ References II

11. Cpp Reddit (C++ discussions, articles, and news): `https://www.reddit.com/r/cpp`
12. Cplusplus Reddit (C++ questions, answers, and discussion): `https://www.reddit.com/r/cplusplus`
13. ACCU Overload Journal: `http://accu.org/index.php/journals/c78/`
14. The C++ Source: `http://www.artima.com/cppsource`

Miscellaneous Talks I

1. Scott Schurr. constexpr: Introduction, CppCon, Bellevue, WA, USA, Sept 19–25, 2015. Available online at `https://youtu.be/fZjYCQ8dzTc`.
2. Scott Schurr. constexpr: Applications, CppCon, Bellevue, WA, USA, Sept 19–25, 2015. Available online at `https://youtu.be/qO-9yiAOQqc`.

C++ Programming Competitions

1. Google Code Jam

   `https://code.google.com/codejam/`

2. Topcoder

   `https://www.topcoder.com/`

3. IEEEXtreme 24-Hour Programming Competition

   `http://www.ieee.org/xtreme`

4. ACM International Collegiate Programming Contest (ICPC)

   `http://icpcnews.com/`

5. CodeChef

   `https://www.codechef.com/`

The Last Word

- Use as many information resources as you can to learn as much as you can about C++.
- Read books, articles, and other documents.
- Watch videos.
- Attend lectures and seminars.
- Participate in programming competitions.
- But most importantly:

  Write code!
  Write lots and lots and lots of code!
- The only way to truly learn a programming language well is to use it heavily (i.e., write lots of code using the language).