M. D. Adams, *An Introduction to the C++ Programming Language (Version 2015-02-03)*, University of Victoria, Victoria, BC, Canada, Feb. 2015, xii+702 slides, ISBN 978-1-55058-537-7 (paperback), ISBN 978-1-55058-538-4 (PDF).

Please Show Your Support for These Lecture Slides

If you like these lecture slides, please show your support for them by doing one or more of the following:

1. Post a review of the lecture slides on Google Play Books and/or Google Books. **This is, by far, the most helpful thing that you can do.**
2. Rate the lecture slides on Google Play Books and/or Google Books.
3. Give a +1 to the lecture slides on Google Play Books and/or Google Books.

For your convenience, the URLs for the lecture slides on both Google Play Books and Google Books are given below. Each URL is also given in the form of a QR code.

- The lecture slides on Google Play Books:
  https://play.google.com/store/books/details?id=cqGBBgAAQBAJ
- The lecture slides on Google Books:
  http://books.google.com/books?id=cqGBBgAAQBAJ

This page is intentionally left blank.

An Introduction to the C++ Programming Language

(Version: 2015-02-03)

Michael D. Adams

Department of Electrical and Computer Engineering
University of Victoria, Victoria, BC, Canada

University of Victoria

For additional information and resources related to these lecture slides, please visit:
`http://www.ece.uvic.ca/~mdadams/cppbook`

Published by the University of Victoria, Victoria, BC, Canada

This document was typeset with LaTeX.

ISBN 978-1-55058-537-7 (paperback)
ISBN 978-1-55058-538-4 (PDF)

License I

```
Creative Commons Legal Code

Attribution-NonCommercial-NoDerivs 3.0 Unported

    CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
    LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN
    ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS
    INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES
    REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR
    DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE
COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY
COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS
AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE
TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY
BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS
CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND
CONDITIONS.

1. Definitions

 a. "Adaptation" means a work based upon the Work, or upon the Work and
    other pre-existing works, such as a translation, adaptation,
    derivative work, arrangement of music or other alterations of a
    literary or artistic work, or phonogram or performance and includes
    cinematographic adaptations or any other form in which the Work may be
    recast, transformed, or adapted including in any form recognizably
    derived from the original, except that a work that constitutes a
    Collection will not be considered an Adaptation for the purpose of
    this License. For the avoidance of doubt, where the Work is a musical
    work, performance or phonogram, the synchronization of the Work in
    timed-relation with a moving image ("synching") will be considered an
    Adaptation for the purpose of this License.
 b. "Collection" means a collection of literary or artistic works, such as
    encyclopedias and anthologies, or performances, phonograms or
```


License II

```
    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 copyrightable 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

License IV

Adaptations. Subject to 8(f), all rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights set forth in Section 4(d).

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested.
b. You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.
c. If You Distribute, or Publicly Perform the Work or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for

License V

```
    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
```


License VI

```
        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
```

License VII

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/.

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;`".

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.

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

C++

- created by Bjarne Stroustrup, Bell Labs
- originally C with Classes, renamed as C++ in 1983
- most recent specification of language in ISO/IEC 14882:2014 (informally known as “C++14”)
- 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
- includes C as subset (two languages for price of one) and C is not going to disappear anytime soon
- general purpose
- easy to move from C++ to other languages but not in other direction
- many other popular languages derived/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

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

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

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

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)

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

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.

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

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

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:

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

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

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

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

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)

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

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)

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)

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

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

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

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.

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.

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

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.

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

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++, 2014.

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

Evolution of C++

- 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

- 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++, 2014.
- ISO JTC1/SC22/WG21 web site. `http://www.open-std.org/jtc1/sc22/wg21/`.

Section 2.2

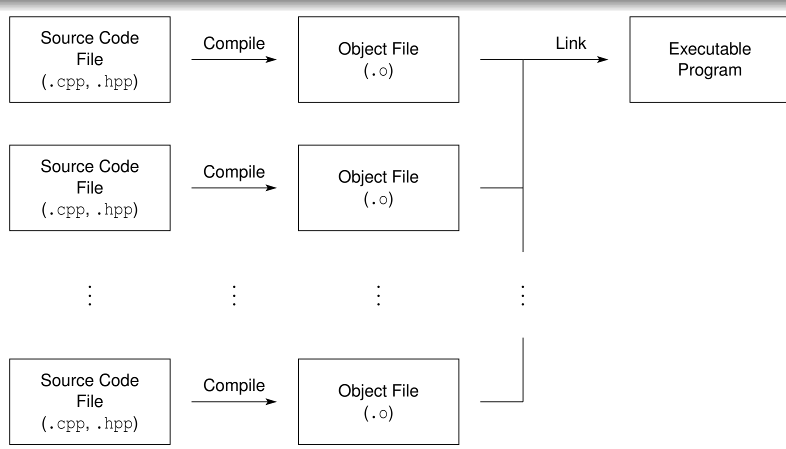
Getting Started

hello Program: hello.cpp

```cpp
#include <iostream>

int main(int argc, char** argv)
{
    std::cout << "Hello, world!\n";
    return 0;
}
```

- 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)
- `<<` is output operator

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* ...
- 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* ...
- particularly useful `g++` command-line options include:

`-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++14`	conform to C++14 standard
`-pthread`	enable concurrency support (via pthreads library)
`-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*
`-Wall`	enable all warning messages

- web page: `http://www.gnu.org/software/gcc`

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., make utility)

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 page: `http://www.gnu.org/software/make`

`make` Command

- 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*: one or more targets to be built
- 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
- *options*: additional options (see below)
- most common command-line options include:

`-n`	show commands that would be executed but do not actually execute them
`-f` *makefile*	use makefile *makefile*

Makefile

- makefile specifies targets and rules for building targets
- each rule in makefile has following form:

```
targets : prerequisites
	commands
	...
```

- 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

Makefile for `hello` Program

```
# CXX: The C++ compiler command.
# CXXFLAGS: The C++ compiler options.
# LDFLAGS: The linker options (if any)
CXX = g++
CXXFLAGS = -g -O
LDFLAGS =

# The all target builds all of the programs handled by
# the makefile.
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
```

Commentary on Makefile for `hello` Program

- chain of dependencies: `all` $\longrightarrow$ `hello` $\longrightarrow$ `hello.o` $\longrightarrow$ `hello.cpp`
- `all` target: builds all of the programs handled by the makefile (e.g., `hello`)
- `hello` target: compiles and links the hello program
- `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
- normally a target is associated with file of the same name
- phony target is target that is not associated with any file (e.g., `all`, `clean`)
- some special make variables:

`$@`	target of the rule
`$<`	name of the first prerequisite
`$^`	names of all of the prerequisites with spaces between them

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*

`gdb` Commands

`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.

gdb Commands

`list`
: Type the source code lines in the vicinity of where the program is currently stopped.

`break` *function*.
: Set a breakpoint at function.

`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

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 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 14882:2014 (informally known as "C++14")
- next version of standard expected in 2017
- 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

- identifiers used to name entities such as:
  - types
  - objects (i.e., variables)
  - functions
- valid identifier is sequence of one or more letters, digits, and underscore characters that does not begin with a digit
- 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

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**
- floating-point types:
  - **float**
  - **double**
  - **long double**
- void (i.e., incomplete/valueless) type: **void**
- null pointer type: `std::nullptr_t`

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
-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	**char16_t**
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

Character	Escape Sequence
backslash (\)	\\
question mark (?)	\?
single quote (')	\'
double quote (")	\"
octal number ooo	\ooo
hex number hhh	\xhhh

- examples of character literals:

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

Character Literals (Continued)

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

String Literals

- 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'`

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
-123
1'000'000'000'000ULL
0xdeadU
```

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:

```
-1.5
1.414
1.25e-8
```

- examples of **float** literals:

```
-1.5f
1.414F
1.25e-8f
```

- examples of **long double** literals:

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

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
- 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;
};
```


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]
```

- in C++ rarely ever need to use arrays
- use `std::array` or `std::vector` type instead (as this has many practical advantages over array)
- will revisit `std::array` and `std::vector` types later

Array Example

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

sizeof(**int**) is 4
array a starts at address 1000

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**
- if `p` is pointer, `*p` is object to which pointer refers (i.e., "`*`" is dereference operator)
- if `x` is object of type `T`, `&x` is 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

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

`sizeof(int)` is 4
`sizeof(int*)` is 4
`&i` is `((int*)1000)`
`&p` is `((int**)1004)`

Address		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
```

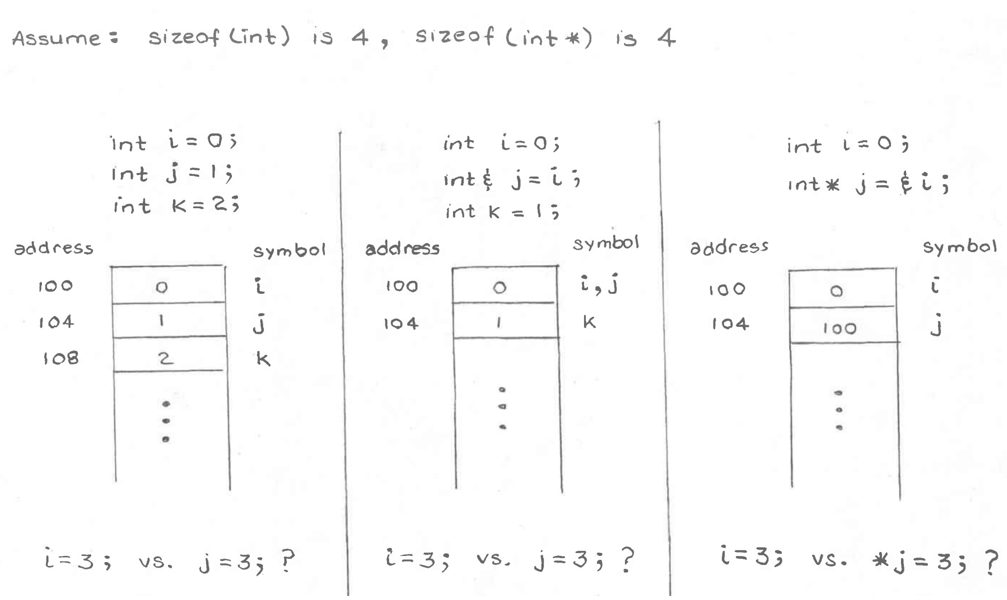
References Example

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

sizeof(**int**) is 4
&i is ((**int***)1000)

Address		Name
1000	42	i, j

Addresses, Pointers, and References



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

- **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)
- 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
```

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 cannot optimize away reads 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
- 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 most of these 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

Section 2.3.2

Operators and Expressions

Operators

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)`
allocate storage	**`new`** `T`
allocate storage (array)	**`new`** `T[a]`
deallocate storage	**`delete`** `a`
deallocate storage (array)	**`delete`**`[] a`

Operators (Continued 4)

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`

Operator Precedence

Precedence	Operator	Name	Associativity
1	`::`	scope resolution	none
2	`.`	member selection (object)	left to right
	`->`	member selection (pointer)	
	`[]`	subscripting	
	`()`	function call	
	`()`	value construction	
	`++`	postfix increment	
	`--`	postfix decrement	
	`typeid``()`	type name	
	`const_cast`	type cast	
	`dynamic_cast`	type cast	
	`reinterpret_cast`	type cast	
	`static_cast`	type cast	

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`	create object	
	`delete`	destroy object	
	`()`	cast	

Operator Precedence (Continued 2)

Precedence	Operator	Name	Associativity
4	`.*` `->*`	member selection (objects) member selection (pointers)	left to right

5	`*`
	`/`
	`%`
		multiplication
		division
		modulus
			left to right

6	`+` `-`	addition subtraction	left to right
7	`<<` `>>`	left shift right shift	left to right

8	`<`
	`<=`
	`>`
	`>=`
		less than
		less than or equal
		greater than
		greater than or equal
			left to right

9	`==` `!=`	equality inequality	left to right

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	`!=`

Expressions

- An **expression** is a sequence of operators and operands that specifies a computation.
- An expression has a value and a type.
- 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`

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
- 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
- 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 **sizeof** Operator

- **sizeof** operator is used to query size of object or type (i.e., amount of storage required)
- for 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 **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 2.3.3

Control-Flow Constructs: Selection and Looping

The **if** Statement

- allows conditional execution of code
- syntax has form:

  if (*expression*)
  $\quad$ $statement_1$
  else
  $\quad$ $statement_2$

- 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*)
  $\quad$ $statement_1$

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

  if ($expression_1$)
  $\quad$ $statement_1$
  else if ($expression_2$)
  $\quad$ $statement_2$
  ...
  else
  $\quad$ $statement_n$

The **if** Statement (Continued)

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

  if (*expression*) {
  $statement_{1,1}$
  $statement_{1,2}$
  $statement_{1,3}$
  ...
  } **else** {
  $statement_{2,1}$
  $statement_{2,2}$
  $statement_{2,3}$
  ...
  }

- 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: 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 **switch** Statement

- allows conditional execution of code based on value of integer expression
- 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 integer expression; $const\_expr_i$ is constant integer expression (e.g., `2`, `5+3`, `3*5-11`)
- 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: Example

```
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 **while** Statement

- looping construct
- syntax has form:

```
while (expression)
      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

```
while (expression) {
      statement1
      statement2
      statement3
      ...
}
```

- 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:

  for ($statement_1$; $expression$; $statement_2$)
  $statement_3$

- first, execute statement $statement_1$;
  then, while expression $expression$ is true, execute statement $statement_3$ followed by statement $statement_2$
- 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:

  for ($statement_1$; $expression$; $statement_2$) {
  $statement_{3,1}$
  $statement_{3,2}$
  ...
  }

- any objects declared in $statement_1$ 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:

  {
  $statement_1$;
  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
```


The 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 (int& x : array) {
    x *= 3;
}
```

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 2.3.4

Functions

Functions

- function has general syntax:

  return_type *function_name* (*argument_declarations*)
  {
  statements
  }

- or, alternatively, can use syntax with trailing return type:

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

- **return** statement exits function, passing specified return value back to caller
- code in function executes until **return** statement is reached or execution falls off end of function
- if function does not return any value, return type is **void**
- function parameters can be passed ***by value*** (i.e., function given copy of object from caller) or ***by reference*** (i.e., function given reference to object from caller)
- to pass parameter by reference, use ***reference type*** for parameter

Functions (Continued)

- can also use syntax with automatic deduction of return type:

  auto *function_name* (*argument_declarations*)
  {
  statements
  }

- if function has no **return** statement, return type deduced to be **void**
- otherwise, return type deduced to match type in expression of **return** statement
- if multiple return statements, must use same type for all **return** expressions

Parameters and Arguments

- **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

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

void increment(int& n) { // n is parameter
    ++n;
}

int main() {
    int i = 3;
    int j = square(i); // i is argument
    increment(j); // j is argument
}
```


Function: Examples

```
long factorial(long n) {
    long result = 1;
    while (n > 1) {
        result *= n;
        --n;
    }
    return result;
}

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

auto square(double x) -> double {
    return x * x;
}

auto square(double x) {
    return x * x;
}
```

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:

```
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

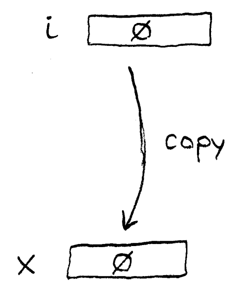
Pass-By-Value Versus Pass-By-Reference

- **pass by value**: function is given copy of object from caller
- **pass by reference**: function is given reference to object from caller
- if object being passed to function is ***expensive to copy*** (e.g., a very large data type), always faster to pass by reference
- if function needs to ***change value of object in caller***, must pass by reference
- example:

```
void increment0(int x) {
    ++x; // Increment x by one.
}

void increment(int& x) {
    ++x; // Increment x by one.
}

void func() {
    int i = 0;
    increment0(i); // i is passed by value
    // i still equals 0 (i was not incremented)
    increment(i); // i is passed by reference
    // i equals 1 (i was incremented)
}
```

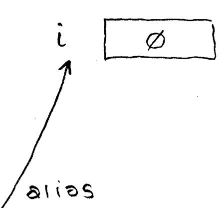
Pass By Value

```
void func() {
    int i = 0;
    increment0(i); // i unchanged
}

void increment0(int x) {
    ++x;
}
```



Pass By Reference

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

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



Pass-By-Reference Example

```
void func() {
    double a;
    RealVector v = getVector();
    a = average(v);
}

double average(RealVector& x) {
    double a;
    // initialize a here
    return a;
}
```

v

alias

- above code is incorrect

Pass-By-Reference Example (Continued)

```
void func() {
    double a;
const RealVector v = getVector();
    a = average(v);
}

double average(RealVector& x) {
    double a;
    // initialize a here
    return a;
}
```

v

alias

- code will not compile

Inline Functions

- **inline function**: 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
- can request function be made inline by including **`inline`** qualifier along with function return type
- inline typically used for ***very short functions*** (where overhead of calling function is large relative to cost of executing code within function itself)
- inline function definition must be visible at point of use
- 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 functions are implicitly inline
- constexpr function very restricted in what it can do (e.g., can only call constexpr functions)
- example:

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


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
- if set of best matches is empty (i.e., no match found), error
- if set of best matches has more than one element (i.e., multiple best matches found), error since call is ambiguous
- matches attempted in following order:
  1. exact match (only trivial conversions such as `T` to `T&`, `T&` to `T`, making **`const`**, making **`volatile`**)
  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

Argument Matching: Example

- example:

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

int i, j, k;
double a, b, c;
// ...
k = max(i, j); // calls max(int, int)
c = max(a, b); // calls max(double, double)
c = max(i, b); // ERROR: ambiguous
```


Section 2.3.5

Input/Output (I/O)

Basic I/O

- **#include** <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

- example:

```
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
- include header file `iomanip`
- 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)

I/O Manipulators Example

- example:

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

int main(int argc, char** argv)
{
    const double pi = 3.1415926535;
    const 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";

    return 0;
}
```

- 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 2.3.6

Miscellany

Namespaces

- mechanism for reducing likelihood of naming conflicts (i.e., attempt to use same identifier to have different meaning in various places in code)
- has general syntax:

```
namespace name {
        code
}
```

- all identifiers (e.g., variable names, function names, type names) declared/defined in code *code* (i.e., code contained in namespace body) made to belong to namespace *name*
- identifiers only have to be unique within a single namespace
- same identifier can be re-used in different namespaces
- scope-resolution operator (i.e., `::`) used to specify namespace to which particular identifier belongs
- **`using`** statement can be used to make identifiers declared in different namespaces appear as if they were in current namespace

Namespaces: Example

```
using std::cout;

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;
    }
}

mike::initialize(); // call initialize in namespace mike
fred::initialize(); // call initialize in namespace fred
using mike::initialize;
initialize(); // call initialize in mike namespace
```

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
```


Section 2.4

Classes

Section 2.4.1

Classes, Members, and Access Specifiers

Classes

- **class** is user-defined type
- class specifies:
  1. how objects of class are ***represented***
  2. ***operations*** that can be performed on objects of class
- class consists of ***zero or more members***
- members can be of various types: data member, function member, and others (e.g., type member)
- **data members** define representation of object of class
- **function members** (also called member functions) provide operations on such objects
- **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

Access Specifiers (Public and Private)

- can control ***level of access*** that users of class have to its members
- three levels of access: private, protected, and public
- **private**: member can only be accessed by other members of class and friends of class
- **public**: member can be accessed by any code
- **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 MyClass // The class is named MyClass.
{
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 MyClass {
    // ...
};
```

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


The `struct` Keyword

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

```
struct MyClass {
    // ...
};
```

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

Data Members

- 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.
};

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:
    double x; // The x component of the vector.
    double y; // The y component of the vector.
    void initialize(double x_, double y_);
};

void Vector_2::initialize(double x_, double y_) {
    x = x_;
    y = y_;
}

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 object of class as ***implicit parameter***

The **this** Keyword

- member function always has object of class as ***implicit parameter***
- implicit parameter passed in form of pointer using special variable called **this**
- normally, we do not explicitly write "**this**", however
- example:

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

MyClass x;
x.updateValue(5);
  // in MyClass::updateValue, variable this equals &x
```


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:
    typedef double Coordinate; // Coordinate type.
    Coordinate x; // The x coordinate of the point.
    Coordinate y; // The y coordinate of the point.
};

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 SomeClass; // forward declaration of SomeClass

class MyClass {
    // ...
    friend void myFunc(); // function myFunc is
                          // friend of MyClass
    friend class SomeClass; // class SomeClass is
                            // friend of MyClass
    // ...
};
```

Class Example

```
class MyClass {
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() {
    MyClass x;
    x.doNothing(); // OK: friend
    x.value = 5; // OK: friend
}

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


const Member Functions

- 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;
    }
    void setCount(int newCount) {
        count = newCount;
    }
    void incrementCount() {
        ++count;
    }
private:
    int count;
};

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

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`).
- C++ has always supported copying via copy constructors and copy assignment operators.
- C++11 adds formal support for moving (e.g., move constructors, move assignment operators).

Section 2.4.2

Constructors and Destructors

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`**)
- normally, constructor ***not called explicitly*** (exception is placement new)
- 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 parameters known as **default constructor**
- if ***no constructors*** specified, default constructor ***automatically provided*** that calls default constructor for each data member
- "default constructor" for built-in type does nothing

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

Copy Constructor

- constructor taking lvalue reference to object as first parameter that can be called with one parameter 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, one is ***automatically provided*** that copies each data member using the data member's copy constructor
- "copy constructor" for built-in type does bitwise copy

```
class Vector { // Two-dimensional vector class.
public:
    // ...
    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 w(v); // calls Vector(const Vector&)
Vector z = u; // calls Vector(const Vector&)
```


Move Constructor

- constructor taking rvalue reference to object as first parameter that can be called with one parameter 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 certain conditions are satisfied, a move constructor is automatically provided that moves each data member using the data member's move constructor

```
class Vector { // Two-dimensional vector class.
public:
    // ...
    Vector(Vector&& v) { // Move 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 x(); // declares function x that returns Vector
Vector y = x(); // calls Vector(Vector&&) if move not elided
```

Constructor Example

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

Vector u; // calls Vector(); u set to (0,0)
Vector v(1.0, 2.0); // calls Vector(double, double)
Vector w(v); // calls Vector(const Vector&)
Vector z = u; // calls Vector(const Vector&)
Vector x(); // declares function x that returns Vector
Vector y = x(); // calls Vector(Vector&&) if move not elided
```

- four constructors provided

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
- 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
- sometimes, automatically provided destructor ***will not*** have correct behavior

Destructor Example

- example:

```
class MyClass {
public:
    MyClass(int bufferSize) { // Constructor.
        // allocate some memory for buffer
        bufferPtr = new char[bufferSize];
    }
    ~MyClass() { // 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 the meaning of operator whose operands are one or more user-defined types through process known as **operator overloading**
- operators that can be overloaded:

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

- not possible to change precedence/associativity or syntax of operators
- meaning of operator specified by operator function, where name of function is **`operator`** followed by operator itself (e.g., **`operator`**+)

Operator Overloading (Continued 1)

- binary operator can be defined either by: 1) member function taking one argument, or 2) global function taking two arguments
- for any binary operator `@`, `a@b` can be interpreted as `a.operator@(b)` or `operator@(a, b)`
- unary operator can be defined either by: 1) member function taking no arguments, or 2) global function taking one argument
- for any unary operator `@`, `@a` can be interpreted as `a.operator@()` or `operator@(a)`
- for any postfix unary operator `@`, `a@` can be interpreted as `a.operator@(int)` or `operator@(a, int)` (where second argument only exists to distinguish postfix operators from prefix ones)
- if member and global functions both defined, argument matching rules determine which is called
- assignment, function-call, subscript, and member-selection operators must be overloaded as member functions
- if first operand of overloaded operator not object of class type, must use global function

Operator Overloading (Continued 2)

- 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	Global 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();
}

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
};

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: Global Versus Member Functions

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

```
class Complex { // Complex number type.
public:
    Complex(double re, double im) : re_(re), im_(im) {}
    double real() const { return re_; }
    double imag() const { return im_; }
    Complex operator+(const double&);
private:
    double re_; // The real part.
    double im_; // The imaginary part.
};

// Overload as global function.
Complex operator+(const Complex& a, const double& b) {
    return Complex(a.real() + b, a.imag());
}

// Overload as member function.
Complex Complex::operator+(const double& b) {
    return Complex(real() + b, imag());
}

// This can only be accomplished with global function.
Complex operator+(const 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 global or member function
                       // operator+(a, r) or a.operator+(r)
    Complex d = r + a; // must use global 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, one automatically provided that assigns to each data member using the data member's assignment operator
- 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***
- 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 re = 0.0, double im = 0.0) :
      re_(re), im_(im) {}
    Complex(const Complex& a) : re_(a.re_), im_(a.im_) {}
    Complex(Complex&& a) : re_(a.re_), im_(a.im_) {}
    Complex& operator=(const Complex& a) { // Copy assign
        if (this != &a) {
            re_ = a.re_; im_ = a.im_;
        }
        return *this;
    }
    Complex& operator=(Complex&& a) { // Move assign
        re_ = a.re_; im_ = a.im_;
        return *this;
    }
private:
    double re_; // The real part.
    double im_; // 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))
}
```


Section 2.4.4

Miscellany

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)
- 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.
```


Assignment Operator Example: Buffer

- example:

```
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

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 MyClass {
public:
    MyClass() {
        ++count; // one more object in existence
    }
    ~MyClass() {
        --count; // one less object in existence
    }
private:
    static int count; // total number of MyClass
                      // objects in existence
};

// Define (and initialize) count member.
int MyClass::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 MyClass {
public:
    // ...
    // convert degrees to radians
    static double degToRad(double deg) {
        return (M_PI / 180.0) * deg;
    }
private:
    // ...
};

double rad;
rad = MyClass::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

```
// 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(Vector&& v) : x_(v.x_), y_(v.y_) {}
    Vector& operator=(const Vector& v) {
        if (this != &v) {
            x_ = v.x_; y_ = v.y_;
        }
        return *this;
    }
    constexpr double x() const {return x_;}
    constexpr double y() const {return y_;}
    constexpr double squaredLength() const {
      return x_ * x_ + y_ * y_;
    }
    // ...
private:
    double x_; // The x component of the vector.
    double y_; // The y component of the vector.
};
```


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

```
#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
};
```


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 extractor should be readable by inserter)

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;
}
```


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(a + a);
    b = a + b;
}
```


Temporary Objects Example (Continued)

Original code:

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

Code showing temporaries (assuming no optimization):

```
int main() {
    Complex a(1.0, 2.0);
    Complex _tmp1(a + a);
    Complex b(_tmp1);
    Complex _tmp2(a + b);
    b = _tmp2;
}
```

Original code:

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

Code showing temporaries:

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

```
#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;
}
```

Return Value Optimization (RVO)

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

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

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

- without RVO: return value of function (which is local to function) is copied to new temporary object (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 one copy constructor and destructor call
- any good compiler should support RVO, although RVO cannot always be applied in all circumstances

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(); // copy construction
}
```

- 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., copy constructor and destructor calls eliminated)

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) {
        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

```
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 function parameterized by type
- syntax for template function has general form:

  template <*parameter_list*> *function*

- *parameter_list*: parameters on which template function depends; each parameter can be type (designated by **class** or **typename** keyword) or constant
- *function*: function declaration or definition
- example:

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

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

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

  function<*parameters*>

- example:

  `max<int>` refers to `int max(int, int)`

  `max<double>` refers to `double max(double, double)`

Template Functions (Continued)

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

Template Function: Example

- example:

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

// compute square of value
template <class 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;
}
```


Template Function Overloading Resolution

- overload resolution proceeds (in order) as follows:
  1. look for an exact match on (nontemplate) functions; if found call it
  2. look for function template from which function that can be called with exact match can be generated; if found, call it
  3. try ordinary overloading 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;
}

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

Section 2.5.2

Class Templates

Motivation for Class Templates

- consider almost identical complex number classes:

```
class ComplexDouble {
    ComplexDouble(double re = 0.0, double im = 0.0) : re_(re), im_(im) {}
    double real() const { return re_; }
    double imag() const { return im_; }
    // ...
private:
    double re_; // real part
    double im_; // imaginary part
};

class ComplexFloat {
    ComplexFloat(float re = 0.0, float im = 0.0) : re_(re), im_(im) {}
    float real() const { return re_; }
    float imag() const { return im_; }
    // ...
private:
    float re_; // real part
    float im_; // imaginary part
};
```

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

```
class Complex {
    Complex(T re = T(0), T im = T(0)) : re_(re), im_(im) {}
    T real() const { return re_; }
    T imag() const { return im_; }
    // ...
private:
    T re_; // real part
    T im_; // imaginary part
};
```

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

Class Templates

- **class template** is class parameterized on types and/or constants
- syntax has general form:

  template *<parameter_list> class*

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

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

template <class T, unsigned int size>
class MyArray { // definition
    // ...
    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)
- be careful when nesting angle brackets, since << and >> may be parsed as left shift and right shift operators in some contexts (e.g., prior to C++11 `std::vector<std::complex<double>>` would lead to parsing error)

Class Template: Example

- example:

```
template <class T>
class Complex { // complex number class template
public:
    Complex(T re = T(0), T im = T(0)) :
      re_(re), im_(im) {}
    T real() const {
        return re_;
    }
    T imag() const {
        return im_;
    }
    // ...
private:
    T re_; // real part
    T im_; // 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 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;
    return 0;
}
```

- 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>
class MyClass
{
public:
    struct Thing {
        T array[3];
    };
    Thing x;
    typedef T* Pointer;
    int i;
};
```

- names `Thing` and `Pointer` are 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**
- following code is invalid and will cause compile error:

```
template <class T>
class MyClass {
    std::vector<T> vec; // ERROR?
    std::vector<T>::iterator iter; // ERROR
    std::vector<T>::value_type val; // ERROR
    // ...
};
```

- must use code like following instead:

```
template <class T>
class MyClass {
    typename std::vector<T> vec;
    typename std::vector<T>::iterator iter;
    typename std::vector<T>::value_type val;
    // ...
};
```


Section 2.6

C++ Standard Library

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	runtime 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)

Commonly-Used Header Files (Continued 1)

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`)

Commonly-Used Header Files (Continued 3)

Numerics Library

Header File	Description
`cmath`	C math library, similar to `math.h` from C (e.g., `M_PI` on POSIX-compliant systems, `sin`, `cos`)
`complex`	complex numbers (e.g., `complex`)
`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)
fstream	file streams (e.g., fstream)
sstream	string streams (e.g., stringstream)
iomanip	manipulators (e.g., setw, dec)

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 2.6.1

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
- three sequence containers provided:
  1. `vector` (one-dimensional array)
  2. `deque` (double-ended queue)
  3. `list` (doubly-linked list)
- **associative container**: collection in which position of element in depends on its value or associated key and some predefined sorting criterion
- four associative containers provided:
  1. `set` (collection sorted by value, duplicate values not allowed)
  2. `multiset` (collection sorted by value, duplicate values allowed)
  3. `map` (collection of key/value pairs, sorted by key, duplicate keys not allowed)
  4. `multimap` (collection of key/value pairs, sorted by key, duplicate keys allowed)

Containers (Continued)

- some member functions typically provided by 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`	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

- example:

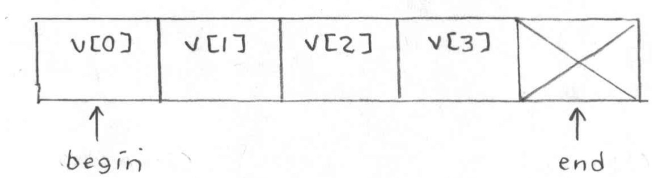
```cpp
#include <iostream>
#include <vector>

int main(int argc, char** argv) {
    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";

    return 0;
}
```

- program will produce output:

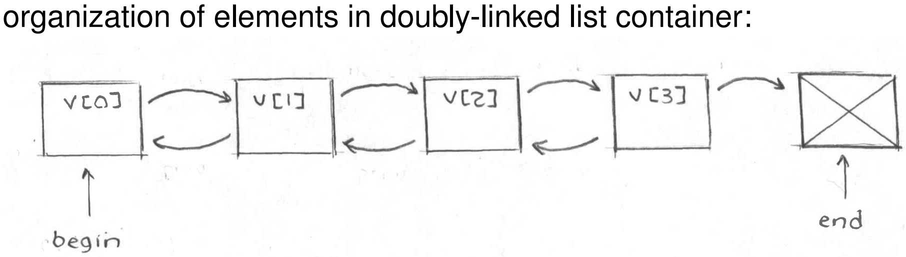
```
0 1 2 3 4 5 6 7 8 9
```


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:

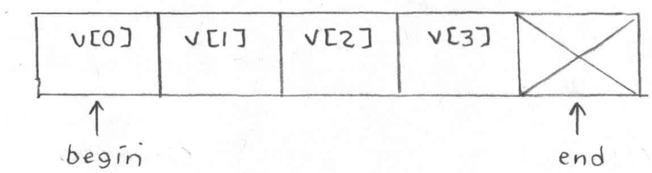


- organization of elements in doubly-linked list container:



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_map`
Bidirectional	Reads and writes forward and backward	`list`, `set`, `multiset`, `map`, `multimap`
Random access	Reads and writes with random access	array, `vector`, `deque`, `string`

Input Iterators

Expression	Effect
`T(a)`	copies iterator (copy constructor)
`*a` `a->m`	dereference as rvalue (i.e., read only); can only be dereferenced once
`++a`	steps forward (returns new position)
`a++`	steps forward (returns old position)
`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
`++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 as lvalue (i.e., write only); can only be dereferenced once
`++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]`	deference 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

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

int main(int argc, char** argv) {
    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.begin(); i != values.end(); ++i) {
        std::cout << *i << " ";
    }
    std::cout << "\n";

    return 0;
}
```


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`

Functions (Part 1)

Non-modifying Sequence Operations

Name	Description
`for_each`	apply function to range
`find`	find values in range
`find_if`	find element in range
`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

Functions (Part 2)

Modifying Sequence Operations

Name	Description
`copy`	copy range of elements
`copy_backward`	copy range of elements backwards
`swap`	exchange values of two objects
`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_copy`	copy range replacing value
`replace_copy_if`	copy range replacing value
`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

Functions (Part 3)

Modifying Sequence Operations (Continued)

Name	Description
`remove`	remove value from range
`remove_if`	remove elements from range
`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
`random_shuffle`	randomly permute elements in range
`partition`	partition range in two
`stable_partition`	partition range in two (stable ordering)

Functions (Part 4)

Sorting

Name	Description
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

Functions (Part 5)

Merge (operating 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

Heap

Name	Description
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 (Part 6)

Minimum/Maximum

Name	Description
min	get minimum of two values
max	get maximum of two values
min_element	get smallest element in range
max_element	get largest element in range
lexicographic_compare	lexicographic less-than comparison
next_permutation	transform range to next permutation
prev_permutation	transform range to previous permutation

Numeric Operations

Name	Description
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

Algorithms Example

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

int main(int argc, char** argv) {
    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::random_shuffle(values.begin(), values.end());
    std::cout << "random order: ";
    for (std::vector<int>::const_iterator i =
      values.begin(); i != values.end(); ++i)
        std::cout << *i << " ";
    std::cout << "\n";
    std::sort(values.begin(), values.end());
    std::cout << "sorted order: ";
    for (std::vector<int>::const_iterator i =
      values.begin(); i != values.end(); ++i)
        std::cout << *i << " ";
    std::cout << "\n";
    return 0;
}
```

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 object of type 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:
    // state information
    double factor_; // multiplicative factor
};

int main() {
    MultiplyBy mb(2.0);
    std::vector<double> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    // v contains 1 2 3
    std::transform(v.begin(), v.end(), v.begin(), mb);
    // v contains 2 4 6
}
```

Section 2.6.2

The `vector` Class Template

The `vector` Class Template

- one-dimensional array, 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 using custom storage allocator, 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`

Member Types

Member Type	Description
reference	Allocator::reference
const_reference	Allocator::const_reference
iterator	***random-access*** iterator type
const_iterator	const ***random-access*** iterator type
size_type	type used for measuring size (typically unsigned integer type)
difference_type	type used to measure distance (typically signed integer type)
value_type	element type
allocator_type	Allocator
pointer	Allocator::pointer
const_pointer	Allocator::const_pointer
reverse_iterator	reverse iterator type (reverse_iterator<iterator>)
const_reverse_iterator	const reverse iterator type (reverse_iterator<const_iterator>)

Member Functions (Part 1)

Construction, Destruction, and Assignment

Member Name	Description
constructor	construct vector (overloaded)
destructor	destroy vector
operator=	assign vector
assign	assign vector content
get_allocator	get allocator used by vector

Iterators

Member Name	Description
begin	return iterator to beginning
end	return iterator to end
rbegin	return reverse iterator to beginning
rend	return reverse iterator to end

Member Functions (Part 2)

Capacity

Member Name	Description
size	return size
max_size	return maximum size
resize	change size
capacity	return allocated storage capacity
empty	test if vector is empty
reserve	request change in capacity

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

Member Functions (Part 3)

Modifiers

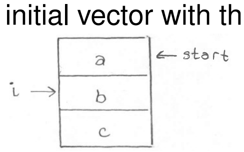
Member Name	Description
push_back	add element at end
pop_back	delete last element
insert	insert elements
erase	erase elements
swap	swap content of two vectors
clear	clear content

Invalidation of References, Iterators, and Pointers

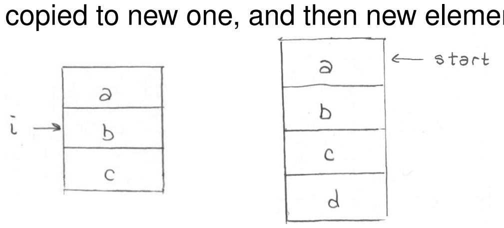
- **capacity**: total number of elements that vector could hold without requiring reallocation of memory
- any operation that causes reallocation of memory used to hold elements of vector invalidates ***all*** iterators, references, and pointers referring to elements in vector
- any operation that changes 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`, `resize`, `reserve`, **`operator`**`=`, `assign`, `clear` (`swap`?)

Iterator Invalidation Example

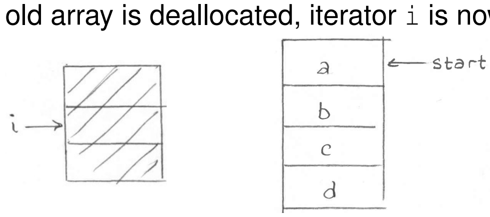
- `start` denotes pointer to first element in array holding vector elements
- `i` is iterator for `vector` (e.g., `vector<T>::const_iterator`, `vector<T>::iterator`)
- initial vector with three elements and capacity of three:



- `push_back(d)` results in new larger array being allocated, contents of old array copied to new one, and then new element added:



- old array is 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)
```


vector Example

```
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
// zero elements found.
int count = 0;
for (std::vector<double>::const_iterator i =
  values.begin(); i != values.end(); ++i) {
    if (*i == 0.0)
        ++count;
}
std::cout << "number of zero values: " << count <<
  "\n";
```

Section 2.6.3

The basic_string Class Template

The 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 char_traits<T>)
- Allocator: type of object used to handle storage allocation (unless using custom storage allocator, use default allocator<T>)
- 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 string, see http://www.cplusplus.com/reference/string/string

Member Types

Member Type	Description
traits_type	traits
value_type	character type (i.e., traits::char_type)
allocator_type	Allocator
size_type	type used for measuring size (typically unsigned integer type)
difference_type	type used to measure distance (typically signed integer type)
reference	Allocator::reference
const_reference	Allocator::const_reference
pointer	Allocator::pointer
const_pointer	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 (Part 1)

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
rbegin	return reverse iterator to reverse beginning
rend	return reverse iterator to reverse end

Member Functions (Part 2)

Capacity

Member Name	Description
`size`	get length of string
`length`	same as `size`
`max_size`	get maximum size of string
`resize`	resize string
`capacity`	get size of allocated storage
`reserve`	change capacity
`clear`	clear string
`empty`	test if string empty

Element Access

Member Name	Description
`operator``[]`	access character in string (no bounds checking)
`at`	access character in string (with bounds checking)

Member Functions (Part 3)

Modifiers

Member Name	Description
`operator``+=`	append to string
`append`	append to string
`assign`	assign content to string
`insert`	insert into string
`erase`	erase characters from string
`replace`	replace part of string
`copy`	copy sequence of characters from string
`swap`	swap contents with another string

Member Functions (Part 4)

String Operations

Member Name	Description
`c_str`	get C-string equivalent
`data`	get string data
`get_allocator`	get allocator
`find`	find content in string
`rfind`	find last occurrence of content in string
`find_first_of`	find character in string
`find_last_of`	find character in string from end
`find_first_not_of`	find absence of character in string
`find_last_not_of`	find absence of character in string from end
`substr`	generate substring
`compare`	compare strings

`std::string` Example

```
#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 (std::string::const_reverse_iterator i = s.rbegin();
    i != s.rend(); ++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";
}
```

Section 2.6.4

Time Measurement

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";
}
```


Part 3

More C++

Section 3.1

Exceptions

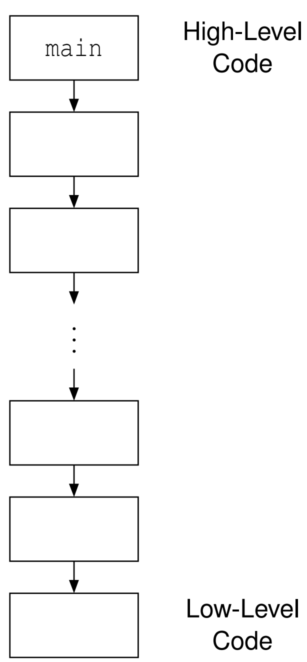
Section 3.1.1

Preliminaries

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

```
#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

```
#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 3.1.2

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

Standard Exception Classes

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

Standard Exception Classes (Continued)

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

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

Section 3.1.3

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)

- advisable for type of exception object to be user defined to reduce likelihood of different parts of code using type in conflicting ways
- if thrown object is class object, copy/move constructor and destructor must be accessible
- `throw x;` initializes temporary of type of `x` with `x`
- temporary may be moved/copied several times before caught

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";}
}
```


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
}
```

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 3.1.4

Exception Specifications

noexcept Specifier

- **noexcept** specifier in function declaration indicates whether or not function can throw exceptions
- two forms for **noexcept** specifier
- **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
```

noexcept Specifier (Continued 1)

- 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
}
```


noexcept Specifier (Continued 2)

- 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
- if exception due to parameter passing must be avoided, pass by reference or ensure **`noexcept`** move and/or copy constructor as appropriate
- if exception due to return by value must be avoided, ensure **`noexcept`** move or copy constructor as appropriate

`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 3.1.5

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 3.1.6

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, `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 be provided by:
  - destructors
  - 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 (provided container element type has nonthrowing move)
  - `insert` on `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 3.1.7

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 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 3.1.8

Smart Pointers and Other RAII Classes

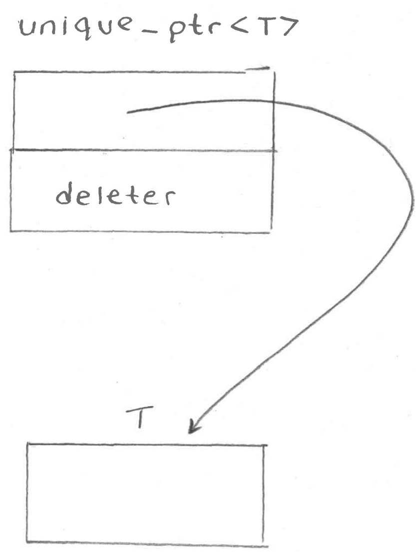
The std::unique_ptr Template Class

- 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 both efficiency and exception-safety reasons)

The `std::unique_ptr` Template Class (Continued)



Example: 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());
    // ...
}
```

Example: std::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());
}
```

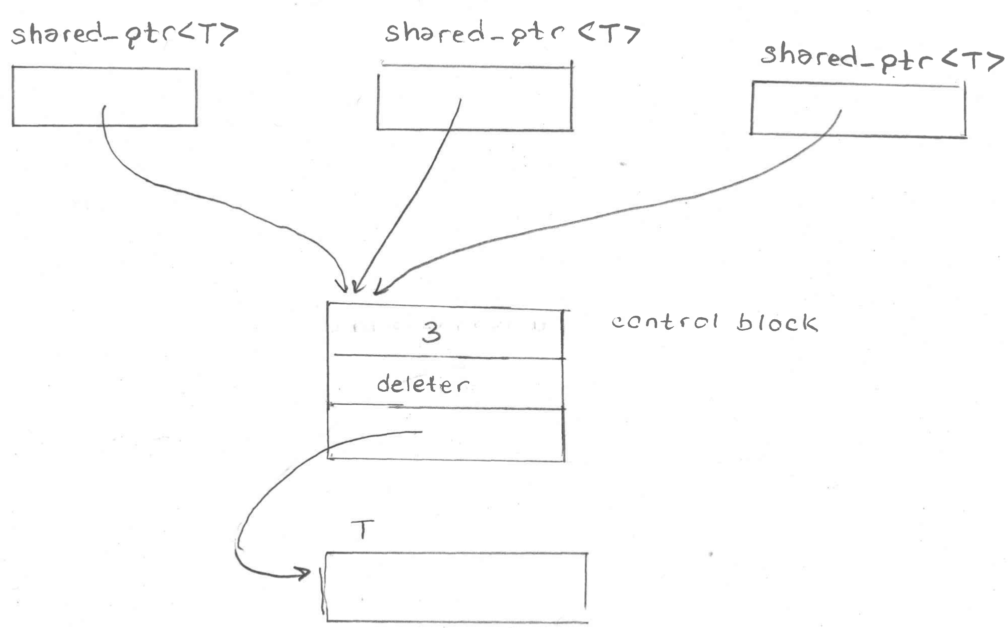

The std::shared_ptr Template Class

- `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 or last remaining owning `shared_ptr` object assigned another pointer via assignment or `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` 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

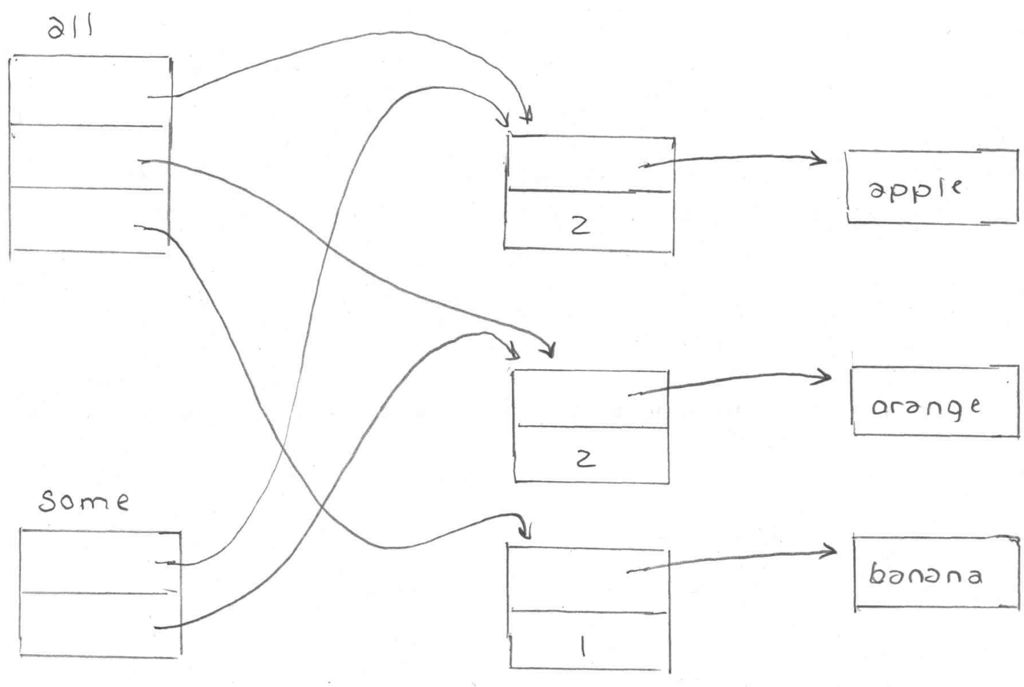
The std::shared_ptr Template Class (Continued)



Example: std::shared_ptr

```
#include <memory>
#include <vector>
#include <string>
#include <iostream>

using namespace std::literals;

int main() {
    std::vector<std::shared_ptr<std::string>> all;
    all.emplace_back(
      std::make_shared<std::string>("apple"s));
    all.emplace_back(
      std::make_shared<std::string>("orange"s));
    all.emplace_back(
      std::make_shared<std::string>("banana"s));

    std::vector<std::shared_ptr<std::string>> some(
      all.begin(), all.begin() + 2);

    for (auto& x : all) {
        std::cout << *x << " " << x.use_count() << "\n";
    }
}

/* output:
apple 2
orange 2
banana 1
*/
```

Example: `std::shared_ptr`



RAII Example: Stream Formatting Flags

```
#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

Section 3.1.9

Exception Gotchas

shared_ptr Example: Not Exception Safe

```
#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

```
#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 3.1.10

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 3.1.11

References

Copyright © 2015 Michael D. Adams

References I

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.

Talks I

1. Jon Kalb. Exception-Safe Code, CppCon, Bellevue, WA, USA, Sep 7–12, 2014. (This talk is in three parts.)
2. Jon Kalb. Exception-Safe Coding in C++Now, Aspen, CO, USA, May 13–18, 2012. (This talk is in two parts.)

Section 3.2

Rvalue References

Section 3.2.1

Preliminaries

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
```


Motivation Behind Rvalue References

- new language feature in C++11
- provide move semantics (i.e., mechanism for moving objects as opposed to copying them)
- allow for perfect forwarding
- perfect forwarding: passing a generic function's actual arguments to a second function without rejecting any arguments that can be passed to that second function, without losing any information about the arguments' cv-qualifications or lvalue/rvalue-ness, and without overloading
- in C++03, best approximations turn all rvalues into lvalues and require two overloads

Section 3.2.2

Copying and Moving

Propagating Values: Copying and Moving

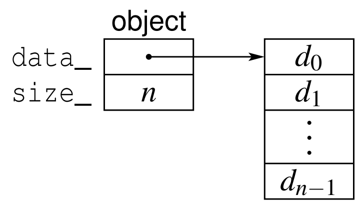
- Suppose that we have two objects of the same type and we want to propagate the value of one object (i.e., the source) to the other object (i.e., the destination).
- This can be accomplished in one of two ways: 1) copying or 2) moving.
- **Copying** propagates the value of the source object to the destination 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`).
- C++ has always supported copying via copy constructors and copy assignment operators.
- C++11 adds formal support for moving (e.g., move constructors, move assignment operators, `std::move`).

Vector Example: Moving Versus Copying

- 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:



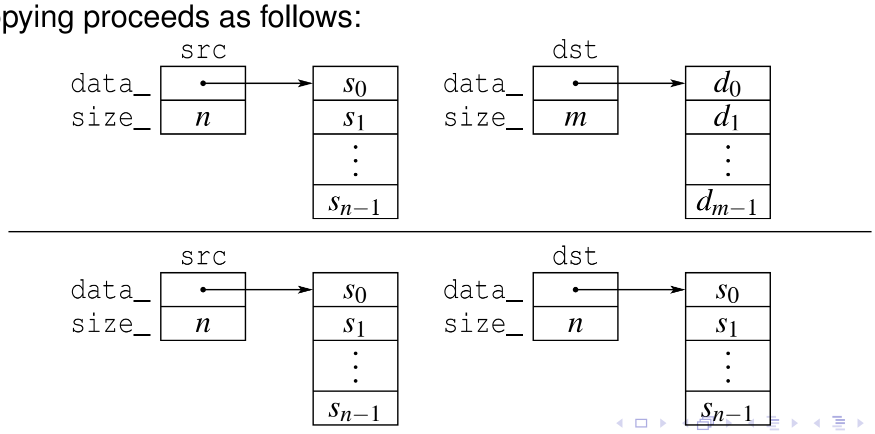
- How would copying be implemented?
- How would moving be implemented?

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:

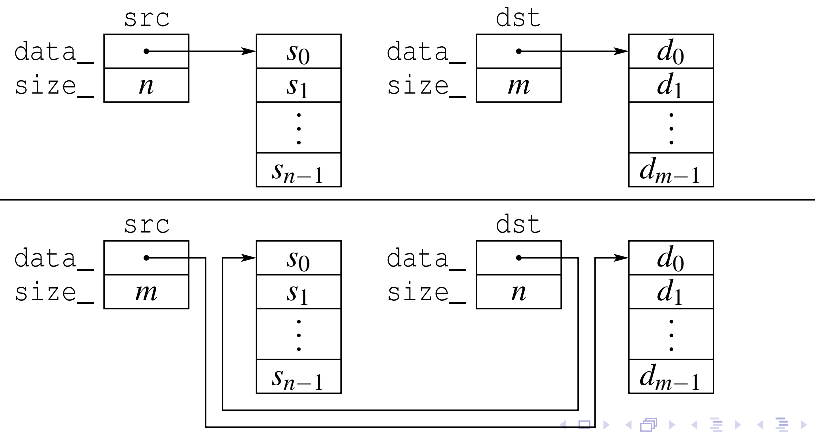


Vector Example: Moving

- 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 usually more efficient than copying, often by very large margin
- prefer moving to copying
- can safely replace copy by move when subsequent code does not depend on value of source object
- would be convenient if language could provide some way to automatically move (instead of copy) in situations where always guaranteed to be safe to do so
- for reasons of efficiency, desirable to provide mechanism whereby programmer can override normal behavior and force move (instead of copy) in situations where might not normally be safe but is safe due to additional knowledge of program behavior
- rvalues references provide this mechanism

Section 3.2.3

References and Expressions

References

- From the beginning, C++ has always had lvalue references (which used to be simply called 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
```

- C++11 added the notion of rvalue references.
- 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
```

- Lvalue and rvalue references are similar in that they are references (i.e., aliases).
- Lvalue and rvalues references differ only in their properties relating to reference binding and overload resolution.

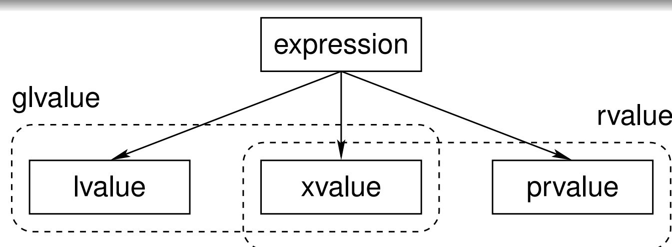
Expressions

- An **expression** is a sequence of operators and operands that specifies a computation.
- An expression has a value and a type.
- 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).
- Therefore, the language should never automatically move (instead of copy) when the source is an lvalue.
- 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.

Moving and Lvalues and Rvalues

- if source is rvalue (i.e., prvalue or xvalue), using move instead of copy is safe
- if source is lvalue, using move instead of copy is not guaranteed to be safe
- want language to automatically use move in rvalue case and copy otherwise

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 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

```
#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";
}
```


Exercise: Expressions

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

std::vector<int>&& func1(std::vector<int>& x) {
    return static_cast<std::vector<int>&&>(x);
    // x?
    // static_cast<std::vector<int>&&>(x)?
}

int main() {
    std::vector<int> x = {1, 2, 3};
    std::vector<int> y;
    int a;

    for (auto i = x.begin(); i != x.end(); ++i) {
        // x.begin()? ++i?
        *i += 5;
        // i?  *i?  *i += 5?
    }

    a = x[0];
    // x[0]?
    ++a; a++;
    // ++a?  a++?

    y = func1(x);
    // func1(x)?  y = func1(x)?
}
```

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*** require operands that are lvalues or rvalues or yield lvalues or rvalues is determined by the parameter types and return types of the operator function.
- member selection operator and lvalues/rvalues
- ternary conditional operator rvalue/lvalue and type of expression

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 3.2.4

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 string getValue();
string&& r1 = getValue(); // ERROR: drops const
const 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 string getConstValue();
string& r1 = getConstValue(); // ERROR: drops const
const 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&& r = i; // ERROR: cannot bind to lvalue
```

- 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

```cpp
#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

```cpp
#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
*/
```


Exercise: Overloading

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

typedef std::complex<double> Complex;
const Complex getConst() {return Complex(1.0, 2.0);}

void func1(const Complex& a) {std::cout << "1a\n";}
void func1(Complex& a) {std::cout << "1b\n";}
void func1(Complex&& a) {std::cout << "1c\n";}

void func2(const Complex& a) {std::cout << "2a\n";}
void func2(const Complex&& a) {std::cout << "2b\n";}

int main() {
    const Complex j(0.0, 1.0);
    Complex a(1.0, 1.0);
    const Complex* p = &j;
    func1(a);
    func1(j);
    func1(a * a);
    func1(getConst());
    func1(*p);
    func2(a + a);
    func2(j);
    func2(getConst());
}
```

- What output will this program produce when run?

Section 3.2.5

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&)
```


Why Rvalue References Cannot Bind to Lvalues (Revisited)

- If an rvalue reference could bind to an lvalue, this would violate the principle of type-safe overloading.

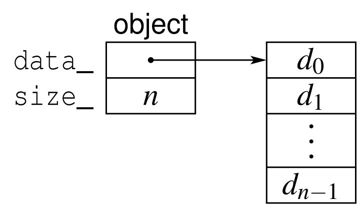
```
#include <iostream>
#include <string>

using namespace std;

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() {
    string s("Hello");
    Container<string> c;
    // What would happen here if lvalues
    // could bind to rvalue references?
    c.push_back(s);
    cout << s << "\n";
}
```

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
};
typedef Vector<std::complex<double>> Vec;
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
};
typedef Vector<std::complex<double>> Vec;
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

- 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>
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.

Section 3.2.6

Miscellany

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;
int& j = i;
auto& k = j; // 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`

- 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:

```
typedef int& IntRef;
int i = 0;
IntRef&& r = i; // r is int& (i.e., lvalue reference)
```

- Example:

```
typedef int& IntRef;
typedef int&& IntRefRef;
typedef const int&& ConstIntRefRef;
typedef const int& ConstIntRef;
typedef const IntRef& T1;     // T1 is int&
typedef const IntRefRef& T2; // T2 is int&
typedef IntRefRef&& T3;       // T3 is int&&
typedef ConstIntRef&& T4;     // T4 is const int&
typedef ConstIntRefRef&& T5; // 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&
```


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?
}
```


Why Distinguish Between Lvalues and Rvalues

- By distinguishing between lvalues and rvalues, we can write more efficient code.
- Scenario 1:

```
void doSomething(std::complex<double>& z) {
    // can the caller detect a change in z?
}

std::complex<double> z(1.0, 0.0);
doSomething(z);
```

- Scenario 2:

```
void doSomething(std::complex<double>&& z) {
    // can the caller detect a change in z?
}

doSomething(std::complex<double>(1.0, 2.0));
```

- A function parameter that is bound to a modifiable rvalue can be changed without any observable effect outside the function.
- This gives us more freedom in how we deal with the object whose change in value cannot be observed.
- For example, this freedom can be used to replace some copies by moves.

Section 3.3

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 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

```cpp
#include <iostream>

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

```cpp
#include <iostream>

struct Hello {
    void operator()() const {
        std::cout << "Hello, World!\n";
    }
};

int main() {
    Hello hello;
    hello();
}
```


Comparison Functor Example

```cpp
#include <iostream>
#include <algorithm>
#include <cstdlib>

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 abs(x) < abs(y);});
    for (auto x : v) std::cout << x << "\n";
}
```

```cpp
#include <iostream>
#include <algorithm>
#include <cstdlib>

struct abs_less {
    bool operator()(int x, int y) const
      {return abs(x) < 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
- personally I recommend explicitly listing all objects to be captured to avoid capturing objects accidentally (e.g., due to typos)
- to capture class members within member function, capture **`this`**
- capture of **`this`** probably best done by value (since likely to yield more efficient code)

`std::transform`

- (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

```cpp
#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";
}
```

```cpp
#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

```cpp
#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`

- `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

```
#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";
}
```

```
#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

```
#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;}
```

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`

```
#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

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?

```
#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";
}
```

Triangle Scan Conversion

- in SPLEL software, triangle scan conversion performed by `scan_triangle` template function
- declaration:

```
template <class T, class F>
void scan_triangle(T a_x, T a_y, T b_x, T b_y,
  T c_x, T c_y, unsigned mask, F scan_line);
```

- `scan_line` is functor to handle single horizontal scan within triangle
- `scan_line` has signature:

```
void scan_line(T y, T x_min, T x_max,
  unsigned left_mask, unsigned right_mask,
  unsigned mid_mask);
```

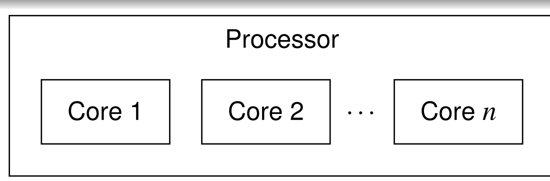

Section 3.4

Concurrency

Section 3.4.1

Preliminaries

Processors

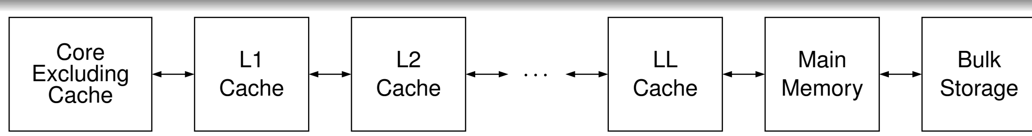


- A **core** is an independent processing unit that reads and executes program instructions, and consists of registers, an arithmetic logic unit (ALU), a control unit, and usually a cache.
- A **processor** is a computing element that consists of one or more cores, an external bus interface, and possibly 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.

Processors (Continued)

- 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 3.4.2

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 data can be shared 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 3.4.3

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

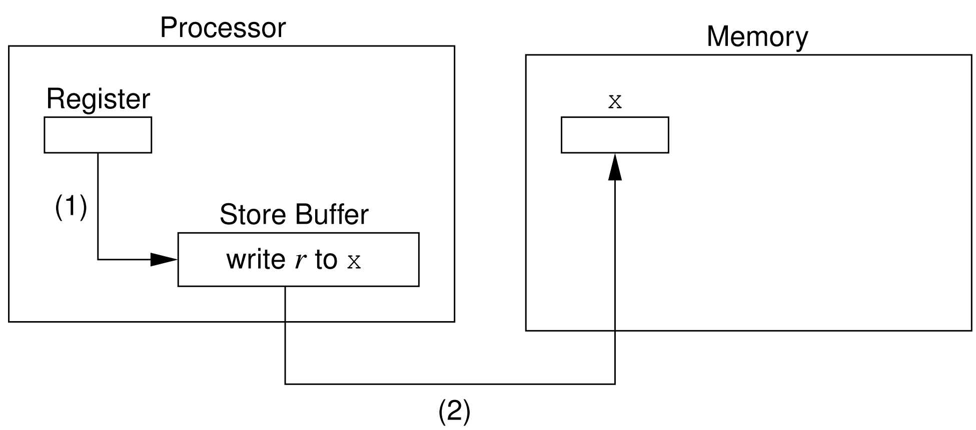
```
x = 1;
a = y;
```

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

```
x = 1;
a = y + z;
```

Thread 2 Code

```
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 3.4.4

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, and IDs are ***unique*** for all joinable `thread` objects and ***same*** for all unjoinable ones

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

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)

```cpp
#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();
}
```

Example: Thread-Function Argument Passing (Reference Semantics)

```
#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<int> 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<int> v{1, 2, 3, 4};

    // move semantics
    std::thread t1(doWork, std::move(v));
    t1.join();
}
```

Example: Moving Threads

```
#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>

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 allocated when thread begins and deallocated when thread ends
- 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 counter = 0;
    // equivalent to:
    // static thread_local counter = 0;
}
```


Example: Thread Local Storage

```
#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();}
}
```

Section 3.4.5

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.

Critical Sections

- 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)

```
#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 3.4.6

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 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 thread entity

Example: Avoiding Data Race Using Mutex (Counter) (mutex)

```
#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 mutex
destructor	destroy mutex

Example: Avoiding Data Race Using Mutex (Counter) (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<std::mutex> 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";
}
```


Example: Avoiding Data Race Using Mutex (IntSet) (lock_guard)

```
#include <thread>
#include <iostream>
#include <set>
#include <mutex>

class IntSet {
public:
    bool contains(int i) const {
        std::lock_guard<std::mutex> lg(m_);
        return s_.find(i) != s_.end();
    }
    void add(int i) {
        std::lock_guard<std::mutex> 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::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 when 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 mutex
destructor	destroy mutex
`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

std::unique_lock Members (Continued)

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<std::mutex> 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
- 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

```
#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;
        std::unique_lock<std::mutex> lock1(m_, std::defer_lock);
        std::unique_lock<std::mutex> 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) a.swap(b);
    });
    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<std::timed_mutex> 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_timed_mutex` Class

- `std::shared_timed_mutex` class provides shared mutex
- `shared_timed_mutex` also allows timeout for acquiring mutex

std::shared_timed_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
`try_lock_for`	try to acquire exclusive ownership of mutex without blocking
`try_lock_until`	try to acquire exclusive ownership of mutex without blocking
`unlock`	release exclusive ownership of mutex

std::shared_timed_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
`try_lock_shared_for`	try to acquire shared ownership of mutex without blocking
`try_lock_shared_until`	try to acquire shared ownership of mutex without blocking
`unlock_shared`	release shared ownership of mutex

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_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` can be used

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<std::shared_timed_mutex> 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<std::shared_timed_mutex> lock(counterMutex);
            c = counter;
        }
        {
            std::lock_guard<std::mutex> 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();
}
```

std::once_flag and std::call_once

- sometimes may want to perform action only once in code executed in multiple threads
- `std::once_flag` class represents flag used to track if action performed
- `std::call_once` template function calls function only once based on value of `std::once_flag` object
- useful for one-time initialization

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 3.4.7

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<std::mutex> 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<std::mutex> 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

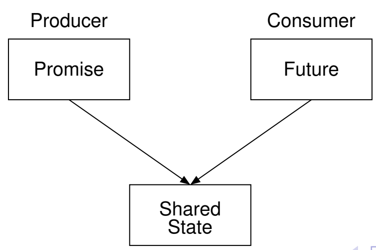
- with `std::condition_variable` class, `std::unique_lock<std::mutex>` class must be used for wait operation
- `std::condition_variable_any` class allows any mutex type (meeting certain basic requirements) to be used
- interface of `std::condition_variable_any` class similar to that of `std::condition_variable` class
- prefer `condition_variable` to `condition_variable_any` since former may be more efficient

Section 3.4.8

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

`std::promise` Members

Construction, Destruction, and Assignment

Name	Description
constructor	construct object
destructor	destroy object
`operator``=`	move assignment

`std::promise` Members (Continued)

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

std::future Members

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

Example: std::shared_future

```
#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<int> 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`)

```
#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<double> 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 3.4.9

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.

The `std::atomic_flag` Class

- `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<SpinLockMutex> 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`)

`std::atomic` Members

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

`std::atomic` Members (Continued 3)

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

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<bool> ready(false);
std::atomic<bool> done(false);
std::atomic<int> 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 3.4.10

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***.

```
        |   Thread 1 (on Core 1)     Thread 2 (on Core 2)
        |
  Time  |      x = 1; //A
        |
        |                                y = x; //B
        v
```

- 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 the 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

```
#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

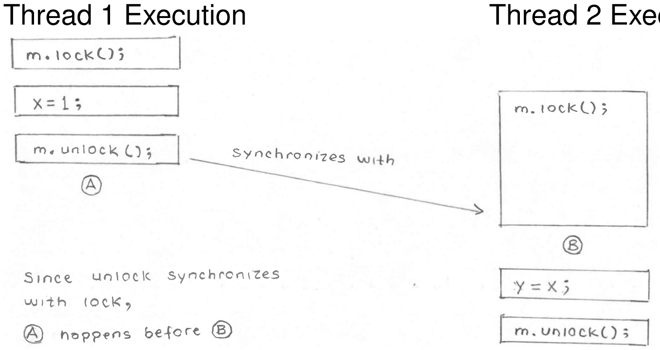
```
m.lock();
x = 1;
m.unlock();
```

Thread 2 Code

```
m.lock();
y = x;
m.unlock();
```

Thread 1 Execution

Thread 2 Execution



Memory Orders

- 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 Orders (Continued 1)

- 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:
  1. whether all writes to atomic objects become visible to ***all*** threads ***simultaneously***; and
  2. 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 Orders (Continued 2)

- 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***.

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 the variable `x` (in one thread) ***synchronizes with*** a sequentially-consistent operation on `x` (in another thread) that reads the value written by W.
- This model allows for relatively ***easy reasoning*** about code behavior.

Example: Sequentially-Consistent Order

```
#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 ***synchronizes with*** a read-acquire operation that reads the value stored 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 and `x2 == 0` and `y2 == 1` in thread 4

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<SpinLockMutex> 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.
- 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.
- 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 that reads the value stored 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<int> 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 using value of counter)

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).

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 weary 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 3.4.11

References

References I

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++. Excellent
2. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Burlington, MA, USA, 2008.
   A good reference for concurrent programming.

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. (This talk is in two parts.)
2. Herb Sutter. C++ Concurrency, C++ and Beyond, Asheville, NC, USA, Aug. 5–8, 2012.
3. Herb Sutter. Lock-Free Programming (Or, Juggling Razor Blades), CppCon, 2014. (This talk is in two parts.)
4. Hans-J. Boehm. Threads and Shared Variables in C++11. Going Native, Redmond, WA, USA, Feb. 2–3, 2012.
5. Mike Long. Introducing the C++ Memory Model. Norwegian Developers Conference, Oslo, Norway, Jun. 15–19, 2014.
6. Herb Sutter. Machine Architecture and You: Things Your Programming Language Never Told You. Northwest C++ Users' Group, Redmond, WA, USA, Sept. 19. 2007. `http://nwcpp.org/september-2007.html`.

Talks II

7. Pablo Halpern. Overview of Parallel Programming in C++, CppCon, Bellevue, WA, USA, Sept. 8, 2014.

Section 3.5

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.

Section 3.6

C++11 Overview

ISO/IEC 14882:2011 (C++11)

- ISO/IEC 14882:2011 (informally known as C++11) most recent version of C++
- adds numerous new features language features
- adds many new features to library
- not fully supported by all C++ implementations
- would not advise using C++11 features yet (due to lack of widespread support for all features and possible compiler/library bugs)
- nevertheless good to know about C++11 for when support is more widespread

Rvalue References

- new reference type: rvalue reference (denoted by `&&`)

```
int&& i = 0;
const int&& j = 1;
```

- rvalue references added to solve two problems:
  1. provide move semantics
  2. allow for perfect forwarding
- move constructors (create new object by move operation)
  for type `T`, looks something like `T::T(T&&)`
- move assignment operators (assignment by move operation)
  for type `T`, looks something like `T& T::operator=(T&&)`

Generalized Constant Expressions

```
#include <iostream>

constexpr int getTen() {return 10;}

// The size of the array must be a constant expression.
int array[getTen() + 20];

int main() {
    // Print the number of elements in the array.
    std::cerr << "array size " <<
      sizeof(array) / sizeof(int) << "\n";
}
```

Initializer Lists

```
#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();
}
```


Initializer Lists

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

int main() {

    // Pre-C++11
    // int a[] = {1, 2, 3};
    // std::vector<int> v1(
    //   a, a + sizeof(a) / sizeof(int));

    // C++11
    std::vector<int> v1 = {1, 2, 3};

    for (std::vector<int>::const_iterator i =
      v1.begin(); i != v1.end(); ++i) {
        std::cout << *i << "\n";
    }
}
```

Type Inference

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

int main() {
    std::vector<int> v;
    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
    }
    for (auto i = v.begin(); i != v.end(); ++i) {
        std::cout << *i << "\n";
    }

    auto i = 0; // i has type int
    auto j = i; // j has type int
    decltype(i) k = 3; // k has type int
    decltype((i)) m = k; // m has type int&
}
```


Range-Based For Loop

```cpp
#include <iostream>

int main() {
    int array[4] = {1, 2, 3, 4};
    // Double the value of each element in the array.
    for (int& x : array) {
        x *= 2;
    }
    // Print the elements of the array.
    for (const int& x : array) {
        std::cout << x << "\n";
    }
}
```

Range-Based For Loop

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

int main() {
    // Create a vector.
    int array[4] = {1, 2, 3, 4};
    std::vector<int> values(array, array + 4);
    // Double the value of each element in the vector.
    for (int& x : values) {
        x *= 2;
    }
    // Print the elements of the vector.
    for (const int& x : values) {
        std::cout << x << "\n";
    }
}
```


Lambda Expressions and Closures

```
#include <iostream>
#include <algorithm>

int main() {
    int array[] = {9, -2, 4 -1, 0, 1};
    const int n = sizeof(array) / sizeof(int);
    // Sort the data by the magnitude of the elements.
    std::sort(array, array + n,
      [](int x, int y){return abs(x) < abs(y);});
    // Print the sorted array.
    for (int i = 0; i < n; ++i) {
        std::cout << array[i] << "\n";
    }
}
```

```
struct __FunctorClass {
    bool operator()(int x, int y) {
        return abs(x) < abs(y);
    }
};

std::sort(array, array + n, __FunctorClass());
```

Suffix Return-Type Syntax

```cpp
#include <iostream>

auto isEven(int n) -> int {
    return !(n % 2);
}

int main() {
    std::cout << (isEven(4) ? "even" : "odd") << "\n";
}
```


Suffix Return-Type Syntax

```cpp
#include <iostream>

/* Not legal C++11
template<class Lhs, class Rhs>
decltype(lhs + rhs) addingFunc(const Lhs& lhs,
  const Rhs& rhs) {
    return lhs + rhs;
}
*/

template<class Lhs, class Rhs>
auto addingFunc(const Lhs& lhs, const Rhs& rhs) ->
  decltype(lhs + rhs) {
    return lhs + rhs;
}

int main()
{
    int x = 1;
    int y = 2;
    int z = addingFunc(x, y);
    std::cout << z << "\n";
}
```

Null Pointer Constant

```cpp
#include <iostream>

int main() {
    char* pc = nullptr; // OK
    int* pi = nullptr;  // OK
    bool b = nullptr;  // OK. b is false.
    // int i = nullptr;   // ERROR
}
```

- **`nullptr`**: null pointer
- `nullptr_t`: null pointer type

Strongly-Typed Enumerations

```cpp
#include <iostream>

// Enumeration values have class scope.
// (apple versus Fruit:apple)
// Integral type for enumeration can
// be specified and defaults to int.
enum class Fruit : char {
    apple,
    orange,
    grape
};

int main() {
    Fruit f = Fruit::apple;
    Fruit g = Fruit::grape;
    std::cout << ((f == g) ? "same" : "different") <<
      "\n";
}
```

Right Angle Bracket

```cpp
#include <vector>

int main() {
    // With a pre-C++11 compiler, the line
    // immediately following this comment
    // will generate an error, since
    // the ">>" is parsed as operator>>.
    std::vector<std::vector<int>> v;
}
```


Alias Templates

```cpp
#include <iostream>

// typedef double Real;
using Real = double;

template <class Type1, class Type2, int N>
class MyClass {
public:
    void print() { std::cout << N << "\n"; }
};

// Illegal: typedef must be fully specified type.
// template<class T>
// typedef MyClass<int, T, 15> MyClass3;

template <class T>
using MyClass2 = MyClass<int, T, 15>;

int main() {
    Real x = 3.14;

    MyClass<int, int, 5> a;
    a.print();

    MyClass2<int> b;
    b.print();
}
```

Variadic Templates

```
#include <iostream>

// Compute the maximum of one element (trivial).
int maximum(int n) {
    return n;
}

// Compute the maximum of two or more elements.
template<typename... Args>
int maximum(int n, Args... args) {
    return std::max(n, maximum(args...));
}

int main() {
    std::cout << maximum(7, 2, 1, 3, 6) << "\n";
}
```


New String Literals

```
#include <iostream>

int main() {
    // UTF-8
    const char s1[] = u8"Hello, World.";
    // UTF-16
    const char16_t s2[] = u"Hello, World.";
    // UTF-32
    const char32_t s3[] = U"Hello, World.";
}
```

User-Defined Literals

```
#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)
```


Explicitly Defaulted and Deleted Special Member Functions

```
class Thing {
public:
    Thing() = default;
    Thing(const Thing&) = delete;
    Thing& operator=(const Thing&) = delete;
    ~Thing() = default;
    void doSomething(int) { /* ... */ }
    void doSomething(double) = delete;
};

int main() {
    Thing first;
    Thing second;

    // The following line will produce an error.
    // Thing third(first);

    // The following line will generate an error.
    // second = first;

    first.doSomething(0);

    // The following line will generate an error:
    // first.doSomething(0.0);
}
```

Other C++11 Language Changes

- modification to the definition of plain old data
- extern templates
- object construction improvement
- explicit overrides final
- explicit conversion operators
- unrestricted unions
- standardized support for multithreading
- thread-local storage
- long long int (**`long long int`**); at least 64 bits
- static assertions (**`static_assert`**)
- allow sizeof to work on members of classes without an explicit object
- attributes

Multithreading Issues

- assume `x` and `flag` both initially zero
- processor 1:

```
while (flag == 0)
    ;
print(x); // What value is printed here?
```

- processor 2:

```
x = 42;
flag = 1;
```

- might (incorrectly) believe that value of `x` printed will always be 42, but this is not the case
- instructions can be executed out of order due to:
  1. reordering of instructions by compiler's optimizer
  2. out-of-order instruction execution on processor
- processor 2 may execute store operations out of order so that `flag` is written before writing `x`, in which case 0 is printed
- processor 1 may execute loads operations out of order so that `x` is read before `flag` is checked, in which case 0 may be printed

C++11 Library Changes

- many library changes
- most changes from C++TR1 adopted
- special mathematical functions from C++TR1 not included

Fixed-Size Array (std::array)

```
#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";
}
```

Other Library Changes

- threading facilities (e.g., thread class)
- tuple types
- hash tables (unordered associative containers) (e.g., `std::unordered_set`, `std::unordered_multiset`, `std::unordered_map`, `std::unordered_multimap`)
- regular expressions (`std::regex`, `std::match_results`, `std::regex_search`, `std::regex_replace`)
- general-purpose smart pointers (i.e., `std::unique_ptr`)
- pseudorandom number generation
- wrapper reference
- polymorphic wrappers for function objects
- type traits for metaprogramming
- method for computing the return type of function objects (e.g., `std::result_of`)

References: Miscellany

- Standard C++ Foundation. `http://isocpp.org`
- Comp.lang.c++.moderated Newsgroup. `http://groups-beta.google.com/group/comp.lang.c++.moderated`
- ACCU Overload Journal. `http://accu.org/index.php/journals/c78/`
- Artima Developer: The C++ Source. `http://www.artima.com/cppsource`

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 (e.g., `char* p = 0; *p;`)
- attempting to modify a string literal or any other const object (excluding mutable data members)
- signed integer overflow
- evaluating an expression that is not mathematically defined (e.g., `double z = 0.0; double x = 1.0 / z;`)
- not returning a value from a value-returning function (other than `main`)
- multiple definitions of the same entity
- performing pointer arithmetic that yields a result outside the boundaries of an array (e.g., `int v[10]; int* p = &v[0]; --p;`)
- using pointers to objects whose lifetime has ended
- left-shifting values by a negative amount (e.g., `int i = 1; i << (-3);`)
- shifting values by an amount greater than or equal to the number of bits in the number (e.g., `int i = 1; i << 10000;`)
- using an automatic variable whose value has not been initialized (e.g., `int i; i++;`)

Examples of Unspecified Behavior

- order in which arguments to a function are evaluated; 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

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());
}
```

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 == false)` is equivalent to `s.fail()`
- `(s == true)` 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)

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_;
};
```

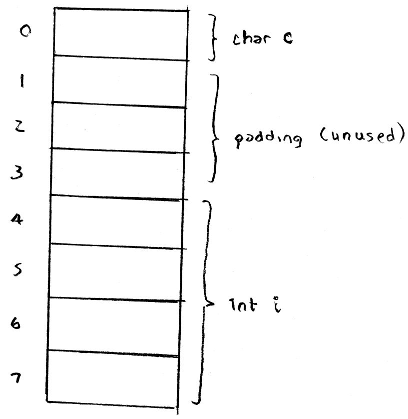

Sizeof Class Versus Sum of Member Sizes

- 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)



- `struct Thing { char c; int i; };`
- suppose `sizeof(int)` is 4 and `alignof(int)` is 4
- implementation adds padding to structure so that `int` data member is suitably aligned (i.e., offset is multiple of 4)

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);
}
```

std::string Concatenation

- 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);
```


std::vector<std::string> Insertion

- 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();
```

Classes Holding Multiple Resources

- 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_;
};
```


Avoid Returning By Const Value

- 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.

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)

Section 4.3

Idioms

Proxy Classes

- 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";
}
```


Part 5

Programming

Section 5.1

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);
```

Error Handling

- 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.

Miscellany

- Avoid ***multiple returns paths*** in functions. That is, only have one point of exit from a function.
- 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 all ***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

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 != 0);
    // 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 5.2

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 Number Representations

- **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., $5 \cdot 10^0 = 0.5 \cdot 10^1 = 0.05 \cdot 10^2$)

Floating-Point Number Representations (Continued)

- 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

IEEE 754 Standard (Continued)

- 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 $E_{\text{min}} \leq E \leq E_{\text{max}}$) 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	$E_{\text{min}} \leq E \leq E_{\text{max}}$	—	$(-1)^s 2^E (1+f)$
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

IEEE 754 Encoding Examples

- 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.

Additional Reading

- D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991
- IEEE Std. 754-1985 — IEEE standard for binary floating-point arithmetic, 1985
- IEEE Std. 754-2008 — IEEE standard for floating-point arithmetic, 2008

Part 6

References

Limits of Knowledge

- 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
  - David Abrahams (one of the founding members of Boost)

C++ References I

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

C++ References II

4. 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.
   Excellent
5. 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.
   Excellent
6. 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. Excellent
7. 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. Excellent

C++ References III

8. 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). Excellent
9. 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++. Excellent
10. 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++. Excellent

C++ References IV

11. 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. Excellent
12. 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. Excellent
13. 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. Excellent
14. 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. Excellent

C++ References V

15. 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.
    Said-To-Be Excellent
16. 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. Excellent

Other C++ References I

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.
6. A. Alexandrescu. *Modern C++ Design*. Addison Wesley, Upper Saddle River, NJ, USA, 2001.

Other C++ References II

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`.

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. C++ FAQ: `http://www.parashift.com/c++-faq/`
6. Newsgroup comp.lang.c++.moderated
7. `http://en.cppreference.com`
8. `http://www.cplusplus.com`
9. Stackoverflow: `http://stackoverflow.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.
- But in addition to all of the preceding things:

  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).