CISC 187 Textbook

Release 4.0.4

Dave Parillo

Jul 07, 2026

LEGAL AND FOREWORD

CHAPTER

ONE

FRONT MATTER

1.1 Copyright Notice

1.2 GNU Free Documentation License

Version 1.3, 3 November 2008

1.2.1 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1.2.2 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

1.2.3 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may

accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

1.2.4 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

1.2.5 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled

"History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties - for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

1.2.6 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

1.2.7 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

1.2.8 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

1.2.9 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

1.2.10 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

1.2.11 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See https://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

1.2.12 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

1.3 Foreword

By Alan Smithee

Don't read this book. Really.

Work this book.

1.4 Preface

There are many books available if you want to learn to program in C++. Does the C++ community really need yet another textbook? Apparently, I think the answer to that question is 'yes'. Why?

- Many excellent books are not appropriate for new language learners.
- Many books, although excellent when written, have not been updated as the language has changed.

  Significant language changes over the past 10 - 15 years justifies teaching new features and retiring old ones.
- Many textbooks claiming to be 'for C++' are not much more than bare bones ports from other languages.

I take exception especially to this last group, as they claim to be C++ textbooks (and charge good money for it), however, they do not teach good **idiomatic** C++.

What do I mean by **idiomatic** C++?

In a nutshell, it means writing C++ code that follows the conventions of the language. Because C++ is a language that has undergone significant change since its inception, the C++ of 1998 is not the C++ of today. Perhaps a short story would help make the point more clearly:

English As She is Spoke

English As She Is Spoke is the common name of a 19th-century book written by Pedro Carolino.

A well-meaning person, to be sure; his goal was to create a Portuguese-to-English phrase book. One minor obstacle stood in Pedro's way. He apparently spoke no English.

Undaunted, and armed with both an earlier Portuguese-French phrase book and a French-English dictionary, he embarked on his quest. He managed to create a book that fails utterly in teaching common English language phrases and idioms to anyone.

He did not intend to write a humor book, but that was the result.

Some of his classic translations from the section *Idiotisms and Proverbs*:

It want to beat the iron during it is hot.
Strike while the iron is hot

A horse baared don't look him the tooth.
Don't look a gift horse in the mouth.

The stone as roll not heap up not foam.
A rolling stone gathers no moss.

> Nobody can add to the absurdity of this book, nobody can imitate it successfully, nobody can hope to produce its fellow; it is perfect.
>
> —Mark Twain, 1883.[1]

[1] Mark Twain, "Introduction to The New Guide of the Conversation in Portuguese and English" (1883) p. 239.

While most C++ textbooks that have been marginally ported from other languages cannot hope to achieve the level of silliness reached by English As She Is Spoke, they still fail their readers. That is, they fail to teach C++ as C++. C++ is not C, Python, Java, Visual Basic, or anything else.

Good, idiomatic, C++ embodies the philosophy behind the C++ Core Guidelines[2], the first few of which are:

- Ideas are expressed in code
- Code in ISO Standard C++
- Prefer compile-time checking to run-time checking

The goal of this book is to describe the current version of the language in a clear, concise style, supplemented by meaningful activities where appropriate. This book is expressly written for students in my course. Others may also find it useful. Users of this book are expected to be 'beginning-intermediate' programmers. You already have a semester of C or C++ completed and want to learn more.

The goal of this book is **not** to describe every language feature in C++. Why? Primarily because C++ is a large language, but also because you don't need to know *all* of C++ to be an effective C++ programmer. If you really want to read about every language feature, then read cppreference.com.

Programming is not a "spectator sport". It is something you do, something you participate in. It would make sense, then, that the book you use to learn programming should allow you to be active.

This book is meant to provide you with an interactive experience as you learn to program. You can read the text, watch videos, and write and execute code. In addition to simply executing code, there are visualizations which allow you to control the flow of execution, and watch variables as they are created and destroyed, in order to gain a better understanding of how the program works.

Different presentation techniques are used where appropriate. In other words, sometimes it might be best to read a description of some aspect of a programming language. On a different occasion, it might be best to execute a small example program. An important goal is to provide multiple options for covering the material in each section. Hopefully, you will find your understanding is enhanced because you are able to experience content in more than just one way.

1.4.1 Overview

Conceptually, the book is separated into 6 modules:

- Introductory topics and review
- Functions
- Sorting
- Classes
- Containers
- Iterators and Trees

Note there are 2 sections before Chapter 1 intended as review and reference material.

The first 2 chapters cover the introductory topics:

Chapter 1
: A review of basic topics from first semester C++. It also introduces how to compile software using CMake.

Chapter 2
: An introduction to `string` and `vector`. Although vector gets more attention later in the book, these 2 types are the workhorses of a great deal of real-world C++ code and it's important that you understand how to use them as early as possible.

[2] Bjarne Stroustrup and Herb Sutter, C++ Core Guidelines

The next 4 chapter explore functional programming in C++. While some simple structs are used, they are limited to basic POD's that would generally be compatible with C.

Chapter 3
: An introduction to functions, including function parameters, namespaces, and scopes. The keyword `const` is introduced and will be revisited over several chapters.

Chapter 4
: An introduction to function overloads and function templates.

Chapter 5
: Introduces pointers in general, the semantics for passing pointers as parameters to functions and the relationship between pointers and arrays. Modern C++ alternatives to raw pointers, sunch as `unique_ptr` are discussed.

Chapter 6
: Introduces recursion, properties of recursive data structures, and introduces the Binary Tree ADT as a recursive data structure.

The next chapter introduces sorting algorithms. Sorting appears before the main class-design chapters so that algorithmic behavior and performance can be discussed while the examples still use familiar sequence containers and functions.

Chapter 7
: Introduces several common sorting algorithms and compares their behavior, costs, and performance characteristics.

The next 3 chapters introduce the foundations of classes in C++. There is more to explore, but other topics related to classes are explored in the context of linear and associative data structures in the later chapters.

Chapter 8
: Introduces classes, starting with how a C++ differs from a *POD*, or *Plain Old Data* in C, continuing with constructors, the importance of class interfaces and their implementation, using `const` in classes, and class enumerations.

Chapter 9
: Expands on the material introduced in Ch 8, discussing more constructor overloads and operator overloads in classes.

Chapter 10
: Focuses on class design concepts: composition and inheritance, multiple inheritance, the Unified Modeling Language (UML), and abstract base classes and interface classes.

The next 2 chapters explore more C++ class concepts using container classes as a springboard.

Chapter 11
: Introduces class templates and begins introducing concepts the rest of the book builds on as it begins to explore the containers in the standard library and uses them as an opportunity to explore more advanced programming topics in general. Introduces container initialization lists and overloading the array index operator.

Chapter 12
: Introduces copy and move semantics in C++. This chapter described copy constructors, then explores lvalues, rvalues, and rvalue references as a way to help explain move constructors and move assignment in C++. A brief introduction to using allocators, what they are for, and how to add them to a container.

The next 2 chapters introduce linear containers and iterators.

Chapter 13
: Introduces the Stack and Queue ADT's and explains how they are implemented in C++ by adapting other containers. The Adapter design pattern is introduced.

Chapter 14
: An exploration of linked lists. The primary motivation for discussing lists at this point is to use a list implementation as a reason for needing a class to have a supporting iterator class. The Iterator design pattern is introduced.

The final chapters explore trees, hash tables, and algorithms. Both data structures **and** algorithms are needed to make useful programs.

Chapter 15
Introduction to Trees, Binary Search Trees, sets and maps, and their application to searching and sorting.

Chapter 16
Introduces hash tables, hash functions, collision handling, and the performance tradeoffs involved in hash table design.

Chapter 17
The STL algorithms, the basic model in the standard library connecting containers, iterators, and algorithms.

1.4.2 How to Contribute

Readers are encouraged to fork the source repository for this book on GitHub. Improve it and submit a pull request. The document GitHub-Forking is an excellent place to get started. Read this first.

Every pull request will be evaluated for inclusion and if not included, I will let you know why.

Footnotes

CHAPTER

TWO

PRELIMINARIES

2.1 C and C++ concepts

This chapter reviews some fundamental topics that everyone should have mastered or at least covered in your first semester course.

They are included here because in my experience these topics have generally **not** been covered during the first semester.

2.1.1 Hello, world!

Welcome to CISC187 and welcome to this book.

This book is intended for students taking CISC187 at SD Mesa. You should already have a solid understanding of the material presented in the vast majority of first semester C and C++ courses.

This entire course focuses on writing code. Even this book provides opportunities for you to write, compile, and evaluate code on may pages. Often the interactive code is on a separate tab, for example:

Example

A C++ construct that may be new to you is the range-based for loop. When your goal is to iterate over all the elements in a container or array, a range-for loop is easier to write and understand.

```
for (auto value: data)           // for each value in data
{
  std::cout << "value is " << value << '\n';
}
```

Introduced in C++11, a range-for loop extracts values from containers one at a time and assigns them to a variable.

Run It

A complete example program using the code explained on the *Example* tab.

```
// compiled with: -std=c++11

#include <iostream>

int main () {
  int data[] = { 1, 2, 3, 5, 8 };
  for (auto value: data)           // for each value in data
  {
    std::cout << "value is " << value << '\n';
```

(continues on next page)

(continued from previous page)

```
  }
  return 0;
}
```

Feel free to modify this code and re-run it to see what happens when you change it.

The remaining section on this page list some things I expect everyone to know on the first day of class. Review this material and if anything looks unfamiliar then read the linked content and ask questions.

Source files and header files

One of the primary goals of this course is to begin creating programs more complex than those written previously. One of the core skills required when writing large programs is to split different parts of the program source into separate files.

You may have only had to do this a few times previously, but you should know by now:

- What are the differences between source and header files?
  - Why do they exist?
  - What are *header guards*?
  - What is `#pragma once`?

  See preprocessor/include for more information.

Compilation vs. linking

- What happens during compilation? Linking?
- How to use function *main()*, *argc*, and *argv*
- cout and the meaning of statements like:

```
#include <iostream>
#include <stdio.h>

int main() {
  std::cout << "Hello C++!" << std::endl;
  puts("Hello C!");
  printf("Hello Alice!\n");
  printf("Hello %s!\n", "Bob");
}
```

You may not have seen printf and puts before. They are output functions C++ inherits from C. Normally, in C++ we use stream I/O functions and classes, but the old C functions are still there if you need them.

Built-in types, variables and operations

You should already be familiar with declaring fundamental types (`int`, `char`, `double`, `unsigned`, etc.). You should also know that other fixed width integer types exist (`int16_t`, `uint64_t`, etc.) even if you haven't used them very much. You should also be familiar with the basic math operations and operators (+, -, =, ==, etc.). Including the shortcut operators (++, +=, etc.). We will be expanding our knowledge of operators and operations extensively during this course.

You should know the difference between *declaring*, *initializing*, and *assigning a value* to a variable. It is (sometimes) valid to assign variables of one type to those of a different type, for example, *double x = 12;* assigns the integer *12* to the *double x*. This is a **widening conversion** and is always safe. The opposite of a widening conversion is a **narrowing**

conversion. A narrowing conversion frequently involves the loss of information. Most compilers will warn about narrowing conversions even in cases where they are allowed.

Keep in mind that a common source of error in programs is unintentional narrowing conversions that occur during math operations. For example:

Example

What is the output, given the following?

```
double value = 3 / 2;
```

Run It

```
#include <iostream>

int main() {
  double value = 3 / 2;
  std::cout << "The value is: " << value << ".\n";
}
```

Fix this program so that the correct value is displayed.

You should know how to explicitly cast fundamental types from one type to another. Most people should be familiar with the `static_cast` form:

```
auto almost_pi = static_cast<int>(3.14159);
```

Some people may have also learned the C-style cast:

```
auto almost_pi = (int)3.14159;
```

These two casts are roughly equivalent, but the first is preferred. We discuss why later in the book. We will be learning other ways to explicitly cast that are a bit more consistent with C++11's more uniform initialization syntax.

Finally, you should know the basic keywords of the language, at least those common to both C and C++, and legal identifier names for functions and variables.

User-defined types

Although you may not have done any object oriented programming yourself, you probably have used objects, even if you weren't aware of it. The C++ standard provides many classes. Two of the oldest classes handle stream formatted input and output: cin and cout.

You should have already encountered code like:

```
std::string  name;
std::cout << "Enter your name: ";
std::cin  >> name;
std::cout << "Hello," << name << "!\n";
```

You may have been taught the basics of string and vector. It is hard to do much (non-embedded) C++ programming without ever using either. A bit like writing a paragraph in English without using the letter 'e'. Try that sometime!

We will be working with `std::string` and `std::vector` often in this course, so if you haven't used them yet, don't worry - you will.

File input and output

I expect you to know how to use some form of file input and output, whether it is the C-style printf and scanf, or the C++-style input and output file streams: ofstream and ifstream. Both are serviceable, have their own advantages and disadvantages. This course emphasizes *contemporary* C++ and encourages the use of C++ generally, but sometimes `printf` is a perfectly acceptable alternative to `cout`.

Don't panic.

While file I/O is not a primary focus of this course, you will be expected to employ basic I/O in labs and projects.

C style IO

This example uses `printf` and `scanf` to interact with the standard console input and output.

```
#include <cstdio>

int main () {
  char name[20];
  scanf("%s", name);
  printf("Hello, %s!\n", name);
}
```

Try This!

Feel free to change the input to something else to see what happens.

What happens if we enter a name longer tahn our buffer size?

C++ style IO

This example uses `std::cout` and `std::cin` to interact with the standard console input and output.

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

int main () {
  std::string name;
  std::cin >> name;
  std::cout << "Hello, " << name << "!\n";
}
```

Feel free to change the input to something else to see what happens.

C file IO

Reading from a file to access external data:

```
#include <cstdio>

int main() {
   // assuming the file 'poem' exists in the current directory
   FILE* ptr = fopen("poem","r");
   if (ptr == NULL) {
     printf("Unable to open poem.");
```

(continues on next page)

(continued from previous page)

```
    return 1;
  }
  char c;
  // read the text file one byte (char) at a time
  while (fscanf(ptr,"%c",&c) == 1) {
    putchar(c);
  }
  return 0;
}
```

C++ file IO

Reading from a file to access external data:

```
#include <fstream>
#include <iostream>

int main () {
  // assuming the file 'poem' exists in the current directory
  std::ifstream is("poem");
  char c;
  // read the text file one byte (char) at a time
  while (is.get(c)) {
    std::cout << c;
  }
  is.close();
  return 0;
}
```

Try This!

Change this program to read from the poem file one **line** at a time instead of reading single characters at a time.

Hint: change `char` to `std::string` and use getline instead of `get`.

poem

Listing 1: poem

```
          Jabberwocky

"Beware the Jabberwock, my son!
  The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
  The frumious Bandersnatch!"

He took his vorpal sword in hand:
  Long time the manxome foe he sought --
So rested he by the Tumtum tree,
  And stood awhile in thought.
```

(continues on next page)

(continued from previous page)

```
And, as in uffish thought he stood,
  The Jabberwock, with eyes of flame,
Came whiffling through the tulgey wood,
  And burbled as it came!

One, two! One, two! And through and through
  The vorpal blade went snicker-snack!
He left it dead, and with its head
  He went galumphing back.

And, has thou slain the Jabberwock?
  Come to my arms, my beamish boy!
O frabjous day! 'Callooh! Callay!'
  He chortled in his joy.

              -- Lewis Carroll, 1871
```

Keep in mind that each of the I/O examples presented are just one way to solve these problems. Each of them could have been written differently and achieved exactly the same goals. For example, reading files one byte at a time is not generally the most efficient file read solution, but it is extremely simple. At this point, we are not concerned with a thorough treatment of input and output, rather we are just reviewing major concepts and showing the differences between C standard I/O and C++ I/O.

Statements and branching

Writing basic statements and conditionally executing code, or executing blocks of code repeatedly, are fundamental skills common to all programming languages.

Everyone should be **extremely familiar** with writing `if`, `switch`, `for`, and `while` blocks.

You should have used combinations of statements and branching to perform tasks perhaps as complex as:

- Computing an amortization table
- Computing population growth
- Parsing text

For example, loops and conditionals make it easy to print all the odd numbers between 1 and 100, inclusive.

```
#include <iostream>
using std::cout;

int main() {
  cout << "Odd numbers:\n";
  for (int num = 1; num <= 100; ++num) {
    if (num % 2 != 0) {
      cout << '\t' << num << '\n';
    }
  }
}
```

Fixing errors in code

You should know the difference between basic types of errors:

- *Compile-time errors*
- Link-time (linker) errors
- *Runtime errors*
- *Semantic errors*

I expect some basic experience using a debugger in whatever programming environment you may have used previously.

If not, refer to the section *Debugging*.

Note

If **any** of the material in the preceding sections sounds unfamiliar, then

- Consider working through the week 1 example source code in your home directory on buffy. Look in the `examples` directory.
- Review the material from your first semester text

More to Explore

- range-based for loop, for loops, and while loops
- if
- *Debugging*
- Jeff Atwood's blog: Code smells

2.1.2 Types

In C++, objects, references, functions, and expressions all have a property called *type*, which both restricts the operations that are permitted for those entities and provides semantic meaning to the otherwise generic sequences of bits.

A *type* is a collection of values. For example, the Boolean type consists of the values `true` and `false`. The integers also form a type. An integer is a *simple type* because its values contain no subparts. In comparison, a bank account object is a *compound type*. A bank account typically contains several pieces of information such as name, address, account number, and account balance.

The C++ type system consists of the following types:

- Fundamental types
  - The type `void`
  - The type `nullptr_t`
  - Arithmetic types, including floating point types and integral types.
  - Floating point types: `float`, `double`, and `long double`.

  - Integral types: `bool`; character types such as `char`, `signed char`, `unsigned char`, `char16_t`, `char32_t`, and `wchar_t`; signed integer types such as `short int`, `int`, `long int`, and `long long int`; and unsigned integer types such as `unsigned short int`, `unsigned int`, `unsigned long int`, and `unsigned long long int`.

- Compound types
  - Array types
  - Pointer types
  - Reference types
  - Function types
  - Class types
  - Enumeration types

Fundamental representations

Machines view stored data as sequences of bits that can be manipulated by specific instructions. These instructions include shifts, logical operations, integer and floating arithmetic and more. While machines manipulate these bit sequences efficiently, people do not.

High level languages introduce *abstractions* to simplify working with data. Abstractions allow the data stored in memory to be managed not as a sequence of bits, but as an integer, a floating point number, a boolean, a character, or something else. In C++, the abstractions for the built in numeric types (plus the types `void` and `nullptr_t`) are referred to as *fundamental types*.

It is natural to think that the types defined for the computer are the same used in mathematics. That assumption would be incorrect. It is important to understand that all of the numeric types are *abstractions*, that is they are approximations of the basic concepts learned in school.

What exactly do we mean? The unsigned integer type represents the set of whole numbers. In mathematics, this set is infinite: given any number N a new integer M can be generated by simply setting $M = N + 1$. However, this basic axiom does not apply to integers stored on a computer because and integer on a computer occupies a fixed size: that is, an `unsigned` is stored in a finite number of bits. Another way to say this is that every numeric fundamental type has both a minimum and maximum allowed value. This situation is very different from what you are used to in math class.

One consequence of a fixed size is that basic axioms of mathematics are not always true. For example, the ordinary associative law of addition:

$$(x + y) + z = x + (y + z)$$

is satisfied only if both $|x + y| <= maxint$ and $|y + z| <= maxint$.

In practice, the upper limit of an `unsigned` is so large that it is not a limiting factor in many ordinary numeric calculations. When the upper limit is exceeded, the results are typically dramatic. *Overflow* errors are typically obvious. Overflow errors may be more common when the type chosen is too small for its intended purpose.

A slightly more involved example is the problem of how to represent signed integers. In mathematics, negative numbers are represented by prefixing them with a minus ("-") sign. However, on a computer, there are only bit sequences. Once again, programmers are faced with the problem of abstraction. What is the 'best' way to represent a signed integer that is both efficient and unambiguous?

As it turns out, there is no 'best' way. In fact, there are many ways to solve this problem. They all have their own trade-offs, but over time, three commonly used representations have been used:

- Sign and magnitude
- One's complement

- Two's complement

While the *two's complement* representation is now nearly universal, this was not always the case. In fact, during the 60's and 70's, debates raged about the best number format representations.

Sign and magnitude

Arguably the easiest to understand as the representation is very similar to how signed numbers are represented in mathematics. The number's sign is stored in a *sign bit*: setting that bit (often the most significant bit) to 0 means a positive number or positive zero, and setting it to 1 is for a negative number or negative zero. The remaining bits in the number indicate the magnitude (or absolute value).

In eight bits the magnitude can range from `0000000` (0) to `1111111` (127). Numbers ranging from -127 to +127 can be represented once the sign bit (the eighth bit) is added. For example, -43 encoded in an eight-bit byte is `10101011` while 43 is `00101011`. A consequence of using signed magnitude representation is that there are two ways to represent zero, `00000000` (0) and `10000000` (-0).

Some early binary computers (e.g., IBM 7090) use this representation for integers, perhaps because of its natural relation to common usage. However, it is slower and requires more complicated hardware than one's complement or two's complement representations.

Example: Sign & Magnitude

Signed magnitude remains the most common way of representing the exponent in floating point values. It is the official IEEE floating point number format.

The program on the run tab isn't meant to be completely understood, but does demonstrate the bits stored in the exponent and mantissa for some floats.

Feel free to modify `main()` and provide your own values.

```
  float u = -1.0;
  print_float(u);
  print_float(0.5);
```

The important item here is that even today, floating point numbers have two different representations for `0`, a side effect of the sign and magnitude representation.

Run It

The following C program [Aspnes2014] prints the sign, exponent, and mantissa of a few small numbers.

```
#include <stdio.h>
#include <math.h>
#include <values.h>

/* endianness testing */
const int EndianTest = 0x04030201;

#define LITTLE_ENDIAN() (*((const char *) &EndianTest) == 0x01)

/* extract nth LSB from object stored in lvalue x */
#define GET_BIT(x, n) ((((const char *) &x)[LITTLE_ENDIAN() ?\
          (n) / CHARBITS :\
          sizeof(x) - (n) / CHARBITS - 1] >> ((n) % CHARBITS)) & 0x01)

#define PUT_BIT(x, n) (putchar(GET_BIT((x), (n)) ? '1' : '0'))
```

(continues on next page)

(continued from previous page)

```
16
17 void print_float_bits(float f) {
18   int i;
19
20   i = FLOATBITS - 1;
21   PUT_BIT(f, i);
22   putchar(' ');
23   for(i--; i >= 23; i--) {
24     PUT_BIT(f, i);
25   }
26   putchar(' ');
27   for(; i >= 0; i--) {
28     PUT_BIT(f, i);
29   }
30 }
31
32 void print_float(float f) {
33   printf("%2g = ", f);
34   print_float_bits(f);
35   putchar('\n');
36 }
37
38 int main(int argc, char** argv) {
39   float u = -1.0;
40   float v = u * 0.0;
41   float w = v * -1.0;
42   puts(" x = S   exp     mantissa");
43   print_float(u);
44   print_float(v);
45   print_float(w);
46   print_float(1.0);
47   print_float(0.1);
48   print_float(0.5);
49   print_float(2.0);
50   print_float(3.0);
51   print_float(6.0);
52   print_float(8.0);
53   print_float(13.0);
54   print_float(21.0);
55 }
```

Try This!

Run and carefully examine the results of the previous program.

How is the value of `0.1` different?

Try to list at least 1 error this might cause in your programs?

Try other values and see which ones have exact floating point representations and which do not. The numbers that *may have* exact representations are called the dyadic rationals.

One's complement

Alternatively, a system known as ones' complement can be used to represent negative numbers. The ones' complement form of a negative binary number is the bitwise NOT applied to each bit in the number. That is, each negative number is the "complement" of its positive counterpart.

C++ provides a complement operator ~ for this purpose. The complement of the 5 bit binary number `11100`, is `00011`, which is the number 3.

Note that like sign-and-magnitude representation, ones' complement has two representations of 0: `00000000` (+0) and 11111111 (-0).

A few of the 4 bit one's complement integers are:

Decimal	-7	-2	-1	-0	0	1	2	7
Binary	1000	1101	1110	1111	0000	0001	0010	0111

One's complement is important both historically, and because it is used to generate two's complement numbers. No modern computers store one's complement signed integers.

Two's complement

A variation of one's complement that avoids the "two zeroes problem" is two's complement. In two's complement, negative numbers are represented by the bit pattern which is one greater (in an unsigned sense) than the one's complement of the positive value. A short 3 bit table comparing one's and two's complement looks like this:

3-bit pattern	100	101	110	111	000	001	010	011
One's complement	-3	-2	-1	-0	0	1	2	3
Two's complement	-4	-3	-2	-1	0	1	2	3
Unsigned value	4	5	6	7	0	1	2	3

Negating a number is done by inverting all the bits (in other words taking the one's complement) and then adding one to that result.

Virtually all modern computers store signed integer values using the two's complement representation. The following program demonstrates the consistency of the two's complement representation.

Example: bitset

A bitset is a simple way to see the sequence of ones and zeros for an integral type.

A `bitset` is a templated type that must be initialized with a size:

```
std::bitset<8> x;
```

The size determines the number of bits stored and doesn't need to match the size of the variable. A value can be provided when declared:

```
std::bitset<4> x = 9;
auto y = std::bitset<4>(9);
```

Run It

```
#include <bitset>
#include <iostream>
using std::cout;
using std::bitset;

int main() {
  int x = -1;
  int y = ~x;  // one's complement of x
  int z = x * 0;

  cout << "x: " << x << "\t(" << bitset<4>(x) << ")\n";
  cout << "y: " << y << "\t(" << bitset<4>(y) << ")\n";
  cout << "z: " << z << "\t(" << bitset<4>(z) << ")\n";
  cout << "~z: " << ~z << "\t(" << bitset<4>(~z) << ")\n";
}
```

Overflow

Why bother with all this obscure discussion about type representation now? Because it is very useful to know how numbers are actually stored when debugging your code. Sometimes a value cannot be represented in the limited number of bits allowed. Examples:

```
unsigned, 3 bits:    8 would require at least 4 bits (1000)
sign mag., 4 bits:   8 would require at least 5 bits (01000)
```

When a value cannot be represented in the number of bits allowed, we say that *overflow* has occurred. Overflow occurs when doing arithmetic operations.

```
example:          3 bit unsigned representation

  011 (3)
+ 110 (6)
---------
   ?  (9)     it would require 4 bits (1001) to represent
              the value 9 in unsigned rep.
```

Overflow

Mistakes happen. Someday, you will write some code that will overflow the amount of space allocated for the type. It helps to understand what is going on if you recognize what overflow looks like for various types.

There is a standard C header to define the exact size of an integral type:

```
#include <cstdint>
```

Once included, a family of fixed size integral types are available:

```
int8_t
uint8_t

int16_t
uint16_t
```

and many others. Use these types like any other standard type you are already familiar with.

Where `int` sizes may vary from machine to machine, the size of `int8_t` is guaranteed to be an 8 bit signed integer always.

For this reason, the programming guidelines of many organizations prefer fixed size integral types to the generic `int` and `long`.

Run It

Ask yourself what is the largest number we can store in an 8 bit **signed** integer, then predict the output of the following program before you run it.

```
#include <iostream>
#include <cstdint>

int main() {
  int8_t x = 128;
  int8_t y = 1;

  std::cout << "x + y = " << (x + y) << '\n';
}
```

Video

This video the representation of int on most computers.

- How data is stored in computer's memory.
- Size and range of int
- Signed and unsigned int
- How negative numbers are stored in binary.

Video

Video URL: https://www.youtube.com/watch?v=zxb8DvLUqcM

Overflow involving signed integral types overflows into the most significant bit, effectively changing the sign, as in the preceding example.

Unsigned integers do not, technically, overflow in that they do not become negative. They are after all, unsigned. They **do** however still 'wrap around' and the result can be that adding two values can result in a value smaller than the sum of the two values.

Try This!

Change the types in the previous program from `int8_t` to `uint8_t`. With no other changes what do you expect the output to be?

Keeping the types as `uint8_t`, change the value of `x` to 256 and run the program again.

What do you expect to see? What did you see?

If any of this was suprising, consider making more changes:

- Try other unsigned types, short, or char
- What happens if you assign `-1` to a variable of unsigned type?

Preventing overflow errors

The C and C++ compilers do not check math overflow for you. You can turn on compiler warnings that can inform you about possible overflow or type conversion problems. A program can report when it happens at runtime, but by that point, the error has already occurred. It is generally preferred to check for possible overflow before attempting a calculation that might overflow.

The following program checks for addition overflow and reports an error if addition overflow would occur.

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

int main () {
  int x = std::numeric_limits<int>::max();
  int y = x - 9;

  if (std::numeric_limits<int>::max() - x < y) {
    std::cerr << "addition failed: result is too big\n";
  } else {
    // addition is safe
    std::cout << "x+y = " << (x+y) << '\n';
  }

}
```

Similarly, checks for multiplication overflow and exponentiation overflow could use the following checks:

```
if (std::numeric_limits<int>::max() / x < y) {
 std::cerr << "multiplication failed: result is too big\n";
}

// number of bits in uint32_t
const num_bits = 32;
if (log2(base)*exponent > sizeof(uint32_t) * num_bits) {
  std::cerr << "exponentiation failed: result is too big\n";
}
```

Other operations

We haven't discussed overflow for subtraction and division.

Do we have to worry about overflow for these two operations?

Why or why not?

Compound types

Most of the compound types will be covered in greater detail later in this book. Those that aren't covered later are discussed now.

Array types

An **array** is a block of memory that holds one or more objects of a given type. Declare an array by giving the type of object the array holds followed by the array name and the size in square brackets:

```
int a[3];               // array of 3 ints
int b[3] = {4, 5, 6};   // array of 3 ints
int c[]  = {1, 2, 3};   // array of 3 ints
char name[64];          // array of 64 characters
```

Arrays **can** be constructed from any fundamental type, pointers, pointers to members, classes, enumerations, or from other arrays (in which case the array is said to be multi-dimensional). Arrays **cannot** be constructed from references, functions, or abstract class types.

Objects of array type cannot be modified as a whole: even though they are lvalues (e.g. an address of array can be taken), they cannot appear on the left hand side of an assignment operator

```
int a[3] = {4, 5, 6};   // array of 3 ints with initial values
int (*b)[3] = &a;       // OK to make an array of pointers using an address
int c[3];               // array of 3 ints with default initial values
c = a;                  // Error. Can't assign to an array
c[0] = a[0];            // OK.
```

Arrays

It's easy to forget that arrays always supply a default value if one is not provided.

This is true even if only part of a multi-dimensional array is initialized with values. Consider the statement:

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

What is declared?

What is initialized?

Then run it.

Run It

```
1 #include <iostream>
2
3 int main () {
4     int a[2][4] = {{1,2,3,4}};
```

(continues on next page)

(continued from previous page)

```
5
6       std::cout << "first: " << a[0][0] << '\n';
7       std::cout << "last:  " << a[1][3] << '\n';
8   }
```

Reference types

A *reference* type declares a named variable as an alias to an *already-existing* object or function.

References are a C++ addition - one of the few types not present in C.

A reference is required to be initialized to refer to a valid object or function. There are no references to `void` and no references to references.

```
int a = 3;
int& r1 = a; // r1 is a reference to a
r1 = 72;     // changes the value of a
int& r2;     // Error. r2 must refer to something
```

Once initialized, a reference always refers to the same object. The value of the object may change, but the address referred to may not.

Note

C++ 11 introduced a new kind of reference, an *rvalue reference*. We will cover this when we get into classes. All of the references discussed until then will be *lvalue references*.

The type `std::size_t`

There exists an implementation defined `typedef` **std::size_t**. The type `std::size_t` represents the maximum number of bytes that can be stored for an object of any type (including array).

In order to use size_t you need include the header `cstdlib`.

This means that `size_t` is **guaranteed** to always be big enough to use safely as an index in any array. This doesn't mean you can't access an invalid element of an array, only that the index can be increased without worrying about the index variable overflowing (see the previous discussion about overflow).

The purpose of `size_t` is to relieve the programmer from having to worry about which of the predefined unsigned types is used to represent sizes.

Code that assumes `sizeof` yields an `unsigned int` is not as portable as code that assumes it yields a `size_t`.

`size_t` is commonly used for array indexing and loop counting. Programs that use other types, such as `unsigned int`, for array indexing may fail, for example, on 64-bit systems when the index exceeds `UINT_MAX` or if it relies on 32-bit modular arithmetic.

So, if you must write a hand-rolled loop to loop though a container that returns its size, then prefer this:

```
for (size_t i = 0; i < foo.size(); ++i)
```

over this:

```
for (int i = 0; i < foo.size(); ++i)
```

As a programmer, you need to use caution if the variable `i` is to be used for anything other than an index --- for example, in an arithmetic expression. Avoid mixing signed and unsigned types as the results can be surprising. Also be aware that C++ uses signed integers for array subscripts and the standard library uses unsigned integers for container subscripts. This makes absolute consistency in all situations impossible.

Later on, we will cover techniques that improve on iterating through data even more.

Try This!

What do you think the output of the following program will be? If you have access to another computer, try compiling in '32 bit' mode

Run it and check your assumptions.

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

int main() {
  char c = 'a';
  std::cout << "size of char:         " << sizeof(c) << '\n'
            << "size of int:          " << sizeof(int) << '\n'
            << "size of unsigned int: " << sizeof(unsigned) << '\n'
            << "size of double:       " << sizeof(double) << '\n'
            << "size of size_t:       " << sizeof(size_t) << '\n'
            << "size of pointer:      " << sizeof(&c) << std::endl;

  double foo[10] = {0.1, 3.14159, 1e1, 2e-6, 5, 6, 7, 8, 9, 10};
  std::cout << "size of double foo[]:     " << sizeof(foo) << '\n'
            << "no. of elements in foo[]: " << sizeof(foo) / sizeof(foo[0]) <<␣
↪std::endl;
}
```

What sizes are different? Why?

What are the implications of these differences when writing code that needs to run on both?

Displaying numbers

The standard library provides facilities to modify the base for integers stored in memory. You can change the base using setbase, or change how a number is displayed using dec, using hex, using oct,

std::hex

These functions are all I/O manipulators. They may be called with an expression such as

```
out << std::hex
```

for any out of type basic_ostream or with an expression such as

```
in >> std::hex
```

for any in of type basic_istream. For example:

```
std::cout << "The number 42 in octal:   " << std::oct << 42 << '\n'
          << "The number 42 in decimal: " << std::dec << 42 << '\n'
          << "The number 42 in hex:     " << std::hex << 42 << '\n';
```

(continues on next page)

(continued from previous page)

```
int n;
std::istringstream("2A") >> std::hex >> n;
std::cout << std::dec << "Parsing \"2A\" as hex gives " << n << '\n';
// the output base is sticky until changed
std::cout << std::hex << "42 as hex gives " << 42
          << " and 21 as hex gives " << 21 << '\n';
```

ASCII Table

This example displays a simple table of the ASCII characters in 3 different bases.

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

int main() {
  using std::cout;
  std::string heading = "\ndec\toct\thex\tchar\n";
  for (int i = 0; i < 128; ++i) {
    if (!(i%20)) cout << heading;
    char c = i;
    // the printable characters are between hex 20 and 7e
    cout << std::dec << i << '\t'
         << std::oct << i << '\t'
         << std::hex << i << "\t'" << c << "'\n";
  }
  return 0;
}
```

Self Check

Q1

Matching question

Match the definition from the left and select the correct concept on the right. Click the "Check Me" button to see if you are correct.

A. A name associated with a memory location. ___ bool

B. A whole number ___ declaring a variable

C. An expression that is either true or false ___ integer

D. Specifying the type and name for a variable ___ variable

Q2

Question

Which declarations are valid?

- ☐ int 3;
- ☐ const double pi;
- ☐ x = 21;
- ☐ char* s = "hello";

☐ constexpr int min = 1;

Q3

Matching question

Match the definition from the left and select the correct concept on the right. Click the "Check Me" button to see if you are correct.

A. a type used to represent decimal values ___ casting

B. An operator that returns the remainder ___ double

C. changing the type of a variable ___ initialize

D. Setting the value of a variable the first time ___ modulus

Q4

Fill in the blank

Given the following:

```
#include <cstdint>
int main() {
  int8_t x = 128, y = 1;
  auto z   = x+y;
}
```

What is value stored in `z`?

_____

Q5

Fill in the blank

Given the following:

```
int x = ~-1;
```

What is value stored in `x`?

_____

Q6

Fix all the errors in the code below:

```
int main() {
  weight = 3
  distance = 8.5;
  cout << "Weight: " << weight << '\n';
  cout << "Distance: " << distance << '\n';
}
```

Q7

Parsons problem

Code fragments

```
#include <iostream>

int x = 3, y = 5;

std::cout << x << std::endl << y << std::endl;

void func() {

x = y * 8;

}
```

Q8

Fill in the blank

Given the following:

```
#include <iostream>

int main () {
    char x[2][3] = {{'a','b','c'}};

    std::cout << x[0][1] << '\n';
}
```

What is value displayed?

_____

More to Explore

- From cppreference.com
  - types and std::size_t
  - typedef and type aliases
- Two's complement
- ISO CPP Super FAQ: Floating point questions
- What every computer scientist should know about floating-point arithmetic
- Dyadic rationals on Wikipedia
- CPP Core Guidelines: ES.100 Don't mix signed and unsigned arithmetic
- An interesting alternative to explore, Google Protocol Buffers use variable length zig-zag encoding
- Secure Coding in C++: Integers

Footnotes

2.1.3 Random numbers

Most computer programs do the same thing every time they are executed, so they are said to be **deterministic**. Usually, determinism is a good thing, since we expect the same calculation to yield the same result. For some applications, though, we would like the computer to be unpredictable. Games are an obvious example.

Making a program truly **nondeterministic** turns out to be not so easy, but there are ways to make it at least seem that way. One of them is to generate "pseudorandom" numbers and use them to determine the outcome of the program. Pseudorandom numbers are not truly random in the mathematical sense, but for our purposes, they will do.

C++ provides a function called uniform_int_distribution that returns a pseudorandomly generated integer value uniformly distributed within the range you specify. It is declared in the header file `random`, which is another part of the standard library.

To see a sample, run this loop:

Take a look at the active code below, which generates 4 random numbers.

```
#include <iostream>
#include <random>

int main () {
   // make a random number generator
   std::random_device device;
   std::default_random_engine gen(device());

    for (int i = 0; i < 4; i++) {
        int upper_bound = 10000;
        int x = std::uniform_int_distribution<int> {0, upper_bound} (gen);
        std::cout << x << '\n';
    }
    return 0;
}
```

Notice there is a lot going on in this small program.

1. Before we create a random number, we have to create a random number generator *engine*.

   The engine is the object that actually does all the hard work.

   The C++ standard library provides many options and variations for pseudorandom engines, since it is such an important topic.

   But for most simple purposes, the default way is good enough.

2. The `uniform_int_distribution` object needs 3 pieces of information:

   - `<int>` - the type of the random value to return
   - `{0, 10000}` - the range of values (inclusive) to select from
   - `(gen)` - the engine to use

Q1

Fill in the blank

Pseudorandom numbers are said to be _____, because different numbers are generated every time the program is executed.

_____

Q2

Question

What header file do we need to declare in order to use `std::uniform_int_distribution`?

- ◯ cstdlib
- ◯ random
- ◯ cmath
- ◯ iostream

Q3

Question

If we wanted to generate a random number between 0 and 12, and we have previously declared

```
std::random_device dev;
std::default_random_engine engine(dev());
```

what should be our next line of code?

- ◯ `int x = std::uniform_int_distribution<int> {0, 12} (gen);`
- ◯ `std::uniform_int_distribution<int> {0, 12} (engine);`
- ◯ `int x = std::uniform_int_distribution<int> {0, 13} (engine);`
- ◯ `int x = std::uniform_int_distribution<int> {0, 12} (engine);`

2.1.4 Code comments

You should have learned different formats for code comments

```
// a one line comment
std::cout << "Hello C++!" << std::endl;  // this is a comment
puts("Hello C!");
/*
a block comment
printf("Hello Alice!\n");
printf("Hello %s!\n", "Bob");
*/
```

What you may not have have learned is **when** to use comments. This is partly a stylistic discussion, and some people have very strong feelings about the use of comments in software. These opinions range from "never use them, ever", to enormous comment blocks at the top of every source file and before each function. Others expect a line of code at the end of nearly every line of source.

I am not any of these people.

The primary focus of this course is on **clarity**.

- The goal of any program in any language is to express ideas in code as *clearly as possible*.
- If you can do that without writing any comments, great.
- If you need to explain some piece of code, add a comment.

  Although, a better solution is usually to rewrite the confusing code so that it doesn't need clarification with a comment.

Comments should always and only state things that cannot be captured well in regular code. For example:

- What are the boundary conditions or limitations of the code.
- What preconditions must exist before using the code?
- What postconditions are guaranteed to exist? Is there a "minimum guarantee"?
- For an algorithm, why was this or that method chosen?

One notable exception to the 'keep comments to a minimum' rule is if you are commenting a public *API* for a library and most readers will only see generated documentation and not the source code.

In that case, you are generally adding comments to be read by a documentation parser/generator such as Doxygen. For example:

```
/**
 * <A short one line description>
 *
 * <Longer description>
 * <May span multiple lines or paragraphs as needed>
 *
 * @param  Description of method's or function's input parameter
 * @param  ...
 * @return Description of the return value
 */
```

Comments in assignments

In this course, I also need everyone to assert that their work is their own. For that reason, the top of every source file should contain your name and student ID:

```
// file.cpp
// Dave Parillo, 123456789
```

Commenting "anti-patterns"

An *anti-pattern* is a common response to a recurring problem that is ineffective. Anti-patterns represent examples that you **should not** copy! As bad as they are, they can still be instructive.

Show Comment Anti-Patterns

In case you are wondering, these anti-patterns are all actual code examples I have received in the past.

One of my pet peeves is writing comments that say **exactly** what the code already says.

```
help(argv[0]);   //passing the 1st arg. to func. help.

for (int i=0; i<10; ++i)       // loop from 0 to 9
{
  printf("counter: %d\n", i);  // print counter
}
```

Perhaps this is not obvious, but what is wrong here:

```
int main( int argc, char* argv[] )  //or alternately char**arg[]
```

What is Wrong?

The comment is actually telling a lie: the alternative will not compile!

```
// from a file named "average.cpp"

int number;          // number of values in the set
double value;        // value entered at keyboard
double average;      // average value
double total;        // sum of all values
char again = 'y';    // repeat running the program
char validElement;   // consider sentinel value -1 is valid

// what is wrong with this code block??
if (total != 0)
  average = total / number;  // calculate the average value

fflush(stdin); // empty input buffer
cin.get();     // read in a character
```

The following series of code blocks are a combination of several commonly provided anti-patterns that have been combined into a single example.

This first code block is composed of comments that add no value. There is only 'noise'. Every comment merely says basically the same thing as the code, just not as well.

Also, we know we are in trouble in this program as the variable names provide little hint about anything this program might actually do.

```
// The FooCalculator class calculates a Foo.
// @author Dave Parillo
struct FooCalculator {
  // The Integer maxFoo defines the maximum foo
  int maxFoo = 100;
  // The Integer foo defines the current foo.
  int foo = 0;
  // The member thing is an array of strings.
  std::string thing[100];
};

FooCalculator f;
```

In this second block, which `isFooSmallEnough` appears to describe what this function is doing, the comments again, add nothing.

The comments for `rammer` and `rammerstat` add no value and are actually misleading, since neither function appears to actually compute anything.

```
// The isFooSmallEnough method determines if the foo is small enough
// @return boolean {@code true} if foo is smaller than max
// @return boolean {@code false} if foo is larger than max
bool isFooSmallEnough() {
  return f.foo < f.maxFoo;
```

```
}

// Compute a rammer
void rammer(std::string stat) {
  if (isFooSmallEnough()) {
    f.thing[f.foo++] = stat;
  }
  std::cout << stat << '\n';
}

// Calculate the ramerstat function.
// @param rammer  A String that stores the rammer value
void rammerstat(std::string x) {
    std::ifstream ram(x);
    std::string stat;
    while (getline(ram,stat)) {
      rammer(stat);
    }
    ram.close();
}
```

The only comment here tells us what we already know. It would be nice to know what is expected of `args` that are passed into our nasty little program, but perhaps the author thought that was obvious?

```
// Main is a global function.
int main(int argc, char** argv) {
  if (argc > 1) {
    for (int i = 1;i < argc; ++i) {
      rammerstat(argv[i]);
    }
  } else {
    puts ("Usage: foo-comments args");
  }
}
```

What *does* this program do if compiled and run?

The preceding advice may conflict with what you have been told in the past about commenting your code. Don't worry too much about that for now. Remember the focus is on **clarity**, not how many comments you write. Eventually some future employer will require you to (hopefully) adhere to some coding standard and you should follow that guidance when you encounter it.

Commenting Dos and Don'ts

I spent a lot of time saying how not to write comments. How *should* you?

First, avoid the four common excuses related to comments. In other words, I don't need to write comments because:

- Good code is self-documenting.

  There are many things that can't be expressed in code.

  - *Why* a particular feature was implemented.
  - What information callers need beyond what is available in the function signature.

  If you never write comments, then you can never explain these important details.

- I don't have time to write comments.

  Writing good comments takes far less time then writing good code. Writing comments first can help make your designs better and prevent rework. With that in mind, comments are a net time-saver.

- Comments get out of date and become misleading

  This often happens when people write documentation 'far' away from the code being documented. If the comments go stale it is because the code changed, but the comments did not. Keeping code and comments close minimizes this problem.

- Comments are worthless, why bother?

  Arguably, the most valid excuse. However, just because other comments are bad does not mean you should not write comments yourself.

The overall idea behind comments is to capture information that was in the mind of the designer but couldn't be captured or expressed in code.

- Comments should describe things that aren't obvious from code.
- Pick a commenting convention and use it consistently. In this course, I generally use the conventions from the tool Doxygen.
- Don't repeat what code says in comments.
  - Use different words in the comments from the name of the things being described.
- Focus on *what* and *why*, not *how*

  The code says *how* an entity is defined or performed. Comments help readers understand *what* the code is doing.
- Write comments before writing code. Use comments as part of the design process when writing code.
- Avoid duplicate comments in the same way we strive to avoid duplicate code.

More to Explore

- comments
- Jeff Atwood's blog: Code tells you how, Comments tell you why
- Eric Lippert's Blog: One more thing about comments
- A Philosphy of Software Design, John Ousterhout. Chapters 15 and 16 focus on comments.
- Doxygen

2.1.5 What you don't need to know (yet)

C++ is a large, relatively complex, language. Due to its size, there are many topics you may have not covered, covered incompletely, or covered incorrectly. Luckily for all of us, there is a relatively simple language buried inside C++. One of the goals of this text is to concentrate on the simpler parts while still solving more advanced problems than found in a typical introductory text.

Pointers, for example. Pointers are tricky for some people to get used to. Very few languages outside of C and C++ give you direct access to pointers and so many ways to manipulate them. Depending on your point of view, you may consider this pure genius, or the most incredibly foolish design decision ever made by a programmer.

Modern C++ adds a variety of tools that make working with pointers easier and safer.

C++

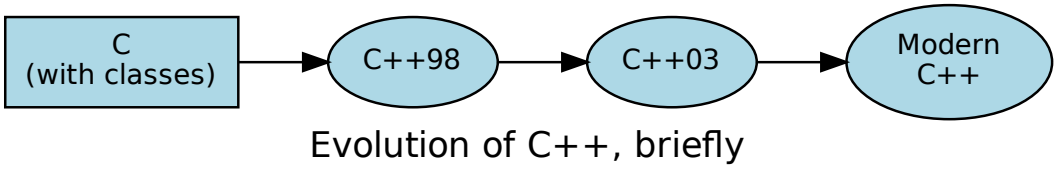
This may sound odd, considering this is supposed to be a second semester course in C++. But as I said, C++ is a very large language. Most likely, you have learned mostly C (probably), with a very small amount of C++ thrown in:

- The classes `cin` and `cout`
- A `class` with *setters* and *getters* - maybe even using `private` members

The version of C++ most likely taught to most is C++98. Modern C++ doesn't look much like the C++ that was written in the 80's and 90's. Primarily because programmers have learned a lot about how to write programs in C++ over the years, but also because the power of the *Standard Template Library* (STL). The STL was not developed until C++ had been used for more than 10 years. While it was incorporated into the first ISO version of the C++ standard (C++98), it took some time for many programmers to recognize the power and flexibility of *template programming* in addition to the *object-oriented programming paradigm* C++ was originally designed to support.

If the language or compiler you learned does not conform to at least the C++98 version of the standard, then it's not C++.

To add to the confusion, C++ is actually a federation of several languages:



Evolution of C++, briefly

Where 'Modern' C++ is C++11 and later. More specifically, the **current** version of the standard.

It is possible to write code in any of these languages compile it with a C++ compiler and call it a "C++ program". This course emphasizes 'modern' C++. Although there is emphasis on newer language features, that does not mean that features released before 2011 should never be used. That would be impossible.

Generally, C++ gives programmers many choices and it is true that some choices are preferred over others. We will try to make preferred design and programming choices clear.

Video

Video URL: https://www.youtube.com/watch?v=AUT201AXeJg

You can test the level of support for the compiler you are using by attempting to compile these examples in your environment.

C++11

A simple test for a modern C++ compiler:

```
// A simple test for C++11 compiler
// compiled with: -Wall -Wextra -pedantic -std=c++11

#include <iostream>

```

(continues on next page)

(continued from previous page)

```
// test C++11 features
int main () {
  int data[] = { 1, 2, 3, 5, 8 };  // initializer list syntax
  for (auto value: data)           // for each value in data
  {
    std::cout << "value is " << value << '\n';
  }
  return 0;
}
```

C++14

The current textbook compiler supports C++14, at least partly.

```
// A simple test for C++14 support
// compiled with: -Wall -Wextra -pedantic -std=c++1y

#include <iostream>
#include <memory>
#include <string>
#include <tuple>
#include <functional>

// this function returns multiple values
auto func() {
   int x = 5;
   return std::make_tuple(x, 7);
}

int main() {
   // function returning multiple values
   int a, b;
   std::tie(a, b) = func();

   // make a unique pointer
   std::unique_ptr<int> p1 = std::make_unique<int>(3);
   std::cout << a << " " << b << " " << *p1 << '\n';

   // quoted numeric literal
   auto c = 299'792'458;

   // std::string literal
   using namespace std::literals;
   auto units = "m/s"s;    // valid c++14

   std::cout << "speed of light: " << c << " " << units << '\n';
}
```

C++17

```cpp
// A simple test for C++17 support
// compiled with: -Wall -Wextra -pedantic -std=c++17

#include <any>
#include <iostream>
#include <type_traits>

template <typename T>
// auto return type deduction: C++14
auto get_value(T value) {
   // constexpr-if: C++17
   // is_pointer type trait: C++11
   if constexpr (std::is_pointer<T>::value) {
       return *value;
   } else {
       return value;
   }
}

int main() {
  // container for any type:   C++17
  auto a = std::make_any<int>(3);
  std::cout << std::any_cast<int>(a) << '\n';
  a = 3.14159;
  std::cout << std::any_cast<double>(a) << '\n';
  int i = 9;
  int* p = &i;

  std::cout << get_value(*p) << '\n';
  std::cout << get_value(i) << '\n';
  return i;
}
```

The textbook online compiler has complete support for C++ up to and including C++17 and most of C++20.

More to Explore

- C++17: constexpr if
- More from cppreference.com:
  - cin, cout, and cerr

2.1.6 Self Check

The questions in this section provide a chance to demonstrate your understanding of the concepts discussed so far.

After reading the material in this chapter, you should be able to answer these questions.

Fill in the blank

Given the following plain english statements:

- Create a variable x with value 8
- Create a variable y with value 5
- Add x and y, storing the result in y

If they were implemented as a program, then what value is `y` when finished?

Parsons problem

Place these statements in their proper order so that the program prompts for input, computes the area, and displays the results.

Code fragments

```
double area = h*w;

double h = 0;
double w = 0;

std::cin >> h;

std::cin >> w;

std::cout << "Enter height:\t";

std::cout << "Enter width:\t";

std::cout << area << '\n';
```

Fix this program so that it compiles.

```
int main() {
  21 = value;
  double value;

  std::cout << value << '\n';
}
```

Fill in the blank

Given the following program:

```
int main() {
  int a = 7;
  int b = 4;

  if (a<=b) {
    a = 99;
  } else {
    int t = a;
```

(continues on next page)

(continued from previous page)

```
    a = b;
    b = t;
  }
  return a;
}
```

What value is returned?

Write a program that accumulates the sum of the numbers 1 - 10 and prints the result.

```
int main() {

}
```

Parsons problem

When assembled in its proper order, the following program segment prints 'Odd numbers:' followed by all the odd numbers from 1 - 100, one per line.

Code fragments

```
  }
}

for(int num=1; num<=100; ++num) {

if(num % 2 != 0) {

if(num * 2 == 0) {  #distractor

int main () {

std::cout << "Odd numbers:\n";

std::cout << '\t' << num << '\n';

}
```

Question

Which of the following statements represent **assignment to** a variable? Check all that apply.

- ☐ int a;
- ☐ a = b;
- ☐ size_t sz = 10;
- ☐ cin >> a;
- ☐ int if = a;

Write a program that stores your name in a local variable and then prints it.

Listing 2: Write a program that prints your name

```
#include <iostream>

int main() {

}
```

Question

Which of the following are legal variable names? Check all that apply.

- ☐ int inner_product_of_a_and_b;
- ☐ double* p2;
- ☐ char 1st_letter;
- ☐ long large num;
- ☐ long double _d;

Question

What are the values of x, y, and z after the following code executes?

```
int x = 0;
int y = 1;
int z = 2;
--x;
++y;
z+=y;
```

- ◯ x = -1, y = 1, z = 4
- ◯ x = -1, y = 2, z = 3
- ◯ x = -1, y = 2, z = 2
- ◯ x = -1, y = 2, z = 2
- ◯ x = -1, y = 2, z = 4

Question

What is the result of 158 % 10?

- ◯ 15
- ◯ 16
- ◯ 8

Question

What is the result of 3 % 8?

- ◯ 3
- ◯ 2
- ◯ 8

Question

What are the values of x, y, and z after the following code executes?

```
int x = 3;
int y = 5;
int z = 2;
x = z * 2;
y /= 2;
++z;
```

- ◯ x = 6, y = 2.5, z = 2
- ◯ x = 4, y = 2.5, z = 2
- ◯ x = 4, y = 2, z = 3
- ◯ x = 4, y = 2.5, z = 3
- ◯ x = 6, y = 2, z = 3

Fill in the blank

Given the following statement:

```
double x = 2 + 2^3
```

What is value stored in `x`?

2.2 Mathematical Background

This chapter presents mathematical notation, background, and techniques used throughout the book. This material is provided primarily for review and reference. You might wish to return to the relevant sections when you encounter unfamiliar notation or mathematical techniques in later chapters.

2.2.1 Sets and Relations

Set Notation

The concept of a set in the mathematical sense has wide application in computer science. The notations and techniques of set theory are commonly used when describing and implementing algorithms because the abstractions associated with sets often help to clarify and simplify algorithm design.

A *set* is a collection of distinguishable *members* or *elements*. The members are typically drawn from some larger population known as the *base type*. Each member of a set is either a *primitive element* of the base type or is a set itself. There is no concept of duplication in a set. Each value from the base type is either in the set or not in the set. For example, a set named $\mathbf{P}$ might consist of the three integers 7, 11, and 42. In this case, $\mathbf{P}$'s members are 7, 11, and 42, and the base type is integer.

The following table shows the symbols commonly used to express sets and their relationships.

Set Symbology

$\{1, 4\}$	A set composed of the members 1 and 4
$\{x \mid x \text{ is a positive integer}\}$	A set definition using a set former Example: the set of all positive integers
$x \in \mathbf{P}$	x is a member of set $\mathbf{P}$
$x \notin \mathbf{P}$	x is not a member of set $\mathbf{P}$
$\emptyset$	The null or empty set
$\lvert\mathbf{P}\rvert$	Cardinality: size of set $\mathbf{P}$ or number of members for set $\mathbf{P}$
$\mathbf{P} \subseteq \mathbf{Q}$, $\mathbf{Q} \supseteq \mathbf{P}$	Set $\mathbf{P}$ is included in set $\mathbf{Q}$, set $\mathbf{P}$ is a subset of set $\mathbf{Q}$, set $\mathbf{Q}$ is a superset of set $\mathbf{P}$
$\mathbf{P} \cup \mathbf{Q}$	Set Union: all elements appearing in $\mathbf{P}$ OR $\mathbf{Q}$
$\mathbf{P} \cap \mathbf{Q}$	Set Intersection: all elements appearing in P AND $\mathbf{Q}$
$\mathbf{P} - \mathbf{Q}$	Set difference: all elements of set $\mathbf{P}$ NOT in set $\mathbf{Q}$
$\mathbf{P} \times \mathbf{Q}$	Set (Cartesian) Product: yields a set of ordered pairs

Here are some examples of this notation in use. First define two sets, $\mathbf{P}$ and $\mathbf{Q}$.

$$\mathbf{P} = \{2, 3, 5\}, \qquad \mathbf{Q} = \{5, 10\}.$$

$|\mathbf{P}| = 3$ (because $\mathbf{P}$ has three members) and $|\mathbf{Q}| = 2$ (because $\mathbf{Q}$ has two members). Both of these sets are finite in length. Other sets can be infinite, for example, the set of integers.

The union of $\mathbf{P}$ and $\mathbf{Q}$, written $\mathbf{P} \cup \mathbf{Q}$, is the set of elements in either $\mathbf{P}$ or $\mathbf{Q}$, which is {2, 3, 5, 10}. The intersection of $\mathbf{P}$ and $\mathbf{Q}$, written $\mathbf{P} \cap \mathbf{Q}$, is the set of elements that appear in both $\mathbf{P}$ and $\mathbf{Q}$, which is {5}. The set difference of $\mathbf{P}$ and $\mathbf{Q}$, written $\mathbf{P} - \mathbf{Q}$, is the set of elements that occur in $\mathbf{P}$ but not in $\mathbf{Q}$, which is {2, 3}. Note that $\mathbf{P} \cup \mathbf{Q} = \mathbf{Q} \cup \mathbf{P}$ and that $\mathbf{P} \cap \mathbf{Q} = \mathbf{Q} \cap \mathbf{P}$, but in general $\mathbf{P} - \mathbf{Q} \neq \mathbf{Q} - \mathbf{P}$. In this example, $\mathbf{Q} - \mathbf{P} = \{10\}$. Finally, the set {5, 3, 2} is indistinguishable from set $\mathbf{P}$, because sets have no concept of order. Likewise, set {2, 3, 2, 5} is also indistinguishable from $\mathbf{P}$, because sets have no concept of duplicate elements.

The *set product* (or Cartesian product) of two sets $\mathbf{Q} \times \mathbf{P}$ is a set of ordered pairs. For our example sets, the set product would be

$$\{(2, 5),\ (2, 10),\ (3, 5),\ (3, 10),\ (5, 5),\ (5, 10)\}.$$

The *powerset* of a set $\mathbf{S}$ (denoted 2^S) is the set of all possible subsets for $\mathbf{S}$. Consider the set $\mathbf{S} = \{a, b, c\}$. The powerset of $\mathbf{S}$ is

$$\{\emptyset,\ \{a\},\ \{b\},\ \{c\},\ \{a, b\},\ \{a, c\},\ \{b, c\},\ \{a, b, c\}\}.$$

A collection of elements with no order (like a set), but with duplicate-valued elements is called a *bag*. To distinguish bags from sets, we will use square brackets [] around a bag's elements. For example, bag [3, 4, 5, 4] is distinct from bag [3, 4, 5], while set {3, 4, 5, 4} is indistinguishable from set {3, 4, 5}. However, bag [3, 4, 5, 4] is indistinguishable from bag [3, 4, 4, 5].

A *sequence* is a collection of elements with an order, and which may contain duplicate-valued elements. A sequence is also sometimes called a *tuple* or a *vector*. In a sequence, there is a 0th element, a 1st element, 2nd element, and so on. We will use angle brackets $\langle\rangle$ to enclose the elements of a sequence. For example, $\langle 3, 4, 5, 4\rangle$ is a sequence. Note that sequence $\langle 3, 5, 4, 4\rangle$ is distinct from sequence $\langle 3, 4, 5, 4\rangle$, and both are distinct from sequence $\langle 3, 4, 5\rangle$.

Relations

A *relation* R over set $\mathbf{S}$ is a set of ordered pairs from $\mathbf{S}$. As an example of a relation, if $\mathbf{S}$ is $\{a, b, c\}$, then

$$\{\langle a, c\rangle, \langle b, c\rangle, \langle c, b\rangle\}$$

is a relation, and

$$\{\langle a, a\rangle, \langle a, c\rangle, \langle b, b\rangle, \langle b, c\rangle, \langle c, c\rangle\}$$

is a different relation. If tuple $\langle x, y\rangle$ is in relation R, we may use the infix notation xRy. We often use relations such as the less than operator ($<$) on the natural numbers, which includes ordered pairs such as $\langle 1, 3\rangle$ and $\langle 2, 23\rangle$, but not $\langle 3, 2\rangle$ or $\langle 2, 2\rangle$. Rather than writing the relationship in terms of ordered pairs, we typically use an infix notation for such relations, writing $1 < 3$.

Define the properties of relations as follows, with R a binary relation over set $\mathbf{S}$.

- R is *reflexive* if aRa for all $a \in \mathbf{S}$.
- R is *irreflexive* if aRa is not true for all $a \in \mathbf{S}$.
- R is *symmetric* if whenever aRb, then bRa, for all $a, b \in \mathbf{S}$.
- R is *antisymmetric* if whenever aRb and bRa, then $a = b$, for all $a, b \in \mathbf{S}$.
- R is *transitive* if whenever aRb and bRc, then aRc, for all $a, b, c \in \mathbf{S}$.

As examples, for the natural numbers, $<$ is irreflexive (because aRa is never true), antisymmetric (because there is no case where aRb and bRa), and transitive. Relation $\leq$ is reflexive, antisymmetric, and transitive. Relation $=$ is reflexive, symmetric (and antisymmetric!), and transitive. For people, the relation "is a sibling of" is symmetric and transitive. If we define a person to be a sibling of themself, then it is reflexive; if we define a person not to be a sibling of themself, then it is not reflexive.

Equivalence Relations

R is an *equivalence relation* on set $\mathbf{S}$ if it is reflexive, symmetric, and transitive. An equivalence relation can be used to partition a set into *equivalence classes*. If two elements a and b are equivalent to each other, we write $a \equiv b$. A *partition* of a set $\mathbf{S}$ is a collection of subsets that are *disjoint* from each other and whose union is $\mathbf{S}$. An *equivalence relation* on set $\mathbf{S}$ partitions the set into disjoint subsets whose elements are equivalent. One application for such *disjoint sets* computing a minimal cost spanning tree.

Example

For the integers, $=$ is an equivalence relation that partitions each element into a distinct subset. In other words, for any integer a, three things are true.

1. $a = a$,
2. if $a = b$ then $b = a$, and
3. if $a = b$ and $b = c$, then $a = c$.

Of course, for distinct integers a, b, and c there are never cases where $a = b$, $b = a$, or $b = c$. So the requirements for symmetry and transitivity are never violated, and therefore the relation is symmetric and transitive.

Example

If we clarify the definition of sibling to mean that a person is a sibling of themself, then the sibling relation is an equivalence relation that partitions the set of people.

Example

We can use the *modulus* function to define an equivalence relation. For the set of integers, use the modulus function to define a binary relation such that two numbers x and y are in the relation if and only if $x \bmod m = y \bmod m$. Thus, for $m = 4$, $\langle 1, 5 \rangle$ is in the relation because $1 \bmod 4 = 5 \bmod 4$. We see that modulus used in this way defines an equivalence relation on the integers, and this relation can be used to partition the integers into m equivalence classes. This relation is an equivalence relation because

1. $x \bmod m = x \bmod m$ for all x;
2. if $x \bmod m = y \bmod m$, then $y \bmod m = x \bmod m$; and
3. if $x \bmod m = y \bmod m$ and $y \bmod m = z \bmod m$, then $x \bmod m = z \bmod m$.

Partial Orders

A binary relation is called a *partial order* if it is antisymmetric and transitive. If the relation is reflexive, it is called a *non-strict partial order*. If the relation is *irreflexive*, it is called a *strict partial order*. The set on which the partial order is defined is called a *partially ordered set* or a *poset*. Elements x and y of a set are *comparable* under a given relation R if either xRy or yRx. If every pair of distinct elements in a partial order are comparable, then the order is called a *total order* or *linear order*.

Example

For the integers, relations $<$ and $\leq$ define partial orders. Operation $<$ is a total order because, for every pair of integers x and y such that $x \neq y$, either $x < y$ or $y < x$. Likewise, $\leq$ is a total order because, for every pair of integers x and y such that $x \neq y$, either $x \leq y$ or $y \leq x$.

Example

For the powerset of the integers, the subset operator defines a partial order (because it is antisymmetric and transitive). For example, $\{1, 2\} \subseteq \{1, 2, 3\}$. However, sets $\{1, 2\}$ and $\{1, 3\}$ are not comparable by the subset operator, because neither is a subset of the other. Therefore, the subset operator does not define a total order on the powerset of the integers.

Acknowledgements

This section is adapted from Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

2.2.2 Selected functions

Factorial The factorial function, written $n!$ for n an integer greater than 0, is the product of the integers between 1 and n, inclusive. Thus, $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. As a special case, $0! = 1$.

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

Note

Factorial is usually denoted with the symbol !, which is not to be confused with the C++ logical operator ! which means NOT.

Notice that when used to negate a boolean, the `!` appears *before* the operand, while factorial appears *after* the operand.

The factorial function grows quickly as n becomes larger. Because computing the factorial function directly is a time-consuming process, it can be useful to have an equation that provides a good approximation. Stirling's approximation states that $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$, where $e \approx 2.71828$ (e is the base for the system of natural logarithms)[1]. Thus we see that while $n!$ grows slower than n^n (because $\sqrt{2\pi n}/e^n < 1$), it grows faster than c^n for any positive integer constant c.

Logic notation We will occasionally make use of the notation of symbolic or boolean logic. $A \Rightarrow B$ means "A implies B" or "If A then B". $A \Leftrightarrow B$ means "A if and only if B" or "A is equivalent to B". $A \vee B$ means "A or B" (useful both in the context of symbolic logic or when performing a boolean operation). $A \wedge B$ means "A and B". $\sim A$ and $\overline{A}$ both mean "not A" or the negation of A where A is a boolean variable.

Floor and ceiling The *floor* of x (written $\lfloor x \rfloor$) takes real value x and returns the greatest integer $\leq x$. For example, $\lfloor 3.4 \rfloor = 3$, as does $\lfloor 3.0 \rfloor$, while $\lfloor -3.4 \rfloor = -4$ and $\lfloor -3.0 \rfloor = -3$. The *ceiling* of x (written $\lceil x \rceil$) takes real value x and returns the least integer $\geq x$. For example, $\lceil 3.4 \rceil = 4$, as does $\lceil 4.0 \rceil$, while $\lceil -3.4 \rceil = \lceil -3.0 \rceil = -3$.

More to Explore

- From cppreference.com
  - Common math functions
  - types - including `bool`
  - floor and ceil

Acknowledgements

This section is adapted from Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

2.2.3 Modulus operator

The modulus operator returns the remainder of an integer division. Sometimes written $n \bmod m$ in mathematical expressions, the syntax in C++ is `n % m`. From the definition of remainder, $n \bmod m$ is the integer r such that $n = qm + r$ for q an integer, and $|r| < |m|$.

Therefore, the result of $n \bmod m$ must be between 0 and $m - 1$ when n and m are positive integers. For example, $5 \bmod 3 = 2$; $25 \bmod 3 = 1$, $5 \bmod 7 = 5$, and $5 \bmod 5 = 0$.

modulus

A common source of error is mixing up modulus and integer division operations.

```
int quotient = 7 / 3;

int remainder = 7 % 3;
```

The first operator, integer division, yields 2. The second operator yields 1. Thus, 7 divided by 3 is 2 with 1 left over.

[1] The symbol "≈" means "approximately equal."

Run It

This program shows the difference between the division operator and the modulus operator.

Listing 3: Modulus Operations

```
#include <iostream>
using std::cout;

int main () {
    int quotient = 7 / 3;
    int remainder = 7 % 3;
    cout << "quotient: " << quotient << '\n';
    cout << "remainder: " << remainder << '\n';
    return 0;
}
```

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if `x % y` is zero, then x is divisible by y.

Also, you can use the modulus operator to extract the rightmost digit or digits from a number. For example, `x % 10` yields the rightmost digit of x (in base 10). Similarly `x % 100` yields the last two digits.

Question

How could you detmine whether a variable `x` is odd?

- ◯ Use x % 2, and if the result is 0, it is odd.
- ◯ Use x % 2, and if the result is 1, it is odd.
- ◯ Use x / 2, and if the result is 0, it is odd.
- ◯ Use x / 2, and if the result is 1, it is odd.

Matching question

Match the modulo expression to its result.

A. 2 % 3	___ 0
B. 3 % 2	___ 1
C. 6 % 2	___ 2
D. 9 % 6	___ 3

Parsons problem

Construct a block of code that prints the remainder of 18 when divided by 13.

Code fragments

```
cout << x % y;

cout << x / y;

cout << y % x;

cout << y / x;
```

(continues on next page)

(continued from previous page)

```
int main () {

int x = 18;
int y = 13;

}
```

Parsons problem

Construct a function that prints whether a number is even.

Code fragments

```
 cout << false;
}

 cout << true;
}

bool is_even (int number) { #distractor

else {

if (number % 2 == 0) {

void is_even (int number) {

}
```

There is more than one way to assign values to q and r, depending on how integer division is interpreted. The most common mathematical definition computes the mod function as $n \bmod m = n - m\lfloor n/m \rfloor$. In this case, $-3 \bmod 5 = 2$. However, Java and C++ compilers typically use the underlying processor's machine instruction for computing integer arithmetic. On many computers this is done by truncating the resulting fraction, meaning $n \bmod m = n - m(\text{trunc}(n/m))$. Under this definition, $-3 \bmod 5 = -3$. Another language might do something different.

Unfortunately, for many applications this is not what the user wants or expects. For example, many hash systems will perform some computation on a record's *key* value and then take the result modulo the hash table size. The expectation here would be that the result is a legal index into the hash table, not a negative number. Implementers of hash functions must either ensure that the result of the computation is always positive, or else add the hash table size to the result of the modulo function when that result is negative.

More to Explore

- From cppreference.com
  - Common math functions
  - remainder and remquo
  - fmod
  - div

Acknowledgements

This section is adapted from Problem Solving with Algorithms and Data Structures using C++, by Brad Miller and David Ranum, Luther College, and Jan Pearce, Berea College released under the CC BY-NC-SA 4.0, and Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

2.2.4 Permutations

A permutation of a sequence **S** is simply the members of **S** arranged in some order. For example, a permutation of the integers 1 through n would be those values arranged in some order. If the sequence contains n distinct members, then there are $n!$ different permutations for the sequence. This is because there are n choices for the first member in the permutation; for each choice of first member there are $n-1$ choices for the second member, and so on.

permute

Sometimes one would like to obtain a random permutation for a sequence, that is, one of the $n!$ possible permutations is selected in such a way that each permutation has equal probability of being selected. A simple function for generating a random permutation is as follows.

```
//Randomly permute the values in array
void permute(int data[], int n) {
  for (int i = n; i > 0; --i) {
    int j = std::uniform_int_distribution<int> {0, i-1} (eng);
    std::swap(data[i-1], data[j]);  // swap data[i-1] with a random
  }                                 // position in the range 0 to i-1.
}
```

Here, the n values of the sequence are stored in positions 0 through $n-1$ of array `data`. Function swap exchanges elements in array `data`, and uniform_int_distribution returns an integer value uniformly distributed in the range 0 to $i-1$.

Run It

```
#include <iostream>
#include <random>
#include <utility>

namespace {
  // make a random number generator
  std::random_device r;
  std::default_random_engine eng(r());
}

//Randomly permute the values in array
void permute(int data[], int n) {
  for (int i = n; i > 0; --i) {
    int j = std::uniform_int_distribution<int> {0, i-1} (eng);
    std::swap(data[i-1], data[j]);  // swap data[i-1] with a random
  }                                 // position in the range 0 to i-1.
}
void print(int data[], int n) {
  for (int i = 0; i<n; ++i) {
    std::cout << data[i] << '\t';
```

(continues on next page)

(continued from previous page)

```
  }
  std::cout << std::endl;
}
int main() {
  int data[] = {1,1,2,3,5,8,13,21,34};

  for (int i = 0; i<5; ++i) {
    permute(data, 9);
    print(data, 9);
  }
  std::cout << '\n';
}
```

shuffle

Randomly shuffling a range of data is a common enough activity that it is implemented in the standard library. The shuffle function does what our permute function does, but a bit more generically.

```
std::shuffle(std::begin(data), std::end(data), eng);
```

Instead of an entire array it takes a range of data and a random number generator.

Run shuffle

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <random>

namespace {
  std::random_device r;
  std::default_random_engine eng(r());  // make a random number generator
}

void print(int data[], int n) {
  for (int i = 0; i<n; ++i) {
    std::cout << data[i] << '\t';
  }
  std::cout << std::endl;
}
int main() {
  int data[] = {1,1,2,3,5,8,13,21,34};

  for (int i = 0; i<5; ++i) {
    std::shuffle(std::begin(data), std::end(data), eng);
    print(data, 9);
  }
  std::cout << std::endl;
}
```

More to Explore

- From cppreference.com
  - Random number generation and random_shuffle
  - Common math functions
  - is_permutation and next_permutation

Acknowledgements

This section is adapted from Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

2.2.5 Logarithms

The logarithm of base b for value y is the power to which b is raised to get y. Normally, this is written as $\log_b y = x$. Thus, if $\log_b y = x$ then $b^x = y$, and $b^{log_b y} = y$.

Logarithms have the following properties, for any positive values of m, n, and r, and any positive integers a and b.

1) $\log(nm) = \log n + \log m$.

2) $\log(n/m) = \log n - \log m$.

3) $\log(n^r) = r \log n$.

4) $\log_a n = \log_b n / \log_b a$.

The first two properties state that the logarithm of two numbers multiplied (or divided) can be found by adding (or subtracting) the logarithms of the two numbers.[1] Property (3) is simply an extension of property (1). Property (4) tells us that, for variable n and any two integer constants a and b, $\log_a n$ and $\log_b n$ differ by the constant factor $\log_b a$, regardless of the value of n. Most runtime analyses we use are of a type that ignores constant factors in costs. Property (4) says that such analyses need not be concerned with the base of the logarithm, because this can change the total cost only by a constant factor.

log base 2

C and C++ have always included functions to compute $\log_e$ and $\log_{10}$. New in C++11 is a function specifically for base 2. As we will see, base 2 logarithms are used frequently and it is convenient to not have to perform change in base calculations when $\log_2$ is needed.

```
double x = std::log(1000);  // natural log of 1000

double x = std::log10(1000);  // base 10 log of 1000

double x = std::log2(1000);  // base 2 log of 1000
```

In this course, nearly all logarithms used have a base of two. This is because data structures and algorithms most often divide things in half, or store codes with binary bits. Whenever you see the notation $\log n$ in this text, either $\log_2 n$ is meant or else the term is being used asymptotically and so the actual base does not matter. For logarithms using any base other than two, we will show the base explicitly.

[1] These properties are the idea behind the slide rule. Adding two numbers can be viewed as joining two lengths together and measuring their combined length. Multiplication is not so easily done. However, if the numbers are first converted to the lengths of their logarithms, then those lengths can be added and the inverse logarithm of the resulting length gives the answer for the multiplication (this is simply logarithm property (1)). A slide rule measures the length of the logarithm for the numbers, lets you slide bars representing these lengths to add up the total length, and finally converts this total length to the correct numeric answer by taking the inverse of the logarithm for the result.

Run It

Listing 4: Log Operations

```cpp
#include <iostream>
#include <cmath>
int main()
{
    std::cout << "log(1) = " << std::log(1) << '\n'
              << "base-5 logarithm of 125 = " << std::log(125)/std::log(5) << '\n';

    std::cout << "log2(65536) = " << std::log2(65536) << '\n'
        << "log2(0.125) = " << std::log2(0.125) << '\n'
        << "log2(0x020f) = " << std::log2(0x020f)
        << " (highest set bit is in position 9)\n"
        << "base-5 logarithm of 125 = " << std::log2(125)/std::log2(5) << '\n';

    // special values
    std::cout << "log2(1) = " << std::log2(1) << '\n'
              << "log2(+Inf) = " << std::log2(INFINITY) << '\n';

}
```

Example

Many programs require an encoding for a collection of objects. What is the minimum number of bits needed to represent n distinct code values? The answer is $\lceil \log_2 n \rceil$ bits. For example, if you have 1000 codes to store, you will require at least $\lceil \log_2 1000 \rceil = 10$ bits to have 1000 different codes (10 bits provide 1024 distinct code values).

Example

Consider the binary search algorithm for finding a given value within an array sorted by value from lowest to highest.

Binary search first looks at the middle element and determines if the value being searched for is in the upper half or the lower half of the array. The algorithm then continues splitting the appropriate subarray in half until the desired value is found. How many times can an array of size (n) be split in half until only one element remains in the final subarray? The answer is $\lceil \log_2 n \rceil$ times.

A useful identity to know is:

$$2^{\log n} = n$$

To give some intuition for why this is true: What does it mean to take the log (base 2) of n? If $\log_2 n = x$, then x is the power to which you need to raise 2 to get back to n. So of course, $2^{\log n} = n$ when the base of the log is 2.

When discussing logarithms, exponents often lead to confusion. Property (3) tells us that $\log n^2 = 2 \log n$. How do we indicate the square of the logarithm (as opposed to the logarithm of n^2)? This could be written as $(\log n)^2$, but it is traditional to use $\log^2 n$. On the other hand, we might want to take the logarithm of the logarithm of n. This is written $\log \log n$.

A special notation is used in the rare case when we need to know how many times we must take the log of a number before we reach a value ≤ 1. This quantity is written $\log^* n$. For example, $\log^* 1024 = 4$ because $\log 1024 = 10$, $\log 10 \approx 3.33$, $\log 3.33 \approx 1.74$, and $\log 1.74 < 1$, which is a total of 4 log operations.

Self Check

Q1

Fill in the blank

Simplify the following expression into the form $log(c)$: $\log(15) - \log(3)$

_____

Q2

Fill in the blank

Simplify the following expression into the form $log(c)$: $\log(3) + \log(4)$

_____

Q3

Fill in the blank

Solve the following expression: $\log_{729}(9)$

_____

More to Explore

- From cppreference.com
  - log, log, and log2

Footnotes

Acknowledgements

This section is adapted from Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

2.2.6 Summations

Most programs contain loop constructs. When analyzing running time costs for programs with loops, we need to add up the costs for each time the loop is executed. This is an example of a *summation*. Summations are simply the sum of costs for some function applied to a range of values. Summations are typically written with the following "Sigma" notation:

$$\sum_{k=1}^{n} f(k)$$

This notation indicates that we are adding the result of $f(k)$ over some range of (integer) values.

The expression parameter and its initial value are indicated below the $\sum$ symbol. Here, the notation $k = 1$ indicates that the parameter is k and that it begins with the value 1. At the top of the $\sum$ symbol is the expression n. This indicates the maximum value for the parameter k. Thus, this notation means to sum the values of $f(k)$ as k ranges across the integers from 1 through n. In other words,

$$\sum_{k=1}^{n} f(k)$$

is simply a way to express the sum

$$f(1) + f(2) + \cdots + f(n-1) + f(n)$$

Given a summation, you often wish to replace it with an algebraic equation with the same value as the summation. This is known as a *closed-form solution*, and the process of replacing the summation with its closed-form solution is known as solving the summation.

A closed form is an expression that can be computed by applying a fixed number of familiar operations to the arguments. For example, the expression $2 + 4 + \cdots + 2n$ is not a closed form, but the expression $n(n+1)$ is a closed form.

For example, the summation $\sum_{k=1}^{n} 1$ is simply the constant expression "1" added n times (remember that k ranges from 1 to n). Because the sum of n 1s is n, the closed-form solution is n. In other words, a summation of a constant expression is equivalent to counting by the constant value "math"*n* times, or multiplying the constant by n.

Summation facts

Fact 1	**Fact 2**
$\sum ca_k = c\sum a_k$	$\sum(a_k + b_k) = \sum a_k + \sum b_k$
Fact 3	**Fact 4**
$\sum a_k x^{i+k} = x^i \sum a_k x^k$	$\sum_{k=m}^{n} a_{k+i} = \sum_{k=m+i}^{n+i} a_k$

Collapsing sums (Fact 5)

and

$$\sum_{k=1}^{n}(a_k - a_{k-1}) = a_n - a_0 \qquad \sum_{k=1}^{n}(a_{k-1} - a_k) = a_0 - a_n$$

Useful forms

Here is a list of useful summations, along with their closed-form solutions.

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

$$\sum_{k=1}^{n} k^2 = \frac{2n^3 + 3n^2 + n}{6} = \frac{n(2n+1)(n+1)}{6}.$$

$$\sum_{k=1}^{\log n} n = n \log n.$$

$$\sum_{k=0}^{n} a^k = \frac{a^{n+1} - 1}{a - 1} \text{ where } a \neq 1. \tag{2.1}$$

Special cases

As special cases to (2.1), we have the following two:

$$\sum_{k=1}^{n} \frac{1}{2^k} = 1 - \frac{1}{2^n}, \tag{2.2}$$

$$\sum_{k=0}^{n} 2^k = 2^{n+1} - 1. \tag{2.3}$$

As a corollary to equation (2.3),

$$\sum_{k=0}^{\log n} 2^k = 2^{\log n+1} - 1 = 2n - 1.$$

Finally,

$$\sum_{k=1}^{n} \frac{k}{2^k} = 2 - \frac{n+2}{2^n}. \tag{2.4}$$

Most of these equalities can be proved using a *proof by induction*. Unfortunately, induction does not help us derive a closed-form solution. Induction only confirms when a proposed closed-form solution is correct.

Example

Find a closed form for the expression $\sum_{k=2}^{n}(k-1)2^{k+1}$.

Solution:

$$\begin{aligned}
\sum_{k=2}^{n}(k-1)2^{k+1} &= \sum_{k=1}^{n-1} k2^{k+2} && \text{Fact 4} \\
&= 2^2 \sum_{k=1}^{n-1} k2^k && \text{Fact 3} \\
&= 2^2\left(2 - n2^n + (n-1)2^{n+1}\right) && \text{Form 5} \\
&= 2^3 - (2-n)2^{n+2}
\end{aligned}$$

Example

Use summation facts and forms to prove $2 + 3 + \cdots + n = \frac{(n-1)(n+2)}{2}$.

Solution:

$$\begin{aligned}
2 + 3 + \cdots + n &= \sum_{k=2}^{n} k \\
&= \sum_{k=1}^{n-1}(k+1) \\
&= \sum_{k=1}^{n-1} k + \sum_{k=1}^{n-1} 1 \\
&= \frac{(n-1)(n)}{2} + (n-1) \\
&= \frac{(n-1)(n+2)}{2}
\end{aligned}$$

Example

Use summation facts and forms to find a closed form for $3 + 7 + \cdots + (3 + 4n)$.

Solution:

$$\begin{aligned}\sum_{k=0}^{n}(3+4k) &= \sum_{k=0}^{n}3 + \sum_{k=0}^{n}4k \\ &= \sum_{k=0}^{n}3 + 4\sum_{k-0}^{n}k \\ &= 3(n+1) + \frac{4n(n+1)}{2} \\ &= (3+2n)(n+1)\end{aligned}$$

Example

Let *count(n)* be the number of times `func()` is executed by the following algorithm as a function of n, where $n \in \mathbb{N}$. Find a closed for for *count(n)*.

```
int i = 1;
while (i < n) {
   i = i + 2;
   for (int j = 1; j <= i; ++j) {
      func();
   }
}
```

Solution:

Each time through the while loop, `i` is incremented by 2. So the values of `i` at the start of the for loop are $3, 5, \cdots, (2k+1)$, where $i = 2k+1 \geq n$ represents the stopping point for the while loop. So we have:

$$\begin{aligned}\textit{count(n)} &= 3 + 5 + \cdots + (2k+1) \\ &= \sum_{i=1}^{k}(2i+1) \\ &= 2\sum_{i=1}^{k}i + \sum_{i=1}^{k}1 \\ &= \frac{2k(k+1)}{2} + k \\ &= k(k+1) + k \\ &= k(k+2)\end{aligned}$$

In order to write *count(n)* in terms of n, we need to do a bit more work. Since $2k+1 \geq n$ is the stopping point for the while loop it follows that $2k-1 < n$ is the last time the while condition is `true`. In otherwords, we have the inequality $2k-1 < n \leq 2k+1$. Solving for k, we have $2k-2 < n-1 \leq 2k$, which gives $k-1 < (n-1)/2 \leq k$. Therefore, $k = \lceil (n-1)/2 \rceil$. Now we can write *count(n)* in terms of n as:

$$\begin{aligned}\textit{count(n)} &= k(k+2) \\ &= \left\lceil \frac{n-1}{2} \right\rceil \left(\left\lceil \frac{n-1}{2} \right\rceil + 2 \right)\end{aligned}$$

Approximating Sums

Not all sums have closed forms. In those cases, we can try to find a suitable approximation. For example, consider the following sum:

$$\begin{aligned}\mathcal{H}_n &= 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \\ &= \sum_{k=1}^{n} \frac{1}{k}\end{aligned}$$

This is called the *Harmonic Series* and it has no closed form. It is closely approximated by $\ln n$ because the definite integral of $1/x$ from 1 to *n* is $\ln n$. The constant known as Euler's constant (γ) with a value close to 0.577, approximates the difference between $\mathcal{H}_n$ and $\ln n$ when n is large.

$$\begin{aligned}\mathcal{H}_{10} - \ln 10 &\approx 2.93 - 2.31 = 0.62 \\ \mathcal{H}_{20} - \ln 20 &\approx 3.00 - 2.60 = 0.60 \\ \mathcal{H}_{40} - \ln 40 &\approx 4.28 - 3.69 = 0.59\end{aligned}$$

More to Explore

- Geometric series
- 700 years of secrets of the Sum of Sums

Acknowledgements

This section is adapted from Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

2.2.7 Estimation

One of the most useful life skills that you can gain from your computer science training is the ability to perform quick estimates. This is sometimes known as "back of the napkin" or "back of the envelope" calculation. Both nicknames suggest that only a rough estimate is produced. Estimation techniques are a standard part of engineering curricula but are often neglected in computer science. Estimation is no substitute for rigorous, detailed analysis of a problem, but it can help to decide when a rigorous analysis is warranted: If the initial estimate indicates that the solution is unworkable, then further analysis is probably unnecessary.

Estimation can be formalized by the following three-step process:

1. Determine the major parameters that affect the problem.
2. Derive an equation that relates the parameters to the problem.
3. Select values for the parameters, and apply the equation to yield an estimated solution.

When doing estimations, a good way to reassure yourself that the estimate is reasonable is to do it in two different ways. In general, if you want to know what comes out of a system, you can either try to estimate that directly, or you can estimate what goes into the system (assuming that what goes in must later come out). If both approaches (independently) give similar answers, then this should build confidence in the estimate.

When calculating, be sure that your units match. For example, do not add feet and pounds. Verify that the result is in the correct units. Always keep in mind that the output of a calculation is only as good as its input. The more uncertain

your valuation for the input parameters in Step 3, the more uncertain the output value. However, back of the envelope calculations are often meant only to get an answer within an order of magnitude, or perhaps within a factor of two. Before doing an estimate, you should decide on acceptable error bounds, such as within 25%, within a factor of two, and so forth. Once you are confident that an estimate falls within your error bounds, leave it alone! Do not try to get a more precise estimate than necessary for your purpose.

Example

How many library bookcases does it take to store books containing one million pages? I estimate that a 500-page book requires one inch on the library shelf (it will help to look at the size of any handy book), yielding about 200 feet of shelf space for one million pages. If a shelf is 4 feet wide, then 50 shelves are required. If a bookcase contains 5 shelves, this yields about 10 library bookcases. To reach this conclusion, I estimated the number of pages per inch, the width of a shelf, and the number of shelves in a bookcase. None of my estimates are likely to be precise, but I feel confident that my answer is correct to within a factor of two. (After writing this, I went to Virginia Tech's library and looked at some real bookcases. They were only about 3 feet wide, but typically had 7 shelves for a total of 21 shelf-feet. So I was correct to within 10% on bookcase capacity, far better than I expected or needed. One of my selected values was too high, and the other too low, which canceled out the errors.)

Example

Is it more economical to buy a car that gets 20 miles per gallon, or one that gets 30 miles per gallon but costs \$3000 more? The typical car is driven about 12,000 miles per year. If gasoline costs \$3/gallon, then the yearly gas bill is \$1800 for the less efficient car and \$1200 for the more efficient car. If we ignore issues such as the payback that would be received if we invested \$3000 in a bank, it would take 5 years to make up the difference in price. At this point, the buyer must decide if price is the only criterion and if a 5-year payback time is acceptable. Naturally, a person who drives more will make up the difference more quickly, and changes in gasoline prices will also greatly affect the outcome.

Example

When at the supermarket doing the week's shopping, can you estimate about how much you will have to pay at the checkout? One simple way is to round the price of each item to the nearest dollar, and add this value to a mental running total as you put the item in your shopping cart. This will likely give an answer within a couple of dollars of the true total.

Acknowledgements

This section is adapted from Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

2.2.8 Pseudocode conventions and definitions

Some examples use 'pseudocode' to represent program logic independent of any mainstream programming language. When pseudocode is used, the objective is to understand the algorithm using simplified representations.

The following notations and conventions are used.

1. Symbol names generally follow C++ snake_case conventions
2. A function is a symbol name followed by `()`
3. Function arguments, if present, will specify a type and a symbol name. For example: `int: x` or `array: a`.

4. The function `print()` denotes a function that sends its arguments to standard output. Generally, the format of the output is not important.
5. The `operator[]` is used in the same manner as a C or C++ array. Arrays are zero-based. **Array overflow is still possible**.
6. Pseudo code operators:
   - <- denotes assignment. The statement x <- y means "assign the value of y to x" You can also read <- as "becomes", as in "x *becomes* y"
   - == denotes the equivalence relation. The statement x == y means or "the object x is equivalent to the object y"
   - `xor` denotes the `exclusive or` as defined in C++. The statement a <- x xor y means "the bits in the integer x are combined with the bits of y and the result is assigned to a"
   - + denotes numeric addition or concatenation, depending on context. If a statement contains a string and a number, then assume the number is concatenated to the string.
7. Other math and logical operations should be interpreted as they are defined in C++.
8. Loop and conditional constructs are formatted as `keyword . . . done keyword`, for example:

```
while (value < 10)
    print (value)
    value <- value + 1
done while

if (value == x)
    print (value)
else
    print (value)
done if
```

Types without an explicit definition can be assumed as defined on first use or the example should explain the initial value.

The concept of *Estimation* might be unfamiliar to many readers. Estimation is not a mathematical technique, but rather a general engineering skill. It is useful to computer scientists doing design work, because any proposed solution whose estimated resource requirements fall well outside the problem's resource constraints can be discarded immediately, allowing time for greater analysis of more promising solutions.

2.3 Algorithm Analysis

This chapter introduces the motivation, basic notation, and fundamental techniques of algorithm analysis. We focus on a methodology known as *asymptotic algorithm analysis*, or simply *asymptotic analysis*. Asymptotic analysis attempts to estimate the resource consumption of an algorithm.

2.3.1 Problems, Algorithms, and Programs

Problems

Programmers commonly deal with problems, algorithms, and computer programs. These are three distinct concepts.

As your intuition would suggest, a *problem* is a task to be performed. It is best thought of in terms of inputs and matching outputs. A problem definition should not include any constraints on *how* the problem is to be solved. The solution method should be developed only after the problem is precisely defined and thoroughly understood. However, a problem definition should include constraints on the resources that may be consumed by any acceptable solution. For

any problem to be solved by a computer, there are always such constraints, whether stated or implied. For example, any computer program may use only the main memory and disk space available, and it must run in a "reasonable" amount of time.

Problems can be viewed as functions in the mathematical sense. A *function* is a matching between inputs (the *domain*) and outputs (the *range*). An input to a function might be a single value or a collection of information. The values making up an input are called the *parameters* of the function. A specific selection of values for the parameters is called an *instance* of the problem. For example, the input parameter to a sorting function might be an array of integers. A particular array of integers, with a given size and specific values for each position in the array, would be an instance of the sorting problem. Different instances might generate the same output. However, any problem instance must always result in the same output every time the function is computed using that particular input.

This concept of all problems behaving like mathematical functions might not match your intuition for the behavior of computer programs. You might know of programs to which you can give the same input value on two separate occasions, and two different outputs will result. For example, if you type `date` to a typical Linux command line prompt, you will get the current date. Naturally the date will be different on different days, even though the same command is given. However, there is obviously more to the input for the date program than the command that you type to run the program. The date program computes a function. In other words, on any particular day there can only be a single answer returned by a properly running date program on a completely specified input. For all computer programs, the output is completely determined by the program's full set of inputs. Even a "random number generator" is completely determined by its inputs (although some random number generating systems appear to get around this by accepting a random input from a physical process beyond the user's control). The limits to what functions can be implemented by programs is part of the domain of *computability*.

Algorithms

An *algorithm* is a method or a process followed to solve a problem. If the problem is viewed as a function, then an algorithm is an implementation for the function that transforms an input to the corresponding output. A problem can be solved by many different algorithms. A given algorithm solves only one problem (i.e., computes a particular function). This book discusses many problems, and for several of these problems we will see more than one algorithm. For the important problem of sorting there are over a dozen commonly known algorithms!

The advantage of knowing several solutions to a problem is that solution $\mathbf{A}$ might be more efficient than solution $\mathbf{B}$ for a specific variation of the problem, or for a specific class of inputs to the problem, while solution $\mathbf{B}$ might be more efficient than $\mathbf{A}$ for another variation or class of inputs. For example, one sorting algorithm might be the best for sorting a small collection of integers (which is important if you need to do this many times). Another might be the best for sorting a large collection of integers. A third might be the best for sorting a collection of variable-length strings.

By definition, something can only be called an algorithm if it has all of the following properties.

1. It must be *correct*. In other words, it must compute the desired function, converting each input to the correct output. Note that every algorithm implements some function, because every algorithm maps every input to some output (even if that output is a program crash). At issue here is whether a given algorithm implements the *intended* function.
2. It is composed of a series of *concrete steps*. Concrete means that the action described by that step is completely understood --- and doable --- by the person or machine that must perform the algorithm. Each step must also be doable in a finite amount of time. Thus, the algorithm gives us a "recipe" for solving the problem by performing a series of steps, where each such step is within our capacity to perform. The ability to perform a step can depend on who or what is intended to execute the recipe. For example, the steps of a cookie recipe in a cookbook might be considered sufficiently concrete for instructing a human cook, but not for programming an automated cookie-making factory.
3. There can be *no ambiguity* as to which step will be performed next. Often it is the next step of the algorithm description. Selection (e.g., the `if` statement) is normally a part of any language for describing algorithms. Selection allows a choice for which step will be performed next, but the selection process is unambiguous at the time when the choice is made.

4. It must be composed of a *finite* number of steps. If the description for the algorithm were made up of an infinite number of steps, we could never hope to write it down, nor implement it as a computer program. Most languages for describing algorithms (including English and "pseudocode") provide some way to perform repeated actions, known as iteration. Examples of iteration in programming languages include the `while` and `for` loop constructs. Iteration allows for short descriptions, with the number of steps actually performed controlled by the input.
5. It must *terminate*. In other words, it may not go into an infinite loop.

Programs

We often think of a computer *program* as an instance, or concrete representation, of an algorithm in some programming language. Algorithms are usually presented in terms of programs, or parts of programs. Naturally, there are many programs that are instances of the same algorithm, because any modern computer programming language can be used to implement the same collection of algorithms (although some programming languages can make life easier for the programmer). To simplify presentation, people often use the terms "algorithm" and "program" interchangeably, despite the fact that they are really separate concepts. By definition, an algorithm must provide sufficient detail that it can be converted into a program when needed.

The requirement that an algorithm must terminate means that not all computer programs meet the technical definition of an algorithm. Your operating system is one such program. However, you can think of the various tasks for an operating system (each with associated inputs and outputs) as individual problems, each solved by specific algorithms implemented by a part of the operating system program, and each one of which terminates once its output is produced.

Summary

A *problem* is a function or a mapping of inputs to outputs. An *algorithm* is a recipe for solving a problem whose steps are concrete and unambiguous. Algorithms must be correct, of finite length, and must terminate for all inputs. A *program* is an instantiation of an algorithm in a programming language. The following slideshow should help you to visualize the differences.

More to Explore

- Computability and complexity

Acknowledgements

This section is adapted from Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

2.3.2 Comparing Algorithms

How do you compare two algorithms for solving some problem in terms of efficiency? We could implement both algorithms as computer programs and then run them on a suitable range of inputs, measuring how much of the resources in question each program uses. This approach is often unsatisfactory for four reasons. First, there is the effort involved in programming and testing two algorithms when at best you want to keep only one. Second, when empirically comparing two algorithms there is always the chance that one of the programs was "better written" than the other, and therefore the relative qualities of the underlying algorithms are not truly represented by their implementations. This can easily occur when the programmer has a bias regarding the algorithms. Third, the choice of empirical test cases might unfairly favor one algorithm. Fourth, you could find that even the better of the two algorithms does not fall within your resource budget. In that case you must begin the entire process again with yet another program implementing a new algorithm. But, how would you know if any algorithm can meet the resource budget? Perhaps the problem is simply too difficult for any implementation to be within budget.

These problems can often be avoided by using asymptotic analysis. Asymptotic analysis measures the efficiency of an algorithm, or its implementation as a program, as the input size becomes large. It is actually an estimating technique and does not tell us anything about the relative merits of two programs where one is always "slightly faster" than the other. However, asymptotic analysis has proved useful to computer scientists who must determine if a particular algorithm is worth considering for implementation.

The critical resource for a program is most often its running time. However, you cannot pay attention to running time alone. You must also be concerned with other factors such as the space required to run the program (both main memory and disk space). Typically you will analyze the *time* required for an *algorithm* (or the instantiation of an algorithm in the form of a program), and the *space* required for a *data structure*.

Many factors affect the running time of a program. Some relate to the environment in which the program is compiled and run. Such factors include the speed of the computer's CPU, bus, and peripheral hardware. Competition with other users for the computer's (or the network's) resources can make a program slow to a crawl. The programming language and the quality of code generated by a particular compiler can have a significant effect. The "coding efficiency" of the programmer who converts the algorithm to a program can have a tremendous impact as well.

If you need to get a program working within time and space constraints on a particular computer, all of these factors can be relevant. Yet, none of these factors address the differences between two algorithms or data structures. To be fair, if you want to compare two programs derived from two algorithms for solving the same problem, they should both be compiled with the same compiler and run on the same computer under the same conditions. As much as possible, the same amount of care should be taken in the programming effort devoted to each program to make the implementations "equally efficient". In this sense, all of the factors mentioned above should cancel out of the comparison because they apply to both algorithms equally.

If you truly wish to understand the running time of an algorithm, there are other factors that are more appropriate to consider than machine speed, programming language, compiler, and so forth. Ideally we would measure the running time of the algorithm under standard benchmark conditions. However, we have no way to calculate the running time reliably other than to run an implementation of the algorithm on some computer. The only alternative is to use some other measure as a surrogate for running time.

Basic operations and input size

Of primary consideration when estimating an algorithm's performance is the number of *basic operations* required by the algorithm to process an input of a certain size. The terms "basic operations" and "size" are both rather vague and depend on the algorithm being analyzed. Size is often the number of inputs processed. For example, when comparing sorting algorithms the size of the problem is typically measured by the number of records to be sorted. A basic operation must have the property that its time to complete does not depend on the particular values of its operands. Adding or comparing two integer variables are examples of basic operations in most programming languages. Summing the contents of an array containing n integers is not, because the cost depends on the value of n (i.e., the size of the input).

Example

Consider a simple algorithm to solve the problem of finding the largest value in an array of n integers. The algorithm looks at each integer in turn, saving the position of the largest value seen so far. This algorithm is called the *largest-value sequential search* and is illustrated by the following function:

```
largest(array: A)
   max_value <- 1
   index  <- 2
   while index < array_size(A)
      if A[index] > A[max_value]
         max_value = index
      done if
   done while

   return max_index
done largest
```

Here, the size of the problem is `array_size(A)`, the number of integers stored in array `A`. The basic operation is to compare an integer's value to that of the largest value seen so far. It is reasonable to assume that it takes a fixed amount of time to do one such comparison, regardless of the value of the two integers or their positions in the array.

Because the most important factor affecting running time is normally size of the input, for a given input size n we often express the time $\mathbf{T}$ to run the algorithm as a function of n, written as $\mathbf{T}(n)$. We will always assume $\mathbf{T}(n)$ is a non-negative value.

Let us call c the amount of time required to compare two integers in function `largest`. We do not care right now what the precise value of c might be. Nor are we concerned with the time required to increment variable $index$ because this must be done for each value in the array, or the time for the actual assignment when a larger value is found, or the little bit of extra time taken to initialize `max_value`. We just want a reasonable approximation for the time taken to execute the algorithm. The total time to run `largest` is therefore approximately cn, because we must make n comparisons, with each comparison costing c time. We say that function `largest` (and by extension, the largest-value sequential search algorithm for any typical implementation) has a running time expressed by the equation

$$\mathbf{T}(n) = cn.$$

This equation describes the growth rate for the running time of the largest-value sequential search algorithm.

Example

The running time of a statement that assigns the first value of an integer array to a variable is simply the time required to copy the value of the first array value. We can assume this assignment takes a constant amount of time regardless of the value. Let us call c_1 the amount of time necessary to copy an integer. No matter how large the array on a typical computer (given reasonable conditions for memory and array size), the time to copy the value from the first position of the array is always c_1. Thus, the equation for this algorithm is simply

$$\mathbf{T}(n) = c_1,$$

indicating that the size of the input n has no effect on the running time. This is called a *constant running time*.

Example

Consider the following code:

```
sum = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    sum++;
```

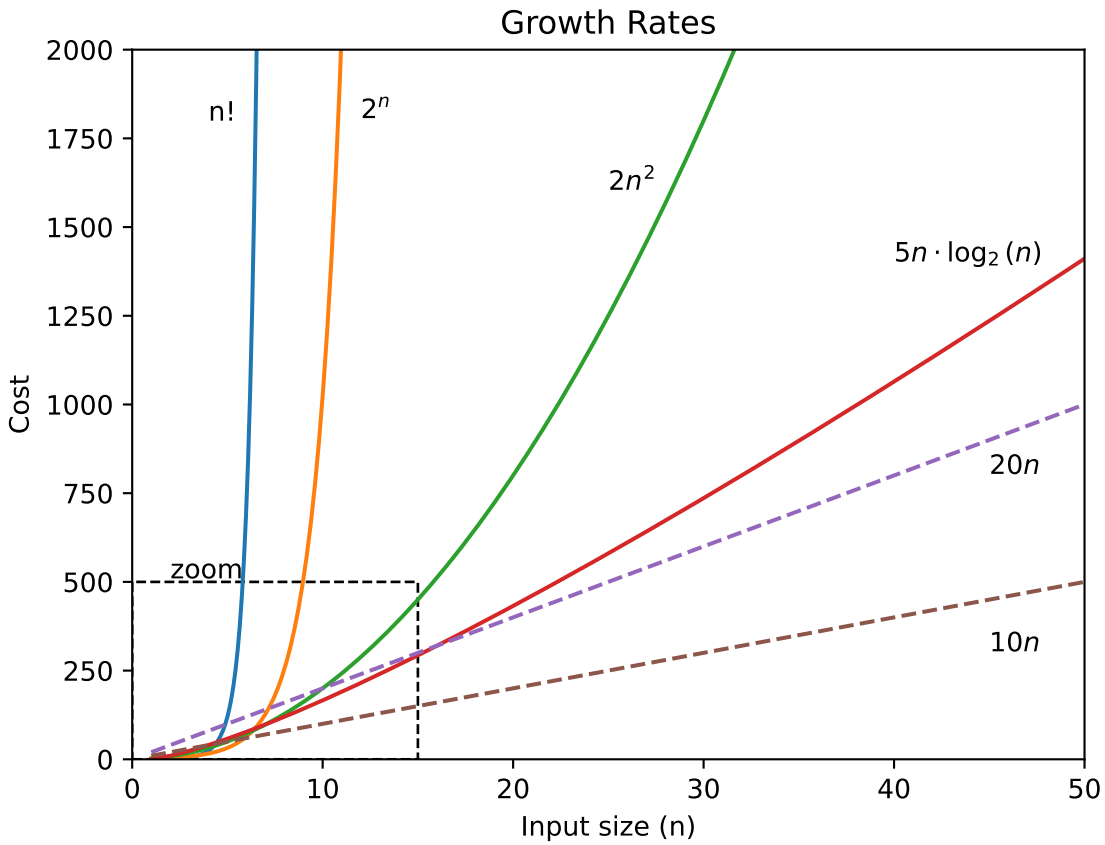
What is the running time for this code fragment? Clearly it takes longer to run when n is larger. The basic operation in this example is the increment operation for variable `sum`. We can assume that incrementing takes constant time; call this time c_2. (We can ignore the time required to initialize `sum`, and to increment the loop counters `i` and `j`. In practice, these costs can safely be bundled into time c_2.) The total number of increment operations is n^2. Thus, we say that the running time is $\mathbf{T}(n) = c_2 n^2$.

Growth Rates

The *growth rate* for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows. The following figure shows a graph for six equations, each meant to describe the running time for a particular program or algorithm. A variety of growth rates that are representative of typical algorithms are shown.

Growth rates

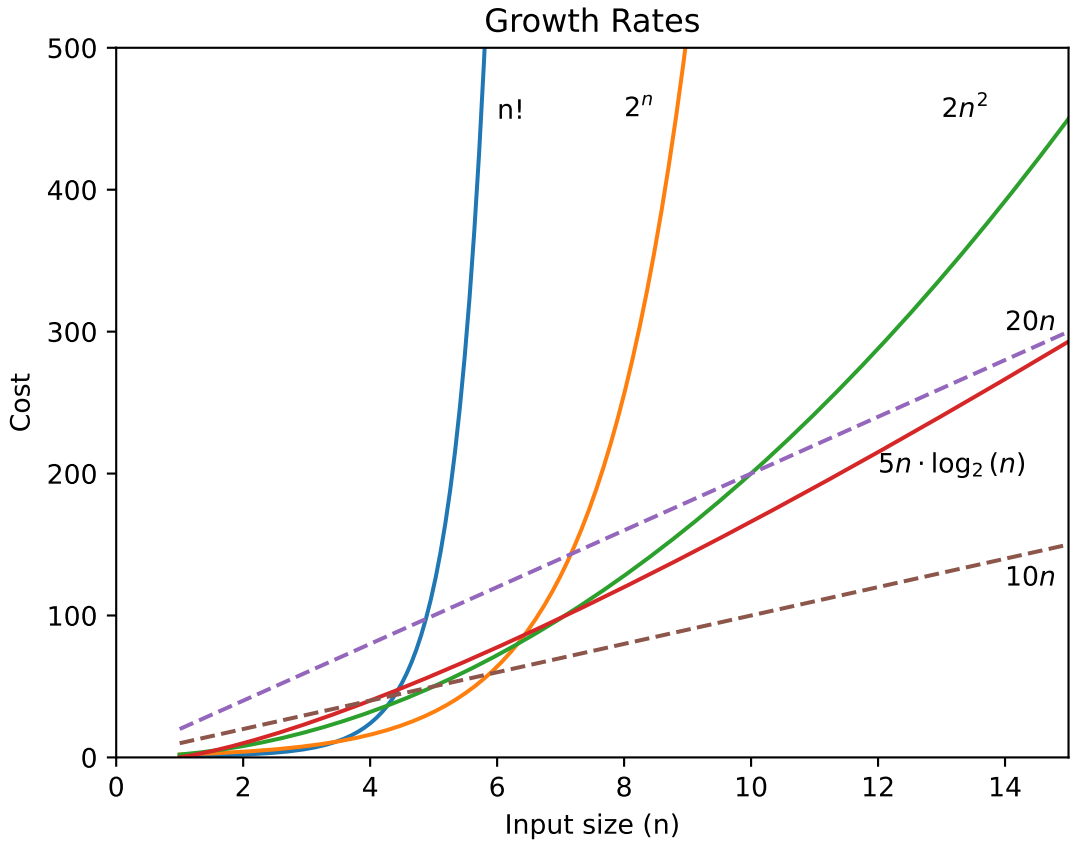
Two views of a graph illustrating the growth rates for six equations. The 'zoom' tab shows in detail the lower-left portion of this graph. The horizontal axis represents input size. The vertical axis can represent time, space, or any other measure of cost.



Zoom

The two equations labeled $10n$ and $20n$ are graphed by straight lines. A growth rate of cn (for c any positive constant) is often referred to as a *linear growth rate* or running time. This means that as the value of n grows, the running time of the algorithm grows in the same proportion. Doubling the value of n roughly doubles the running time. An algorithm whose running-time equation has a highest-order term containing a factor of n^2 is said to have a *quadratic growth rate*. In the figure, the line labeled $2n^2$ represents a quadratic growth rate. The line labeled 2^n represents an *exponential growth rate*. This name comes from the fact that n appears in the exponent. The line labeled $n!$ also grows exponentially.

As you can see from the figure, the difference between an algorithm whose running time has cost $\mathbf{T}(n) = 10n$ and another with cost $\mathbf{T}(n) = 2n^2$ becomes tremendous as n grows. For $n > 5$, the algorithm with running time $\mathbf{T}(n) = 2n^2$ is already much slower. This is despite the fact that $10n$ has a greater constant factor than $2n^2$. Comparing the


Growth Rates
500
400
300
200
100
0
Cost
0
2
4
6
8
10
12
14
Input size (n)
n!
2^n
$2n^2$
$20n$
$5n \cdot \log_2(n)$
$10n$

two curves marked $20n$ and $2n^2$ shows that changing the constant factor for one of the equations only shifts the point at which the two curves cross. For $n > 10$, the algorithm with cost $\mathbf{T}(n) = 2n^2$ is slower than the algorithm with cost $\mathbf{T}(n) = 20n$. This graph also shows that the equation $\mathbf{T}(n) = 5n \log n$ grows somewhat more quickly than both $\mathbf{T}(n) = 10n$ and $\mathbf{T}(n) = 20n$, but not nearly so quickly as the equation $\mathbf{T}(n) = 2n^2$. For constants $a, b > 1, n^a$ grows faster than either $\log^b n$ or $\log n^b$. Finally, algorithms with cost $\mathbf{T}(n) = 2^n$ or $\mathbf{T}(n) = n!$ are prohibitively expensive for even modest values of n. Note that for constants $a, b \geq 1, a^n$ grows faster than n^b.

We can get some further insight into relative growth rates for various algorithms from the following table. Most of the growth rates that appear in typical algorithms are shown, along with some representative input sizes. Once again, we see that the growth rate has a tremendous effect on the resources consumed by an algorithm.

Costs for representative growth rates.

n	$\log \log n$	$\log n$	n	$n \log n$	n^2	n^3	2^n
16	2	4	2^4	$4 \cdot 2^4 = 2^6$	2^8	2^{12}	2^{16}
256	3	8	2^8	$8 \cdot 2^8 = 2^{11}$	2^{16}	2^{24}	2^{256}
1024	≈ 3.3	10	2^{10}	$10 \cdot 2^{10} \approx 2^{13}$	2^{20}	2^{30}	2^{1024}
64K	4	16	2^{16}	$16 \cdot 2^{16} = 2^{20}$	2^{32}	2^{48}	$2^{64\text{K}}$
1M	≈ 4.3	20	2^{20}	$20 \cdot 2^{20} \approx 2^{24}$	2^{40}	2^{60}	$2^{1\text{M}}$
1G	≈ 4.9	30	2^{30}	$30 \cdot 2^{30} \approx 2^{35}$	2^{60}	2^{90}	$2^{1\text{G}}$

Self Check

Q1

Fill in the blank

Given the following growth functions: $n!, 2^n, 2n^2, 5n \log 2n, 20n,$ and$10n$.

For the growth function $n!$, type a value (a positive integer) for which this function is the most efficient of the six. If there is no value for which it is most efficent, type **none**.

Q2

Parsons problem

Order the following functions so that finally the list will contain all the functions in ascending order of their growth rates.

Code fragments

```
5 * log(log(n)

7 * sqrt(n)

9 * n * log(n)

pow(2, sqrt(n))

pow(2,pow(n,2))

pow(n, 4/3)
```

(continues on next page)

(continued from previous page)

```
pow(n,2)

sqrt(n) * log(n)
```

Q3

Parsons problem

Order the following functions so that finally the list will contain all the functions in ascending order of their growth rates.

Code fragments

```
2 * log(n)

pow(2, sqrt(n))

pow(2,n)

pow(n, 4/3)

pow(n,4)

sqrt(n) * log(n)
```

More to Explore

- TBD

Acknowledgements

This section is adapted from Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

2.3.3 Best, worst, and average cases

Consider the problem of finding the factorial of n. For any given input, there is a single output. For example, when $n = 2$, then $n! = 2$ and when $n = 3$, then $n! = 6$ and when $n = 4$, then $n! = 24$.

Find max

Compare the factorial algorithm to a sequential search algorithm.

```
search(array: A)
   max_value <- 1
   index  <- 2
   while index < array_size(A)
      if A[index] > A[max_value]
         max_value <- index
      done if
   done while
```

(continues on next page)

(continued from previous page)

```
   return max_index
done search
```

The algorithm operates on arrays of varying size. Many different array sizes may be passed to this function.

Regardless of the input array size, the algorithm evaluates each storage location exactly once.

Run It

tbd

Compare finding the maximum value to finding a specific value.

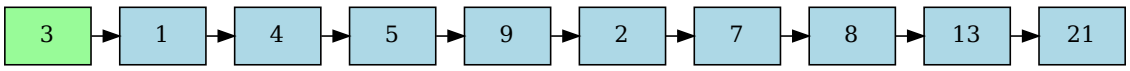
When looking for a value K, we can stop searching as soon as the desired value is found.

This differs from the find max search algorithm in which every location must always be examined.

Best case

```
search(array: A, integer: K)
   index  <- 1
   while index < array_size(A)
      if A[index] == K
         return index
      done if
   done while
   return array_size(A)
done search
```

There are many possible running times for this algorithm. Given the following data:

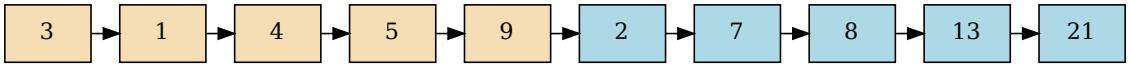


If searching for $K = 3$, we are in luck: we will find it in the first element we examine. This is the **best case**. We can't search for less than one location. In this case, we find the value in the first place we look and return the index `0`.

Average case

```
search(array: A, integer: K)
   index  <- 1
   while index < array_size(A)
      if A[index] == K
         return index
      done if
   done while
   return array_size(A)
done search
```

Given random assortments of data search very many times on arrays of many different sizes, we can expect many different running times.

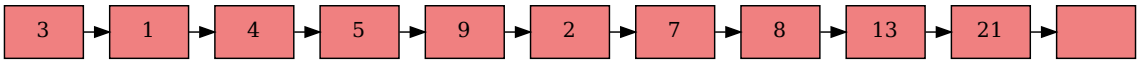


If searching for $K = 9$, then we find the value after we have examined half of the array. On average, the algorithm examines $\frac{n+1}{2}$ values. This is called the **average case** for this algorithm.

Worst case

```
search(array: A, integer: K)
   index  <- 1
   while index < array_size(A)
      if A[index] == K
         return index
      done if
   done while
   return array_size(A)
done search
```

If searching for $K = 72$, then we won't find the value at all. But we don't know this until every element has been examined.



This is the **worst case**. In this case, we find don't find the value and return the size of the container.

Because arrays use zero-based indexing, the size is is always a location *past the end* of the array. Returning a value beyond the range searched is a standard idiom for "The value was not found".

When analyzing an algorithm, should we study the best, worst, or average case? Normally we are not interested in the best case, because this might happen only rarely and generally is too optimistic for a fair characterization of the algorithm's running time. In other words, analysis based on the best case is not likely to be representative of the behavior of the algorithm. However, there are rare instances where a best-case analysis is useful --- in particular, when the best case has high probability of occurring. The Shellsort and Quicksort algorithms both can take advantage of the best-case running time of Insertion sort to become more efficient.

How about the worst case? The advantage to analyzing the worst case is that you know for certain that the algorithm must perform at least that well. This is especially important for real-time applications, such as for the computers that monitor an air traffic control system. Here, it would not be acceptable to use an algorithm that can handle n airplanes quickly enough *most of the time*, but which fails to perform quickly enough when all n airplanes are coming from the same direction.

For other applications --- particularly when we wish to aggregate the cost of running the program many times on many different inputs --- worst-case analysis might not be a representative measure of the algorithm's performance. Often we prefer to know the average-case running time. This means that we would like to know the *typical* behavior of the algorithm on inputs of size n. Unfortunately, average-case analysis is not always possible. Average-case analysis first requires that we understand how the actual inputs to the program (and their costs) are distributed with respect to the set of all possible inputs to the program. For example, it was stated previously that the sequential search algorithm on average examines half of the array values. This is only true if the element with value K is equally likely to appear in

any position in the array. If this assumption is not correct, then the algorithm does *not* necessarily examine half of the array values in the average case.

The characteristics of a data distribution have a significant effect on many search algorithms, such as those based on hashing and search trees such as the binary search tree. Incorrect assumptions about data distribution can have disastrous consequences on a program's space or time performance.

In summary, for real-time applications we are likely to prefer a worst-case analysis of an algorithm. Otherwise, we often desire an average-case analysis if we know enough about the distribution of our input to compute the average case. If not, then we must resort to worst-case analysis.

More to Explore

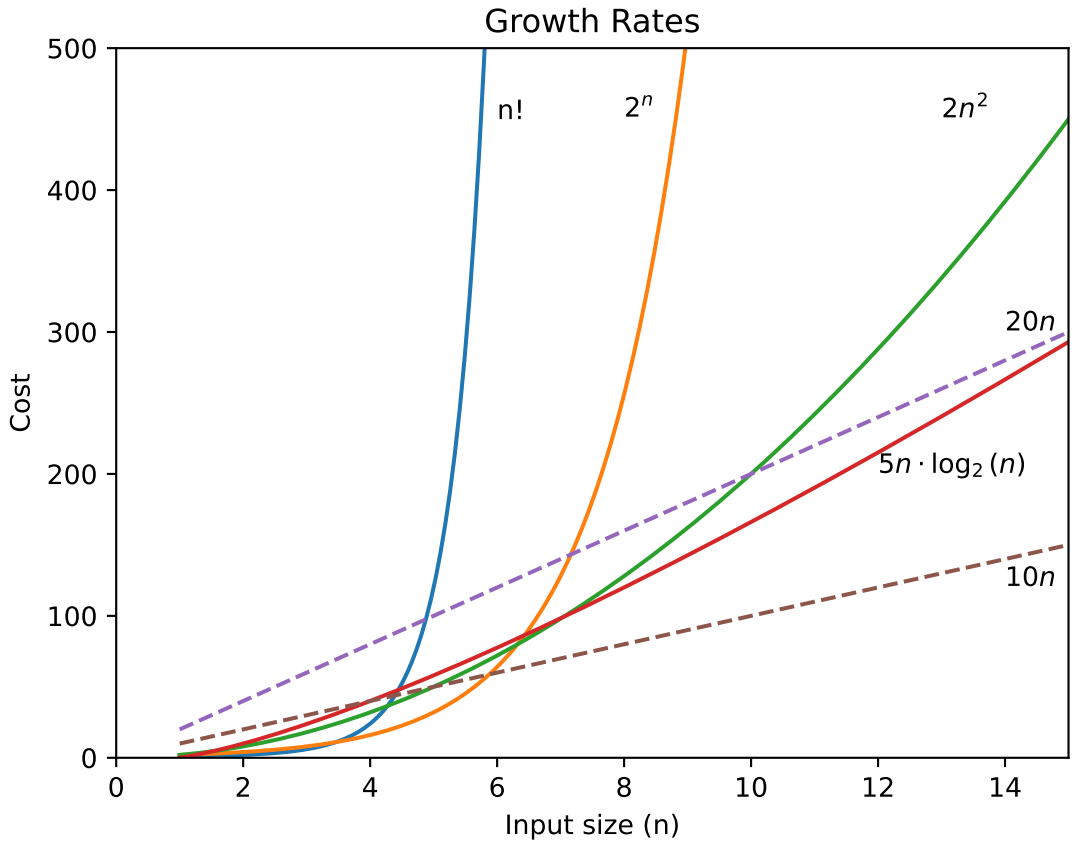
- Average time complexity

Acknowledgements

This section is adapted from Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

2.3.4 Asymptotic Analysis and Upper Bounds

Recall our growth rates from a little while ago.



Despite the larger constant for the curve labeled $10n$ in the figure above, $2n^2$ crosses it at the relatively small value of $n = 5$. What if we double the value of the constant in front of the linear equation? As shown in the graph, $20n$ is surpassed by $2n^2$ once $n = 10$. The additional factor of two for the linear *growth rate* does not much matter. It only doubles the x-coordinate for the intersection point. In general, changes to a constant factor in either equation only shift *where* the two curves cross, not *whether* the two curves cross.

When you buy a faster computer or a faster compiler, the new problem size that can be run in a given amount of time for a given growth rate is larger by the same factor, regardless of the constant on the running-time equation. The time curves for two algorithms with different growth rates still cross, regardless of their running-time equation constants. For these reasons, we usually ignore the constants when we want an estimate of the growth rate for the running time or other resource requirements of an algorithm. This simplifies the analysis and keeps us thinking about the most important aspect: the growth rate. This is called *asymptotic algorithm analysis*. To be precise, asymptotic analysis refers to the study of an algorithm as the input size "gets big" or reaches a limit (in the calculus sense). However, it has proved to be so useful to ignore all constant factors that asymptotic analysis is used for most algorithm comparisons.

Big-O Notation

When trying to characterize an algorithm's efficiency in terms of execution time, independent of any particular program or computer, it is important to quantify the number of operations or steps that the algorithm will require. If each of these steps is considered to be a basic unit of computation, then the execution time for an algorithm can be expressed as the number of steps required to solve the problem. Deciding on an appropriate basic unit of computation can be a complicated problem and will depend on how the algorithm is implemented.

Consider the problem of accumulating a sum. The idea of a simple loop to add values should be familiar.

```
accumulate1(int: N)
   sum <- 0
   count  <- N
   while count > 0
      sum <- sum + count
      count <- count - 1
   done while
   return sum
done accumulate
```

Compare the previous algorithm to this one:

```
accumulate2(int: N)
   if N == 0 return N
   return N + accumulate(N-1)
done accumulate
```

Are both implementations valid?

Is one more efficient than the other?

How do we characterize functions that appear to be different and compare them using a consistent yardstick? Asymptotic analysis to the rescue.

A good basic unit of computation for comparing the summation algorithms shown earlier might be to count the number of assignment statements performed to compute the sum. In the function `accumulate1`, the number of assignment statements is 2 --- assigning `0` to `sum` and assigning `N` to `count`, plus the value of *n* (the number of times we perform $sum = sum + count$ and $count = count - 1$). We can denote this by a function, call it `T`, where $T(n) = 2 + 2n$. The parameter *n* is often referred to as the "size of the problem," and we can read this as "T(n) is the time it takes to solve a problem of size n, namely 2+2n steps."

In the summation functions given above, it makes sense to use the number of terms in the summation to denote the size of the problem. We can then say that the sum of the first 100,000 integers is a bigger instance of the summation

problem than the sum of the first 1,000. Because of this, it might seem reasonable that the time required to solve the larger case would be greater than for the smaller case. Our goal then is to show how the algorithm's execution time changes with respect to the size of the problem.

Computer scientists prefer to take this analysis technique one step further. It turns out that the exact number of operations is not as important as determining the most dominant part of the $T(n)$ function. In other words, as the problem gets larger, some portion of the $T(n)$ function tends to overpower the rest. This dominant term is what, in the end, is used for comparison. The **order of magnitude** function describes the part of $T(n)$ that increases the fastest as the value of *n* increases. Order of magnitude is often called **Big-O notation** (for "order") and written as $O(f(n))$. It provides a useful approximation to the actual number of steps in the computation. The function $f(n)$ provides a simple representation of the dominant part of the original $T(n)$.

In the above example, $T(n) = 2 + 2n$. As *n* gets large, the constants will become less and less significant to the final result. If we are looking for an approximation for $T(n)$, then we can drop them and simply say that the running time is $O(n)$. It is important to note that the constants are certainly significant for $T(n)$. However, as *n* gets large, our approximation will be just as accurate without it.

Try This!

Prove to yourself that the recursive version of the summation in `accumulate2` has the same $O(n)$ performance as `accumulate1`.

Example

Suppose that for some algorithm, the exact number of steps is $T(n) = 5n^2 + 27n + 1005$. When *n* is small, say 1 or 2, the constant 1005 seems to be the dominant part of the function. However, as *n* gets larger, the n^2 term becomes the most important. In fact, when *n* is really large, the other two terms become insignificant in the role that they play in determining the final result. Again, to approximate $T(n)$ as *n* gets large, we can ignore the other terms and focus on $5n^2$. In addition, the coefficient 5 becomes insignificant as *n* gets large. We would say then that the function $T(n)$ has an order of magnitude $f(n) = n^2$, or simply that it is $O(n^2)$.

Self Check

Q1

Question

If the exact number of steps is $T(n) = 2n + 3n^2 - 1$ what is the Big O?

- ◯ O(2n)
- ◯ O(n)
- ◯ O(3n^2)
- ◯ O(n^2)
- ◯ More than one of the above

Q2

Parsons problem

Without looking at the graph above, from top to bottom order the following from most to least efficient.

Code fragments

```
constant

cubic

exponential

linear

log linear

logarithmic

quadratic
```

Q3

Question

Which of the following statements is true about the two algorithms? Algorithm 1: 100n + 1 Algorithm 2: n^2 + n + 1

- ◯ Algorithm 1 will require a greater number of steps to complete than Algorithm 2
- ◯ Algorithm 2 will require a greater number of steps to complete than Algorithm 1
- ◯ Algorithm 1 will require a greater number of steps to complete than Algorithm 2 until they reach the crossover point
- ◯ Algorithm 1 and 2 will always require the same number of steps to complete

Using summation facts

Sometimes we can combine our knowledge of asymptotic analysis and math facts to make algorithms more efficient.

Sum

Starting with the original accumulate algortihm.

```
accumulate1(int: N)
   sum <- 0
   count  <- N
   while count > 0
      sum <- sum + count
      count <- count - 1
   done while
   return sum
done accumulate
```

When we discussed *Summations*, we saw that is loop is equivalent to

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

We can use this fact to transform our $O(n)$ algorithm into $O(1)$. The cost of $O(1)$ algorithms is *constant*. *Constant time* algorithms do not grow more expensive as the size of n grows large.

Run It

```
#include <ctime>
#include <iostream>

clock_t start() {
   return clock();
}
void time_since(clock_t start) {
   clock_t end = clock();
   double elapsed_secs = double(end - start) / CLOCKS_PER_SEC;
   std::cout << std::fixed
             << " took "<< elapsed_secs << " seconds\n";
}

long accumulate1(long n){
   long sum = 0;
   for (long i = n; i > 0; --i){
       sum = sum + i;
   }
   return sum;
}

long accumulate2(long n){
    return (n*(n+1))/2;
}

int main(){

    for (int N=1000; N<1e6; N*=10) {
       clock_t begin = clock();
       std::cout << "N: " << N
                 << ", Sum1 = " << accumulate1(N) << '\t';
       time_since(begin);
    }

    for (int N=1000; N<1e6; N*=10) {
       clock_t begin = clock();
       std::cout << "N: " << N
                 << ", Sum2 = " << accumulate2(N) << '\t';
       time_since(begin);
    }

    return 0;
}
```

There are two important things to notice about this output. First, the times recorded above are shorter than any of the previous examples. Second, they are very consistent no matter what the value of n. It appears that `accumulate2` is hardly impacted by the number of integers being added.

But what does this benchmark really tell us? Intuitively, we can see that the iterative solutions seem to be doing more work since some program steps are being repeated. This is likely the reason it is taking longer. Also, the time required for the iterative solution seems to increase as we increase the value of n. However, there is a problem. If we run the same function on a different computer or used a different programming language, we would get different results. It could take even longer to perform `accumulate2` if the computer were older.

Asymptotic analysis gives us the tools to definitively state, without resorting to measuring execution time, that the `accumulate2` runs in *constant time*, while the `accumulate1` function runs in $O(n)$ time.

Pitfall: Confusing upper bound and worst case

A common mistake people make is confusing the upper bound and worst case cost for an algorithm. The upper bound represents the highest growth rate an algorithm may have for size n. The sequential search algorithm we discussed in *Best, worst, and average cases* involved 3 key input cases:

1. When the target value was in the first element (base case)
2. When the target value was not found (worst case)
3. The average cost for all possible locations, which works out to $n/2$

In the best case, only a single element is visited. Accordingly, the upper bound for this algorithm in the best case is $O(1)$. Even when n grows large, the cost for the base case is constant.

In the worst case, every element is visited. Accordingly, the upper bound for this algorithm in the worst case is $O(n)$. No matter the value of n, for some constant c, cn is bigger than n.

In the average case, about $\frac{n}{2}$ elements are visited. The upper bound for this algorithm in the average case is also $O(n)$. As n grows large, the denominator becomes insignificant. No matter the value of n, for some constant c, cn is bigger than $n/2$.

Therefore, question we should always consider is: *what is the upper bound of our algorithm in the best / average / worst case*? And the answer should be (sequential search):

- $O(1)$ in the **best case**
- $O(n)$ in the **worst case**
- $O(n)$ in the **average case**

More to Explore

- TBD

Acknowledgements

This section is adapted from Problem Solving with Algorithms and Data Structures using C++, by Brad Miller and David Ranum, Luther College, and Jan Pearce, Berea College released under the CC BY-NC-SA 4.0.

2.3.5 Lower Bounds

Big-O notation describes an upper bound. In other words, big-O notation states a claim about the greatest amount of some resource (usually time) that is required by an algorithm for some class of inputs of size n (typically the worst such input, the average of all possible inputs, or the best such input).

Similar notation is used to describe the least amount of a resource that an algorithm needs for some class of input. Like big-O notation, this is a measure of the algorithm's growth rate. Like big-O notation, it works for any resource, but we most often measure the least amount of time or storage required. And again, like big-O notation, we are measuring the resource required for some particular class of inputs: the worst-, average-, or best-case input of size n.

The *lower bound* for an algorithm (or a problem, as explained later) is denoted by the symbol Ω, pronounced "big-Omega" or just "Omega". The following definition for Ω is symmetric with the definition of big-O.

> For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \geq cg(n)$ for all $n > n_0$.

Example

Assume $\mathbf{T}(n) = c_1 n^2 + c_2 n$ for c_1 and $c_2 > 0$. Then,

$$c_1 n^2 + c_2 n \geq c_1 n^2$$

for all $n > 1$. So, $\mathbf{T}(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

It is also true that the equation of the example above is in $\Omega(n)$. However, as with big-O notation, we wish to get the "tightest" (for Ω notation, the largest) bound possible. Thus, we prefer to say that this running time is in $\Omega(n^2)$.

Recall the sequential search algorithm to find a value K within an array of integers. In the average and worst cases this algorithm is in $\Omega(n)$, because in both the average and worst cases we must examine *at least* cn values (where c is 1/2 in the average case and 1 in the worst case).

An alternate definition

An alternate (non-equivalent) definition for Ω is

> $\mathbf{T}(n)$ is in the set $\Omega(g(n))$ if there exists a positive constant c such that $\mathbf{T}(n) \geq cg(n)$ for an infinite number of values for n.

This definition says that for an "interesting" number of cases, the algorithm takes at least $cg(n)$ time. Note that this definition is *not* symmetric with the definition of big-O. For $g(n)$ to be a lower bound, this definition *does not* require that $\mathbf{T}(n) \geq cg(n)$ for all values of n greater than some constant. It only requires that this happen often enough, in particular that it happen for an infinite number of values for n. Motivation for this alternate definition can be found in the following example.

Assume a particular algorithm has the following behavior:

$$\mathbf{T}(n) = \begin{cases} n & \text{for all odd } n \geq 1 \\ n^2/100 & \text{for all even } n \geq 0 \end{cases}$$

From this definition, $n^2/100 \geq \frac{1}{100} n^2$ for all even $n \geq 0$. So, $\mathbf{T}(n) \geq cn^2$ for an infinite number of values of n (i.e., for all even n) for $c = 1/100$. Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

For this equation for $\mathbf{T}(n)$, it is true that all inputs of size n take at least cn time. But an infinite number of inputs of size n take cn^2 time, so we would like to say that the algorithm is in $\Omega(n^2)$. Unfortunately, using our first definition will yield a lower bound of $\Omega(n)$ because it is not possible to pick constants c and n_0 such that $\mathbf{T}(n) \geq cn^2$ for all $n > n_0$. The alternative definition does result in a lower bound of $\Omega(n^2)$ for this algorithm, which seems to fit common sense more closely. Fortunately, few real algorithms or computer programs display the pathological behavior of this example. Our first definition for Ω generally yields the expected result.

Clearly asymptotic bounds notation is not a law of nature. It is merely a powerful modeling tool used to describe the behavior of algorithms.

2.3.6 Theta Notation

The definitions for big-O and Ω give us ways to describe the upper bound for an algorithm (if we can find an equation for the maximum cost of a particular class of inputs of size n) and the lower bound for an algorithm (if we can find an equation for the minimum cost for a particular class of inputs of size n). When the upper and lower bounds are the same within a constant factor, we indicate this by using Θ (big-Theta) notation. An algorithm is said to be $\Theta(h(n))$ if it is in $O(h(n))$ *and* it is in $\Omega(h(n))$. Note that we drop the word "in" for Θ notation, because there is a strict equality for two equations with the same Θ. In other words, if $f(n)$ is $\Theta(g(n))$, then $g(n)$ is $\Theta(f(n))$.

Because the sequential search algorithm is both in $O(n)$ and in $\Omega(n)$ in the average case, we say it is $\Theta(n)$ in the

average case.

Given an algebraic equation describing the time requirement for an algorithm, the upper and lower bounds always meet. That is because in some sense we have a perfect analysis for the algorithm, embodied by the running-time equation. For many algorithms (or their instantiations as programs), it is easy to come up with the equation that defines their runtime behavior. The analysis for most commonly used algorithms is well understood and we can almost always give a Θ analysis for them. However, the class of NP-Complete problems all have no definitive Θ analysis, just some unsatisfying big-O and Ω analyses.

Even some "simple" programs are hard to analyze. Consider the *Collatz Conjecture*. The Collatz Conjecture or 3x+1 problem can be summarized as follows:

> Take any positive integer n. If n is even, divide n by 2. If n is odd, multiply n by 3 and add 1. Repeat the process indefinitely. The conjecture states that no matter which number you start with, you will always reach 1 eventually.

Nobody currently knows the true upper or lower bounds for the conjecture, because it is unknown if it is true.

```
while n > 1
   if n is odd
      n <- 3*n + 1
   else
      n <- n / 2
   done if
done while
```

While some textbooks and programmers will casually say that an algorithm is "order of" or "big-O" of some cost function, it is generally better to use Θ notation rather than big-O notation whenever we have sufficient knowledge about an algorithm to be sure that the upper and lower bounds indeed match. We will use Θ notation in preference to big-O notation whenever our state of knowledge makes that possible. Limitations on our ability to analyze certain algorithms may require use of big-O or Ω notations. In rare occasions when the discussion is explicitly about the upper or lower bound of a problem or algorithm, the corresponding notation will be used in preference to Θ notation.

Classifying Functions

Given functions $f(n)$ and $g(n)$ whose growth rates are expressed as algebraic equations, we might like to determine if one grows faster than the other. The best way to do this is to take the limit of the two functions as n grows towards infinity,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}.$$

If the limit goes to ∞, then $f(n)$ is in $\Omega(g(n))$ because $f(n)$ grows faster. If the limit goes to zero, then $f(n)$ is in $O(g(n))$ because $g(n)$ grows faster. If the limit goes to some constant other than zero, then $f(n) = \Theta(g(n))$ because both grow at the same rate.

Example

If $f(n) = n^2$ and $g(n) = 2n \log n$, is $f(n)$ in $O(g(n))$, $\Omega(g(n))$, or $\Theta(g(n))$? Since

$$\frac{n^2}{2n \log n} = \frac{n}{2 \log n},$$

we easily see that

$$\lim_{n\to\infty} \frac{n^2}{2n \log n} = \infty$$

because n grows faster than $2 \log n$. Thus, n^2 is in $\Omega(2n \log n)$.

Pitfall: Confusing lower bound and best case

A common mistake people make is confusing the lower bound and best case cost for an algorithm. In part this is because for simple algorithms, they look just like the upper bound.

The lower bound represents the least cost an algorithm needs for a problem of size n.

In the best case, only a single element is visited. Accordingly, the lower bound for this algorithm in the best case is $\Omega(1)$. Even when n grows large, the cost for the base case is constant.

In the worst case, every element is visited. $\Omega(n)$ is a lower bound for the cost of the algorithm in the worst case because the worst case must **always** examine n records.

In the average case, about $\frac{n}{2}$ elements are visited. The lower bound for this algorithm in the average case is also $\Omega(n)$. As n grows large, the denominator becomes insignificant. No matter the value of n, for some constant c, cn is less than or equal to the average cost of $n/2$.

For this simple algorithm the upper and lower bounds are the same in the best / average / worst case:

- $O(1)$ in the **best case**
- $O(n)$ in the **worst case**
- $O(n)$ in the **average case**

Then why do we have upper and lower bounds in the first place, if they work out to be the same?

In the case of functions like sequential search that are perfectly understood, they **are** the same. The upper and lower bound will only be different when we are describing what we know about an algorithm that we **don't know the exact cost for**.

This is what Θ is for. It is a shorthand we use to say the upper and lower bounds match. We can say the cost is Θ some value.

Sequential search has worst case cost $\Theta(n)$ because the upper and lower bounds are the same.

Self Check

Q1

Question

Determine the proper relationship between the following pair of functions.

$f(n) = \sqrt{n}$

$g(n) = \log n^2$

- ◯ $f(n)$ is $\Omega(g(n))$
- ◯ $f(n)$ is $O(g(n))$
- ◯ $f(n)$ is $\Theta(g(n))$

Q2

Question

Determine the proper relationship between the following pair of functions.

$f(n) = \log n^2$

$g(n) = \log n + 5$

- ◯ $f(n)$ is $\Omega(g(n))$
- ◯ $f(n)$ is $O(g(n))$

◯ $f(n)$ is $\Theta(g(n))$

More to explore

- Collatz Conjecture on wikipedia.

Acknowledgements

This section is adapted from Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

How long will it take to process the company payroll once we complete our planned merger? Should I buy a new payroll program from vendor X or vendor Y? If a particular program is slow, is it badly implemented or is it solving a hard problem? Questions like these ask us to consider the difficulty of a problem, or the relative efficiency of two or more approaches to solving a problem.

Asymptotic analysis allows us to compare the relative costs of two or more algorithms for solving the same problem. Asymptotic analysis also gives algorithm designers a tool for estimating whether a proposed solution is likely to meet the resource constraints for a problem before they implement an actual program. After reading this chapter, you should understand

- the concept of a *growth rate*, the rate at which the cost of an algorithm grows as the size of its input grows;
- the concept of an *upper bound* and *lower bound* for a growth rate, and how to estimate these bounds for a simple program, algorithm, or problem; and
- the difference between the cost of an *algorithm* (or program) and the cost of a *problem*.

The chapter concludes with a brief discussion of the practical difficulties encountered when empirically measuring the cost of a program, and some principles for code tuning to improve program efficiency.

CHAPTER

THREE

CHAPTERS

3.1 Development tools

This chapter introduces the development environment used throughout the book and what you need to do in order to complete assignments.

Understanding the first 3 section of this chapter are critical - all assignments are turned in using `git` and are compiled and tested using `cmake`. People who plan to complete assignments on a system other than the Mesa server may find the section *Compiling code on your local computer* useful.

3.1.1 Git setup

All of the assignments in this course are turned in using a source code change management tool called `git`. Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. If you continue working as a programmer, even if your team doe not use git, it most certainly will use a tool with similar capabilities.

To use Git on the command line, you'll need to download, install, and configure Git on your computer.

Note

You will not be able to turn in any assignments until **all three** steps have been completed.

Step 1: Install Git

Download and install the version of git appropriate for your operating system from the git download site. You may down load the GUI if you like, however, all of the examples in this text will be on the command line.

If you are only using the Mesa server, you can skip this step. Git has been installed for you.

Step 2: Setup your username and email address in Git

Git uses a username to associate commits with an identity. The Git username is not the same as your GitHub username. In this class, you should use the name you used to register for this course.

1. Open a terminal or Git Bash
2. Set your Git user name for all repositories on this computer:

```
git config --global user.name "Mona Lisa"
```

3. Confirm your Git user name is set correctly:

```
git config --global user.name
```

which should display in this case:

```
Mona Lisa
```

GitHub uses your commit email address to associate commits with your GitHub account. You can choose the email address that will be associated with the commits you push from the command line as well as web-based Git operations you make.

1. Open a terminal or Git Bash
2. Set your Git email address for all repositories on this computer:

```
git config --global user.email "email@example.com"
```

3. Confirm your Git user email is set correctly:

```
git config --global user.email
```

which should display in this case:

```
email@example.com
```

If you want to receive email when I create issues or comment on your code, then you should use a valid email address.

If you want to keep your email address private, you can do this on GitHub. For more information see the Manaing email page on GitHub.

Step 3: Setup authentication

When you connect to a GitHub repository from Git, you'll need to authenticate with GitHub using either HTTPS or SSH.

Starting in 13 August 2021, basic authentication using a password to Git no longer works. GitHub requires the use of token-based authentication (for example, a personal access token, or an SSH key) for all authenticated Git operations. See this blog post for details.

Although GitHub provides many authentication choices, in this class you will need to be able to access GitHub from the Mesa server. This means you need to use either a personal access token or SSH.

You do not need to create both, but you are free to do so.

- A personal access token only works when performing Git operations over HTTPS.
- An SSH key only works when performing Git operations over SSH.

Creating an SSH key

You can connect to GitHub using the Secure Shell Protocol (SSH), which provides a secure channel over an unsecured network.

Using the SSH protocol, you can connect and authenticate to remote servers and services. With SSH keys, you can connect to GitHub without supplying your username and personal access token at each visit.

Note

As a security precaution, GitHub automatically removes your inactive SSH keys that have not been used for a year. For more information, see "Deleted or missing SSH keys."

If you don't already have an SSH key, then you must generate a new SSH key to use for authentication. If you're unsure whether you already have an SSH key, then you can check for existing keys. For more information, see "Checking for existing SSH keys."

To create an SSH key do the following:

1. Login to the Mesa server or open a local terminal or Git Bash shell
2. Paste the text below, substituting in your GitHub email address:

   ```
   ssh-keygen -t ed25519 -C "your_email@example.com"
   ```

   This creates a new ssh key, using the provided email as a label. You should see:

   ```
   Generating public/private ed25519 key pair.
   ```

3. When you're prompted to "Enter a file in which to save the key," press Enter. This accepts the default file location.

   ```
   Enter file in which to save the key (/home/students/fire/fire40/.ssh/id_ed25519):
   ```

   You should see:

   ```
   Created directory '/home/students/fire/fire40/.ssh'.
   ```

4. At the prompt, type a secure passphrase. For more information, see "Working with SSH key passphrases."

   ```
   Enter passphrase (empty for no passphrase): [Type a passphrase]
   Enter same passphrase again: [Type passphrase again]
   ```

   You will see some other output, but most important is the file containing your public key, which will be similar to:

   ```
   Your public key has been saved in /var2/home/fire/fire40/.ssh/id_ed25519.pub.
   ```

   Make note of that, as you'll need it in the next steps.

To configure your GitHub account to use your SSH key, you'll also need to add it to your GitHub account.

1. First, copy your public SSH key.

   ```
   cat ~/.ssh/id_ed25519.pub
   ```

   This command will display the contents of your public key which you can then copy and paste in the browser. The ~ is just an alias for your home directory.

   Caution

   Private keys are sensitive data!

   Treat your private keys like passwords and keep them secret.

> Your private key is stored in a separate file - by default it is the same file name as your public key without the ".pub" file extension.
>
> Never post your private key on a web page or any other public location even for a short time.
>
> If you think your private key has been compromised, you should delete it and create a new one.

2. In the upper-right corner of any page on GitHub, click your profile photo, then click **Settings**.
3. In the user settings sidebar, click **SSH and GPG keys**.
4. Click **New SSH key** or **Add SSH key**.
5. In the "Title" field, add a descriptive label for the new key. For example, if you created this on the Mesa server, you might call this key "Mesa College CISC187".
6. Paste your key into the "Key" field.
7. Click **Add SSH key**.
8. If prompted, confirm your **GitHub password**.

Now you are ready to use your SSH key.

Using an SSH key on the command line

After you've set up your SSH key and added it to your GitHub account, you can use it. Since the setup is more complicated than a simple token, it's a good idea to test it first. When you test your connection, you'll need to authenticate using your SSH key passphrase you created earlier.

Test your SSH keys:

1. Login to the Mesa server (or other location of you public/private SSH key pair)
2. Enter the following:

```
ssh -T git@github.com
```

   This will attempt to connect to github.com over ssh.

   You may see a warning like this:

```
The authenticity of host 'github.com (IP ADDRESS)' can't be established.
RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.
Are you sure you want to continue connecting (yes/no)?
```

3. Verify that the fingerprint in the message you see matches GitHub's RSA public key fingerprint. If it does, then type yes:

```
Hi username! You've successfully authenticated, but GitHub does not
provide shell access.
```

4. Verify that the resulting message contains your username. If you receive a "permission denied" message, see "Error: Permission denied (publickey)".

A common source of error is using your GitHub username over SSH.

Note

Always use the "git" user

All connections, including those for remote URLs, must be made as the "git" user. If you try to connect with your GitHub username:

```
ssh -T GITHUB-USERNAME@github.com
```

then it will fail:

```
Permission denied (publickey).
```

Once your SSH key is setup correctly, the only real difference in using it is the URL you use to clone a repository. To clone using SSH use the following command:

```
git clone git@github.com:DaveParillo/cisc187-TTYY-fireNN.git
```

In the above URL, replace **TT** with either

- **sp** in the spring term
- **fa** in the fall term

and replace **YY** with the 2 digit year of the semester and replace **NN** with your 2 digit fire number assigned to you.

For example:

```
git clone git@github.com:DaveParillo/cisc187-sp21-fire01.git
```

Creating a GitHub personal access token

Personal access tokens (PATs) are an alternative to using passwords for authentication to GitHub when using the git command line.

Note

As a security precaution, GitHub automatically removes personal access tokens that haven't been used in a year.

Follow the instructions on GitHub to create a personal access token

At a minimum, you will want to ensure you give the token access to repositories from the command line. At the "scopes" or "permissions" screen, ensure you have **repo** selected.

After you have copied your token and leave the token creation page, you will not be able to see it again.

Caution

Tokens are sensitive data!

Treat your tokens like passwords and keep them secret.

A PAT is not your password, but it provides access to your source code.

If you think your token has been compromised, delete it on GitHub and make a new token.

Using a token on the command line

Once you have a token, you can enter it instead of your password when performing Git operations over HTTPS.

For example, to get a copy of all your assignments on the command line you would enter the following:

```
git clone https://github.com/DaveParillo/cisc187-TTYY-fireNN.git
Username: your_username
Password: your_token
```

In the above URL, replace **TT** with either

- **sp** in the spring term
- **fa** in the fall term

and replace **YY** with the 2 digit year of the semester and replace **NN** with your 2 digit fire number assigned to you.

Do **not** enter your password at the prompt. Yes, I know it says 'password'. Enter your personal access token Personal access tokens can only be used for HTTPS Git operations.

Instead of manually entering your PAT for every HTTPS Git operation, you can cache your PAT with a Git client. Git will temporarily store your credentials in memory until an expiry interval has passed. You can also store the token in a plain text file that Git can read before every request.

For more information, see "Caching your GitHub credentials in Git."

More to Explore

- Git Documentation
- GitHub quickstart
- Creating SSH Keys
- Creating a GitHub Personal Access Token

3.1.2 Tutorial: Using Git

When you are programming, you will make mistakes. If you program long enough, these will eventually include big mistakes like accidentally deleting all of your source files. You are also likely to spend some of your time trying out things that don't work, at the end of which you'd like to go back to the last version of your program that did work. All these problems can be solved by using a **version control system**.

Think of `git` or any other version control system as a time machine for any work you do. But remember: you can only "go back in time" to versions of your work you saved in version control. More on that later.

A brief summary of git is given below. For more details, see the tutorials available at https://git-scm.com/.

Whatever version control software you use, they all follow the same basic pattern:

1. Initialize repository

   You only need to do this step once.

   If you already have a repository available on a remote server, then you can use `git clone` instead of `git init`.

2. Work with repository

   a. Create new files

b. Add files to repository

c. Edit existing files

d. Commit changes

e. Push local changes to a remote server like GitHub for safe keeping.

Repeat the above as often as needed.

Initialize a git repository

Typically you run git inside a directory that holds some project you are working on (for example, homework). Before you can do anything with git, you need to create or copy a repository, which is a hidden directory .git that records changes to your files.

In this example, we will be walking though a small empty repository.

If we choose to put it in a new directory *git-demo*:

```
$ mkdir git-demo
$ cd git-demo/
$ git init
```

Then we should see something like:

```
Initialized empty Git repository in /home/runner/WorrisomeSophisticatedCybernetics/git-
↪demo/.git
```

Now let's create a file and use `git add` to add it to the repository:

```
$ echo 'int main(int argc, char** argv) { return 0; }' > tiny.cpp
$ git add tiny.cpp
```

The `git status` command will tell us that Git knows about `tiny.cpp`, but hasn't committed the changes to the repository yet:

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

  new file:   tiny.cpp
```

The `git commit` command will save the actual changes, along with a message saying what you did. For short messages, the easiest way to do this is to include the message on the command line:

```
$ git commit -m 'a very short c++ program'
[master (root-commit) 3a6fd19] a very short c++ program
 1 file changed, 1 insertion(+)
 create mode 100644 tiny.cpp
```

Without the `-m` argument, git runs the default editor (vim) to let you edit your commit message. If you don't like vim, you can change the default using `git config`:

```
$ git config --global core.editor "emacs -nw"
```

You can see what commits made so far using `git log`:

```
$ git log
commit 3a6fd19e8ef4662744bd41a20cde9924aad918ed
Author: DaveParillo <DaveParillo@noreply.github.com>
Date:   Sat Jun 10 12:07:51 2017 -0700

    a very short c++ program
```

Try This!

Head on over to replit.com and use the console window in a repl and practice the steps described in this section.

You can run git commands in repl's for any language, but in order to compile `tiny.cpp`, you'll need to be in a C++ repl.

Editing files

Suppose I edit tiny.cpp using my favorite editor to turn it into the classic hello-world program:

```
#include <iostream>

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

I can see what files have changed using git status:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

  modified:   tiny.cpp

no changes added to commit (use "git add" and/or "git commit -a")
```

Notice how Git reminds me to use `git commit -a` to include these changes in my next commit. I can also do `git add tiny.cpp` to only include the changes to tiny.cpp (maybe I made changes to a different file that I want to commit separately).

If I want to know the details of the changes since my last commit, I can run `git diff`:

```
$ git diff
diff --git a/tiny.cpp b/tiny.cpp
index a9b8738..a6501a7 100644
--- a/tiny.cpp
+++ b/tiny.cpp
@@ -1 +1,6 @@
```

(continues on next page)

(continued from previous page)

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

Since I like these changes, I commit them:

```
$ git commit -a -m 'turn tiny into a basic hello world'
[master 170eaf0] turn tiny into a basic hello world
 1 file changed, 6 insertions(+), 1 deletion(-)
```

The repository now contains two commits:

```
$ git log | more
commit 170eaf0461a7f0f865328b73bee6d313c3dbad42
Author: DaveParillo <DaveParillo@noreply.github.com>
Date:   Sat Jun 10 12:23:55 2017 -0700

    turn tiny into a basic hello world

commit 3a6fd19e8ef4662744bd41a20cde9924aad918ed
Author: DaveParillo <DaveParillo@noreply.github.com>
Date:   Sat Jun 10 12:07:51 2017 -0700

    a very short c++ program
```

Renaming files

You can rename a file with `git mv`. This is just like the regular Linux `mv` command, except that it tells Git what you are doing. If you forget to use `git mv` it's not normally a problem. Unless your changes are massive, git is usually good about figuring out when files have been moved:

```
$ git mv tiny.cpp hello.cpp
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  renamed:    tiny.cpp -> hello.cpp
```

Moving a file counts as a change. These changes don't get written to the repository unless you do another git commit:

```
$ git commit -m 'give better name to hello program'
[master 7a603f4] give better name to hello program
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename tiny.cpp => hello.cpp (100%)
```

Adding and removing files

To add a file, create it and call `git add`:

```
$ cp hello.cpp goodbye.cpp
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

  goodbye.cpp

nothing added to commit but untracked files present (use "git add" to track)
$ git add goodbye.cpp
$ git commit -m 'started to make a second program to say goodbye'
[master f41cb3a] started to make a second program to say goodbye
 1 file changed, 6 insertions(+)
 create mode 100644 goodbye.cpp
```

If you add many files at once, you can refer to the directory they are in. If that directory is the current directory, `.` is acceptable. When you specify a directory, then all of the files new or modified are added recursively from that point downward.

Git add and commit best practices

It is easy to inadvertently add files you did not mean to when adding a directory. Check what you have added using `git status` to ensure the files you are about to commit belong in the commit.

If you accidentally add files you did not mean to, then it is easy to "un-add" them using `git revert`.

General rules for adding files:

1. Commit only source files you have created or modified.
2. Avoid committing binary files and generated build artifacts.

To remove a file, use `git rm`:

```
$ git rm goodbye.cpp
rm 'goodbye.cpp'
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  deleted:    goodbye.cpp

$ git commit -m 'on second thought, goodbye.cpp was a bad idea'
[master cbcf75f] on second thought, goodbye.cpp was a bad idea
 1 file changed, 6 deletions(-)
 delete mode 100644 goodbye.cpp
```

Recovering files from the repository

Nothing is ever truly deleted from the repository once checked in. If you accidentally delete something, you can recover it from the repository.:

```
$ ls -a
./  ../  .git/      hello.cpp

$ rm hello.cpp
$ ls -a
./  ../  .git/

# gone, but not forgotten

$ git checkout -- hello.cpp
$ ls -a
./  ../  .git/      hello.cpp
```

Using `git checkout --` gets the most recent version out of the repository, but using the commit id, we can operate on any version:

```
$ git checkout 3a6f -- tiny.cpp
$ ls -a
./  ../  .git/      hello.cpp  tiny.cpp
```

Because tiny.cpp is not part of the current HEAD (most recent version), it is considered a new file, but the checkout did add tiny.cpp and stage it for commit:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  new file:   tiny.cpp
```

More to Explore

- Git Home

  All Git commands take a `--help` argument that brings up their manual page. There is also extensive documentation at https://git-scm.com/.

Endnotes

1. Content in this section is adapted from https://www.cs.yale.edu/homes/aspnes/classes/223/notes.html

3.1.3 Building software

In this section we will learn one way to compile and run programs. If you plan to use the Mesa Linux server, then this section describes what you need to know. If you plan to use other programs to compile and run your programs you **still** need to review at least the section describing CMake. All of the labs and projects use the *CMake* build system.

What do we really mean when we say "build software"? A **software build** (when used as a verb) is the process of converting a set of **text** and related **build files** into machine-readable format. Our job as programmers almost always boils down to converting plain text files into something else. A *build* often creates an executable file or a library that can be included as part of a larger executable. But just as often programmers convert text into some other form of text:

- Convert XML to HTML, for example with XSLT
- Convert JSON to XML
- Convert a language independent interface definition into a language specific one
- Transpile Java code into JavaScript

We could list more, but you get the point. We generally do not perform these conversions manually, we use tools.

In this class we will be using a core set of tools to build and test programs:

CMake
: An open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. It generates native makefiles, projects, and workspaces that can be used in the compiler environment of your choice.

Clang
: The Clang project provides a language front-end and tooling infrastructure for languages in the C language family. For our purposes the Clang project provides:

  - `clang`: a C compiler
  - `clang++`: a C++ compiler
  - `clang-tidy`: a C++ "linter"

    A linting tools is a program that analyzes your source code statically (without running it) to check for possible issues or bugs.
  - `clang-format`: a source code (re) formatter.
  - `lldb`: a debugger.

    A debugger is a program that allows you to control and view internal details of your program. You can pause execution, view or change variables, and see source code. As the name implies, the tool is usually used to help find and fix bugs.

  There are many other tools in the Clang suite, but these are the ones you'll interact with most often.

  Clang is the default compiler on MacOS and is an option on most Linux based systems.

Doctest
: doctest is a fast C++ testing framework based on the popular Catch Unit Test framework. Most labs use doctest to assess your work as you progress through the lab.

GCC
: The GNU Compiler Collection (GCC) provides a tooling infrastructure for C, C++, FORTRAN, and a few other languages. For our purposes GCC provides:

  - `gcc`: a C compiler
  - `g++`: a C++ compiler
  - `gdb`: a debugger.

  GCC is the default compiler for most Unix and Linux based systems.

Git
: Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files.

Make
: The `make` program is a tool used to help make things. Using *make* we define *targets* in a text file and each *target* runs a *recipe*. Using targets and recipes in combination allows make to perform many complex tasks automatically.

On Linux based systems these tools work together like this:

- We use `git` to download the latest copy of our code.
- We use `CMake` to create our build files, which are generally Makefiles.
- Once the Makefile has been created, we use `make` to perform tasks. The simplest command `make` will try to build all the software in the current directory.

  Make will use either `gcc` or `clang` to actually compile code.
- Test cases are written using the `doctest` library and build using `make`.

How to access the Mesa Server

All projects and some of the lab assignments are required to be handed in on the San Diego Mesa College server **buffy**. Access is only available using a *secure shell* client program (ssh). There are a few ways to access the server using `ssh`. Unless you have a very old computer, then you already have ssh installed.

Open Git Bash, or a terminal window, or Power Shell, depending on your operating system. Then type the following in the *GIT Bash* to connect to the server:

```
ssh fireNN@209.129.16.61
```

where *fireNN** is the user name assigned to you by the instructor.

For the impatient: A quick summary

Lab build files are generated using CMake. Once you have a build file generated for your particular environment, then you may compile the software and run tests.

Note

All of these steps are demonstrated on a **nix* style operating system: GNU/Linux, Unix, MacOS, or Cygwin on Windows.

It also assumes you have already cloned your assignments repository.

1. First, open a terminal since all of the command that follow are typed on the command line.
2. Login to the Mesa server using `ssh`.
3. Once logged in, change directory to the folder containing a lab

   For example:

   ```
   cd cisc187-sp23-fire40/lab01-hello
   ```

   Your fire number and semester will be different.

   If you do not have a directory starting with `cisc187-` in your home directory then clone your repository.

4. Create a new directory to hold the build files and have `cmake` generate the makefiles:

```
# make a directory to store build output and configure
mkdir build
cd build
cmake ..
```

   You only need to do this step once when you make the build directory initially.

5. Now you can build the project:

```
make
```

   and run the tests

```
make test
```

That it!

There are many ways to run cmake and steps 4 and 5 above are what you commonly see on the internet. One alternative is:

```
# make a directory to store build output and configure
cmake -S /path/to/lab01-hello -B build


# build the project
cmake --build build


# run the tests
cmake --build build --target test
```

This video demonstrates these steps and shows what normal results should look like.

Video

Video URL: https://www.youtube.com/watch?v=nQ31ApyU7_o

Most of the time you'll just be compiling code and running lab tests:

- *cd build*
- *make*
- *make test*

The make target `test` runs **all** the tests. Usually when working on a lab, you just want to compile and test that step. All labs are 'chunked' into steps with a separate test program to test it.

You can compile a single test step by referring to the numbered lab step, for example:

```
make step1
```

You can run a single test using either the `make` target provided or by running the test using the `ctest` program provided by `CMake`. for example:

```
test/step1
ctest -R step1
```

Both of these commands return the exact same output.

3.1.4 Compiling code on your local computer

The following sections describe briefly how to get started with a local development environment, if you wish. One of the primary motivations for building locally is to avoid a persistent internet connection running your SSH session.

If you have bad or intermittent internet connectivity, then this section is for you.

If you plan to use the Mesa server, then you can ignore the rest of this section.

As with all things C++, you have choices. The options described here do not represent all of the many ways one might build C++ programs locally, but are common choices on Windows and MacOS machines.

In most cases, you still need to install git and CMake.

This book does **not** explain how to install these *IDE's*. Use the documentation provided with your IDE for that.

Compiling using the CISC187 Docker image

Use git to clone your assignments repository to your computer.

The CISC187 Docker image provides an environment much like the development environment on the Mesa server, but locally.

Currently, two compilers are installed on the image:

- GCC 14
- Clang 17

The image includes support tools, debuggers, vim plugins, and checking scripts that are installed on the Mesa server.

Install

In order to use the docker image, you first need to install docker for your operating system.

Note

Windows operating system requirements

Windows 10 Professional or Enterprise is required for Docker on Windows using Hyper-V.

Docker uses a hypervisor with a VM, and the host server (your computer) must support virtualization. Since older Windows versions and Windows 10 Home edition do not support Hyper-V.

For Windows Home or Education builds running under WSL2 is an option. See the install documentation for details.

In any Windows build at least 4GB available RAM is recommended.

Once docker is installed, open a Terminal window, or on Windows, a Powershell terminal and type:

```
docker pull dparillo/cisc187
```

This command will download the CISC187 docker image and make it available to run.

Run

To run the docker image on windows type:

```
docker run --rm -it -v C:\Path\To\Source\Directory:/mnt/cisc187 dparillo/cisc187
```

Note

An important thing to notice is when mounting a volume with `-v` on Windows, the Windows part, left of the `:` uses Windows Path separator characters (\), and on the Linux side, Linux Path separator characters are used (/). The Windows file path must include the drive letter.

The same command on Mac or Linux:

```
docker run --rm -it -v /path/to/source:/mnt/cisc187 dparillo/cisc187
```

Meaning of these options:

`--rm:`
: Automatically remove the container when it exits. There is no need to save it. It is useful to think of docker containers as applications that perform some task and clean up when finished.

  One of the powerful things about this is that it is impossible to damage or corrupt your development environment. If you think you did something bad, exit the container and restart.

`-i:`
: Keep STDIN open even if not attached. Instead of the short `-i`, you can use `--interactive`.

`-t:`
: Allocate a pseudo TTY. This allows you to communicate with your docker container in the window where you started it. Instead of the short `-t`, you can use `--tty`.

`-v:`
: Bind mount a volume from the local computer onto the host. The general syntax is `-v /absolute/local/path:/absolute/container/path` Instead of the short `-v`, you can use `--volume`.

  The idea here is that your source code is never really inside the docker container. Your source code is separate, but visible to the running container.

The container mount point was not chosen at random. The container is set up with `/mnt/cisc187` as the *WORKDIR*. When the container starts, you start in this directory.

The run command has many more options available and docker has many more commands other than the run command, but this is all you need to know to compile assignments.

Once the CISC187 docker container is running you are ready to compile an assignment. Builds are exactly the same as on the Mesa server:

```
mkdir build
cd build
cmake ..
make
make test
```

Compiling with Visual Studio

In this course you need to be using Visual Studio 2022 at a minimum to complete all the assignments.

In order to enable CMake integration with Visual Studio ensure you have the additional C++ CMake tools for Windows installed.

Use git to clone your assignments repository to your computer. Now you are ready to compile an assignment.

GUI

These instructions describe how to build software using the Visual Studio Graphical User Interface (GUI).

1. Open the windows file explorer and go to the location where you cloned your repository.
2. Right-click on the lab you want to build, for example `lab01-hello`.

   Do **not** open the entire cloned repository.
3. Select 'Open in Visual Studio'

Terminal

These instructions describe how to build software using the Using the Visual Studio command line

1. Create a directory named build adjacent, but **not in** your source directory.
2. Open the Visual Studio Developer prompt. *cd* into the build directory created in the previous step.
3. Type *cmake ..*

   This should create a standard Visual Studio solution that you can run from the command line or the IDE.
4. Type *MSBuild lab1.sln* to build all projects in the **Debug** configuation
5. Type *ctest -C Debug* to run all tests

To remove all executable files:

```
MSBuild lab1.sln -target:Clean
MSBuild lab1.sln -t:Clean
```

To build a single test:

```
MSBuild lab1.sln -t:step1
```

To build all files in **Release** configuration, without any Debug symbols:

```
MSBuild lab1.sln -p:Configuration=Release
# run tests
ctest -C Release
```

If this doesn't work, try the instructions on the Microsoft site

Compiling with Code Blocks

Use git to clone your assignments repository to your computer. Now you are ready to compile an assignment.

1. Create a directory named build and open CMake GUI.
2. Select 'Browse Source' and select the folder containing the lab you want to build.
3. Select 'Browse Build' and select the *build* folder.

4. In the lower left corner, select 'Configure' and select 'CodeBlocks - MinGW Makefiles' from the list of available generators.

   Leave the radio selections alone and press 'Finish` when done.

   Campus windows computers may complain about a *sh.exe* program in your path outside of CodeBlocks. To fix this error:

   - Delete the CMake variable *CMAKE_SH* in the variables list.
   - Press 'Configure' a second time.

5. Press 'Generate'. When finished ("Generating done") close CMake GUI.
6. Open the generated "CBP" file in CodeBlocks. It should be in the build folder you pointed at in step 3.

Build the 'all' target to compile and link programs and tests. Test cases must be run individually - there is no target to run all the tests.

Compiling with Xcode

Use git to clone your assignments repository to your computer. Now you are ready to compile an assignment.

Open a terminal in the directory containing your lab, then:

```
mkdir build
cd build
cmake -G Xcode ..
```

Open the Xcode project and build as usual.

Compiling on Linux

Use git to clone your assignments repository to your computer.

You'll need to install a C++ tool chain, the details tend to vary by distribution, however, most Linux distributions have good documentation for installing C++ tools. The only thing you should verify is that your distro has a modern version of a C++ compiler (C++14 at a minimum) available. The GNU Compiler Collection (GCC) or Clang are preferred.

Once you have a tool chain installed, use git to clone your assignments repository to your computer. Now you are ready to compile an assignment.

The process is exactly the same as on the Mesa server. Open a terminal in the directory containing your lab, then:

```
mkdir build
cd build
cmake ..
make
make test
```

Which option should I choose?

There are a lot of options and the choices can be confusing. The short answer is that there is no wrong choice. Also, you can change you mind at any time and even shift from one compile option to another as you prefer.

So how are these options really different from each other?

1. The **I** in IDE stands for *integrated* They frequently include a large collection of tools to help with many tasks professional programmers encounter often.

   For this reason they tend to be large and use a fair amount of CPU and memory.

2. Accessing a remote server like buffy requires minimal CPU and memory locally. Most of the resources you are using are on the remote server. It is also the easiest to access. All the software you need is already installed on the server. You only need an ssh client.

   The main drawbacks are:

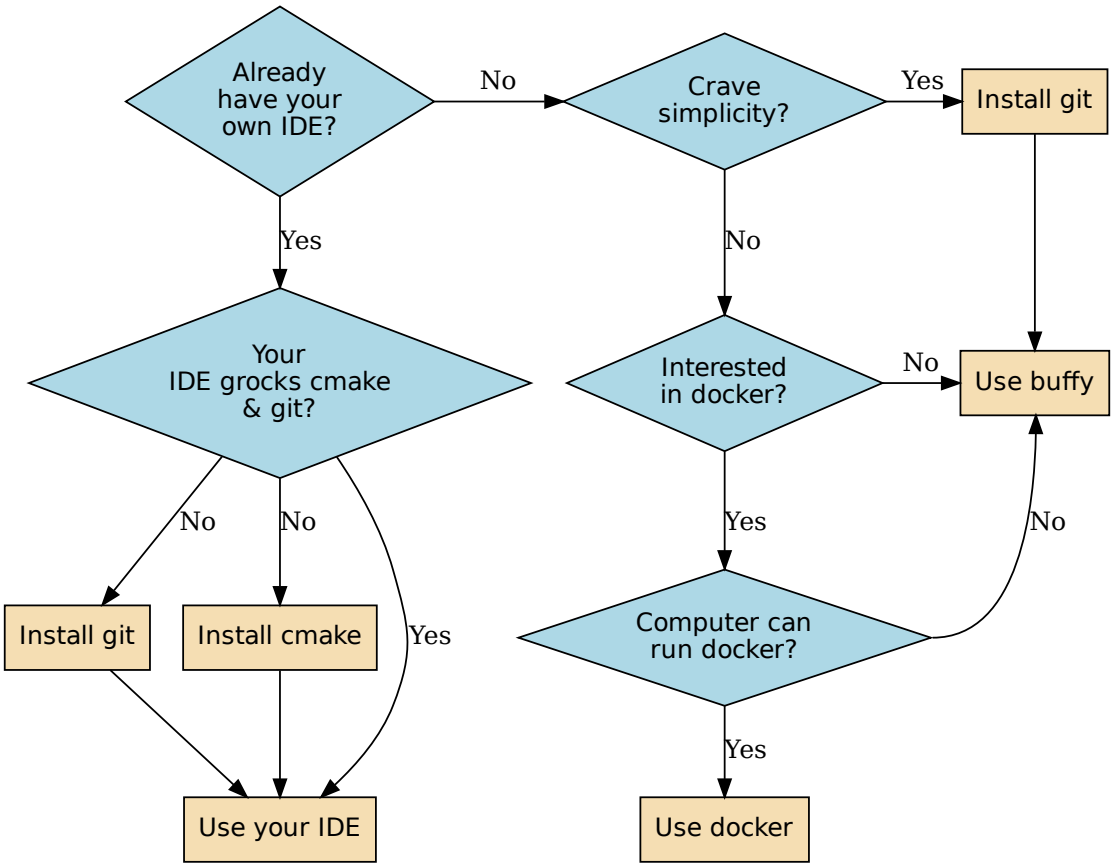
   - You have no control over the environment - you don't own the server.
   - Using the remote server requires good internet. If you lose your internet for any reason, then you will lose your connection.

3. Docker blends the two previous choices. You get a local server separate from your computer that has everything you need installed. It uses less resources than a typical IDE and if needed you can limit the resources it uses and like a local IDE, does not require persistent internet to work. Also, if you want you can modify the docker image and make your own custom version.

   The main drawbacks are:

   - After installing Docker there is anew persistent service running on your computer.
   - It is still not a real replacement for an IDE.

This decision chart may help.



More to Explore

- Sofware build
- Git Documentation
- Clang docs
  - clang-tidy
  - clang format
  - Clang C++ status
- cppfront : an experimental C++ Syntax compiler.
- *Command-line compiling*
- Grok definition

The remaining sections in this chapter are useful at various times throughout the course, but you should not try to master them all in the first week.

If you plan to work mostly on the Mesa server or the docker image then the introduction to GNU/Linux commands and the vim text editor are highly recommended.

3.1.5 Introducing Gnu/Linux

Early in its evolution, the command-line environment of UNIX (its only user interface back then) became dominated by dozens of small utilities and tools which are still in common use today. These tools are small and generally do one thing well. Most read from 'standard input' and write to 'standard output'. The tools are commonly chained together in longer command pipelines, one program passing its output to the next as input, and controlled by a variety of command-line options and arguments.

This is one aspect of UNIX that makes it a powerful environment for processing text-based data, one of its first uses in a corporate environment. Dump some text in one end of a command pipeline and retrieve processed output from the other end. As programmers, this is useful, because at its core, programming involves manipulating text: transforming source code into executable machine code, converting database query results into readable reports, etc.

This section is not intended to make you an expert if you have never used linux before. It is intended to give you enough of a jump start to demystify things somewhat and ease forward progress.

The command line

A command line, or terminal, is a text based interface to the system. You enter commands by typing them on the keyboard and feedback is returned as text.

The command line typically presents you with a prompt. As you type, it will be displayed after the prompt. Most of the time you will be issuing commands. Here is an example:

```
host@user: ls -l /home/user
total 12
-rw-rw---- 1 user student  419 May 29 06:49 main.cpp
-rw-rw---- 1 user student 3320 May 29 06:50 util.h
-rw-r--r-- 1 user student  725 May 29 06:49 vim_tips.md
host@user:
```

What's happening?

- The first line is the **prompt** (host@user). On this first line, a command was also entered `ls`. The `ls` command lists the contents of a directory. In this example, we also supplied some command line arguments (-l /home/user). It is conventional to start command line arguments (or 'switches') with a dash.
- The next 4 lines are output from the 'ls' command. Most commands produce output that is written to standard output, which in the case of running on the command line is the terminal itself.
- The last line displays the command prompt again. At this point, the system is waiting for the next command.

  If no prompt is displayed, this means the command is still running.

Absolute vs relative paths

Look at the argument passed to the `ls` command in the previous example:

```
/home/user
```

Specifically, this indicates the *absolute path* to the directory `/home/user`. There are 2 types of paths we can use, **absolute** and **relative**. All references to a file or directory use one of these forms.

Absolute path
: Specifies a location starting at the **root directory**. An absolute path *must* start with the character '/'.

Relative path
: Specifies a location in relation the current working directory. A relative path will *never* start with the character '/'.

When navigating or specifying paths, there are two special files which reside in every directory: `.` and `..`

.
: The single dot `.` file contains information about the files in the current directory. In many commands, it is synonymous with the current directory.

..
: The two dots `..` file contains information about the files in the directory that is the parent of the current directory. The only directory that does not contain a `..` file is the root directory: `/`.

In addition, the character ~ (tilde) is an abbreviation for the path to any user home directory.

```
ls ~alice/public_html
```

Displays the contents of the public_html folder in her home directory, regardless of where her home directory actually is located.

Self Check

Q1

Question

Which of the following statements are **relative paths**? Check all that apply.

- ☐ ../usr/bin
- ☐ /usr/bin
- ☐ ./
- ☐ /
- ☐ user/files

Q2

Question

Which of the following paths list the contents `/home/user`? Check all that apply.

- ☐ In directory <tt>/tmp</tt>, <tt>../home/user</tt>
- ☐ In directory <tt>/home</tt>, <tt>../home/user</tt>
- ☐ In directory <tt>/home/user</tt>, <tt>..</tt>
- ☐ In directory <tt>/home/user/work</tt>, <tt>..</tt>
- ☐ In directory <tt>/home/user/work</tt>, <tt>../../user</tt>

Basic commands

pwd
: Prints the working directory name. `pwd` prints the full path name of the directory you are currently in.

ls
: Print directory contents. With no arguments, `ls` prints the current directory contents, that is the contents of the directory returned by `pwd`. For each operand that names a file of type directory, ls displays the names of files contained within that directory, as well as any requested, associated information.

  The `ls` command has a **lot** of options.

cd
: Change directory. With no arguments, (just `cd`), this command will take you to **your** home directory. A file after the cd command `cd ../lab/solutions` changes the current working directory to the named path.

  The special name ~ is a shortcut for the users home directory.

mkdir
: Make a new directory. At least one argument is required. For example:

```
host@user: cd
host@user: ls -l
total 4
-rw-rw---- 1 user student  419 May 29 06:49 main.cpp
host@user: mkdir labs
host@user: ls -l
total 8
drwx------ 1 user student  419 May 29 15:06 labs
-rw-rw---- 1 user student  419 May 29 06:49 main.cpp
```

man
: Display manual pages for a command.

  There is an **enormous** amount of information available from the command line. Conceived of and written before the internet existed, it was intended to function as a comprehensive reference for everything a programmer would need to know to be productive in Unix. For example:

```
host@user: man ascii

ASCII(7)               Linux Programmer's Manual              ASCII(7)
NAME
   ascii - ASCII character set encoded in octal, decimal, and hexadecimal
```

(continues on next page)

(continued from previous page)

```
DESCRIPTION
   ASCII  is  the American Standard Code for Information Interchange.
   It is a 7-bit code.  Many 8-bit codes (such as ISO 8859-1, the Linux
   default character  set)  contain  ASCII  as  their lower half.
   The international counterpart of ASCII is known as ISO 646.

   The following table contains the 128 ASCII characters.

   C program '\X' escapes are noted.

   Oct   Dec   Hex   Char                        Oct   Dec   Hex   Char
   ------------------------------------------------------------------------
   000   0     00    NUL '\0'                    100   64    40    @
   001   1     01    SOH (start of heading)      101   65    41    A
   002   2     02    STX (start of text)         102   66    42    B
   003   3     03    ETX (end of text)           103   67    43    C
   004   4     04    EOT (end of transmission)   104   68    44    D
   005   5     05    ENQ (enquiry)               105   69    45    E
   006   6     06    ACK (acknowledge)           106   70    46    F
   007   7     07    BEL '\a' (bell)             107   71    47    G
   010   8     08    BS  '\b' (backspace)        110   72    48    H
   011   9     09    HT  '\t' (horizontal tab)   111   73    49    I
   012   10    0A    LF  '\n' (new line)         112   74    4A    J
   013   11    0B    VT  '\v' (vertical tab)     113   75    4B    K
   014   12    0C    FF  '\f' (form feed)        114   76    4C    L
   015   13    0D    CR  '\r' (carriage ret)     115   77    4D    M
   016   14    0E    SO  (shift out)             116   78    4E    N
   017   15    0F    SI  (shift in)              117   79    4F    O
   020   16    10    DLE (data link escape)      120   80    50    P
   021   17    11    DC1 (device control 1)      121   81    51    Q
   022   18    12    DC2 (device control 2)      122   82    52    R
   023   19    13    DC3 (device control 3)      123   83    53    S
   024   20    14    DC4 (device control 4)      124   84    54    T
   025   21    15    NAK (negative ack.)         125   85    55    U
   026   22    16    SYN (synchronous idle)      126   86    56    V
   027   23    17    ETB (end of trans. blk)     127   87    57    W
   030   24    18    CAN (cancel)                130   88    58    X
   031   25    19    EM  (end of medium)         131   89    59    Y
   032   26    1A    SUB (substitute)            132   90    5A    Z
```

(remainder ommitted)

For these last four 'core' commands, use the `man` command to learn detailed information about how they function, if you need it.

cp
: Copy files or directories

mv
: Move files or directories

rm
: Remove files or directories

passwd

Change your login password

Try This!

Use this example terminal to try the commands described in this section.

It's a sandbox, feel free to play for a bit.

- The top window is a file where you can type and save commands.
- The bottom window is a linux shell. Commands typed directly in the shell are not saved.

Try This!

Modify this program to make the output look like the table in the ascii man page.

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

int main() {
  std::string heading = "\ndec\toct\thex\tchar\n----------------------------\n";
  for (int i = 0; i < 128; ++i) {
    if (i%20 == 0) std::cout << heading;
    char c = i;
    // the printable characters are between hex 20 and 7e
    std::cout << std::dec << i << '\t'
              << std::oct << i << '\t'
              << std::hex << i << "\t'" << c << "'\n";
  }
  return 0;
}
```

More to Explore

- LinuxCommand.org
- UNIX Text Processing - one of the best general references for people new to Unix or GNU/Linux. And you can build it yourself on any Unix or GNU/Linux system.
- GNU / Linux tutorial - from debian.org
- UNIX Philosophy - from Wikipedia
- Learn Enough (TM) Command-Line to be Dangerous tutorial by Michael Hartl.
- Ryans Tutorials: Linux - a decent introduction to the linux command line

3.1.6 Introducing the vim editor

Vim is a highly configurable text editor built to make creating and changing any kind of text very efficient. It is included as `vi` with most *nix systems and with Apple OS X. Among its features are:

- persistent, multi-level undo tree

- extensive plugin system
- support for hundreds of programming languages and file formats
- powerful search and replace
- integrates with many tools

If you plan to program in C++, you should have at least a minimal familiarity with `vi`. `vi` is a very old, but very capable text editor with many features specifically designed for writing software. Technically, `vi` came first, and `vim` is a separate program (Vi IMproved), but often typing either 'vi' or 'vim' invokes the same program, and the distinction is unimportant until you begin exploring some of the features `vim` has added. This text uses the two terms interchangeably.

Vi has thousands of features. No one knows them all, that is not the point. Vi is a very efficient editor that has intuitive commands. Because `vi` was created by programmers, for programmers, most `vi` commands 'reward' you, if you know how to touch type. The idea is to spend less time messing with menu commands and to spend more time writing code.

Benefits of learning Vim

Once the basics of the `vi` editor have been mastered, you'll find the skills learned to master vi translate to other tasks you'll perform as a programmer as well.

1. It is well integrated into Unix and Linux. Many Linux tools use the same key bindings as vim to perform similar actions, so while learning vim, you are also learning these commands without even knowing it.
2. The command shell history can be edited using vi commands.
3. Files can be edited directly over a network connection.
4. Vim can edit zip files directly with no need to extract their contents first.
5. Vim can easily compile programs and has rules for building many kinds of programs

Vim help

Vim is delivered with an extensive set of searchable, hyperlinked documentation. Access vim help using `:help` (or just `:h`. By default a help window open with basic vim navigation and how to jump to hyperlinked topics within the documentation.

Modes

One of the first things to get used to with vi is that it is a **modal** editor. Anyone who has used a computer to edit the contents of a text file has most likely used a "mode-less" editor; for example, programs like Microsoft Word, Notepad or Notepad++. In a "mode-less" editor there is actually one mode: the text input mode. For composing an essay this is more than likely all that is needed.

The vi editor has more modes than just "insert text". Modes allow vi to do two basic things:

- accept a command, such as deleting a line
- accept text, written by you

Some commands tell vim to enter a new *mode* which affects what your next keystrokes will do. In vi, type `:help vim-modes` to get help on modes and how to use them.

In the vi editor, each of these tasks is achieved by putting the editor into a particular *mode of operation* (normally just called a *mode*). When you wish to give vi a command, you enter command mode, and when you want to enter text, you enter insert mode.

It is important to set the correct mode before you begin writing, but this is simple to do. When you first start vi, it is automatically in normal mode.

Name	Description	help page
normal	For navigation and manipulation of text. This is the mode that vim will usually start in, which you can usually get back to with ESC.	`:help Normal-mode`
insert	For inserting new text. The main difference from vi is that many important "normal" commands are also available in insert mode - provided you have a keyboard with enough meta keys (such as Ctrl, Alt, Windows-key, etc.).	`:help Insert-mode`
visual	For navigation and manipulation of text selections, this mode allows you to perform most normal commands, and a few extra commands, on selected text.	`:help Visual-mode`
select	Similar to visual, but with a more MS Windows-like behavior.	`:help Select-mode`
command line	For entering editor commands - like the help commands in the 3rd column.	`:help Command-line-mode`
Ex-mode	Similar to the command-line mode but optimized for batch processing.	`:help Ex-mode`

There are many sources of vim tutorials and resources for learning about vim. The best place to start is at your terminal, if you have one. At the command prompt, type `vimtutor`:

```
host@user: vimtutor
```

Will launch a short tutorial designed to get you started with the basics.

The site https://openvim.com/ hosts a web-based vim tutor that is easy to use.

Linux.com also has a decent introduction to vim.

Motions

A motion is simply a command that moves the cursor. There's plenty of them, with `h`, `j`, `k`, and `l` being the most easily understood: move left, move up, move right, move down; all by only a single character.

Word motions

`w` To the beginning of the next word.

`e` To the end of the current word.

`W` To the beginning of the next WORD.

`E` To the end of the current WORD.

`b` Go backward a word.

`B` Go backward a WORD.

Other motions

`gg` Go to the first line.

`G` Go to the very last line.

`0` Go to the very first character of the line.

`^` Go to the very first non-whitespace character of the line.

`$` Go to the very last character of the line.

Selections

Selections are slightly different than motions in that they don't move the cursor but they do alter the way in which commands work. They will apply a command or operation to something like a word, or a sentence, or a block of text inside paired parenthesis, etc. A single selection can do a lot to make life in Vim easy.

`aw` A word

`aW` A WORD

`iw` An inner word

`iW` An inner WORD

Command & motion examples

- Re-indent everything `gg=G`
- Re-indent a block of code, including braces `=aB` or `=a{`
  - If the cursor is on a brace, `=%` is the same as `=aB`
- Re-indent 'inner block', excluding braces `=iB` or `=i{`
- Delete to the next word: `daw`
- Delete to the next `x` character (replace 'x' with your character): `dtx`
- Change a sentence: `cas`
- Change 3 letters: `3cl`
- Delete everything inside parenthesis: `di)`
- Visually select a paragraph: `vap`

Creating 'Hello World' using vim

It is best to review this section after completing the *vimtutor*. The *vimtutor* does a good job of explaining the basics of vim, but doesn't provide information specific to compiling and running programs.

1. Before jumping into the editor, first let's plan to save our project in a directory of its own:

   ```
   host@user: mkdir work
   host@user: cd work
   ```

   I chose to work in the directory, `work`, but you could have named it `hello`, `project1`, or anything else. Just pick something that works for you, preferably related to the task at hand. Recommendation: avoid spaces in directory names.

2. At the command prompt, open a new cpp source file:

   ```
   host@user: vi hello.cpp
   ```

   You'll now be in `vi`, which initially, will show a blank canvas. Use the commands you learned in *vimtutor* to enter insert mode and begin typing your program:

   ```
   #include <iostream>

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

3. When done typing, press ESC to leave insert mode.
4. Save your work by typing `:w`.
5. Still within vim, run the *make command*: type `:make hello`. You should see something similar to:

```
c++ hello.cpp   -o hello

Press ENTER or type command to continue
```

6. If you type `:make hello` a second time (when hello already exists and you haven't yet modified hello.cpp), then you should see:

```
make: `hello' is up to date.
```

7. At this point, you can run your program, also without leaving `vi`, by typing `:!./hello`. The `:!` command (bang command) executes shell commands from vim. If you typed the above source code, you should see:

```
Hello, World!

Press ENTER or type command to continue
```

Navigating to *make* errors

It is inevitable that sometimes you will create a program that does not compile. Vim provides tools that make navigating the error list and jumping to source code easy. The idea is to save the error messages from the compiler in a file and use Vim to jump to the errors one by one. You can examine each problem and fix it, without having to remember all the error messages.

Given the following, almost correct version of our *hello, world* program:

```
#include <iostream>

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

Typing `make foo` results in:

```
c++ -std=c++11 -Wall -Wextra -pedantic    foo.cpp   -o foo
foo.cpp:4:5: error: use of undeclared identifier 'cout'; did you mean
↪'std::cout'?
    cout << "Hello, World!\n";
    ^~~~
    std::cout
/usr/include/c++/v1/iostream:50:33: note: 'std::cout' declared here
extern _LIBCPP_FUNC_VIS ostream cout;
                                ^
1 error generated.
make: *** [foo] Error 1

Press ENTER or type command to continue
```

When you press ENTER, the `make` output is hidden, but stored in a special vim buffer: the quickfix window. Open the quickfix window with the command `:copen`. This will open the output from the last `make` command in a new pane.

You can move through the errors with the commands `:cnext` (or just `:cn`) and `:cprevious` (or just `:cp`). When done, close the quickfix window with `:close` (or just `:ccl`)

There are many more quickfix commands, see `:help quickfix` for more information.

The `make` utility is described in more detail in the *Command-line compiling* section.

More to Explore

- Learning the vi Editor
- vi Reference
- Vim Tips wiki
- vim.org
- vi reference cards - available in several languages
- vi lovers home page
- Swarthmore Dept. of Computer Science vim tips and tricks page
- UNIX Text Processing - has a chapter dedicated to vi.

Text editor downloads

- Notepad++ (Windows only) - also available on campus
- Vim
- Emacs

Both vim and emacs are included on the Mesa server. If you have already downloaded `git`, then you also have vim. Git provides a subset of the Gnu/Linux utilities, including vim and ssh, but not make or g++.

The `gcc` and `make` section is purely for reference, but does provide some insight into how software gets built and can help remove some of the 'magic' from the build process.

3.1.7 Command-line compiling

Why in the 'modern' era, is there still such focus on command line and text based tools for programming? Several actually.

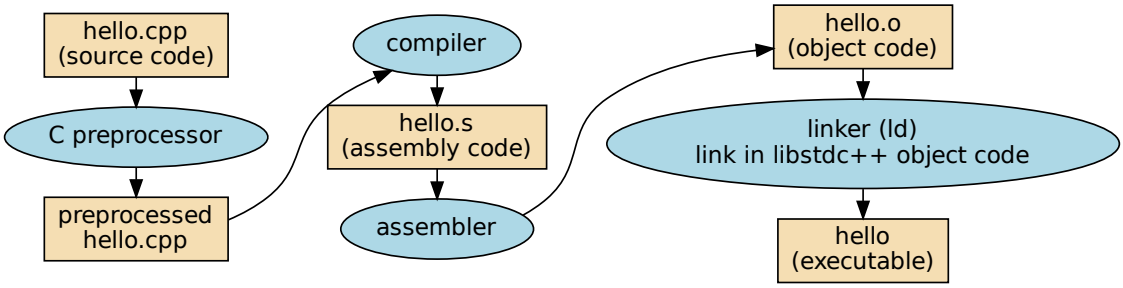
- Command line applications can easily be used in batch files or scripts, which is essential for automated testing or builds. On large software projects, even if your team uses an Integrated Development Environment (IDE), you'll also need a command line tool-chain that a machine can execute.
- The input, output and error streams can easily be redirected, allowing information to be sent or received from files or other applications. This can mean test data can be easily supplied or output captured.
- Use over remote shell or similar connections makes it possible to efficiently perform tasks remotely, such as when a developer is connecting from home or on the road.
- There are defined standards for help (such as the Unix **man** command or passing -? as an argument). If someone does not know a feature, they can easily discover what the tool provides.
- If you are writing such tools rather than using them, the command line is easy to develop for.

  Accepting arguments on the command line is trivial and output to a text stream is similarly easy.

The C++ compiler

There are many programs that will make executables from C++ source. One popular choice is `gcc`, the Gnu project C and C++ compiler. `gcc` is free and open-source, meaning that it can be made to run on many different operating systems. It also has the advantage of being able to compile more than just C and C++, although it was developed initially for that. One of the advantages of using a multi language tool like gcc is that it potentially minimizes the number of new tools you have to learn to begin working in a new language.

When you invoke `gcc`, it normally does preprocessing, compilation, assembly and linking.



Standard executable build steps

`gcc` is a complex program with **many** options. You can execute only one of these steps, some of them, all of them, or change the behavior of one or more steps.

Generally, we are interested in building our programs as simply as possible. Given a single source file `main.cpp`, that contains all the code needed to describe a working program, it is enough to type:

```
g++ main.cpp
```

This command will perform all of the steps to make an executable and write the executable to a file named `a.out`. Normally, however, we want to give our programs meaningful names:

```
g++ main.cpp -o main
```

The `-o` switch tells g++ to write the executable to a file named `main`. We could have added a file extension, like `.exe`, but that is ordinarily only done on the Windows (C) operating system. On most other systems, the extension is used for people to sort out different types of files, but the operating system doesn't need them to figure things out.

If our `main.cpp` contained syntax from the C++11 standard, or included any headers that did, then in order for it to compile, the following switch must also be added:

```
g++ main.cpp -std=c++11 -o main
```

Note

The default C++ version varies between compilers and may change with new releases. For years the default (on non-Windows compilers) was C++98. That started changing around 2016 or so.

As of GCC11 (2021), the default is C++17. Clang made the switch to a C++17 default in 2023 with Clang 16. As of Visual Studio 2022, the default is C++14

A summary of useful command line arguments:

o
: Specify the output executable filename.

g
: Include debugging symbols in the executable output. Debug level #2 is the default.

  You must include this if you want to debug your program using a debugger.

Wall
: Print all warnings

Werror
: Treat warnings as errors and do not write an executable output file.

Wextra
: Print extra warnings above and beyond 'all'.

pedantic
: Warn about code not compliant with the ISO standard.

std=c++11
: Instruct the compiler the program contains C++11 syntax.

O3
: Compile with optimization level #3 enabled.

The order of command line arguments (generally) does not matter.

The make utility

The `make` program automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them. Note how this is different from the previous sections. The `make` program doesn't actually compile anything. It doesn't know how. What `make` *does know* is

- What source files are normally associated with C++ programs
- What commands are used to compile and link C++ source files

`make` actually 'knows' some other things as well, and can be given more information to make it even more useful, but these first two things are the most important.

To prepare to use 'make', you must write a file called the "makefile" that describes the relationships among files in your program and provides commands for updating each file. By default, `make` searches for a file named **Makefile**.

Once a file named Makefile exists, each time you change some source files, this simple shell command:

```
make
```

is enough to perform all necessary recompilations. The `make` program uses the makefile data and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the recipes recorded in the makefile.

A simple makefile consists of "rules" with the following shape:

```
TARGET ... : PREREQUISITES ...
   RECIPE
   ...
   ...
```

A "target" is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as 'clean'. Since these don't result in object files or executables, they are called *phony targets*.

A "prerequisite" is a file that is used as input to create the target. A target often depends on several files.

A "recipe" is an action that 'make' carries out. A recipe may have more than one command, either on the same line or each on its own line.

Note

You need to put a tab character at the beginning of every recipe line! This is an obscurity that catches the unwary.

A *very* simple Unix C++ makefile:

```
CXX = c++
CXXFLAGS=-Wall -Wextra -pedantic -std=c++11 -O3

my_prog : clean
   ${CXX} ${CXXFLAGS} -o my_prog *.cpp

clean:
   rm -f my_prog
```

This makefile always deletes the executable file 'my_prog' whether any source files have changed or not and recompiles every cpp file in the current directory to regenerate the executable. Not particularly efficient, but for small programs it gets the job done and requires no maintenance if new files are added.

Assuming you keep your work separated into directories so that all the source code in a directory applies only to a single executable, then this simple makefile is sufficient for everything you'll need to worry about over course duration.

Suffice it to say that make provides many tools and techniques to define 'smarter' recipes that do not suffer the constraints listed above.

The make program defines a large number of built-in variables to control how programs get built. While there are many, the two to focus on for now are those used in the previous makefile.

CXX
: Specify which C++ compiler to use

CXXFLAGS
: Specify command line arguments to pass to the C++ compiler

A slight variation on this theme creates several executables from a list. The primary assumption in this makefile is that each program in the list `PROGS` contains a main and is a single file. The makefile makes use of the fact that `make` can make a simple program without a makefile, if the program is contained within a single file.

```
CXX = c++
CXXFLAGS=-Wall -Wextra -pedantic -std=c++11 -O3

PROGS = asciitable \
        hello \
        phrase-o-matic \
        printodds \
        std11-test \

all: $(PROGS)

clean:
    rm -f $(PROGS)
```

Even without a Makefile, `make` is smart enough to produce sensible defaults for simple programs. For example, given our `main.cpp` example from the previous section, we can make it using make:

```
make main
```

The make program will actually invoke:

```
g++ main.cpp -o main
```

If you have exported an environment variable that make uses, such as `CXXFLAGS`, then those will be used by make:

```
export CXXFLAGS=-Wall -Wextra -pedantic -std=c++11
```

Now the make program will invoke:

```
make main
g++ main.cpp -Wall -Wextra -pedantic -std=c++11 -o main
```

Which is exactly what the previous makefile is doing, but on all of the programs listed.

Running your programs

If you compile your own program, you will need to prefix it with ./ on the command line to tell the shell that you want to run a program in the current directory (called '.') instead of one of the standard system directories. So for example, if I've just built a program called *hello*, I can run it by typing:

```
./hello
```

More to Explore

- Textbook FAQ: *Why not use an IDE?*
- GNU Make Manual
- Prefer compile-time and link-time errors to runtime errors
  - Effective C++ item #46
- Pay attention to compiler warnings
  - Effective C++ item #48

You may choose to defer reading about debugging until you actually need to use it and can treat it like a quick primer. When you are working on project 1 is a good time to read this section.

3.1.8 Debugging

Different kinds of errors can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:

1. Syntax errors are produced when compilers or interpreters translate source code into machine readable form. A syntax error is the failure to follow the basic rules of the language. For example, mis-matched parentheses or braces, or misplaced keyword, or a variable used where a keyword is expected.
2. Link errors occur when each compilation unit compiles correctly, but the next stage, the linker is unable to combine all the object code into a single valid executable file.

3. Runtime errors are produced by the executable system if something goes wrong while the program is running. Runtime error messages may include information about where the error occurred and what functions were executing. Detailed runtime error output is typically only available in compiled languages such as C++ with debugging symbols explicitly compiled into the final program.
4. Semantic errors are problems with a program that compiles and runs but doesn't do the right thing. Example: An expression may not be evaluated in the order you expect, yielding an unexpected result.

The first step in debugging is to figure out which kind of error you are dealing with. Although the following sections are organized by error type, some techniques are applicable in more than one situation.

Syntax errors

Syntax errors are usually easy to fix once you figure out what they are. Unfortunately, the error messages are often not helpful. The errors reported vary from compiler to compiler and can be long and cryptic. This is especially true of errors involving templates.

Often syntax errors in one part of a program can cascade, generating more downstream errors in other parts of the program. This is common for simple errors such as typographical mistakes. Therefore, it is a good idea to attack errors from the top of the error list first. Commonly, fixing a few minor errors reported at the top of the error list eliminates dozens of lines of reported errors. Bottom line: don't let the size of an error output intimidate you, the compiler is just trying to be thorough.

On the other hand, the messages from most syntax errors do tell you almost exactly where in the program the problem occurred. Actually, it tells you where the compiler noticed a problem, which is not necessarily where the error is. Sometimes the error is prior to the location of the error message, often on the preceding line.

If you are copying code from a book, start by comparing your code to the book's code very carefully. Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.

Here are some ways to avoid the most common syntax errors:

1. Make sure you are not using a keyword for a variable name.
2. Make sure you are compiling with the `-std=c++11` argument.
3. Check that indentation is consistent. You may indent with either spaces or tabs but it's best not to mix them. Each level should be nested the same amount. Neatness catches more errors than you might expect.
4. Make sure that any strings in the code have matching quotation marks.
5. If a function declares a return value, ensure your function returns a value.
6. An unclosed bracket -- (, {, or [-- makes the compiler continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.
7. Check for the classic = instead of == inside a conditional.

A simple dropped semi-colon may generate slightly different results using different compilers. For example, given the following:

```
2  int foo() {
3    return 1;
4  }
5
6  int main() {
7    foo()
8  }
```

The `gcc` compiler, reports the error on line 8. It notices an end brace reached } but a semi-colon was expected:

```
g++ -std=c++11 -Wall -Wextra -pedantic    foo.cpp   -o foo
g++: warning: couldn't understand kern.osversion '14.5.0
foo.cpp: In function 'int main()':
foo.cpp:8:1: error: expected ';' before '}' token
 }
 ^
```

The `clang` compiler knows where the semi-colon belongs and in this case, provides a slightly more specific error:

```
The ``clang`` compiler, reports the error on line 7.
clang++ -std=c++11 -Wall -Wextra -pedantic    foo.cpp   -o foo
foo.cpp:7:8: error: expected ';' after expression
  foo()
       ^
       ;
1 error generated.
```

I can't get my program to compile no matter what I do.

If the compiler says there is an error and you don't see it, that might be because you and the compiler are not looking at the same code. Check your programming environment to make sure that the program you are editing is the one you are actually compiling. If you are not sure, try putting an obvious and deliberate syntax error at the beginning of the program. Now compile it again. If the compiler doesn't find the new error, there is probably something wrong with the way your environment is set up.

If this happens, one approach is to start again with a new program like Hello, World!, and make sure you can get a known program to run. Then gradually add the pieces of the new program to the working one.

Link errors

If you encounter a link error, then the basic syntax of each compilation unit is correct. The most common mistake new programmers make is to start changing code in response to a link error. This will only make matters worse, as the program syntax was OK to begin with. The simplest kind of link error is when the linker can't find a `main` function to start execution:

```
g++: warning: couldn't understand kern.osversion '14.5.0
Undefined symbols for architecture x86_64:
  "_main", referenced from:
      __start in crt1.o
ld: symbol(s) not found for architecture x86_64
collect2: error: ld returned 1 exit status
```

Although cryptic looking, there are several clues that we do not have a syntax problem:

1. The first line of output says "Undefined symbols ...". This is a clear indication that the linker could not find something it expected.
2. The next 4 lines expand on this to tell us that the linker program (`ld`) was unable to find the symbol `_main`, which is the *mangled name* this particular compiler gives to the function `main()`. Compilers are free to mangle function names as they see fit to generate an executable in which every function signature is unique.

Since link errors or other problems related to setting up your environment are not covered in detail in this course, this is a perfectly fine time to ask your instructor for help. Since the problem is either in your environment, your `Makefile`, or both.

Runtime errors

Once your program is syntactically correct, you can create an executable and start running it. What could possibly go wrong?

My program does absolutely nothing.

This problem is most common when your file consists of functions and classes but does not actually invoke anything to start execution. This may be intentional if you only plan to import this module to supply classes and functions.

If it is not intentional, make sure that your program has a `main()` function.

My program hangs.

If a program stops and seems to be doing nothing, we say it is hanging. Often that means that it is caught in an infinite loop or an infinite recursion.

1. If there is a particular loop that you suspect is the problem, add a `cout` or `puts` statement immediately before the loop that says entering the loop and another immediately after that says exiting the loop.
2. Run the program. If you get the first message and not the second, you've got an infinite loop. Go to the Infinite Loop section below.
3. Most of the time, an infinite recursion will cause the program to run for a while and then produce a RuntimeError: StackOverflow error. If that happens, go to the Infinite Recursion section below.
4. If you are not getting this error but you suspect there is a problem with a recursive method or function, you can still use the techniques in the Infinite Recursion section.
5. If neither of those steps works, start testing other loops and other recursive functions and methods.
6. If that doesn't work, then it is possible that you don't understand the flow of execution in your program. Go to the Flow of Execution section below.

One last possibility is that your program is simply waiting for input and there is no visual indication that input is expected. If you don't suspect an infinite loop, try typing something and pressing *Enter*. If your program does **anything**, including crashing, then you don't have an infinite loop. You have a logic error. Go to the Semantic error section below.

Infinite loops

If you think you have an infinite loop and you think you know what loop is causing the problem, add a `print` statement at the end of the loop that prints the values of the variables in the condition and the value of the condition.

For example:

```
while ( x > 0 && y < 0) {
  // do something to x
  // do something to y

  std::cout << "x: " << x << '\n';
  std::cout << "y: " << y << '\n';
  std::cout << "condition: " << (x > 0 && y < 0) << '\n';
}
```

Now when you run the program, you will see three lines of output for each time through the loop. The last time through the loop, the condition should be `0`. If the loop keeps going, you will be able to see the values of `x` and `y`, and you might figure out why they are not being updated correctly.

In a development environment like *CodeBlocks*, *Visual Studio*, or using command line debuggers such as *gdb* one can also set a breakpoint at the start of the loop, and single-step through the loop. While you do this, inspect the values of `x` and `y` by hovering your cursor over them.

Of course, all programming and debugging require that you have a good mental model of what the algorithm ought to be doing: if you don't understand what ought to happen to `x` and `y`, printing or inspecting its value is of little use. Probably the best place to debug the code is away from your computer, working on your understanding of what should be happening.

Infinite recursion

Most of the time, an infinite recursion will cause the program to run for a while and then produce a `Stack overflow` error.

If you suspect that a function or method is causing an infinite recursion, start by checking to make sure that there is a base case. In other words, there should be some condition that will cause the function or method to return without making a recursive invocation. If not, then you need to rethink the algorithm and identify a base case.

If there is a base case but the program doesn't seem to be reaching it, add a `print` statement at the beginning of the function or method that prints the parameters. Now when you run the program, you will see a few lines of output every time the function or method is invoked, and you will see the parameters. If the parameters are not moving toward the base case, you will get some ideas about why not.

Once again, if you have an environment that supports easy single-stepping, breakpoints, and inspection, learn to use them well. It is our opinion that walking through code step-by-step builds the best and most accurate mental model of how computation happens. Use it if you have it!

Semantic errors

In some ways, semantic errors are the hardest to debug, because the compiler and the runtime system provide no information about what is wrong. If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages. However, your program will not do the right thing. It will do something else. Specifically, it will do what you told it to do. Only you know what the program is supposed to do, and only you know that it isn't doing it.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is actually doing. One of the things that makes that hard is that computers run so fast.

You will often wish that you could slow the program down to human speed, and with some debuggers you can. But the time it takes to insert a few well-placed `print` statements is often short compared to setting up the debugger, inserting and removing breakpoints, and walking the program to where the error is occurring.

General debugging tips

Before you can effectively use debugging tools, you need to know what your program is *supposed* to do. The basic method of all debugging:

1. Know what your program is supposed to do.
2. Detect when it doesn't.
3. Fix it.

A tempting mistake is to skip step 1, and just try randomly tweaking things until the program works. Better is to see what the program is doing internally, so you can see exactly where and when it is going wrong. A second temptation is

to attempt to intuit where things are going wrong by staring at the code or the program's output. Avoid this temptation as well: let the computer tell you what it is really doing inside your program instead of guessing.

My program doesn't work

You should ask yourself these questions:

1. Is there something the program was supposed to do but which doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should.
2. Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.
3. Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves invocations to functions or methods in other compilation units. Read the documentation for the functions you invoke. Try them out by writing simple test cases and checking the results.

In order to program, you need to have a mental model of how programs work. If you write a program that doesn't do what you expect, very often the problem is not in the program; it's in your mental model.

The best way to correct your mental model is to break the program into its components (usually the functions) and test each component independently. Ask yourself if each function is truly doing one thing in your program. Small function that do one thing well makes solving semantic errors much easier. Once you find the discrepancy between your model and reality, you can solve the problem.

Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

I've got a big hairy expression and it doesn't do what I expect

Having a "big hairy expression" is your first problem. Ask your self if this is the simplest solution for the problem you are trying to solve.

Writing complex expressions is fine as long as they are **clear**, but they can be hard to debug. Consider breaking a complex expression into a series of assignments to temporary variables.

For example:

```
this.hands[i].add_card (this.hands[this.neighbor(i)].top())
```

This can be rewritten as:

```
auto neighbor = this.neighbor (i);
auto picked = hands[neighbor].top();
hands[i].add_card (picked);
```

The explicit version is easier to read because the variable names provide additional documentation, and it is easier to debug because you can check the types of the intermediate variables and display or inspect their values.

Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, if you are translating the expression `x/2pi` into code, you might write:

```
y = x / 2 * M_PI;
```

That is not correct because multiplication and division have the same precedence and are evaluated from left to right. So this expression computes `(x/2)pi`.

A good way to debug expressions is to add parentheses to make the order of evaluation explicit:

```
y = x / (2 * M_PI);
```

Whenever you are not sure of the order of evaluation, use parentheses. Not only will the program be correct (in the sense of doing what you intended), it will also be more readable for other people who haven't memorized the rules of precedence.

I've got a function that doesn't return what I expect

If you have a `return` statement with a complex expression, you don't have a chance to print the `return` value before returning. Again, you can use a temporary variable. For example, instead of:

```
return this.hands[i].remove_matches();
```

you could write:

```
auto count = hands[i].remove_matches();
return count;
```

Now you have the opportunity to display or inspect the value of `count` before returning.

Assertions

The include <assert.h> defines a very handy **assert** macro. The assert macro tests if a condition is true and halts your program with an error if it is false.

```
#include <assert.h>

int main() {
  assert (5 == 2+2);
}
```

When compiled an run, the output is:

```
Assertion failed: (5 == 2+2), function main, file foo.cpp, line 4.
```

Line numbers and everything, even if you compile with the optimizer turned on. Much nicer than a mere segmentation fault, and if you run it under the debugger, the debugger will stop exactly on the line where the assert failed so you can poke around and see why.

Debugging tools

There are many tools to help programmers find and fix errors. The simplest thing you can do is add print statements or assertions to your code. This is the slowest way to debug your code as it requires a recompile each time you want to look at something different.

It is better, generally to use a more sophisticated tool. Every compiler and language provides some sort of debugging tool to assist developers in writing software.

Nearly every *IDE* comes with a graphical debugger. Most of them are very good. Linux provides a variety of debugging tools. The program `ddd` is a graphical debugger for linux. The program `gdb` is a text-based debugger for linux.

The GNU debugger (gdb)

The standard debugger on GNU/Linux is called gdb. This lets you run your program under remote control, so that you can stop it and see what is going on inside.

Given the small, buggy program:

```
#include <iostream>

int main() {
  int sum = 0;

  for (int i = 0; i -= 1000; ++i) {
    sum += i;
  }
  std::cout << "sum = " << sum << '\n';
}
```

Note that we are going to add the flag -g3 to tell the compiler to include debugging information. Debug level 3 is the most detailed debug level. Debug levels 2 and 3 allow gdb to translate machine addresses back into identifiers and line numbers in the original program for us.

Let's compile and run it and see what happens:

```
$ g++ bogus.cpp -std=c++11 -Wall -Wextra -pedantic -g3 -o bogus
$ ./bogus
sum = -34394132
$
```

That doesn't look like the sum of 1 to 1000. So what went wrong? If we were clever, we might notice that the test in the for loop is using the shortcut -= operator instead of the <= operator that we probably want. But let's suppose we're not so clever right now - it's four in the morning, we've been working on bogus.cpp for twenty-nine straight hours, and there's a -= up there because in our befuddled condition we know in our bones that it's the right operator to use. We need somebody else to tell us that we are deluding ourselves, but nobody is around this time of night. So we'll have to see what we can get the computer to tell us.

The first thing to do is fire up gdb, the debugger. This runs our program in stop-motion, letting us step through it a piece at a time and watch what it is actually doing. In the example below gdb is run from the command line:

```
$ gdb bogus
GNU gdb (GDB; openSUSE 13.1) 7.6.50.20130731-cvs
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i586-suse-linux".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.opensuse.org/>.
Find the GDB manual and other documentation resources online at:
<https://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
..
```

(continues on next page)

(continued from previous page)

```
Reading symbols from /var2/home/dparillo/bogus...done.
(gdb) run
Starting program: /var2/home/dparillo/bogus
sum = -34394132
[Inferior 1 (process 32083) exited normally]
```

So far we haven't learned anything. To see our program in action, we need to slow it down a bit. We'll stop it as soon as it enters main, and step through it one line at a time while having it print out the values of the variables.

```
(gdb) break main
Breakpoint 1 at 0x8048719: file bogus.cpp, line 4.
(gdb) run
Starting program: /var2/home/dparillo/bogus

Breakpoint 1, main () at bogus.cpp:4
4     int sum = 0;

(gdb) display sum
1: sum = -1209683968
(gdb) next
6     for (int i = 0; i -= 1000; ++i) {
1: sum = 0
(gdb) next
7       sum += i;
1: sum = 0
(gdb) display i
2: i = -1000
(gdb) next
6     for (int i = 0; i -= 1000; ++i) {
2: i = -1000
1: sum = -1000
(gdb) n                       # getting lazy and used 'n' instead of 'next'
7       sum += i;
2: i = -1999
1: sum = -1000
(gdb) n
6     for (int i = 0; i -= 1000; ++i) {
2: i = -1999
1: sum = -2999
(gdb) quit
A debugging session is active.

  Inferior 1 [process 32187] will be killed.

Quit anyway? (y or n) y
```

Here we are using **break main** to tell the program to stop as soon as it enters main, **display** to tell it to show us the value of the variables `i` and `sum` whenever it pauses, and **n** (or **next**) to execute the program one line at a time.

When stepping through a program, `gdb` displays the line it will execute next as well as any variables you've told it to display. This means that any changes you see in the variables are the result of the previous displayed line. Bearing this in mind, we see that `i` drops from 0 to -1000 the very first time we hit the top of the for loop and drops to -1999 the next time. So something bad is happening in the top of that for loop, and we might begin to suspect that `i -= 1000` is

not doing what we intended.

Useful gdb commands

help
Get a description of gdb commands.

run
Runs your program. You can give it arguments that get passed in to your program just as if you had typed them to the shell. Also used to restart your program from the beginning if it is already running.

quit
Leave gdb, killing your program if necessary.

break
Set a breakpoint, which is a place where gdb will automatically stop your program. Some examples:

- `break function_name` stops before executing the first line in `function_name`.
- `break 117` stops before executing line number 117.

list
Show part of your source file with line numbers (handy for figuring out where to put breakpoints). Examples:

- `list function_name` lists all lines of `function_name`.
- `list 117-123` lists lines 117 through 123.

next
Execute the next line of the program, including completing any function calls in that line. This command executes, but does not *step into* functions.

step
Execute the next step of the program, which is either the next line if it contains no function calls, or the entry into the called function.

finish
Continue until you get out of the current function (or hit a breakpoint). Useful for getting out of something you stepped into that you didn't want to step into.

cont
(Or continue). Continue until

a) the end of the program,

b) a fatal error like a Segmentation Fault or Bus Error, or

c) a breakpoint.

If you give it a numeric argument (e.g., `cont 1000`) it will skip over that many breakpoints before stopping.

print
Print the current value of some expression once, e.g. `print i`.

display
Like `print`, but runs automatically every time the program stops. Useful for watching values that change often.

Self Check

Q1

Question

Debugging is:

- ◯ tracking down programming errors and correcting them.
- ◯ removing all the bugs from your house.
- ◯ finding all the bugs in the program.
- ◯ fixing the bugs in the program.

Q2

Question

Which of the following is a semantic error?

- ◯ Attempting to divide by 0.
- ◯ Forgetting a semi-colon at the end of a statement where one is required.
- ◯ Forgetting to divide by 100 when printing a percentage amount.

Q3

Question

Which of the following is a syntax error?

- ◯ Attempting to divide by 0.
- ◯ Forgetting a colon at the end of a statement where one is required.
- ◯ Forgetting to divide by 100 when printing a percentage amount.

Q4

Question

Which of the following is a run-time error?

- ◯ Attempting to divide by 0.
- ◯ Forgetting a colon at the end of a statement where one is required.
- ◯ Forgetting to divide by 100 when printing a percentage amount.

Q5

Question

Who or what typically finds syntax errors?

- ◯ The programmer.
- ◯ The compiler / interpreter.
- ◯ The computer.
- ◯ The teacher / instructor.

More to Explore

- From: cppreference.com: assert and static_assert.
- GDB tips
- DDD

3.2 String and Vector

A brief introduction to `string` and `vector`. Although vector gets more attention later in the book, these 2 types are the workhorses of a great deal of real-world C++ code and it's important that you understand how to use them as early as possible.

3.2.1 Using string and vector

Many useful things get done using the standard fundamental types. Often, however, we need to group a set of related types together. In order to group them together, we need some sort of **container**.

Two of these containers you should learn first: std::string and std::vector. They come with many useful functions that help solve many common problems. While there is some cost in using them, often the cost is worth the benefits in terms of reduced development time, code clarity, and other advantages.

Both `string` and `vector` are *container classes*. That is, their primary job is to make it easy to work with the data stored inside the container.

The simplest 'batch of data' is a 'batch of characters' Another word for 'batch of characters' is 'string'. A `string` is a container for data of a single type: char.

A `vector` can store data of **any** type: even types that you make up. We will see how that is possible in a bit, but first let's focus on how to use these two important types.

3.2.2 String abstractions in C

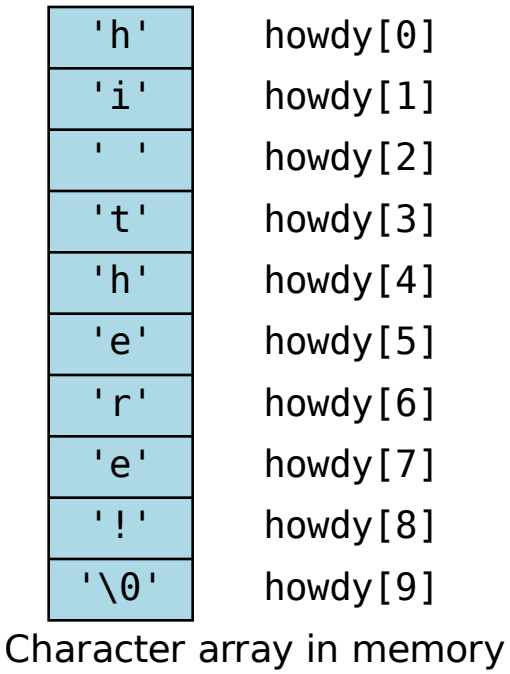
In the C language, the abstract idea of a string is implemented with an array of characters.

```
// the char array must be null terminated
char a[] = {'h', 'e', 'l', 'l', 'o', '\0'};  // null == '\0'

char b[] = {'h', 'e', 'l', 'l', 'o', 0};     // null == 0 also

// a quoted literal is just a special case of a char array
char* c = "hello";
```

Arrays of `char` that are null terminated are commonly called byte strings or C strings. Given the byte string:

```
const char* howdy = "hi there!";
```

In memory, `howdy` is automatically transformed into:



Character array in memory

The last character in the array, `'\0'` is the *null character*, and is used to indicate the end of the string. The null character is a `char` equal to 0.

```
#include <iostream>

int main () {
   if (const char null1 = '\0', null2 = 0;              // C++17
       null1 == null2) {
        std::cout << "these values are the same\n";
    } else {
        std::cout << "not the same\n";
    }
}
```

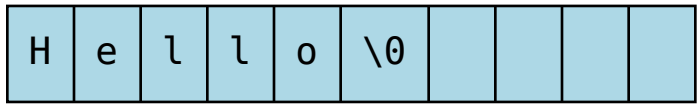
Note

Care must be taken to ensure that the array is large enough to hold all of the characters AND the null terminator. Forgetting to account for null, or having a 'off by one error' is one of the most common mistakes when working with C strings.

A character array may allocate more memory that the characters currently stored in it. An array declaration like this:

```
char hi[10] = "Hello";
```

results in an in-memory representation like this:



Character array with reserve memory

The array elements after the null are unused, but could be. So, an array of size 10 has space for 4 more characters, 9 total.

C strings have an advantage of being extremely lightweight and simple. Their main disadvantage is that they are too simple for many applications. Their simplicity makes them a pain to work with, which is why the Standard Template Library (STL) contains the string class.

3.2.3 Working with C strings

The C programming language has a set of functions implementing operations on strings (character strings and byte strings) in the standard library. Various operations, such as copying, concatenation, tokenization, and searching are supported.

The complete list of byte string functions is available from cppreference.com.

It's important to know how to work with byte strings in C++ because the C string functions that C++ inherits from C continue to provide a few capabilities not implemented elsewhere in the STL.

In addition, when the type you have is a byte string, it's just easier and more efficient to manipulate the byte string directly, rather than create a temporary `std::string` merely to perform an operation and then convert back. In general, you want to try to avoid these kinds of unnecessary type conversions.

```
#include <cstdio>   // printf
#include <cstring>  // for strcpy function

// In C, a string is literally an array of char
//
// This is not the same as the string class from the STL.

int main()
{
  char a[] = {'h', 'e', 'l', 'l', 'o', '\0'};  // the char array must be null terminated
  char b[] = {'h', 'e', 'l', 'l', 'o', 0};     // null == 0
  const char* c = "hello";                     // a string is just a special case of a␣
↪char array

  for (int i = 0; a[i]; ++i) {                 // explain why this loop terminates
    printf ("%c %c %c\n", a[i], b[i], c[i]);
  }

  // copying strings
  char* d = 0;
```

(continues on next page)

(continued from previous page)

```
  d = a;      // we only copied a pointer here!
  if (a == d) {
    d[0] = 'H';
  }
  printf ("\na and d strings:\n  %s %s\n", a, d);

  char e[sizeof(b)];
  strcpy(e, b);
  e[0] = 'H';
  printf ("\nb and e strings:\n  %s %s\n", b, e);

  // a = e;    // compile error.  C strings are not assignable

  return 0;
}
```

Change character case

Many languages provide utilities to change character case as part of the string class.

Not C++.

C++ uses the legacy null-terminated byte strings library to provide these features.

Changing character case is a common task and unless you choose to write your own version of these functions, these functions from the STL are the ones you should use.

Many string conversion functions are defined in the `cctype` header. These functions which C++ inherited from C are often perfectly acceptable, however, there are some notable exceptions, such as the toupper function.

toupper(std::locale())

This version of `std::toupper` function takes a single `char`, which can be **any** character type, is not modified, and returns a character of the *same type* as the character type provided.

Because of this, the C++ version that uses `std::locale` is preferred.

```
#include <iostream>
#include <cctype>    // C toupper
#include <locale>    // C++ toupper

int main() {
  using std::cout;
  char eng[14] = "hello, world!";

  // Use C version
  for (const auto& c: eng) {
    cout << std::toupper(c) << ' ';
  }
  cout << '\n';
  // Use locale aware version - no conversion
  for (const auto& c: eng) {
    cout << "'" << std::toupper(c, std::locale())  << "' ";
  }

```

(continues on next page)

(continued from previous page)

```
  return 0;
}
```

toupper()

The `std::toupper` function takes a single `char`, which is not modified, and returns an `int`. The return value can be used as the upper case version of the input character.

Note

Use the right toupper!

The C version of toupper returns **int** values, *not* character values. This can cause unexpected behavior or conversions.

For these resaons, the `std::locale()` version of toupper is preferred.

See the previous tab for details.

`toupper` is defined in header `cctypes`.

This function uses the default **C** locale to replace the lowercase letters `abcdefghijklmnopqrstuvwxyz` with respective uppercase letters `ABCDEFGHIJKLMNOPQRSTUVWXYZ`. Non-ASCII characters are not handled.

Recall that `char` implicitly convert to `int`.

```
#include <cctype>
#include <iostream>

int main() {
  char value[15] = "hello, world!";
  char& first = value[0];
  // failure to assign the return value of toupper
  // to a variable is a common source of error.
  first =  std::toupper(first);
  std::cout << value << '\n';
  return 0;
}
```

Copying and comparing C strings

Unlike most of the types we work with in C++, byte strings are simple arrays. Arrays cannot be assigned to each other using `operator=`. Values in array must be copied one elements at a time, for example using a loop.

Similarly, if you compare two arrays for equivalence, `operator==` will only return `true` if both arrays share the same memory address.

This is not what we usually want.

Like copying, in order to check a pair of arrays for equivalence a loop is used to compare each element one at a time until a difference is found.

The copy and compare functions are defined in the `cstring` header.

strcpy()

The strcpy function takes two byte strings as parameters and copies the source character array including the null terminator, to the destination character array.

```
#include <cstring>
#include <iostream>

int main()
{
    const char* src = "Take the test.";

    // src[0] = 'M';        // can't modify string literal
    char dest[16];           // mutable destination
    std::strcpy(dest, src);
    dest[0] = 'M';
    std::cout << src << '\n' << dest << '\n';
}
```

Note the order of the arguments.

A common source of error is to swap the order of the arguments.

strncpy()

The strncpy function copies byte strings, but will copy at most a provided `count` number of characters.

```
#include <cstring>
#include <iostream>

int main() {
    const char* src = "hi";
    char dest[6] = {'a', 'b', 'c', 'd', 'e', 'f'};
    std::strncpy(dest, src, 5);

    std::cout << "The contents of dest are: ";
    for (char c : dest) {
        if (c) {
            std::cout << c << ' ';
        } else {
            std::cout << "nul" << ' ';
        }
    }
    std::cout << '\n';
}
```

strcmp()

The strcmp function takes two byte strings as parameters and returns a `0` if every element in both arrays is equal.

If the first operand is greater than the second, then a positive value is returned. If the first operand is less than the second, then a negative value is returned.

```
#include <cstdlib>
#include <cstring>
```

(continues on next page)

(continued from previous page)

```
#include <iostream>

int main() {
  const int count = 3;
  const char* argv[count] = {"test_prog", "--value", "42"};
  int value = 0;

  for (int i=1; i < 3; ++i) {
    if (strcmp(argv[i], "--value") == 0) {
      ++i;
      if (i < count) {
        value = atoi(argv[i]);
      } else {
        std::cerr <<
          "Error using '--value' argument: no value specified\n";
        break;
      }
    } else {
      std::cerr << "Unknown command received!";
      break;
    }
  }
  std::cout << "value: " << value << '\n';
}
```

Note

These functions are not locale-aware.

If you need to make locale aware comparisons, then use strcoll.

strncmp()

The strncmp function takes two byte strings as parameters and returns a `0` if every element in both arrays is equal.

However, this function only compares the at most a specified number of characters.

```
#include <cstring>
#include <iostream>

void demo(const char* lhs, const char* rhs, int sz) {
    int compare = std::strncmp(lhs, rhs, sz);

    if(compare == 0) {
        std::cout << "First " << sz << " chars of ["
                  << lhs << "] equal [" << rhs << "]\n";
    } else if(compare < 0) {
        std::cout << "First " << sz << " chars of ["
                  << lhs << "] precede [" << rhs << "]\n";
    } else if(compare > 0) {
        std::cout << "First " << sz << " chars of ["
                  << lhs << "] follow [" << rhs << "]\n";
```

(continues on next page)

(continued from previous page)

```
    }
}

int main() {
    demo("Hello, world!", "Hello, everybody!", 13);
    demo("Hello, everybody!", "Hello, world!", 13);
    demo("Hello, everybody!", "Hello, world!", 7);
    demo("Hello, everybody!" + 12, "Hello, somebody!" + 11, 5);
}
```

Note

These functions are not locale-aware.

If you need to make locale aware comparisons, then use strcoll.

Self Check

Q1

Fill in the blank

Given the following:

```
char text[32];
strcpy(text, "hello");
int len = strlen(text);
```

What is the value of `len`?

Q2

Fix the errors in the `printf` line below:

```
#include <cstdio>
#include <string>

int main() {
  std::string yazoo = "ritish alternative band";
  char c = 'B';

  printf ("%c%s\n",c, yazoo);
}
```

Q3

Fill in the blank

Which `#include` is required to use functions such as `std::atoi` and `std::atof`?

Q4

Fill in the blank

Which `#include` is required to use functions such as `std::stoi` and `std::stol`?

_____

More to Explore

- cppreference.com byte strings
- Bjarne Stroustrup's C++11 FAQ: Raw String literals
- Locales:
  - Thinking in C++: Locales
  - Differences between the C Locale and the C++ Locales

3.2.4 The string class

Like a byte string, a `std::string` is storage for a character sequence:

```
#include <string>        // access std::string functions

using std::string;       // alias type std::string to 'string'

int main() {
  string x;                           // empty string
  string greet =  "Hello, World!";    // create from C string
  string hello   ("Hello, World!");   // as above, constructor style syntax
  string howdy = {"Hello, World!"};   // C++11 only
  string howdy   {"Hello, World!"};   // as above, = is optional
  return 0;
}
```

What is different is that a `std::string` is a full-fledged *object*. It knows it's own size, and comes with many convenience functions.

Notice that unlike a built-in variable declaration such as `int x;`, the declaration `string x` is **not** incomplete. The variable `x` is a complete and valid `string` object that stores no characters.

Simple operations

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

using std::string;

int main() {
  string a = "hello";
  a += ", world!";        // joining strings is pretty easy

  // Copying or creating one string from another feels as natural
```

(continues on next page)

(continued from previous page)

```
  // as a fundamental type.
  string b = a;

  if (a == b)             // Same goes for comparisons
  {
    // modify values
    b[0] = 'H';           // and a string feels like an 'array of char'
    b[7] = 'W';
  }

  std::cout << a << '\n'; // and has stream support
  std::cout << b << '\n';

  return 0;
}
```

front() and back()

This tab shows alternate functions for accessing the first and last characters in a string.

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

int main() {
  std::string value = "hello, world!";

  std::cout << "first: " << value.front() << '\n';
  std::cout << "last: " << value.back() << '\n';
  return 0;
}
```

append()

The append function allows you to append *N* copies of a character or an array of characters to the end of a string.

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

int main() {
 using std::cout;
  std::string hi = "hello";
  std::string howdy = hi;

  cout << "original: " << hi    << '\n';

  hi.append(5, 'o');   // append 5 o's
  hi.append(", world!");
  cout << hi << '\n';

  cout << "original: " << howdy   << '\n';

  // append returns a new string value, so
```

(continues on next page)

(continued from previous page)

```
  // calls to append can be chained together
  howdy.append(5, 'o').append(", world!");
  cout << howdy << '\n';

  return 0;
}
```

insert()

The `insert` function allows you to insert 1 character or an array of characters at a position in a string.

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

int main() {
  std::string value = "hello, world!";

  std::cout << "original: " << value << '\n';

  value.insert(0,3,'*');       // insert "***" at position 0
  std::cout << value << '\n';

  // insert a char array before the '!'
  value.insert(value.size()-1," (this means you)");
  std::cout << value << '\n';

  return 0;
}
```

Using the operator[] to access select characters in a string is, like an array, not range checked. This means that if you use an index referring to an invalid position, then your program might have undefined behavior, or fail unexpectedly. You can use the function at anywhere `operator[]` is allowed. The `at` function is range checked. While there is a cost associated with this check, if the index provided is out of range, then an std::out_of_range exception is thrown, which must be caught, otherwise the program will terminate.

```
if (a == b)
{
  b.at(0)  = 'H';   // might be OK
  b.at(-1) = 'W';   // never OK.  throws exception
}
```

Remember that a `std::string` is **not** a byte string. `std::string` is a class. A decision was made long ago that in order to remain more compatible with C, double quited strings should evaluate to byte strings. Declarations like this are a common source of confusion for new programmers:

```
auto my_string = "Howdy!";
```

What type is `my_string`?

Show

`my_string` is **not** a std::string.

The default type for characters enclosed in double quotes is `const char*`.

This is one of those situations where auto may not be deducing the type you actually want. There are several simple ways to use auto *and* get the type deduced to be a `std::string`.

Example

C++14 Feature

Adds the concept of a *std::string literal* to the language.

You can simply append an `s` to the end of the quoted string. This identifies the literal as type std::string instead of `const char*`.

```
auto my_string = "Howdy!"s;     // preferred
```

Alternatively, you can call the string constructor explicitly, which works for C++ versions older than C++14.

```
auto my_string = string("Howdy!");

auto your_str  = string{"Howdy!"};  // C++11 initialization syntax
```

Run It

```
#include <string>
#include <iostream>
#include <iomanip>  // std::quoted

int main()
{
    using namespace std::string_literals;
    using std::string;

    auto my_string = "Howdy!"s;
    auto howdy2 = string("Howdy!");
    auto your_str  = string{"Howdy!"};  // C++11 initialization syntax

    string s1 = "abc\0\0def";
    string s2 = "abc\0\0def"s;

    std::cout << "s1: " << s1.size() << ' ' << std::quoted(s1) << '\n';
    std::cout << "s2: " << s2.size() << ' ' << std::quoted(s2) << '\n';
}
```

Note

Look carefully at the size values printed for `s1` and `s2`.

Are those sizes expected? Why or why not?

Getting information out of a string

A `string` knows its own size and can provide other useful information.

```
#include <cassert>
#include <string>
using std::string;
int main() {
  string my_string = "Hello";

  assert( my_string.size() == 5 );  // .length() is available also
  if (!my_string.empty()) {
    my_string += ", there.";       // my_string == "Hello, there."
  }
  return 0;
}
```

And the string class provides many functions dedicated to finding substrings within a string.

Example: find

The simplest example is the find function.

Given any string object, for example, this string:

```
  if(pos == std::string::npos) {
```

defined using the string literal syntax, creates a new object `us`.

Once we have a `string`, calling the string member function find always returns a position:

- Either a position within the string, or
- The special value std::string::npos which means the value was not found in the string.

We can use this to check if we found what we were looking for:

```
    std::cout << "There is no 'i' in 'team'\n";
  }
  auto hi = "Hello world"s;
  pos = hi.find("wor");                 // pos == 6
```

The position returned by find is a zero-based index into the string.

Find can also take a sequence of characters. In that case, it returns the position to the first `char` where the entire sequence is matched.

Reverse find

Similar to `find`, rfind performs the same task as find, but iterates through the string in reverse order: starting at the end and moving towards the first character. Keep in mind that the position returned is still based on the same index positions as regular `find`.

Find first of

The find_first_of function takes a `char` sequence, but unlike find where the entire sequence is used to find a match, `find_first_of` examines each character in the sequence, on at a time, and returns the *minimum position* of **any** of the characters listed as function arguments in the string. For example:

The function returns the position of 'e' in 'Hello world', even though 'e' and 'o' are both present, because 'e' is first.

The order of the character arguments do not matter. The results would be exactly the same if the arguments were 'uoiea'. Don't take my word for it, try it yourself.

Run It

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

using namespace std::literals;

int main() {
  auto us = "team"s;
  size_t pos = us.find('i');
  if(pos == std::string::npos) {
    std::cout << "There is no 'i' in 'team'\n";
  }
  auto hi = "Hello world"s;
  pos = hi.find("wor");                 // pos == 6

       if (hi.find('o') == hi.rfind('o')) {
    std::cout << "There must be exactly 1 'o' in this string\n";
       }
  pos = hi.find_first_of("aeiou");      // pos == 1  (e)
  pos = hi.find_first_not_of("aeiou");  // pos == 0  (H)

  return 0;
}
```

The special value std::string::npos is used both as an end of string indicator by functions that expect a string and as an indicator of *not found* by functions that return an index (like find).

Video

Video URL: https://www.youtube.com/watch?v=nkKeA74p3RY

Convert a byte string to a number

Another common string task is parse a C string and extract a numeric value.

The number conversion functions are part of the `string` header. There are number conversion functions that C++ inherits from C. They are all in the `cstdlib` header. Of these C functions, the `atoi` and `atof` functions should not be used. They fail silently. That is if something bad happens, there is no way to check and even detect whether the function completed its task.

Use stoi and stod instead.

stoi()

stoi extracts an integer value from a string.

`stoi` discards any whitespace characters until the first non-whitespace character is found, then takes as many characters as possible to form a valid integer number representation and converts them to an integer value.

The valid integer value consists of the following parts:

- an optional plus or minus sign
- the digits 0-9

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

int main() {
    std::cout << std::stoi("42") << '\n';

    const char* str_pi = "3.14159";
    std::cout << std::stoi(str_pi) << '\n';
}
```

stod()

stod extracts an floating point value from a byte string.

Other than return value, it functions similarly to `stoi`.

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

int main()
{
    std::cout << std::stod("42") << '\n';

    std::cout << std::stod("0.0000000123") << "\n"
              << std::stod("0.012") << "\n"
              << std::stod("15e16") << "\n"
              << std::stod("-0x1afp-2") << "\n"
              << std::stod("inf") << "\n"
              << std::stod("Nan") << "\n";
}
```

All of these functions allow for error handling, which we will cover in a future chapter. For now, know that these function will generate errors if given bad or unexpected input.

Converting a std::string to a C string

You cannot use `std::string` in a function that expects `const char*` - you must convert it to a null terminated character array.

```
auto my_name = "Alice"s;


printf ("Hello again, %s\n", my_name);       // compile error!
```

(continues on next page)

(continued from previous page)

```
// the c_str() function converts a string into a c string
printf ("Hello again, %s\n", my_name.c_str());
```

Self Check

Q1

Fill in the blank

Given the following:

```
std::string x = "The rain in Spain. . . ";
size_t pos = x.find("in");
```

What is the value of pos?

_____

Q2

Parsons problem

Code fragments

```
auto snowflake = us.find_first_of("Korea");

if (snowflake == std::string::npos) {

int main() {

std::cout << "Did not find anything\n";

std::cout << "Found it!\n";

std::string  us = "Team USA";

}

}
return snowflake;

} else {
```

Q3

Fill in the blank

Given the following:

```
#include <string>

int main (){
  std::string s = "Donald Duck";
  int value = 0;
  if (s.find_first_of(' ') == s.find_last_of(' ')) {
```

(continues on next page)

(continued from previous page)

```
    value = 3;
  } else {
    value = 5;
  }
  return value;
}
```

What value is returned from main?

_____

More to Explore

- cppreference.com Strings library overview
- YoLinux String class tutorial
- Bjarne Stroustrup's C++11 FAQ: Raw String literals

3.2.5 Analysis of String Operators

Prior to C++11 the string class was not required to store its character elements contiguously. Now string acts much like the vector class, except for some string optimizations and other minor differences.

C++11 strings use contiguous storage locations in an underlying (typically larger) array just like vector. Due to this, the character elements in strings can be accessed and traversed with the help of iterators, and they can also be accessed randomly using indexes.

Like vectors, strings have a dynamic size meaning that whenever a new character is inserted or deleted, their size changes automatically. Just like vectors, new elements can be inserted into or deleted from any part of a string, and automatic reallocation for other existing items in the string is applied.

Indexing and assigning a new character to an index position that already exists both take $O(1)$, in other words, the same amount of time no matter how large the string is.

Now that we have seen how performance can be measured concretely you can look at the *string operations table* to see the average complexity of all the basic string operations and you will see a striking resemblance to vectors because the implementations are so similar.

Table 1: **Big-O of C++ string operations**

Operation	Big-O Efficiency
index []	O(1)
index assignment =	O(1)
push_back()	amortized O(1)
pop_back()	O(1)
erase(i)	O(n)
insert(i, item)	O(n)
find(b, e, item)	O(log n) or O(n)
reserve()	O(n)
begin()	O(1)
end()	O(1)
size()	O(1)

push_back() is $O(1)$ unless there is inadequate capacity. Then the entire string is moved to a larger contiguous underlying array. Copying all the old string data to a new location is $O(n)$.

The previous table says that `find` could be $O(n)$ or $O(\log(n))$. One might ask why not just write a little for loop instead? Searching for a value seems like such a simple thing. Why go through the effort to figure out how to use all these string functions? Let's find out.

In the program below, the time to perform operations is measured using the std::chrono library. It provides a flexible collection of types that track time with varying degrees of precision. steady_clock::now returns the current time. The elapsed time between two time points is stored in a duration object. Duration objects make it obvious what the time units are and also makes it easy to convert. The is one of the major advantages over the C time functions.

To use the `steady_clock` to time an algorithm or function, create two time points. To get the total runtime, subtract the begin time from the end time.

This example creates a series of increasingly long strings.

Every character except for one is the same.

The program prints how long it takes to find it when placed at the midpoint of the string.

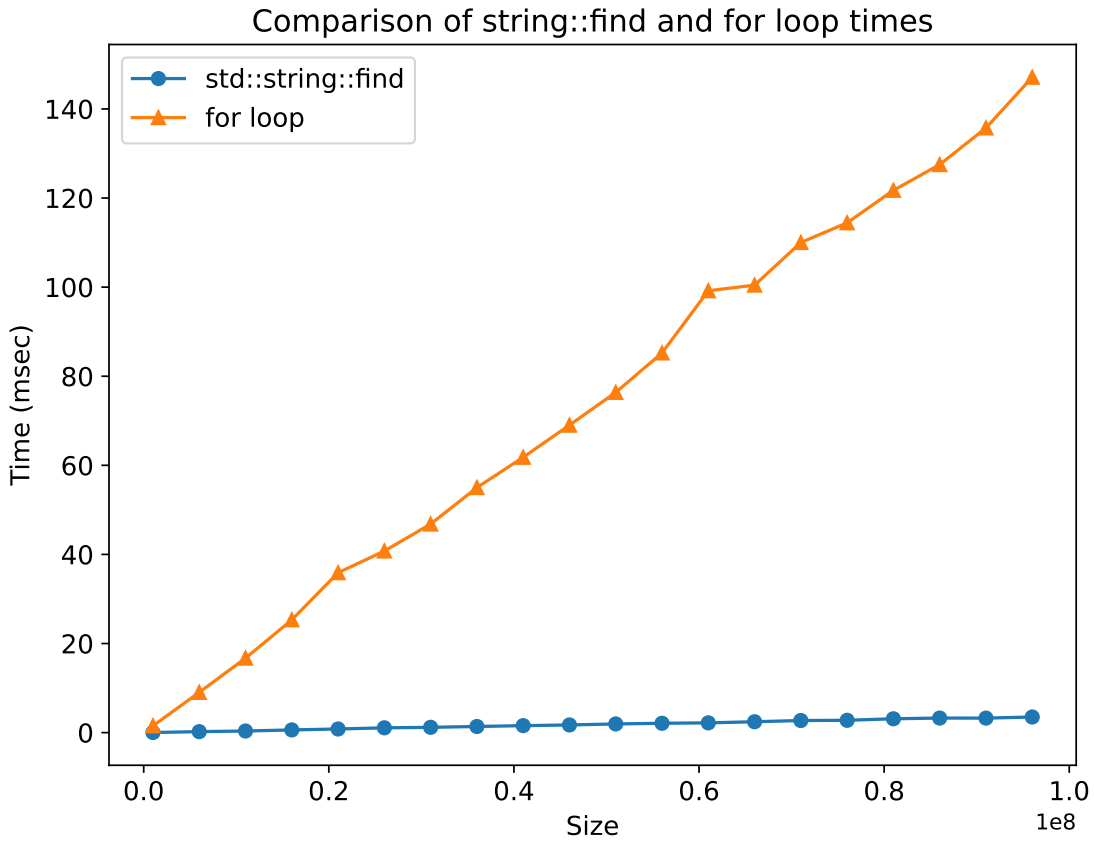
```
#include <chrono>
#include <iomanip>
#include <iostream>
#include <string>

int main() {
    using std::cout;
    using std::chrono::steady_clock;
    using msec_t = std::chrono::duration<double, std::milli>;

    cout << std::setw(6) << "size\t\t"
         << std::setw(8) << "string::find\t"
         << std::setw(8) << "for loop (all times in msec)\n";
    for(int size = 1e6; size < 1e8; size += 5e6) {
        // create a big string
        std::string haystack (size, 'h');

        // insert a unique character
        auto needle = 'n';
        haystack[size/2] = needle;

        // search for needle in haystack using string find function
        auto begin = steady_clock::now();
        if (haystack.find(needle) == std::string::npos) {return -1;}  // error
        auto end = steady_clock::now();
        msec_t elapsed_msecs = end - begin;

        // search for needle in haystack using a for loop
        auto begin_for = steady_clock::now();
        for (auto k = 0; k < size; ++k) {
            if (haystack[k] == needle) { break; }
            if (k > size/2)            { return -2; }  // error
        }
        auto end_for = steady_clock::now();
        msec_t for_msecs = end_for - begin_for;
```

(continues on next page)

(continued from previous page)

```
        cout << std::setw(6) << size << '\t';
        if (size < 9e6) { cout << '\t'; }
        cout << std::setprecision(6) << std::fixed
             << std::setw(8) << elapsed_msecs.count() << '\t'
             << std::setw(8) << for_msecs.count() << '\n';
    }
    return 0;
}
```

A graph of the loops in the preceding code should look something like this:



Try This!

What is the Big-O of find? $O(n)$ or $O(\log(n))$?

What happens if the needle value is in a location other than the midpoint? Try the beginning and end to see what happens.

Challenge: Try putting the needle in a random location to see what happens.

More to Explore

- *Asymptotic Analysis and Upper Bounds*
- cppreference.com Strings library overview
- Mike Shahar post: Exploring std::string
- Average time complexity

3.2.6 The vector class

A vector is intended to behave like a dynamically sized array. It is a *template*, so unlike a string, which is a container for characters only, a vector can serve as a container for any type. More on templates later, for now, we just need to know enough to know how declare a vector.

As with strings, in standard C, the typical way to work with a collection of data is with a 'raw' array:

```
int a[] = {3, 1, 4, 1, 5, 9};
```

Some downsides to raw arrays are that they:

- Do not know their own size
- Need to have their size specified when declared
- Decay into pointers easily
- Provide no convenience functions

The `vector` class solves these problems for us and a few others besides. Declaring a `vector` is quite similar to the `string` declarations from the previous section. In order to access the STL vector capabilities, use `#include <vector>`.

Declare

To properly declare a vector, the *type* of data stored in the vector must be declared as a type parameter.

The `<int>` and `<std::string>` represent the *template parameters* passed to the `vector`. It is these template parameters that allow the vector class to serve as a container for (almost) any type. There are some limits we will cover later, but for now, know that any normal type you already have learned about can be stored in a vector.

```
// an empty vector of int
vector<int> x;

// initialize and store: "x", "x"
vector<std::string> dos_equis (2, "x");

// C++11 initialization list syntax
vector<int> pi_digits = {3,1,4,1,5,9};
```

Unlike a fundamental type, the declaration `vector<int> x;` does **not** create an uninitialized variable. It creates a fully formed vector with no elements stored in it yet. This is perfectly OK and normal.

However, a common error is to forget to include the template parameter:

```
vector x;       // compile error
```

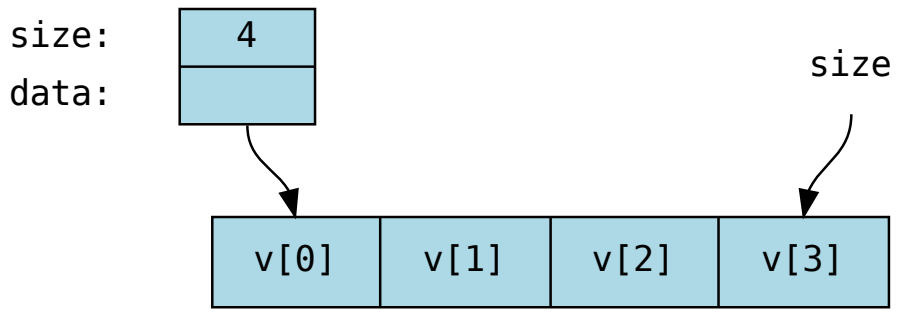
Run It

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

using std::vector;       // alias type std::vector

int main() {
  vector<int> x;                          // empty vector of int

  for (const auto& value: x)
  {
    std::cout << value << ',';
  }
  std::cout << '\n';

  vector<std::string> dos_equis (2, "x");  // "x", "x"

  vector<int> pi_digits = {3,1,4,1,5,9};  // C++11
  for (const auto& value: pi_digits)
  {
    std::cout << value << ',';
  }
  std::cout << '\n';

  return 0;
}
```

Given a vector declared as:

```
std::vector<int> v(4);
```

A container capable of storing 4 integers is created:



Although the `vector` object is initialized, its contents are not. Many compiler implementations will initialize the contents to zero, but don't rely on this behavior. Explicitly initialize with a default value, if that is what you want:

```
std::vector<int> v(4, -1);
```

A vector comes with a rich assortment of convenience functions. Like an array, the operator[] can be used to access elements without bounds checking. Like a string, the at function provides bounds checking and will throw a std::out_of_range exception if an out of bounds index is used on the `vector`.

Access operations

Like arrays, indexes are zero-based.

```
// read vector elements
std::cout << "First element: " << numbers[0];
std::cout << "First element: " << numbers.at(0);

// write vector elements
numbers[0] = 5;
numbers.at(0) = 5;
```

A common source of error occurs when printing a vector. A vector feels like a built-in type and this seems like it should work:

```
// compile error
std::cout << "all numbers: " << numbers;
```

The vector type does not 'know' how to send it's values to an output stream by default.

Something to consider

Why do you think this feature is not built into the standard library?

Run It

This example demonstrates an out of range error.

How can we fix this while changing the least amount of code possible?

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

int main() {
  std::vector<int> numbers {2, 4, 6, 8};
  std::cout << "Size: " << numbers.size() << '\n';
  std::cout << "Second element: " << numbers[1] << '\n';

  numbers.at(0) = 5;
  numbers.at(4) = numbers[3] + 2;  // out of range error.
                                   // index 4 is out of bounds

  std::cout << "All numbers:";
  for (const auto& num : numbers) {
    std::cout << ' ' << num;
  }
  std::cout << '\n';
```

(continues on next page)

(continued from previous page)

```
  return 0;
}
```

Fill and print vector

A vector 'hello world':

1. Fill a vector in one simple way, using push_back
2. iterate through the vector and print

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

using std::cout;
using std::string;
using std::vector;


int main() {
  vector<string> words = {
     "reach", "clear", "fall", "set", "yard",
     "liquid", "wise", "badge", "four", "coherent"
  };

  cout << "Word list: \n";
  for (string s: words) {
    cout << s << ", \thas " << s.size() << " letters\n";
  }
}
```

The vector class also provides:

front and back
: return a reference to the first and last elements

size
: return the number of elements

empty
: return `true` if the container is empty

Although there are more functions, these are the ones we need to worry about for now. We will be looking more at memory management in vectors in *The std::vector class*.

Something to consider

What is the difference between a `std::string` and `std::vector<char>`?

Why did the developers of the STL decide it was important to include both?

Comparisons between vectors are also automatically handled by the class. In the case of a vector, operator==, or an equality comparison between two vectors `a` and `b`, means the two vectors are equal if `a.size() == b.size()` and each element in `a` compares equal with each element in `b` in the same position in the vector.

Compare operations

Vectors support the same syntax as the built in types.

```
// declare 2 vectors, one empty and one not

std::vector<int> x {2, 4, 6, 8};
std::vector<int> y;

bool test = (x == y);   // test is false

y = x;
```

Run It

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

int main() {
  std::vector<int> x {2, 4, 6, 8};
  std::vector<int> y;

  if (x == y) {
    std::cout << "x and y are equal\n";
  } else {
    std::cout << "x and y differ\n";
  }

  y = x;          // copy all data from x into y
  if (x == y) {
    std::cout << "x and y are equal\n";
  } else {
    std::cout << "x and y differ\n";
  }

  return 0;
}
```

Try This!

Create two vectors of strings containing the same values and check them for equality.

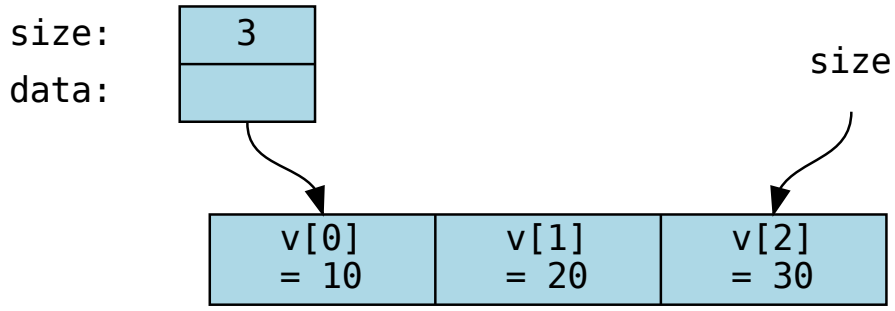
```
#include <iostream>

int main() {
}
```

Adding data to a vector

How do we solve the out_of_range exception from a few examples ago? How do we dynamically add data to a `vector`?

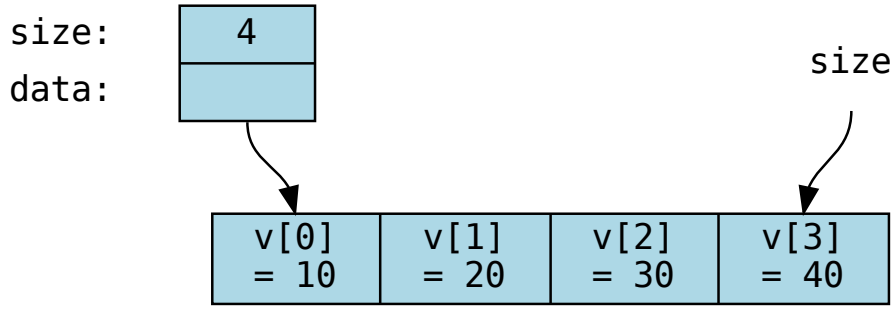
A simple way is to use the push_back function.

Given an vector of 3 int's:




```
values.push_back(40);
```

Appends the value 40 to the end of the vector.



push_back and pop_back

push_back appends an element to the end and increases the capacity of the vector, if needed.

pop_back reduces the size of the vector by one. The last element is no longer available.

Note that pop_back does not return a value.

If you need that last element, remember to save it first.

```
std::vector<char> letters {'a', 'b', 'c'};

letters.push_back('d');  // add 'd' to the end of the vector
letters.pop_back();      // pop_back is the opposite:
```

Run It

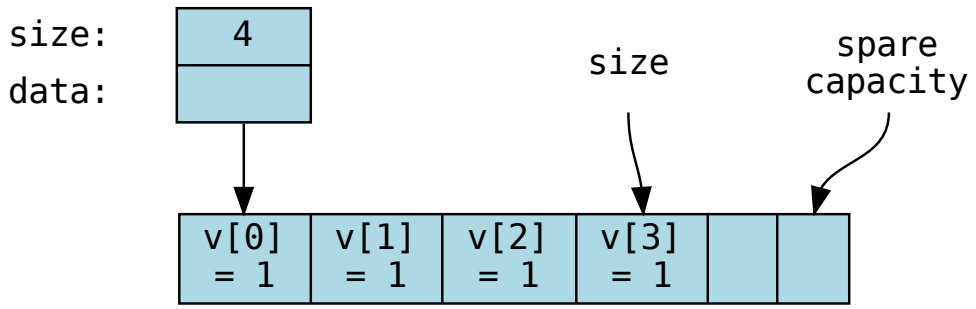
```cpp
#include <vector>
#include <iostream>

int main() {
  std::vector<char> letters {'a', 'b', 'c'};

  letters.at(0) = 'z';
  letters.push_back('d');  // add 'd' to the end of the vector
  char ch = 'e';
  letters.push_back(ch);  // add 'e' to the end
  letters.pop_back();     // pop_back is the opposite:
                          //  - removes the end element from the vector

  std::cout << "All letters:";
  for (const auto& c : letters) {
    std::cout << ' ' << c;
  }
  std::cout << '\n';
  letters.clear();         // clear all contents from vector
  return 0;
}
```

Vector capacity

A vector exposes an interface that 'feels like' an array, but the underlying storage grows to accommodate new data as required. With an array, you either have to allocate as much memory as you *might* need in the worst case, even if only a small fraction is used most of the time or you allocate 'just enough' and when more memory is required, copy all the data into a new array. A vector does do this also, but the implementation is hidden and you don't have to worry about it.

Something to be aware of - when `pop_back` is called, no actual storage is deleted. The memory is still available in the vector and available for reassignment with `pop_back`.

This extra memory after the current size is referred to as the *total capacity* of the vector, or just *capacity*.



Managing the storage capacity in addition to the vector data is one of the things that make vectors efficient.

Phrase O'matic

The 'phrase-o-matic' is a port of a fun little java program from Head First Java, 2nd ed. ISBN-13: 978-0596009205

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

int main() {
  // initialize a random number generator
  std::random_device r;
  std::default_random_engine eng(r());
  using rand = std::uniform_int_distribution<std::uint64_t>;

  const char* list_one[] = {
    "24/7", "multi-tier", "30,000 foot", "B-to-B", "win-win",
    "front-end", "web-based", "pervasive", "smart", "six-sigma",
    "critical-path", "dynamic", "extreme", "three-tier", "agile"
  };

  // we prefer vector over arrays
  const std::vector<std::string> list_two = {
    "empowered", "sticky", "value-added", "oriented", "centric",
    "distributed", "clustered", "branded", "outside-the-box",
    "positioned", "networked", "focused", "leveraged", "aligned",
    "targeted", "shared", "cooperative", "accelerated"
  };

  const std::vector<std::string> list_three = {
    "process", "tipping-point", "solution", "architecture",
    "core competency", "strategy", "mind-share", "portal",
    "space", "vision", "paradigm", "mission"
  };

  const std::size_t one_size = 14;  // arrays don't know their size
  auto r1 = rand {0, one_size} (eng);
  auto r2 = rand {0, list_two.size()-1} (eng);   // vectors know their size
  auto r3 = rand {0, list_three.size()-1} (eng);

  std::string phrase = {list_one[r1]};
  phrase += " " + list_two[r2] + " " + list_three[r3];

  std::cout << "What we need is a " << phrase << '\n';

  // or could have omitted temporary phrase and simply:
  // std::cout << "What we need is a " << list_one[r1] << ' '
  //                                   << list_two[r2] << ' '
  //                                   << list_three[r3] << '\n';
  return 0;
}
```

vector test

A short test program to demonstrate the parts of the vector interface.

What other tests can you make?

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

using std::string;
using std::vector;

vector<string> make_vector() {
  return {
     "reach", "clear", "fall", "set", "yard",
     "liquid", "wise", "badge", "four", "coherent"
   };
}

void check (const string& name, const string& actual, const string& expected)
{
  std::cout << std::left << std::setfill('.')
            << std::setw(50) << name
            << std::setw(7) <<  std::left;
  if(actual == expected) {
    std::cout << " OK      \n";
    return;
  }
  std::cout << " FAILED\n";
  std::cout << "\treceived [" << actual
            << "], but expected [" << expected << "]\n";
  exit(1);
}

// write test cases
void test_simple_access(vector<string> words) {
   check("test at()", words.at(0), "reach");
   check("test operator[]", words[0], "reach");
   check("test at() end", words.at(9), "coherent");
   check("test operator[]", words[9], "coherent");

   // foo.at(i) and foo.[i] refer to the same thing
   for (auto i=0U; i< 3; ++i) {
   // for (auto i=0U; i< words.size(); ++i) {
      check("at and operator[] are the same", words.at(i), words[i]);
   }

   const string empty;
   // Try uncommenting these lines to see what happens
   //check("test at out of bounds", words.at(-1), empty);
   //check("test at out of bounds", words.at(11), empty);
   //check("test [] out of bounds", words[-1], empty);
```

(continues on next page)

(continued from previous page)

```
   //check("test [] out of bounds", words[11], empty);
}

void test_other_access(vector<string> words) {
   check("test front()", words.front(), words.at(0));
   check("test back()", words.back(), words.at(words.size()-1));
}

void test_assignment(vector<string> words) {
   check("test original value", words[2], "fall");
   words[2] = "falldown";
   check("test new value", words[2], "falldown");
}

// call test cases
int main() {
  auto words = make_vector();

  test_simple_access(words);
  test_other_access(words);
  test_assignment(words);
}
```

More to Explore

- cppreference.com std::vector
- WikiBooks.org C++ Programming STL Containers

3.2.7 Analysis of Vector Operators

Accessing data using an index and assigning to an index position that already exists both take the same amount of time no matter how large the vector is. When an operation like this is independent of the size then it is $O(1)$.

As we have seen, one way to create a longer vector is to use the `push_back()` method. The `push_back()` method is typically $O(1)$, provided there is adequate capacity in the underlying array.

First we'll use `push_back()` method. *The following code* shows the code for making our vector.

```
#include <vector>
using std::vector;

void test_push_back(int size){
    vector<int> vect;
    for (int i = 0; i < size; ++i){
        vect.push_back(i);
    }
}
```

And we can time how long it takes to push 10,000 values into a vector.

```cpp
#include <chrono>
#include <iostream>
#include <vector>
using std::vector;

void test_push_back(int size){
    vector<int> vect;
    for (int i = 0; i < size; i++){
        vect.push_back(i);
    }
}

int main(){
    using msec_t = std::chrono::duration<double, std::milli>;

    auto begin = std::chrono::steady_clock::now();
    test_push_back(10'000);
    auto end = std::chrono::steady_clock::now();
    msec_t elapsed_time = end - begin;

    std::cout << "push_back (msec): " << elapsed_time.count() << '\n';

    return 0;
}
```

In the experiment above the statement that we are timing is the function call to `test_push_back`. From the experiment, we see the amount of time taken by the push_back operation. Not only is the `push_back()` function call duration being measured, but the time to allocate space is being measured.

We can improve the runtime a bit further by setting an adequate reserve for the vector in advance. Doing this will keep us from having to move the entire vector to an adequately sized space in memory as the vector grows.

```cpp
#include <chrono>
#include <iostream>
#include <iomanip>
#include <vector>
using std::vector;

void test_push_back(int size){
    vector<int> vect;
    for (int i = 0; i < size; ++i){
        vect.push_back(i);
    }
}

void test_reserve(int size){
    vector<int> v(size);
    for (int i = 0; i < size; ++i){
        v[i] = i;
    }
}

int main(){
```

(continues on next page)

(continued from previous page)

```
    using std::cout;
    using std::chrono::steady_clock;
    using msec_t = std::chrono::duration<double, std::milli>;

    cout << std::setw(6) << "size\t"
         << std::setw(8) << "push_back\t"
         << std::setw(8) << "pre-allocated vector (all times in msec)\n";

    for(int size = 1'000; size < 1'000'000; size += 50'000) {

      auto begin = steady_clock::now();
      test_push_back(size);
      auto end = steady_clock::now();
      msec_t elapsed_1 = end - begin;

      auto begin2 = steady_clock::now();
      test_reserve(size);
      auto end2 = steady_clock::now();
      msec_t elapsed_2 = end2 - begin2;

      cout << std::setprecision(6) << std::fixed
           << size << '\t'
           << std::setw(8) << elapsed_1.count() << '\t'
           << std::setw(8) << elapsed_2.count() << '\n';
    }
    return 0;
}
```

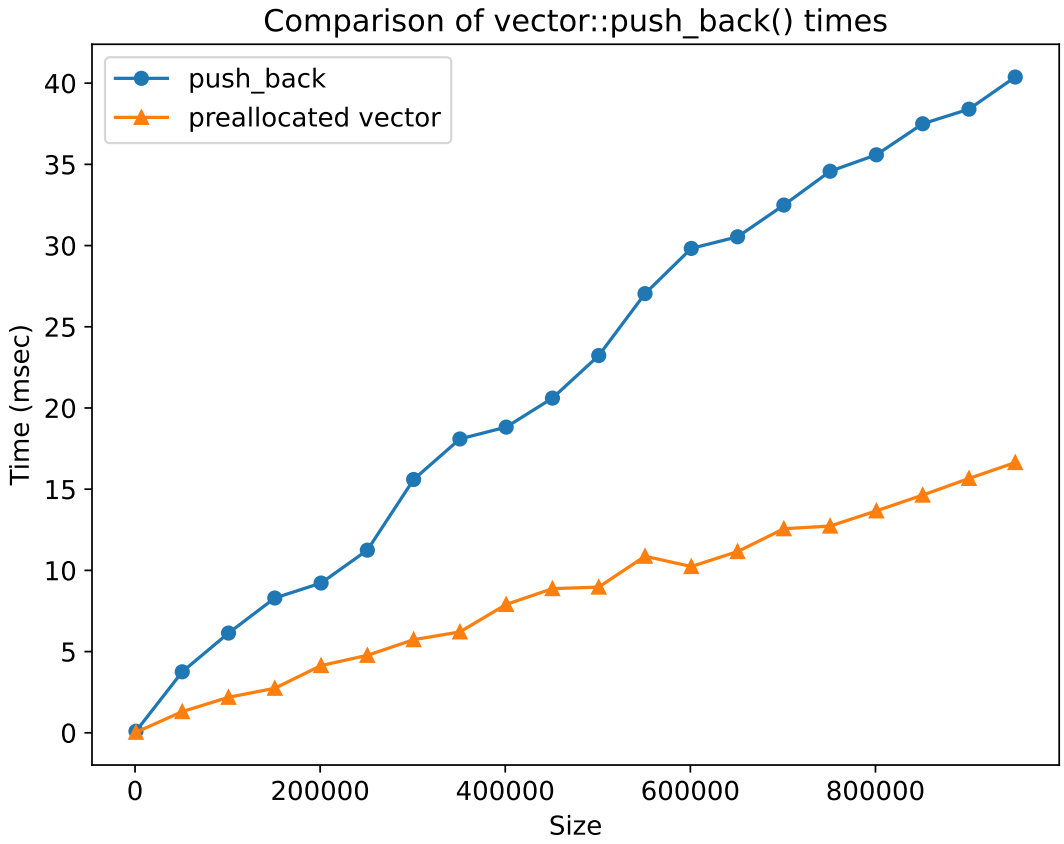
A graph of the loops in the preceding code should look something like this:

Now that we have seen how performance can be measured concretely you can look at *the table below* to see the complexity of some basic vector operations. When `pop_back()` is called, the vector size is reduced by 1 and it takes constant time: $O(1)$. However, when `erase()` is called the time is $O(n)$. The reason for this lies in how C++ chooses to implement vectors. When an item is taken from the front of the vector, in C++ implementation, all the other elements in the vector are shifted one position closer to the beginning. This implementation also allows the index operation to be $O(1)$. This is a trade-off that the C++ implementers thought was a good one.

Table 2: **Complexity of C++ vector operations**

Operation	Complexity
index []	O(1)
index assignment =	O(1)
push_back()	amortized O(1)
pop_back()	O(1)
erase(i)	O(n)
insert(i, item)	O(n)
find(b, e, item)	O(n)
reserve()	O(n)
begin()	O(1)
end()	O(1)
size()	O(1)

The `push_back()` operation is $O(1)$ unless there is inadequate capacity, in which case the entire vector is moved to a


Comparison of vector::push_back() times
push_back
preallocated vector
Time (msec)
Size
0
5
10
15
20
25
30
35
40
0
200000
400000
600000
800000

larger contiguous underlying array, which is $O(n)$. However, since over the long term, as n grows large, then number of vector copies is small. So on average, even though there are some $O(n)$ operations, it turns out that `push_back()` is constant time.

As a way of demonstrating the difference in performance between `pop_back()` and `erase()`, let's do another timing experiment. Our goal is to be able to verify the performance of the `pop_back()` operation on a vector of a known size when the program pops from the end of the vector using `pop_back()`, and again when the program pops from the beginning of the vector using `erase()`. We will also want to measure this time for vectors of different sizes. What we would expect to see is that the time required to pop from the end of the vector will stay constant even as the vector grows in size, while the time to pop from the beginning of the vector will continue to increase as the vector grows.

The following code shows one way to measure the difference between the `pop_back()` and `erase()`.

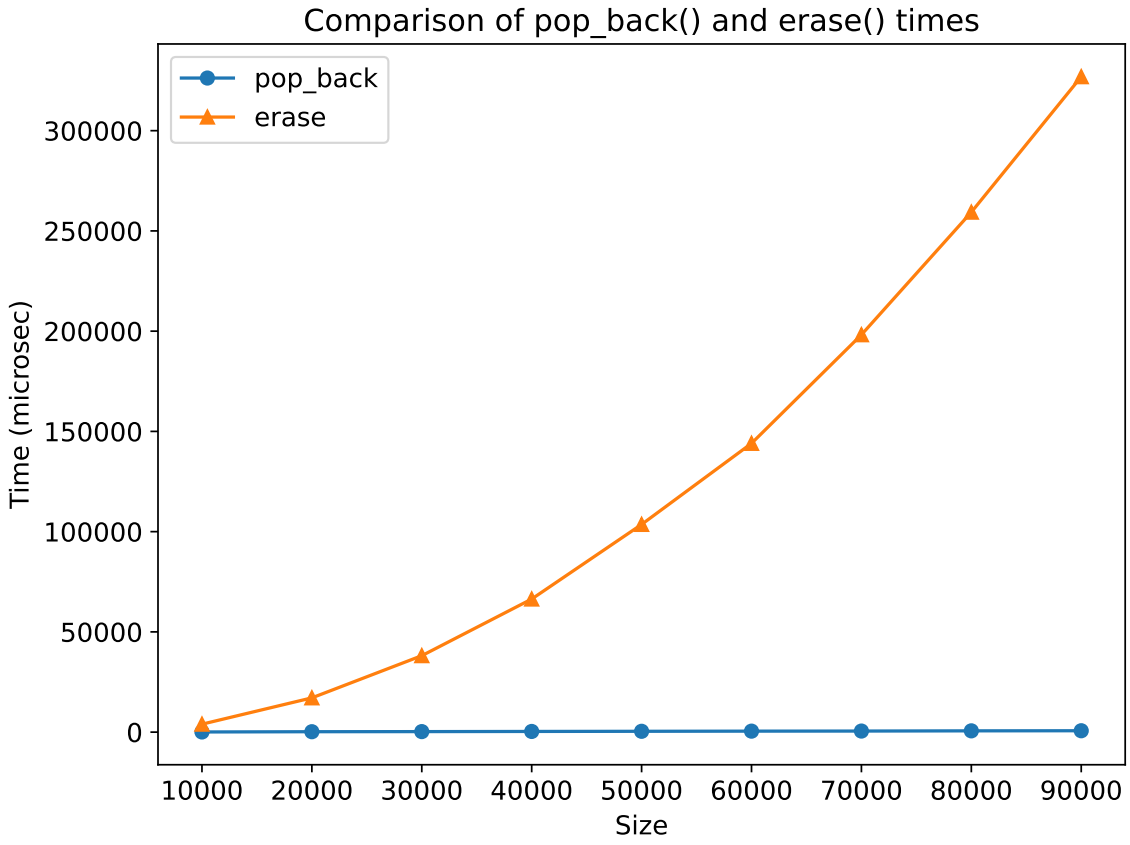
```
#include <chrono>
#include <iostream>
#include <iomanip>
#include <numeric>
#include <vector>
using std::vector;

int main(){
    using std::cout;
    using std::chrono::steady_clock;
    using msec_t = std::chrono::duration<double, std::micro>;

    cout << std::setw(6) << "size\t"
         << std::setw(8) << "pop_back\t"
         << std::setw(8) << "erase\t\t"
         << std::setw(8) << "how much faster is pop_back?\n";
    cout << std::setw(19) << "(microsec)\t"
         << std::setw(10) << "(microsec)\n";

    for(int size = 10'000; size < 100'000; size += 10'000) {
        // Create 2 identical vectors with values 0..N
        vector<int> data1(size);
        std::iota(data1.begin(), data1.end(), 0);
        vector<int> data2(data1);


      auto begin1 = steady_clock::now();
      for (int i = 0; i < size; i++){
        data1.pop_back();
      }
      auto end1 = steady_clock::now();
      msec_t elapsed_1 = end1 - begin1;

      auto begin2 = steady_clock::now();
      for (int i = 0; i < size; i++){
        data2.erase(data2.begin());
      }
      auto end2 = steady_clock::now();
      msec_t elapsed_2 = end2 - begin2;

      cout << std::setprecision(6) << std::fixed
```

(continues on next page)

(continued from previous page)

```
            << size << '\t'
            << std::setw(8) << elapsed_1.count() << '\t'
            << std::setw(8) << elapsed_2.count() << "\t"
            << std::setprecision(0)
            << std::setw(8) << elapsed_2.count() / elapsed_1.count() << " times\n";
    }
    return 0;
}
```

Although erase is $O(n)$, a graph showing how much faster `pop_back()` can be as the size of a vector grows can still be surprising.



Self Check

Q1

Matching question

Select the complexity associated with the listed operation(s).

A. begin(), end()	___ O(1)
B. erase(i), insert(i, item)	___ O(1)
C. find(srt, stp, item)	___ O(1)
D. index assignment =	___ O(1)
E. push_back(), pop_back()	___ O(log n)
F. reserve()	___ O(n)
G. size(), index []	___ O(n)

More to Explore

- *Asymptotic Analysis and Upper Bounds*
- cppreference.com std::vector overview
- Average time complexity

3.3 Introduction to functions

An introduction to functions, including function parameters, namespaces, and scopes. The keyword `const` is introduced and will be revisited over several chapters.

The special characteristics of `main` and command line argument handling are introduced.

3.3.1 Introduction to functions

A *function* is a group of statements that work together to perform a task. A function is composed of:

- A *name*
- Zero or more *parameters*
- A return value
  - A function that returns no value must still declare that fact by specifying a return type `void`.

```
return_type function_name (argument list)
{
   // zero or more statements
}
```

The part outside the braces is called the **function declaration**. The braces and their contents is called the **function body**. Once defined, a function may be called and the task it defines can be executed as often as needed.

Example

Some simple, specific examples:

```
int area (int height, int width) {
  return height*width;
}

void say_hello() {
```

(continues on next page)

(continued from previous page)

```
  std::cout << "hello";
}
```

Run It

```
#include <iostream>

int area (int height, int width) {
  return height*width;
}

void say_hello() {
  std::cout << "hello";
}

int main () {
  say_hello();
  std::cout << "\narea = " << area(4,3) << '\n';
  return 0;
}
```

Listing 1: Simple functions

```
#include <iostream>

int area (int height, int width) {
  return height*width;
}

void say_hello() {
  std::cout << "hello";
}

int main () {
  say_hello();
  std::cout << "\narea = " << area(4,3) << '\n';
  return 0;
}
```

Function declarations and compilation units

By default, functions have global *scope*: they can be used anywhere in your program, even in other files. If a file doesn't contain a *declaration* for a function before it is used, then the compiler will complain.

The solution is to either:

(a) Move the function definition before any functions that use it; or

(b) Put in a declaration without a body before any functions that use it, in addition to the declaration that appears in the function definition.

Option (b) is generally preferred, and is the only option when the function is used in a different file.

To make sure that all declarations of a function are consistent, the usual practice is to put them in an include file. For example, if the `area` function is used in a lot of places, we might put it in its own file `area.cpp`:

```
#include "area.h"

int area (int height, int width) {
  return height*width;
}
```

The file `area.cpp` above uses an `#include` directive to include a copy of the following header file area.h:

```
#ifndef AREA_H
#define AREA_H

/* Returns the area of a rectangle */
int area (int height, int width);

#endif /* AREA_H */
```

Note that the declaration in `area.h` doesn't have a body. Instead, it's terminated by a semicolon, like a variable declaration. A *function declaration* serves the same purpose as a *variable declaration*: they both introduce a new name and its type into a *scope*.

The `#ifndef`, `#define`, and `#endif` together form a pattern called a *header guard* or *include guard*. They ensure the functions in include files are defined only once.

By convention, the documentation for functions is primarily in the include file. The idea is that `area.h` is the public interface of this module, and so the explanation of how to use the function should be there. The reason `area.cpp` includes `area.h` is to get the compiler to verify that the declarations in the two files match.

- `area.h` contains the function declaration
- `area.cpp` contains the function definition (which includes a declaration)

Best Practice

Keep your declarations and definitions separate.

The source file that *defines* a function should include the file that *declares* a function.

Every other file that needs to use the `area` function uses an include directive `#include "area.h"` at the top of the file that uses it:

```
1 #include "area.h"
2
3 bool too_small (int x, int y) {
4   const int min_size = 10;
5   return area(x, y) < min_size;
6 }
```

The `#include` on line 1 uses double quotes instead of angle brackets; this tells the compiler to look for `area.h` in the current directory instead of the system include directory (typically /usr/include). Using `make`, you can add directories to the include search path using `-I`.

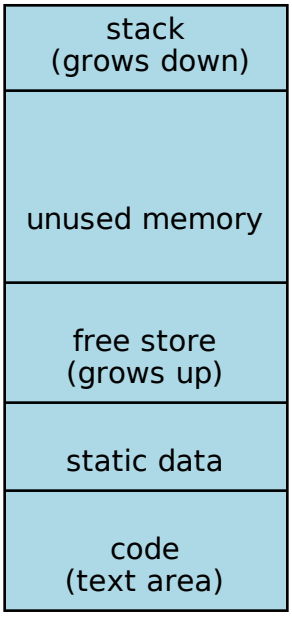
See also

Scope

The call stack

Functions are routinely called from many places and more than one function can be 'active' at any one time. The CPU needs a mechanism to keep track of every function call, all function parameters, and local variables, so that the CPU can execute each instruction in its proper order.

Some of this information will be stored in **registers**, memory locations built into the CPU itself, but most will go on the *stack*, a region of memory that on typical machines grows downward, even though the most recent additions to the stack are called the "top" of the stack.



Typical program memory layout

Typically, each called function and any local variables, return values, or parameters passed in, is stored in a special data structure called a **stack frame** or an **activation record**. Each function call pushes another activation record onto the stack.

The location of the top of the stack is stored in the CPU in a special register called the **stack pointer**. So a typical function call looks like this internally:

1. The current instruction pointer or program counter value, which gives the address of the next line of machine code to be executed, is pushed onto the stack.
2. Any arguments to the function are copied either into specially designated registers or onto new locations on the stack. The exact rules for how to do this vary from one CPU architecture to the next, but a typical convention might be that the first few arguments are copied into registers and the rest (if any) go on the stack.
3. The instruction pointer is set to the first instruction in the code for the function.
4. The code for the function allocates additional space on the stack to hold its local variables (if any) and to save copies of the values of any registers it wants to use (so that it can restore their contents before returning to its caller).

5. The function body is executed until it hits a return statement.
6. Returning from the function is the reverse of invoking it:
    - Any saved registers are popped back from the stack,
    - The return value is copied to a standard register,
    - The values of the instruction pointer and stack pointer are restored to what they were before the function call.

From the programmer's perspective, the important point is that both the arguments and the local variables inside a function are stored in freshly allocated locations that are thrown away after the function exits. After a function call the state of the CPU is restored to its previous state, except for the return value. Any arguments passed to a function are passed as copies by default, so changing the values of the function arguments inside the function has no effect on the caller. Any information stored in local variables is lost.

Try This!

Read the code below and predict what the output should be **before** stepping through it.

```
#include <iostream>

// forward function declarations
void dig();
void deeper();

int main() {
  std::cout << "Programs always start in function main.\n";

  dig();

  std::cout << "Returned to main.\nexiting.";
  return 0;
}

void dig() {
  std::cout << "Digging...\n";
  deeper();
  std::cout << "Still digging...\n";
}

void deeper() {
  std::cout << "now even deeper....\n";
}
```

More to Explore

- Basic intro to functions from Buckys C++ Programming Tutorials.
- From: cppreference.com: function declarations.
- cppplusplus.com tutorial on functions

3.3.2 Passing parameters

In C and C++, parameter passing defaults to **pass by value**. Unless you specify otherwise, function parameters are initialized with *copies* of the actual arguments, and function callers get back a *copy* of the value returned by the function. Pass by value is the simplest way to get data into and out of functions.

```
void printFavorite(int x) {
  // 'x' is a copy of 'favorite'
  std::cout << "my favorite number is " << x << '\n';
}

int main() {
  int favorite = 72;
  printFavorite(favorite);
}
```

The important point is that two copies of my favorite number are stored. The one declared in main, `favorite`, and the one declared in printFavorite, `x`. The parameter `x` is initialized using the value of `favorite` in main.

```
#include <iostream>

// define the function
void printFavorite(int x) {
  std::cout << "my favorite number is " << x << '\n';
}

int main() {
  int favorite = 72;
  printFavorite(favorite);
  return 0;
}
```

More than one parameter can be passed. For example, a function to add two numbers makes copies of both parameters adds them together and returns a result, which is also copied.

Step through this example and see how the copies of both local variables and return values are managed on the stack.

```
#include <iostream>

// This function takes two parameters.
int add_numbers(int x, int y){
  int answer = x + y;
  return answer;
}

int main() {
  int a = 13;
  int b = 21;
  int sum = add_numbers(a, b);
  std::cout << sum << '\n';
  return 0;
}
```

One benefit of pass by value is that local changes to parameters do not impact the caller. That is, the caller can trust their data has not been modified.

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

void print_n (const std::string message, int repeat) {
  while (repeat > 0) {
    std::cout << message << '\n';
    --repeat;
  }
}

int main() {
  int n = 3;
  print_n ("hello, world", n);
  std::cout << "n = " << n;
}
```

For large / complex data types, however, pass by value becomes expensive even in small programs. An alternative to pass by value, is called **pass by reference**. The function parameter passed into the function is still a new variable. That does not change. However, rather than passing a *copy* of the entire object, instead we *bind the address* of the original object to a new variable. Only the object reference is passed to the function.

Pointers and References

In this respect, a reference behaves much like a `const pointer`.

- Both require an initial value in order to compile
- Neither can refer to (or point to) a different object

```
int  n = 3;          // typical int declaration
int  m = 5;

int& a = n;          // a refers to n
int* p = &n;         // p points to n,
                     // but could point to something else

int* const p2 = &m;  // p2 point to m and can only point there
```

The following are compile errors:

```
int& b;              // a reference that doesn't refer to anything
p2 = &n;             // attempt to change what a const pointer points to
```

If the pointer comparison is confusing, do not worry. We will delve more deeply into pointers soon, this is just for comparison for those people who have an introduction into pointers.

Run It

```
#include <iostream>

int main() {
  int  n = 3;          // typical int declaration
  int  m = 5;

```

(continues on next page)

(continued from previous page)

```
  int& a = n;           // a refers to n
  int& b = m;           // b refers to m

  int* p = &n;          // p points to n,
                        // but could point to something else

  int* const p2 = &m;  // p2 points to m and can only point there


  // any use of a is equivalent to thing the variable
  // that a refers to
  a = b;
  std::cout << "n = " << n << '\n';

  a = 0;

  p = &m;       // OK
  //p2 = &n;   // compile error

  std::cout << "n = " << n << '\n'
            << "m = " << m << '\n'
            << "a = " << a << '\n'
            << "b = " << b << '\n'
            << "p = " << * p << '\n'
            << "p2 = " << * p2 << '\n';
}
```

We use the *address of operator* & to declare that only the address of the variable is passed, rather than a copy. The primary advantage is that since all addresses are the same size, the cost of passing is the same, regardless of how large the object is.

Understanding references is critical to understanding how C++11 and later version of the language function. References are a major new language feature and we will be using them often from now on.

Try This!

Modify the `print_n` function signature so that the variable `repeat` is a **reference** instead of a copy.

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

void print_n (const std::string message, int repeat) {
  while (repeat > 0) {
    std::cout << message << '\n';
    --repeat;
  }
}


int main() {
  int n = 3;
  print_n ("hello, world", n);
```

```
  std::cout << "n = " << n;

  assert ( n == 0 );        // program terminates if false
}
```

A common source of confusion when starting out with references is keeping the `operator&` straight.

The meaning of this operator depends on how it is used.

On the left-hand side of an assignment, or in function parameters, **& always** defines a reference to a type:

```
int& a = 3;
const int& cr(a);     // cr refers to a,
                      // but we can't change the value of a using cr

void show_usage (std::string& message);

const double& pi = 3.1415926;
```

On the right-hand side of an assignment, **& almost always** means address of a variable. The only exception is when casting to a reference type:

```
int n = 3;
int* p = &n;      // p points to the address of the variable n

const int& cr(a);     // const reference

// cast away the 'const' part of cr
int& r2 = const_cast<int&>(cr);
```

In the last code block, notice that both `cr` and `r2` refer to `a`, however, `r2` can change the value of `a` because we cast away the `const` modifier that was part of `cr`.

Although the language allows casting away `const` like this, you should use this feature very sparingly.

There is a lot going on in the following program. You should step through this code and make sure you understand what is happening to the variables in `main` and the functions called from `main`.

```
#include <iostream>

// A copy of x is passed to this function.
// Changes to x are not reflected in the caller.
void by_value(int x) {
  std::cout << "in by_val the address of x is   " << &x << '\n';
  x = 99;
}

// A reference to x is passed to this function.
// Changes to x are not reflected in the caller.
void by_reference (int& x) {
  std::cout << "in by_ref the address of x is   " << &x << '\n';
  x = -1;
}

```

(continues on next page)

(continued from previous page)

```
int main () {
  auto alpha = 11;
  auto beta = 11;

  std::cout << "in main the address of alpha is " << &alpha << '\n';
  std::cout << "in main the address of beta is  " << &beta << '\n';

  by_value(alpha);
  by_reference(beta);

  std::cout << "alpha is now " << alpha << '\n';
  std::cout << "beta is now " << beta << '\n';
  return 0;
}
```

Q1

Given the following program:

```
#include <iostream>

int change_and_add(int &a, int &b) {
  a = 3;
  b = 4;
  return a + b;
}

int main() {
  int a = 1;
  int b = 2;
  int c = change_and_add(a, a);
  std::cout << a << b << c;
}
```

Fill in the blank

What is the output from this program?

_____

More to Explore

- Reference initialization
- const_cast conversion
- Value categories

3.3.3 The `main` function

C++ programs are made up of functions and every executable program must have exactly one function named `main` that serves as the entry point for the program. Libraries do not need a main, because they are intended to link with another program that contains a main. Several restrictions apply to the main function that don't apply to any other C++ functions. The main function:

- Does not need to be declared
- Cannot be overloaded
- Cannot be declared as inline or static
- Cannot have its address taken
- Cannot be called from your program

main

There are only a few main signatures that are valid entry points:

```
// if main takes no arguments
int main()

// if main takes command-line arguments
int main(int argc, char* argv[])
```

The names `argc` and `argv` are traditional. You could use any valid identifier, but most programs use these.

- `int argc`: the total number of arguments in argv, strings separated by *white space* (space or tab characters)
- `char *argv[]`: an array of these strings

  `char *argv[]` can also be specified as `char **argv`, which is the same thing, if you remember pointers from your first semester. If not, we'll cover it soon.

Some compilers allow passing the current system environment variables as a third argument passed into main. The environment variables are also passed in as an array of C strings.

```
int main(int argc, char** argv, char** envp)
```

If the linker can't find a function that evaluates to one of the above, then it will fail and your program build is incomplete.

Run It

A simple command line argument handling program. Compile, link, run (coliru) provides a c++ compiler and a very basic command line.

Press the "edit" button and then 'Compile, link and run" to compile and run the program.

Replit provides an interactive shell. You can run this program manually in the shell by typing

```
./main
```

Another convention is to store the name of the program as invoked on the command line in `argv[0]`. One side effect of this is that `argc` is **never** equal to zero and the array `argv` always contains at least 1 c string.

Why bother with command line programs?

Command line arguments make programs more flexible. They allow users to run the same program in different ways, often to provide more or less detailed output or otherwise change the behavior at runtime.

C++ is primarily used in *systems programming* and is a fundamental part of all **nix* programs. **nix* is short for *Unix* (and friends), *MacOS X*, and *GNU/Linux*. The combination of command line argument handling and taking input from *standard input* and writing output to *standard output* is the core around which most **nix* programs are designed.

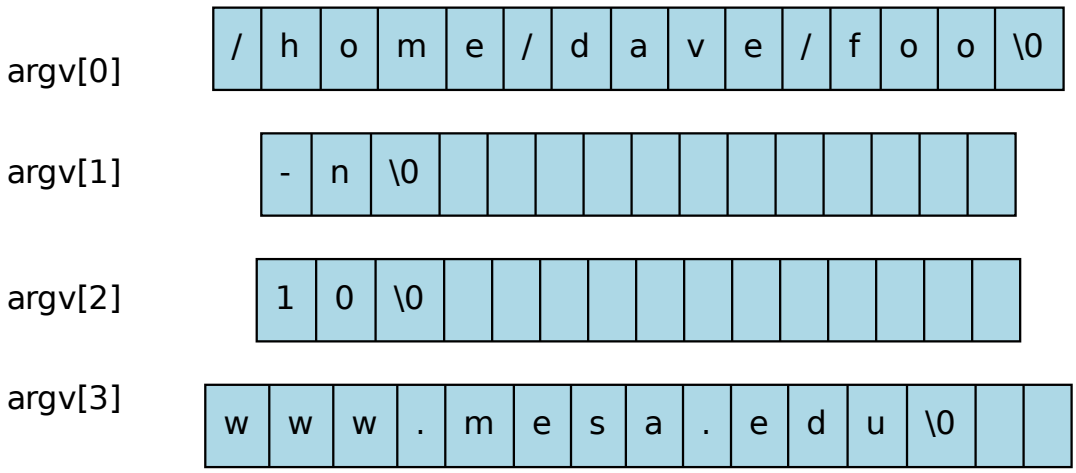
Parsing command line arguments

Parsing the command line is all about getting the user entered C strings from the command line and into our program in a useful form.

The important thing to remember is that `argc` and `argv` are passed automatically to main and are available for use. If you run a program named `foo` invoked as:

```
/home/dave/foo -n 10 www.sdmesa.edu
```

Then `argc` would be set = 4 and array `argv` would contain 4 arrays of length 15:



The two dimensional argv array

Different program `foo` invocations would result in different values for argc and argv.

There is nothing special about the character `-`. It is a convention used to distinguish command line arguments with special meaning (the switches) from other content.

echo

A simple echo program can demonstrate using command line parameters in a program.

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

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

  // why did I initialize this to 1 instead of 0?
  for (int i = 1; i < argc; ++i) {
    std::cout << "Hello, " << argv[i] << '\n';
  }

}
```

Try This!

Run `echo` with a variety of inputs, such as:

```
San Diego
"Mesa College"
```

Can you explain the differences?

Parsing values

Everything that is passed to main through `argv` is a C string. If you expect to receive a number on the command line, you need to transform the value from a character array into the appropriate numeric value yourself.

Traditional command line argument parsing proceeds as follows:

```
foreach argument
do
   if the current value equals an expected value
      process the argument
   else if the current value equals a different expected value
      process the argument
   else
      let the user know we received something unexpected
   done if
done foreach
```

There are many ways to check if two character arrays are equivalent. In this example, we use strcmp:

```
if (std::strcmp(argv[i], "-h") == 0) {
  // display help text
  break;
}
```

The strcmp and related functions are defined in the legacy C string header `<cstring>`. The function compares two null-terminated byte strings lexicographically (the way they would sort alphabetically). The sign of the result is the sign of the difference between the values of the first pair of characters (both interpreted as unsigned char) that differ in the two strings. The behavior is undefined if either argument are not pointers to null-terminated strings.

If the function returns `0`, the the two arrays are considered equivalent.

Sometimes a command line argument is used to communicate that a value of a particular type is expected to follow. Let's say we want our hello world program to repeat its message a certain number of times. We need a way to communicate this information to the program.

```
if (std::strcmp(argv[i], "-r") == 0) {
  // We should try to repeat,
  // increment the loop counter based on argc
  ++i;
  if (i < argc) {               // is there really a next argument?
      repeat = std::stoi(argv[i]);
  } else {
    std::cerr << "Error using '-r' argument: no repeat value provided\n";
}
```

There are many other ways to process the command line and many libraries exist to aid in the task. The technique presented here is simple and only uses facilities from the standard library.

Run It

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

using std::string;

//
// This function simply returns any text provided.
//
string echo(const string& text)
{
  return text;
}

string usage(const char* name) {
   string msg = "Usage: ";
   return msg.append(name).append(" [-h] [-r] [-n name]\n");
}

string help (const char* name) {
  auto msg = usage(name);
  constexpr auto text = R"help_text(
Options:
  -h   Show this text
  -r   Number of times to repeat.  Default = 1.
  -n   A name to say hello to. Default = "world".
)help_text";

  return msg.append(text);
}


int main(int argc, char** argv) {
  int repeat = 1;
```

(continues on next page)

(continued from previous page)

```
  string who = "world";
  // why did I initialize this to 1 instead of 0?
  for (int i=1; i < argc; ++i) {
      if (std::strncmp(argv[i], "-h", 2) == 0) {
      std::cerr << help(* argv);
      break;
    } else if (std::strncmp(argv[i], "-r", 2) == 0) {
          ++i;
      if (i < argc) {
          repeat = std::stoi(argv[i]);
      } else {
        std::cerr << "Error using '-r' argument: no repeat value provided\n";
      }
    } else if (std::strncmp(argv[i], "-n", 2) == 0) {
          ++i;
      if (i < argc) {
          who = argv[i];
      } else {
        std::cerr << "Error using '-n' argument: no name provided\n";
      }
    } else
        std::cerr << "Unknown argument '" << argv[i] << "' provided\n";
    }
  }

  do {
    std::cout << "Hello, " << who << "!\n";
    --repeat;
  } while (repeat > 0);

  return 0;
}
```

Try This!

Run this program with a variety of inputs and see what happens.

Try passing no arguments or switches, the same switch more than once, and a switch with no value after it.

A common source of confusion is distinguishing between 'standard input' and the command line. Parameters passed to a program after the program name are only stored in the array `argv`. Most operating systems allow you to use the special characters <, > (redirection operators) and | pipe operators to direct data into the standard input of a program. Information sent to a program using redirection or pipes is immediately available for use by any facility that can process the standard input stream, such as cin.

You can also use cin to manage a 'scripted conversion' with a user, where you prompt for input using cout and process the input using cin, however, processing standard input using redirection is far more flexible in terms of creating reusable programs that work together.

This idea is the foundation of Unix and its many derivatives, including GNU/Linux and Mac OS.

More to Explore

- Using the getopt function
- Textbook: *Pointers*
- From cppreference.com:
  - cin, cout, and cerr
  - strcmp, and strncmp
  - stoi, and strtol

3.3.4 Scope

Each name that appears in a C++ program is only valid in some *possibly discontiguous* portion of the source code called its *scope*.

Within a scope, unqualified name lookup can be used to associate the name with its declaration.

```
#include <cassert>

int n = 3;                  // global variable

int main() {
  int n = 42;               // local variable
  assert (n == 42);         // local scope definition replaces the global
}

void func () {
  assert (n == 3);          // only name 'n' in scope is the global one
}
```

It is important to remember that global variables are exactly that -- global. They are visible everywhere in every scope. Once created, there is no way to make a global variable go away. And since there is only 1 "global scope" it is easy to accidentally create name conflicts with variables in local scopes. This is one of (but not only) reason why global variables are considered something to avoid on most programming projects.

Macro assert

The `assert` macro from C is a simple way to validate something in your program that **must** be true before you proceed. To use the assert macro, `#include <cassert>`, then call the function `assert`. The assert function takes a boolean variable or an expression that evaluates to a `bool`.

Use the assert macro in your own code to validate function parameters or any other important checks.

In the following example, all of the assertions about `n` are `true`:

```
#include <cassert>

int n = 3;

int main() {
  assert (n == 3);
```

(continues on next page)

(continued from previous page)

```
  int n = 42;           // local n shadows global version
  assert (n == 42);

  // a new block scope can be created anywhere
  {
    double n = -3.14159;          // inner scope shadows both outer scopes
    assert (n - -3.1415 < 0.01);

  } // double n is no longer in scope

  assert (n == 42);
}
```

Try This!

Remove the braces that form the block scope around `double n`.

What do you expect to happen?

The global variable `n` is created before main is ever called and exists after main exists. Global variables always exist for the entire lifetime of a program. They are one of the few things in a program that outlive the `main()` function.

Global variables are created in the order they are encountered in a program. Because global variables are stored in special memory locations. If they are constants, they are stored in the `code` program segment. If they are variable, they are stored in the `data` program segment. Because these memory segments function like stacks, they are created and destroyed like stacks. In the case of global variables, the first variable created will be the last variable destroyed.

One last mention about global variables: globals of primitive types are initialized by default. This is not the same behavior as local variables, which are uninitialized by default. The confusion about when variables are initialized is a common source of error.

Local variables

You may read somewhere that global variables are evil, harmful, or whatever. There are no *harmful* language features.

There *are* some language features that can get you into trouble if you have not carefully weighed the pros and cons of their use. That is why in this book, we generally like to avoid absolutes and to use *prefer*.

We *prefer* keeping scope to the absolute minimum to get something done. Because we prefer keeping scope to a minimum, we *prefer*:

- Local variables over global variables.
- Declaring variables in the block where they are used.

There is no reason in modern languages to declare all your variables at the start of a function. This is actually a requirement of older languages like FORTRAN and the original K&R dialect of C. In modern languages, it's an old habit some people have carried forward for no good reason.

Errors related to scope are common, especially for beginning programmers who tend to rely more on global variables, or variables with more scope than needed. For example, what is the output of the following?

```
#include <iostream>

int main() {
```

(continues on next page)

(continued from previous page)

```
  int i1;
  for (i1 = 0; i1 < 10; i1++) {
    std::cout << i1 << ' ';
  }
 std::cout << '\n';
  int i2;
  for (i2 = 0; i1 < 10; i2++) {
    std::cout << i2 << ' ';
  }
}
```

Show Answer

If you guessed `0 1 2 3 4 5 6 7 8 9` , then you got it!

Can you explain *why*?

Hint: What is the value of `i2` when the end of main is reached?

What should be done to fix this program?

Try This!

Change the preceeding program and put the declarations for `i1` and `i2` in their respective `for` loops. Don't make any other changes.
What happens?
Why is this better than the unmodified program?

```
#include <iostream>

int main() {
  int i1;
  for (i1 = 0; i1 < 10; i1++) {
    std::cout << i1 << ' ';
  }
  std::cout << '\n';
  int i2;
  for (i2 = 0; i1 < 10; i2++) {
    std::cout << i2 << ' ';
  }
}
```

Even a local variable can have too much scope, as in the preceding example.

One of the simplest things you can do as a programmer to improve the clarity and maintainability of your code and reduce errors is to minimize the scope of local variables. The most powerful technique for minimizing the scope of local variables is to declare them where they are first used.

The more distance between the top of a function and the place where a variable is actually used, the more important this advice is. If a variable is initialized far from it's first use, the reader has probably forgotten the initial value, or even if the variable is local or global.

Long methods with dozens of local variable are part of the reason special variable naming schemes became popular in some programming circles in the 70's - 90's. Some are still popular even today (Hungarian notation). Generally, all of

these systems attempt to compensate for confusion created by poor design choices.

Declaring a variable too early creates the problems described earlier, but also creates the problem of causing the scope to extend later than needed. When a variable is declared before the block in which it is used, it also exists after the block scope ends.

This is the problem with our two `for` loops in the previous example.

This is a classic *semantic error*. The code compiles and runs, but does not produce the expected result.

One of the reasons we prefer `for` loops to `while` loops is that for loops allow us to initialize the loop variables within the for block. While loops, in contrast, nearly always define a variable controlling the exit condition outside the scope of the while block.

Prefer initialized local variables to unitialized ones. Uninitialized variables are a common source of error. If you code contains conditional logic or loops and the variable is accessed in its uninitialized state only occasionally, the result can be bugs that are difficult to identify and fix.

More to Explore

- From: cppreference.com: scope and initialization
- CPP Core guidelines:
  - Keep scopes small
  - Don't introduce a variable (or constant) before you need to use it
  - Avoid non-const global variables
  - Keep functions short and simple
  - Express intent

3.3.5 Namespaces

When we introduced functions, we noted that all functions are *by default* global. Another way of saying this is that they are by default in the *global namespace*. The `namespace` keyword provides a mechanism to avoid polluting the global namespace with too many names.

A `namespace` is simply a named block that defines a scope. Namespaces provide a method for preventing name conflicts in large projects.

Symbols declared inside a namespace block are placed in a named scope that prevents them from being mistaken for identically-named symbols in other scopes.

Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

```
// declare some things in the mesa namespace
namespace mesa {
  int i = 0;
  double pi = 3.1416;
  void details (char);
}

void mesa::details (char c) { // define the function declared earlier
```

(continues on next page)

(continued from previous page)

```
  // do something
}

//void mesa::oops () {       // error: oops not yet declared in mesa namespace
//}

namespace mesa {  // a separate mesa namespace block
  void oops ();
  namespace cisc {
    double pi = 3.14159265358979;  // not the same variable as mesa::pi
  }
}

int main () {
  using mesa::cisc::pi;
  pi = 3.f;
  mesa::details('a');
}
```

The larger your project, the more important it is to partition the global namespace. By default, all symbols are declared in the *global namespace* (`::`).

What is the problem with the global namespace?

- There is only 1 of them
- Name conflicts can be common on large projects
- Complicates mixing third party libraries

Well-behaved third party libraries will not put much (if anything) in the global namespace.

You can put anything in a namespace, except `main`. The function main has a few special rules and one is that it must be in the global namespace.

The `using` directive allows all the names in a namespace to be used without the namespace-name as an explicit qualifier. Use a using directive in an implementation file (i.e. `*.cpp`) if you are using several different identifiers in a namespace; if you are just using one or two identifiers, then consider a using declaration to only bring those identifiers into scope and not all the identifiers in the namespace. If a local variable has the same name as a namespace variable, the namespace variable is hidden. It is an error to have a namespace variable with the same name as a global variable.

Prefer using declarations to `using namespace std`

What's wrong with `using namespace std;`?

Nothing, technically. It was a simplification in your first semester classes. The intent was to avoid 'burdening' you with having to care about this technical detail.

BUT

The using-directive `using namespace std;` at any namespace scope introduces *every* name from the namespace std into the global namespace (since the global namespace is the nearest namespace that contains both std and any user-declared namespace), which may lead to undesirable name collisions. This, and other using directives are generally considered bad practice at file scope of a header file. Additionally, shadowing names in the standard namespace can lead to unexpected behaviors.

Example: namespace std

It can be hard to remember every name that might be imported when `using namespace std;`. Even when only 1 header is included.

The following example seems innocent enough, until you learn that showpoint is a name in `std::ios`

Run the following example twice, first as is, then remove the line `bool showpoint = true;`:

```
#include <iostream>

using namespace std;

int main () {
  bool showpoint = true;

  cout << "1.0 with showpoint: " << showpoint << 1.0 << '\n'
       << "1.0 with noshowpoint: " << noshowpoint << 1.0 << '\n';

}
```

Errors using namespace directives are seldom this obviously wrong.

Stack Overflow

Here is a simplification of a real question asked on stack overflow:

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

struct point {
  double x;
  double y;
};
std::ostream& operator<<(std::ostream& os, const point& p) {
    return os << '{' << p.x << ',' << p.y << '}';
}

// calculate the distance between two points
float distance(const point& p1, const point& p2) {
  return sqrt((p1.x - p2.x)*(p1.x - p2.x) +
              (p1.y - p2.y)*(p1.y - p2.y));
}

int main() {
  vector <point> pair {{0,0}, {100,100}};

  cout << "Distances between points:\n";
  for (auto ii = pair.begin(); ii != pair.end(); ii++) {
    for (auto jj = pair.begin(); jj != pair.end(); jj++) {
      cout << * ii << ',' << * jj << ": " << distance(ii,jj) << '\n';
    }
  }
```

(continues on next page)

(continued from previous page)

```
29   return 0;
30 }
```

This code compiles and runs. It also seems to say the distance from `(0,0)` to `(100,100)` is both 1 or -1! I think we can all agree that is not the correct answer for two points `(0,0)` and `(100,100)`.

Try This!

What is wrong with the program?

Can you fix it?

Hint:

This section is about *namespaces*.

One final word from two experts:

> **Summary**
>
> Namespace usings are for your convenience, not for you to inflict on others: Never write a `using` declaration or a `using` directive before an `#include` directive.
>
> Corollary: In header files, don't write namespace-level using directives or using declarations; instead, explicitly namespace-qualify all names. (The second rule follows from the first, because headers can never know what other header `#include` might appear after them.)
>
> **Discussion**
>
> In short: You can and should use namespace using declarations and directives liberally in your implementation files after `#include` directives and feel good about it. Despite repeated assertions to the contrary, namespace `using` declarations and directives are not evil and they do not defeat the purpose of namespaces. Rather, they are what make namespaces usable.
>
> —Herb Sutter and Andrei Alexandrescu, C++ Coding Standards

More to Explore

- From: cppreference.com: namespace declarations and namespace aliases
- From stack overflow: Why is using namespace std considered bad practice?

3.3.6 Keyword: `const`

Use const to instruct the compiler to hold something constant. The `const` keyword can modify the behavior of anything, depending on where it is used:

- fundamental types
- structs or classes
- functions and function parameters
- pointers and references
  - or the values stored in pointers and references

Programs typically use a lot of constants. Mathematical constants like π, or e, and unit conversions are common. Obviously, it would be bad if a program changed the value of π in the middle of its execution. As programmers, we can make a promise to never change those variables that should not change. But the language allows a way to enforce the idea.

A `const` is a named constant. It can't be assigned a new value after initialization. For example:

```
const double pi = 3.14159265359;

double area = pi * r * r;           // OK to read pi like any other variable

pi = 3;                             // this is a compile error
```

Returning to our by_reference function from the previous section, we have seen that pass by reference makes passing data to functions cheap:

```
void by_reference (int& x) {
  std::cout << "in by_ref the address of x is   " << &x << '\n';
  x = -1;
}
```

But one of the side-effects of making passing cheap, is that now the called function might change the value provided. As programmers, we could promise to not change values in functions that shouldn't, but generally, we don't like promises. Programmers prefer contracts. Fortunately, `const` allows us to define just such a contract in the declaration:

```
void by_reference (const int& x) {
  std::cout << "in by_ref the address of x is   " << &x << '\n';
  x = -1;  // this is now a compile error
}
```

This is called **passing by constant reference**, or "const ref", for short. Passing by const ref allows us to enforce our intentions. If a variable is declared as `const`, it is a compile error to change it. Any callers can use this function and rest assured that their data cannot change. The more time you spend programming, the more you will appreciate how powerful that guarantee is.

C99 added the `const` keyword, so now it's in both languages, but you don't see it nearly as often in C. Many programmers use `#define` instead.

Prefer `const` to `#define`

We also prefer inline specifier and enum declaration over define.

There are good reasons to avoid `#define` where alternatives exist.

`#define` is parsed by the *preprocessor*, not the compiler. This mean that effectively, all `#define` directives are literally strings. Fundamentally they are no different from any other pre-processor directive (`#include`, `#ifdef`, etc.), except that people commonly use `#define` as a placeholder for a numeric *type*, or a function.

For example:

```
#define ASPECT_RATIO 1.653
```

is an old fashioned way to define a constant, but you'll likely see it 'in the wild'. The pre-processor literally copies the value '1.653' every place in the source code it encounters the string 'ASPECT_RATIO'. Then the program is compiled.

Prefer this instead:

```
const double ASPECT_RATIO = 1.653;
```

This version preserves the name 'ASPECT_RATIO' which can simplify debugging. It is possible to also preserve macro names using certain debugging compiler switches, such as `-g3`.

Function-like macros using `#define`

If you use `#define` to create a function-like macro, then unexpected behaviors are possible. For example, a macro to call some function `f()` with the larger of either `a` or `b`:

```
#include <iostream>

#define CALL_WITH_MAX(a,b) f((a) > (b) ? (a) : (b))

int f(const int x) {
  return x;
}

int main() {
  int a = -5;
  const int b = 0;

  for (int i = 1; i < 11; ++i) {
    CALL_WITH_MAX(++a, b);                  // call f, but throw away the result
    std::cout << i << ", a: " << a << '\n'; // a is one larger each iteration
  }

  a = -5;
  for (int i = 1; i < 11; ++i) {
    CALL_WITH_MAX(++a, b+10);
    std::cout << i << ", a: " << a << '\n';
  }
}
```

The number of times `a` is incremented depends on **how** `CALL_WITH_MAX` is called. In this case, the value of `b` affects whether `a` is incremented once or twice. **Ouch!**

`#define` may seem like a shortcut. It's not.

Use it when no better alternative exists.

Keyword: `constexpr`

C++11 Feature

The keyword constexpr was added in C++11.

It looks similar to `const`, but it is different.

The `constexpr` specifier declares that it is *possible* to evaluate the expression, object, or function at compile time.

So while this is OK:

```
constexpr double pi = 3.14159265359;

// and so is this
constexpr double pi = acos(-1);

// and so is this
constexpr double area (const double radius) {
   return pi * radius * radius;
}
```

This is not OK in C++11:

```
constexpr double area (const double radius) {
   assert (radius > 0);
   return pi * radius * radius;
}
```

Adding a simple assertion causes this function to no longer compile:

```
g++ -std=c++11 -Wall -Wextra -pedantic     area.cpp   -o area
area.cpp: In function 'constexpr double area(double)':
area.cpp:8:1: error: body of constexpr function 'constexpr double area(double)' not a␣
↪return-statement
  }
  ^
```

On compilers that support C++14, if you compile with `-std=c++11` you may see a warning like:

```
warning: use of this statement in a constexpr function is a C++14 extension [-Wc++14-
↪extensions]
  assert (radius > 0);
```

There are some exceptions, but in C++11, any function more complex than `return (some_expression)` is not able to be evaluated at compile time, therefore, it won't compile as a `constexpr` expression. You should still use it when you can.

C++26 Feature

The rules for `constexpr` changed in every release since C++11 and continue to evolve to include the current standard. In general most changes have simply increased the number of places where `constexpr` can be used, so for most programmers using `constexpr` has gotten easier with each release.

Guidelines for now

- When creating local variables
  - Ask: "Does this variable ever change?"
  - If not, consider `const` or `constexpr`
- Recall `constexpr` is more restrictive
  - Constant expression is evaluated at *compile time*
- When passing parameters to functions

- Consider passing by `const` reference
  - Applies only to object types
    - Pass fundamental types by value if they should not change

Try This!

Given the following:

How many simple changes can you make to the function `area` that are valid if the function signature is

```
const double area (double radius)
```

but invalid if the function signature is unchanged?

More to Explore

- From: cppreference.com: const qualifier and constexpr
- C++ Core Guidelines for constexpr

3.3.7 Keyword: `auto`

C++11 Feature

In C++11, the auto specifier meaning has changed significantly from it's definition prior to C++11. Prior to C++11, `auto` designated variables assigned to the automatic storage class duration. Every variable is assigned a storage duration which limits its lifetime. The automatic duration applied to local variables in a block scope. Because it was the default storage duration for variables in a block, it was rarely, if ever, used. It is still the default storage duration for block variables, but no C++11 keyword is reserved for this behavior.

Since C++11, auto specifier instructs the compiler to automatically deduce the type of a variable based on the initializer used.

```
auto new_variable = initial_value;
```

Since the initializer provides the information used to deduce the *type*, an initial value is required when using `auto`:

```
auto x = 3;   // OK
auto y;       // Not OK

auto int z;   // Error in C++11
              // This is how auto was used prior to C++11
```

You can use `auto` in nearly any statement that includes an initializer:

```
auto x = 3;          // x is an int
auto q = '?';        // q is a char
auto r = &x;         // r is an int
                     //  - a reference to a variable is implicitly dereferenced
```

(continues on next page)

(continued from previous page)

```
auto i = 0, d = 0.0; // Error.
                     // If using auto to deduce multiple types,
                     // then the types must all match
auto i = 0, *p = &i; // OK. Both i and the pointer to i are int
```

This is a far better use for the keyword auto. Later, we will see that some types are long and complicated - having to type them out, as was required prior to C++11, is a chore. `auto` frees us from having to declare a type the compiler already knows about. Additionally, some types, such as those returned by *lambda expressions*, are hidden from us - auto is the only option in these cases.

When variables are initialized using auto, they inherit the *type* from the initializer, but not its *constness*.

```
int          val = 0;
int&         ir  = val;
auto         x   = ir;
```

What type is x?

Show Answer

If you said, `int`, excellent job!

`ir` is a reference to `val`, which makes `ir` just another name for `val`. `auto x = ir;` is exactly the same as if we had written `auto x = val;` here.

Given the following code:

```
const double val = 3.14;
auto         y   = val;
```

What type is y?

Show Answer

If you said, `double`, excellent job!

Just because `val` is `const`, it has nothing to do with whether `y` is `const`.

If we had wanted `y` to be `const`, then we would have needed to say so:

```
const auto y = val;
```

The `auto` keyword is a simple way to declare a variable that has a complicated type. We will get into the details of using auto in this way in later sections. But accept for now that you can use auto to declare a variable where the initialization expression involves iterators, templates, pointers to functions, or pointers to members. You can also use `auto` to declare and initialize a variable to a lambda expression. You can't declare the type of a lambda yourself because the type of a *lambda expression* is known only to the compiler.

`auto` is also commonly used when a type would be redundant, especially in *range-for loops*:

```
std::vector<double> numbers = {1.1, 2.2, 3.3, 5.5, 8.8};
for (const auto& n: numbers) {
```

(continues on next page)

(continued from previous page)

```
  std::cout << n << '\n';
}
```

In this case, the compiler already knows the type stored in the `vector` numbers. It doesn't need to be told again.

When first introduced to `auto`, many programmers balk. It *feels* sloppy and imprecise, and it *seems* as if we are sacrificing clarity. In fact using `auto` is just as strongly typed as a manual declaration and it aids clarity because it forces delaying variable declaration until you have a value to initialize it with.

Guideline

Prefer to declare local variables using `auto x = expr;` when you don't need to explicitly commit to a type. It is efficient by default and guarantees that no implicit conversions or temporary objects will occur.

It is important to note that `auto` may not always deduce the type you expect or the type you need. For example:

```
auto s = "Hello, world!";
```

What type is s?

Show Answer

If you said, `const char*`, excellent job!

If you guessed `string`, you are not alone. This is a common mistake and one that gives new C++ programmers a lot of headaches. String literals in C++ are **always** `const char*`.

If you need a `std::string`, you have to explicitly declare it:

```
std::string s = "Hello, world!";
```

One final note.

The `auto` keyword is a *placeholder* for a type, but it is **not** a type. Therefore, `auto` cannot be used in casts or operators such as `sizeof` and `typeid`.

More to Explore

- From: cppreference.com: The auto specifier and decltype specifier.
- Herb Sutter's GOTW blog
  - #92 (auto part 1)
  - #93 (auto part 2)
  - #94 (almost always auto)

3.3.8 Error handling

Programmers are expected to create programs that function correctly and run as expected. Programs need to function 'correctly' even in the face of unexpected or unusual conditions.

When the unexpected happens, we need to recover as gracefully as possible. Sometimes the best option is to clearly communicate what happened and exit. Abruptly halting or crashing is not generally acceptable.

Error handling involves:

- Detecting an error
- Transmitting information about an error
- Preserving the valid state of a program
- Avoiding resource leaks

The rest of this section describes simple error reporting strategies that are compatible with C.

One tried and true way to communicate errors is by returning error values from functions. We already have been introduced to the assert macro to handle errors in the section *Assertions*.

The assert macro is a 'function-like' macro that evaluates a boolean expression and aborts the program if the condition is `false`. The assert macro is most useful for debugging, but be aware that since it can easily be disabled, it is hard to depend on in production software.

The macro `NDEBUG`, if defined, will disable all the assert functions in a program.

assert

The assert macro takes a single expression. It does not provide a built-in mechanism for a c

Try changing the assertions to generate errors.

Then define `NDEBUG` and see what happens.

```
// uncomment to disable assert()
// #define NDEBUG
#include <cassert>
#include <iostream>

int main()
{
    assert(2 + 2 == 4);
    std::cout << "Checkpoint #1\n";

    assert(2*2*2 == 8);
    std::cout << "Checkpoint #2\n";
}
```

assert messages

The assert macro does not provide a built-in mechanism for a custom user message, but there are a few tricks we can use to create a compound expression.

While it is treated as a single expressin by assert, it provides a way to insert a customer error message when the assertion fails.

Using the comma operator is one technique. The entire expression is still a single boolean exppression.

After the left-hand side expression is evaluates, it is discarded. Any side-effects from the evaluated expression remain. This is what the comma operator does and why it's use is generally discouraged. However, in this case, it's our message and the program does not need it.

Next the right-hand side is evaluated. If the expression evaluates to false, the entire expression, along with the string literal is displayed by assert before the program exits.

```
#include <cassert>
int main()
{
    assert((void("The 'void' here avoids an 'unused value' warning"), 1 == 0));
}
```

Using the relational 'and' operator && is another technique. The entire expression is still a single boolean exppression.

If the left-hand side expression is true, then the right-hand side is evaluated, but since it is a non-zero literal, it will always be `true`.

If the expression evaluates to false, the entire expression, along with the string literal is displayed by assert before the program exits.

```
#include <cassert>
int main()
{
    assert((010 + 010 == 15) && "What is 8+8 again?");
}
```

How does the assert macro support our error handling goals?

- Detecting an error

  Detection is handled in the boolean expression passed to the macro. It can only be a single expression and some caution needs to be taken because "function like" macros are not functions and can behave in unexpected ways.

- Transmitting information about an error

  The assert macro immediately aborts the program and prints the line number of the source were the error occurred.

- Preserving the valid state of a program

  The program immediately terminates. If your program manages external resources like a file, it may be corrupted if the program left it in an indeterminate state on exit.

- Avoiding resource leaks

  This is not applicable since the program terminates. Any resources opened by the program (memory, file handles, etc.) will be recovered by the operating system when the program exits.

Another facility C++ inherits from C is the errno macro. errno is a preprocessor macro used for error indication. The exact definition is implementation defined, but expands to a modifiable `int`. Several standard library functions indicate errors by writing positive integers to `errno`. Typically, the value of errno is set to one of the error codes, listed in the header cerrno as macro constants that begin with the letter **E**, followed by uppercase letters or digits.

errno

The value of errno is `0` at program startup, and although library functions are allowed to write positive integers to errno whether or not an error occurred, library functions never store `0` in errno.

Print an error if we use the log function incorrectly.

```cpp
#include <cerrno>
#include <cmath>
#include <cstring>
#include <iostream>

int main()
{
    const double value = std::log(-1.0);
    std::cout << "Log result: " << value << '\n';

    if (errno == EDOM)
    {
        std::cout << "log(-1) failed: " << std::strerror(errno) << '\n';
    }
}
```

How does errno support our error handling goals?

In all 4 cases, the answer is the same: *it's up to you*.

You need to check `errno` to see if it has been set. It is your resposibility to reset error if needed. No function that sets `errno` will ever reset it to 0. Any messages communicated are yours. No error messages are automatically generated. It is also your responsibility to preserve the state of your program and cleanup resources that may be partially or improperly allocated.

One big advantage of `errno` is that for functions that use it, you get a simple error code you can use to recover from an error without the entire program aborting.

Handling multiple errors at once

Each of the previous error handling techniques are simple, but each allows us to communicate only a single error at a time. Sometimes we need to communicate more information.

We could create a data structure to store each error we care about in a `bool`.

```cpp
struct my_errors {
   constexpr const bool busy = false;
   constexpr const bool cancelled = false;
   constexpr const bool domain_error = false;
   constexpr const bool invalid = false;
};
```

However, this approach does have some limitations. There is no easy way, for example to discover that no errors are set, which hopefully is the normal situation for our program. As programmers, we always want the typical uses or our data structures to be as simple as possible. We want the atypical ones to be simple too!

Can we make this easier to work with? Yes.

Once way is to pack all the boolean values into a single variable.

There are several ways to accomplish this. Here we discuss two of them. Both of them use a single bit to represent the true or false state. Starting from the previous code, we change it like this:

```cpp
constexpr const unsigned error_none = 0;
constexpr const unsigned error_busy = 1;
constexpr const unsigned error_cancelled = 2;
```

(continues on next page)

(continued from previous page)

```
constexpr const unsigned error_domain = 4;
constexpr const unsigned error_invalid = 8;
```

The values assigned to each of these variables is not coincidence. Each (other than 0) represents an increase in the power of two: $2^0, 2^1, 2^2, 2^3$. Each of these numbers sets exactly 1 bit in an `unsigned int` and no others. So now we can use these values to set the bits in the variable we want to use to keep track of errors.

bitmask

We use unsigned integers and bitwise operators to 'flag' each error. This technique is called a bitmask and it has a long history in programming.

Note

Shifting bits into the sign bit of a signed integer type is implementation defined in C++. To avoid surprising behavior, it is a best practice to only use unsigned integer types when manipulating bits.

Set and print some bits.

We can print the value stored in `errors`, but we don't really care about the numeric value, we care about the individual bits in the number.

The maybe_unused attribute suppresses warnings about unused variables.

```
#include <iostream>

int main()
{
   using std::cout;
   [[maybe_unused]] constexpr const unsigned error_none = 0;
   [[maybe_unused]] constexpr const unsigned error_busy = 1;
   [[maybe_unused]] constexpr const unsigned error_cancelled = 2;
   [[maybe_unused]] constexpr const unsigned error_domain = 4;
   [[maybe_unused]] constexpr const unsigned error_invalid = 8;

   unsigned errors = error_none;

   // Use bitwise OR to set a bit in the variable
   errors = errors | error_busy;
   errors = errors | error_invalid;

   // Use bitwise XOR to unset a bit
   errors = errors ^ error_busy;

   // show error value and each bit
   cout << "errors: " << errors << ", bits: (";
   for (auto err_num = error_invalid; err_num > error_none; err_num /= 2) {
       // Use bitwise AND to test if a bit has been set
       if (err_num & errors) {
           cout << 1;
       } else {
           cout << 0;
```

(continues on next page)

(continued from previous page)

```
        }
    }
    cout << ")\n";
}
```

The approach using bitwise operations is simple once you know the tricks, but C++ provides a type that provides the ability to perform the same operations: std::bitset.

bitset

We use the same bitwise operators for bitset that we used with unsigned integers. But bitsets provide some additional features that can make them easier to work with.

Set and print some bits in a bitset.

The maybe_unused attribute suppresses warnings about unused variables.

```
#include <iostream>
#include <bitset>

int main()
{
    using std::cout;
    [[maybe_unused]] constexpr const std::bitset<8> error_none = 0;
    [[maybe_unused]] constexpr const std::bitset<8> error_busy = 1;
    [[maybe_unused]] constexpr const std::bitset<8> error_cancelled = 2;
    [[maybe_unused]] constexpr const std::bitset<8> error_domain = 4;
    // we can also initialize using binary if desired
    [[maybe_unused]] constexpr const std::bitset<8> error_invalid   = {0b0000'1000};
    [[maybe_unused]] constexpr const std::bitset<8> error_length    = {0b0001'0000};
    [[maybe_unused]] constexpr const std::bitset<8> error_underflow = {0b0010'0000};
    [[maybe_unused]] constexpr const std::bitset<8> error_overflow  = {0b0100'0000};
    [[maybe_unused]] constexpr const std::bitset<8> error_range     = {0b1000'0000};

    std::bitset<8> errors = error_none;

    // Use bitwise OR to set a bit in the variable
    errors = errors | error_busy;
    errors = errors | error_invalid;

    // Use bitwise XOR to unset a bit
    errors = errors ^ error_busy;

    std::cout << "errors: " << errors << '\n';
    if ((errors & error_busy).any() != 0) {
        cout << "error busy is on\n";
    } else {
        cout << "error busy is off\n";
    }
    if (errors.test(4) == 1) {
        cout << "\nbit 4 is true" << '\n';
    }

}
```

More to Explore

- Wikibooks: C Error Handling
- On cpp reference.com:
  - The assert macro
  - The errno macro
  - Keyword static_assert
- bitmask
- Learn C++ article O.3 - Bit manipulation with bitwise operators and bit masks

3.3.9 Exception handling

An alternative to returning values from functions is to use **exceptions**.

A C++ exception is a response to an exceptional circumstance that occurs while a program is running, such as an attempt to open a file that does not exist.

Exceptions provide a way to transfer control from one part of a program (where the error occurs) to another (where the error is 'handled'). C++ exception handling is built upon three keywords: `try`, `catch`, and `throw`.

throw
: A program throws an exception when a problem shows up.

try
: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more `catch` blocks.

catch
: A program *handles* an exception with one or more catch blocks.

Basic anatomy of an exception

1. Surround potentially error throwing code in a `try` block
2. After the `try` block, include 1 or more `catch` blocks
   a. The parameter in the catch identifies the type of exception the catch block will handle

```
try {
  // execute potentially dangerous statements
}

// catch a specific class of exceptions
catch (const std::exception& e) {
  std::cout << "Exception occurred.\n";
  std::cout << "Details: " << e.what() << std::endl;
}
```

If you specify a `try`, you must include at least 1 `catch`.

Passing catch parameters by *const reference* is considered a best practice

- std::exception.what()

  - Returns a description of what caused the error
- More than one `catch` block is acceptable
- The special catch

```
catch (...)
```

will catch any exception. If you use this, then you should be prepared to really handle *anything*.

Standard exceptions

The standard exceptions in C++ are organized in a class hierarchy.

- std::exception is the base class for all exceptions
- Classes derived from `std::exception`
  - bad_alloc: thrown by `new` and other memory allocation errors
  - bad_cast: thrown by `dynamic_cast` and similar
  - bad_typeid: thrown by `typeid`
  - bad_exception: runtime unexpected or pointer exceptions
  - logic_error: exceptions that *should* be detected by reading the code
  - runtime_error: exceptions that theoretically can't be detected by reading the code
  - `logic_error`, and `runtime_error` are also exception bases
- Classes derived from logic_error
  - domain_error: invalid mathematical domain
  - invalid_argument: bad parameters or arguments used
  - length_error: Thrown when a std::string is too large
  - out_of_range: Used for range checked access, vector.at(x)
- Classes derived from runtime_error
  - overflow_error: mathematical overflow
  - range_error: Thrown when storing an out of range value
  - underflow_error: mathematical underflow

This list is just a partial set of the exceptions in the standard library. It includes the exceptions that have been in the language since before C++11. Check your compiler manual or the C++ reference for the latest exceptions.

Using exceptions

C++ exceptions are designed to support *error handling*.

Use `throw` only to signal an error. Use `catch` only to specify error handling actions when you know you can handle it. Possibly by translating it to another type and re-throwing an exception of that type. For example, catching a `bad_alloc` and re-throwing a `no_space_for_file_buffers` exception.

Do not use `throw` to catch a coding error in usage of a function. Instead, use assert or other mechanism to either stop the program or log the error.

Do not use `throw` if you discover an unexpected violation of an invariant of your component. Instead, use `assert` or other mechanism to terminate the program. Throwing an exception will not cure memory corruption and may lead to further corruption of important user data.

Use `try` and `catch` blocks if the logic is more clear than checking a condition and returning a value. For example, if you need to propagate errors several levels up the stack:

```
void f1() {
  try {
    f2();
  } catch (const some_exception& e) {
    // ... handle error
  }
}
void f2() { ...; f3(); ...; }
void f3() { ...; f4(); ...; }
void f4() { ...; f5(); ...; }
void f5()
{
  if ( /*...some error condition...*/ )
    throw some_exception();
}
```

Only the code that detects the error, `f5()`, and the code that handles the error, `f1()`, have any clutter. None of the other functions have to worry about passing error codes either in return values or in extra parameters that would have to be mutable.

Do not use `try` blocks to reclaim resources. This is a Java technique, which is great for Java, but is not needed in C++. In C++, use Resource Acquisition Is Initialization (*RAII*).

Use constructors to allocate resources and use destructors to clean up resources,

Do not use `try` blocks as a proxy for error return codes. This results in too many `try` blocks cluttering up functions, which harms readability if nothing else.

Exceptions and I/O streams

I/O streams can be configured to throw exceptions with `std::basic_ios::exceptions`. This object gets and sets the exception mask of the stream. The exception mask set in the program determines which error states in the stream will throw an exception if an error is encountered. If no exception bits are set, then the I/O streams in C++ will not throw any exceptions.

For example:

```
std::ifstream ifs("in.txt");
ifs.exceptions(std::ifstream::failbit);
```

At this point, only the `failbit` will trigger an exception.

C++11 Feature

I/O Streams may throw `ios_base::failure` but since C++11 this class inheritance changed.

`ios_base::failure` inherits from `std::system_error`

The end result is that `ios_base::failure` now has an `error_code` member to the exception object it didn't used to have.

```
catch (const ios_base::failure& e) {
  std::cout << "I/O exception occurred.\n";
  std::cout << "Details: " << e.what() << std::endl;
  std::cout << "Code: " << e.code() << std::endl;
}
```

More to Explore

- CPP Core Guidelines: Error Handling
- ISO C++ FAQ Exceptions
- Overview of the error handling library and exceptions
- try, catch, and throw
- Post from Eric Lippert on vexing exceptions

3.3.10 Static functions and variables

By default, all functions are global; they can be used in any file of your program whether or not a declaration appears in a header file. To limit access to the current file, declare a function or variable `static`, like this:

```
// assume all these definitions are in a single file "foo.cpp"

// static variable used by non-static functions
static bool verbose = false;

bool is_verbose() {
  return verbose;
}

// vprint could reside in another file
void vprint (std::string message) {
  if (is_verbose()) {
    std::cout << message << '\n';
  }
}
// this function only works if it is the same file
// as the one where verbose is defined
void verbose_print (std::string message) {
  if (verbose) {
    std::cout << message << '\n';
  }
}

// local static function can only be called in this compilation unit
static void helloHelper(void) {
  puts("hi!");
}

// anyone can call `hello`
void hello(int repetitions) {
  for(int i = 0; i < repetitions; ++i) {
    helloHelper();
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Similar to file static functions and variables, the keyword `static` can also be used inside functions. Static variables are initialized only the first time the function is called, for example:

```
size_t counter() {
  static size_t count = 0;
  return ++count;
}
```

The first time `counter` is called, the variable `count` is initialized to zero. Each call thereafter, `count` is increased by 1 and the new value is returned.

Another appropriate use of static variable in functions: when defining a constant that should only be initialized once. For example, our earlier `too_small` function, could be:

```
#include "area.h"

bool too_small (int x, int y) {
  static const int min_size = 10;
  return area(x, y) < min_size;
}
```

Under very rare circumstances, it may be useful to have a variable local to a function that persists from one function call to the next. You can do so by declaring the variable static. For example, here is a function that counts how many times it has been called:

```
// return the number of times the function has been called
int counter(void) {
  static count = 0;
  return ++count;
}
```

Static local variables are stored in the same memory space as global variables. But they are only visible inside the function that declares them. This makes them slightly less troublesome than global variables; there is no fear that some unrelated code elsewhere will quietly change their value. Static variables are rarely used in practice, however, because they do not work well in multi-threaded applications.

Anonymous namespaces

It is possible to define a namespace without a name. Unnamed namespace members have potential scope from their point of declaration to the **end of the translation unit**.

In other words, they behave a bit like global variables, visible to all functions, but *only* within the source file where the namespace is defined.

Unnamed (or anonymous) namespaces are considered a 'modern' C++ alternative to declaring variables as `static` within a translation unit.

```
static int i;

// anonymous namespace
namespace {
```

(continues on next page)

(continued from previous page)

```
  int i;
}
```

At one point the C++ standards committee planned to deprecate the use of `static` in this way and force the use of namespaces, but that decision was reversed in 2009.

Unnamed namespaces are preferred over the use of the keyword static for several reasons:

- The `static` keyword can have different meanings depending on context

  Namespaces only have one purpose: to define and enclose a scope.

  Namespaces provide a uniform and consistent way of controlling visibility. You don't have to use different tools for the same thing.

- A namespace can encapsulate anything. Only functions and objects can be declared static.

  User defined types, which is the focus of the second half of this course cannot be declared static.

  ```
  // valid statements
  static void my_function() { /* function body */ }
  static int  my_variable;

  // invalid
  static class sample_class { /* class body */ };
  static struct sample_struct { /* struct body */ };

  // valid
  namespace
  {
    class sample_class { /* class body */ };
    struct sample_struct { /* struct body */ };
  }
  ```

- When using an anonymous namespace, the function/object name is mangled properly, which allows you to see something like "(unique namespace)::xyz" in the symbol table after de-mangling, and not just "xyz" with static linkage.

Example

At compile time, This definition is treated as a definition of a namespace with a unique name and a using-directive in the current scope.

```
namespace {
  int i;       // defines ::(unique)::i
}
```

The unique name of the namespace is hidden. Since it is not known, no code outside the current translation unit can access it.

It is technically possible to have more than one unnamed namespace in the same translation unit.

```
namespace {
  int i;       // defines ::(unique)::i
}
```

(continues on next page)

(continued from previous page)

```
namespace A {
    namespace {
        // reusing the name 'i' in this scope is a bad idea . . .
        int i; // A::(unique)::i
        int j; // A::(unique)::j
    }
    void g() { i++; } // A::unique::i++
}
```

As a best practice, you should keep unnamed namespaces to a minumum and declare them near the top of your translation unit so that they stand out.

Run It

```
#include <iostream>

namespace {
  int i = 0;       // defines ::(unique)::i
}

void f() {
  ++i;         // increments ::(unique)::i
}

namespace A {
  namespace {
    // reusing the name 'i' in this scope is a bad idea . . .
    int i = 3;          // A::(unique)::i
    int j = 5;          // A::(unique)::j
  }
  void g() { ++i; } // A::unique::i++
}

using namespace A;  // introduces all names from A into global namespace

void h() {
  // error: ::(unique)::i and ::A::(unique)::i are both in scope
  // i++;

  A::i++; // ok, increments ::A::(unique)::i
  j++;    // ok, increments ::A::(unique)::j
  f();
}

int main() {
}
```

More to Explore

- From: cppreference.com:

- Function declarations
- Storage class specifiers
- Namespace declarations and Namespace aliases

3.4 Function overloads and templates

This section focuses on C++ extensions to simple functions, especially overloaded functions and function templates.

3.4.1 Function overloads

Unlike C, in C++, two different functions can have the same name as long as their parameter lists are different. When two functions have the same name, but different parameters, the functions are said to be **overloaded**.

In order to count as a valid overload, either the number of parameters must be different, or the parameter types must be different, or a combination of both. For example:

```
#include <iostream>
using std::cout;

// add two ints
constexpr
int add (int a, int b) {
  return a+b;
}
// add two doubles
constexpr
double add (double a, double b) {
  return a+b;
}

int main () {
  int x=5, y=2;
  double p=3.14, e=2.718;
  cout << add (x,y) << '\n';
  cout << add (p,e) << '\n';

  // error: call to overloaded function add (double, int) ambiguous
  // cout << add (31.4, 10) << '\n';

  // explicit conversion is OK
  cout << add (31.4, double(10)) << '\n';
}
```

Function overloads are a huge advantage over C where (nearly) every function is global and every function name must be unique. For example:

- C defines 7 different functions just for absolute value
  - `abs`, `llabs`, `fabs`, `fabsf`, etc. see abs
- and 13 different functions for different types of division operations

This just adds to the amount of stuff programmers have to commit to memory. In C++, you only have to remember a single function to compute the absolute value: `abs`.

Video

Video URL: https://www.youtube.com/watch?v=IAMzWp3kS_k

Another example: a family of functions to compute volume.

```cpp
#include <cmath>
#include <iostream>
#include <numbers>

// volume of a cube
constexpr
double volume (const double a) {
  return a * a * a;
}

// volume of a cylinder
constexpr
double volume (const double r, const double h) {
  // starting with C++20, numeric constants are preferred over
  // macros like `M_PI`
  return std::numbers::pi * r * r * h;
}

// volume of a cuboid
constexpr
double volume (const double a, const double b, const double c) {
  return a * b * c;
}

int main() {
  std::cout << "volume of a 2 x 2 x 2 cube: "
            << volume(2) << '\n'

            << "volume of a cylinder, radius 2, height 3: "
            << volume(2, 3) << '\n'

            << "volume of a 2 x 3 x 4 cuboid: "
            << volume(2, 3, 4) << '\n';
  return 0;
}
```

Note

The return type is **not** part of the overload.

Two functions in the same namespace that differ only in return type will not compile.

Overloading anti-patterns

How many parameters are too many?

This is an often asked question, with no clear cut answer. It is primarily a question of *clarity* and *design*.

For example, given:

```
int operate (float a, int b, long c, double d);
```

In this case, the parameters and function name provide no guidance on how to call this function. So four is probably too many parameters, simply because future usage errors are likely.

Keep in mind that more parameters equal more complexity. Limit the number of parameters you need in a given method, or use a `struct` to combine parameters. Also, be wary of overloads with the same number of parameters and different types. For example:

```
int operate (double a, int b);
int operate (int a, double b);
```

Depending on what `operate` does with it's parameters, reversing the order of the parameters could have drastic consequences. We just don't know without looking at the source code. In this case even two parameters is too many. It is almost certain someone will invoke the wrong version occasionally.

More to Explore

- From: cppreference.com: overload resolution

3.4.2 Operator overloads

Very few languages allow overloading basic operators. In this section, we won't discuss all possible overloading, but we are introducing some of the more common overloads that generally are implemented as ordinary 'free' functions.

In C++, operators are overloaded in the form of functions with special names. For example, `a+b` and `operator+(a,b)` both call the same function.

Most C++ operators can be overloaded. You cannot change the meaning of operators for built-in types in C++. Operators can only be overloaded when at least 1 operand is a user-defined type. Other rules of overloads still apply: overloads for a specific function signature can only be used once.

Some of the most commonly overloaded operators are << and the relational operators: ==, `!=`, <, <=, >, and >=.

It doesn't always make sense to overload all of the relational operators. For example, a complex number does not have a *natural order*, so you may only want to overload == and `!=` for a complex number.

If you overload == you should always overload `!=`. If you overload == and <, then you should overload all 6 relational operators.

Relational operators

Standard algorithms such as std::sort and containers such as set expect `operator <` to be defined, by default, for the user-provided types, and expect it to implement strict weak ordering. Strict weak ordering defines members of a set as *comparable* to each other. The general signature for these non-member functions is:

```
// In this example, T is a placeholder for your type.
// Note that this is not a function template.
inline bool operator<(const T& lhs, const T& rhs)
{
   // compare the data in left-hand side and right-hand side objects
   // for less than
}

inline bool operator==(const T& lhs, const T& rhs)
{
   // compare the data in left-hand side and right-hand side objects
   // for equality
}
```

An idiomatic way to implement strict weak ordering for a structure is to use lexicographical comparison provided by std::tie:

```
struct Record
{
    std::string name;
    unsigned int floor;
    double weight;
};

inline bool operator<(const Record& lhs, const Record& rhs)
{
   // parameters passed to each tie must be in the same order
   // or this will always return false
   return std::tie(lhs.name, lhs.floor, lhs.weight)
        < std::tie(rhs.name, rhs.floor, rhs.weight);
}
```

If some of the data required for the comparison is private and has no function to access the data members, then you may need to make your relational operators friends.

Once you have defined `operator<` and `operator==`, there is no need to rewrite the comparison logic again. It is much better to implement the remaining comparison functions in terms of < and ==.

```
// note the operands swapped inside the function body
inline bool operator> (const T& lhs, const T& rhs){ return   rhs < lhs; }

inline bool operator<=(const T& lhs, const T& rhs){ return !(lhs > rhs); }
inline bool operator>=(const T& lhs, const T& rhs){ return !(lhs < rhs); }

inline bool operator!=(const T& lhs, const T& rhs){ return !(lhs == rhs); }
```

Note

It is a common programming anti-pattern to reimplement all the logic for each relational overload.

This is a common source of error and can lead to bugs that are very difficult to track down.

Insertion and extraction overloads

The bitshift operators << and >>, although still used in hardware interfacing for the bit-manipulation functions they inherit from C, have become more prevalent as formatted stream operators in C++.

The overloads of `operator >>` and `operator <<` that take a std::istream reference or std::ostream reference as the left hand argument are known as insertion and extraction operators.

The canonical forms are:

```
std::ostream& operator<<(std::ostream& os, const T& rhs)
{
  // write rhs to stream
  return os;
}

std::istream& operator>>(std::istream& is, T& rhs)
{
  // read rhs from stream
  if( /* could not construct T from stream */ ) {
    is.setstate(std::ios::failbit);
  }
  return is;
}
```

When we use classes such as cout, the << operator looks at the type on the right-hand side to determine which overload to call.

These two lines of code call the exact same function:

```
std::cout << "howdy!";

operator<< (std::cout, "howdy!");
```

More to Explore

- Operator overloading in C++ from stackoverflow. Much of the content in this section was taken from there.
- Comparison operators from cppreference.com.

3.4.3 Function templates

Function overloads allow programmers to reuse function names. Functions with the same name may or may not accomplish the same task.

What if our functions actually are intended to do exactly the same thing, but merely on different types?

Function overloads allow us to write functions with the same names and different parameter lists, but each function still requires its own function body, even if it's only to call another function.

Example

These two overloads each add their parameters, only on different types:

```
int    sum (int a, int b);

double sum (double a, double b);
```

Run It

These two overloads each add their parameters, only on different types:

```
#include <iostream>

constexpr
int sum (int a, int b) {
  return a+b;
}

constexpr
double sum (double a, double b) {
  return a+b;
}

int main () {
  std::cout << sum (10,20) << '\n';
  std::cout << sum (1.0,1.5) << '\n';
}
```

C++ provides a way to write a single piece of code that can stand in for an entire *class of functions* that all do exactly the same thing.

In C++, we can define a **template** for a function. The template defines a *function generating recipe* using a *generic type* as a placeholder. Templates are created using the `template` keyword, followed by zero or more template parameters in angle brackets <>. For example:

```
template <template-parameters> function-declaration
```

Note

Template parameters are optional.

It looks strange to have a template with no template parameters, but it is perfectly legal. For example, as we will explore as part of hash tables, the overloads of `std::hash<>` normally do not include template parameters:

```
struct point {
  int x;
  int y;
}

namespace std {
  template <>
  struct hash<point>
  {
      // hash function implementation for a point
  };
}
```

Example

Using templates, our previous sum functions collapse down to:

```
template <class T>
T sum (T a, T b) {
  return a+b;
}
```

Run It

```
#include <iostream>

template <class T>
constexpr
T sum (T a, T b) {
  return a+b;
}

int main () {
  std::cout << sum<int> (10,20) << '\n';
  std::cout << sum<double> (1.0,1.5) << '\n';
}
```

When identifying template parameter types, it is common to see either `typename` or `class`. As we will see later, a `class` defines a type, so for the purposes of a template, they are the same. Whether you use 'typename' or 'class' is a matter of preference.

The identifier `T` is traditional, but any valid variable name could be used. In introductory template tutorials `AnyType` is not uncommon.

Templates are normally completely specified in header files. Because templates are not either declarations or definitions, it is an error to write a template in a cpp source file and then try to use it in another source file.

Using templated functions

In short, using functions generated by templates is not very different from a non-templated function.

You can explicitly provide the type:

```
std::cout << sum<int> (10, 20) << '\n';
std::cout << sum<double> (1.0, 1.5) << '\n';
```

Or let the compiler deduce the type:

```
std::cout << sum (10, 20) << '\n';
std::cout << sum (1.0, 1.5) << '\n';
```

Given a template of a single generic type, take care when mixing types when two or more parameters are involved:

```
#include <iostream>

template <class T>
```

(continues on next page)

(continued from previous page)

```
4  T sum (const T a, const T b) {
5    return a+b;
6  }
7
8  int main () {
9    std::cout << sum <double> (10,1.5) << '\n';  // OK
10   std::cout << sum <int>    (10,1.5) << '\n';  // Compiles with warning
11   std::cout << sum          (10,1.5) << '\n';  // Compile error
12
13   return 0;
14 }
```

The call to `sum <double>` implicitly converts the `int` to `double` without warning.

The warning for `sum <int>` happens because we have explicitly declared the function to take type `int`, but the second argument is a double. The warning says that the copy of `1.5` passed to sum will be truncated to `1`, which is a narrowing conversion.

The error is due to the compiler not be able to find a function overload that meets the calling requirements. Even though sum is a template, the compiler will say:

```
no matching function for call to 'sum'

note: candidate template ignored: deduced conflicting types
for parameter 'T' ('int' vs. 'double')
```

In the third call to sum, we asked the compiler to deduce the types. Since the template defines a function with a single type for both arguments and the return value, it doesn't know which to choose. Both `int` and `double` are equally valid choices.

The examples on lines 9 and 10 are valid because the compiler does not need to deduce the type, it was explicitly told the type of the function to generate.

Multiple template parameters

A `sum` function that only adds numbers of the same type is not particularly useful. Templates also allow defining multiple types to be used in a template with each parameter potentially having a different type.

```
#include <iostream>

template <typename T1, typename T2>
bool are_equal (const T1& a, const T2& b) {
  return (a==b);
}

int main () {
  if (are_equal(10, 10.0)) {
    std::cout << "x and y are equal\n";
  } else {
    std::cout << "x and y are not equal\n";
  }
}
```

There is no 'rule' that says each template parameter can be used only once in the function declaration.

```
template <typename T1, typename T2>
  T2 foo (const T1& x, const T2& y) {
    T1 tmp_x = x;
    T2 tmp_y = y < 1? 1: y*y;
    while (tmp_y < tmp_x) {
      ++tmp_x;
      tmp_y *= 2;
    }
    T2 result = tmp_y;
    return result;
  }
```

Non-generic template parameters

Not every template parameter has to be a class or a typename. Any specific type is a valid template parameter.

The following example defines a template that defines a function that multiplies a value of type `T` by a provided `int N`. The template parameter `int N` can be used in the function body just like any other local variable or function parameter.

Non-generic template parameters may be specified as `const` if the function body will not modify them.

Example

```
template <class T, int N>
T multiply (const T& val) {
  return val * N;
}
```

The `multiply` template **must** be called using template parameters

```
const double two_pi = multiply<double,2>(3.14159);
```

Run It

```
#include <iostream>

// it is possible to forward declare a template
template <class T, const int N>
constexpr
T multiply (const T& val);

int main() {
  std::cout << multiply<double,2>(3.14159) << '\n';
  std::cout << multiply<long,3>(10) << '\n';
}

// note the definition includes ALL of the declaration
// including the template information
template <class T, int N>
T multiply (const T& val) {
  return val * N;
}
```

This is exactly what the std::array class template does. As a wrapper around a raw array, when this container is used, the size of the array is a required template parameter:

```
std::array<int,4> numbers {2, 4, 6, 8};
```

Note

Templates that include non-generic template parameters can't use auto type deduction. For example, `std::array` needs both the type and the array size. Our `multiply` example needs both the type and the operand `N`. While it might be possible to deduce the type based on the argument provided, there is no way for the compiler to 'deduce' the second operand `N`.

```
template <class T, int N>
T multiply (const T& val) {
  return val * N;
}

int main() {
  multiply(3);     // compile error: multiply times what?
}
```

More to Explore

- From cppreference.com:
  - Template parameters
  - Implicit type conversion and arithmetic conversions
  - The utlity function std::hash

3.4.4 Concepts overview

One of the pitfalls when using templates, especially templates in unfamiliar libraries, is that C++ templates can be *too generic*. When you declare a template with either a `class` or `typename`, literally **any** `class` or `typename` could be passed in.

What if your template assumes one of the provided types has a `push_back` function? C++ has a mechanism to resolve this ambiguity and enforce additional requirements on template arguments used as parameters.

- Concepts
- The `requires` keyword

Concepts

Without resorting to the experimental technical specification you can still get some of the readability improvements from concepts, just by defining an alias for the type your template expects:

```
#include <iostream>
// A 'concept' for a numeric type
#define NumericType typename
```

(continues on next page)

(continued from previous page)

```
// Same template as before
// 'typename' replaced with 'NumericType'
template <NumericType T, int N> T multiply (T val) {
  return val * N;
}
```

We haven't actually made any functional change here. We have simply made a change that allows our source code to indicate our *intent*.

Until the Concepts/Constraints Technical Specification is implemented, expressing our intent is about all we can do.

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

#define InputIterator typename

namespace mesa {
  struct point {
    int x = 0;
    int y = 0;
  };

  // T must overload operator >>
  template <InputIterator T>
    T get(std::string prompt = "Enter a single value: ") {
      while(true) {
        std::cout << prompt;
        std::string line;
        std::getline(std::cin, line);

        // If we can't stream into our type T
        // then the input was not valid for that type.
        std::istringstream buf(line);
        T result;
        if(buf >> result) {
          // check for any extra input and reject input if found
          char junk;
          if(buf >> junk) {
            std::cerr << "Unexpected character.\n";
          } else {
            return result;
          }
        } else {
          std::cerr << "Not a valid input.\n";
        }
      }
    }
}

int main() {
  auto a = mesa::get<int>();
```

(continues on next page)

(continued from previous page)

```
  auto b = mesa::get<int>("Enter an integer: ");
  auto c = mesa::get<float>("Enter a float: ");

  std::cout << "Values: " << a << ", "
                          << b << ", "
                          << c << '\n';

  // auto p = mesa::get<mesa::point>();  // compile error!
  return 0;
}
```

Attempting to 'get' a `mesa::point` is a compile error because our point object does not have a definition for the `operator>>` function overload. The compiler first displays the error, which may look something like this:

```
concept.cpp:25:16: error: invalid operands to binary expression ('std::istringstream'␣
↪(aka 'basic_istringstream<char>') and 'mesa::point')
        if(buf >> result)
concept.cpp:41:18: note: in instantiation of function template specialization 'mesa::get
↪<mesa::point>' requested here
  auto p = mesa::get<mesa::point>();
```

The compiler will then display an exhaustive list of every type it tried:

```
/usr/include/c++/v1/cstddef:135:3: note: candidate function template not viable: no␣
↪known conversion from 'std::istringstream' (aka 'basic_istringstream<char>') to 'byte'␣
↪for 1st argument
operator>> (byte  __lhs, _Integer __shift) noexcept
^
/usr/include/c++/v1/istream:625:1: note: candidate function template not viable: no␣
↪known conversion from 'mesa::point' to 'unsigned char *' for 2nd argument
operator>>(basic_istream<char, _Traits>& __is, unsigned char* __s)
^
/usr/include/c++/v1/istream:633:1: note: candidate function template not viable: no␣
↪known conversion from 'mesa::point' to 'signed char *' for 2nd argument
operator>>(basic_istream<char, _Traits>& __is, signed char* __s)
^

// many others omitted
```

The more code you have written and the more code pulled in from `#include` directives, the longer the list will be. It can run on for thousand of lines. Clearly we'd like something better, but the C++ standard doesn't offer a solution until C++20.

Keyword: `requires`

A *requires clause* is an additional constraint on template arguments or a function. It is planned for release in C++20.

You will sometimes encounter *named requirements* in C++ code.

The named requirements listed are the named requirements used in the C++ standard to define the expectations of the standard library.

Some of these requirements are being formalized in C++20 using the concepts language feature. Until then, the burden is on the programmer to ensure that library templates are instantiated with template arguments that satisfy these requirements. Failure to do so may result in very complex compiler errors and warnings.

Even though they do not enforce any specific compiler rule or constraint (yet), they can improve the intent of expected template types.

More to Explore

- a bit of background for concepts and C++17 Bjarne Stroustrup
- Concepts C++ from Wikipedia
- From cppreference.com
  - Concepts Library
  - Named requirements

3.4.5 Trailing return types

Problem:

- You created a template, but the return type needs to be a type other than one of the template parameters.

You'd like to be able to use auto to simply:

```
template<class T, class U>
auto mystery_function(T t, U u);
```

Starting in C++17, this syntax works much more often than in previous version of the standard, because the rules for deducing types have expanded. Earlier version of C++ need to resort to a **trailing return type**.

Even in C++17, depending on what a function does, return type deduction may not always work. If it is possible for our function to return different types, then a simple auto return will not compile:

```
auto f(bool val)
{
    if (val) return 123; // deduces return type int
    else return 3.14f;   // error: deduces return type float
}
```

You can use `auto`, together with the `decltype` type specifier, to delay the evaluation of a function return value until after the function parameters have been declared.

Use `auto` and `decltype` to declare a function whose return type depends on the types of its arguments.

To understand what's going on, first we have to understand `decltype`.

Keyword: decltype

Added in C++11, the `decltype` type specifier yields the **type** of a specified expression, object, or literal value. We use decltype when we want to define a variable based on the result of an expression, but we don't want to use the expression to initialize the variables value. For example:

```
int i = 42;
decltype(i) j = i * 2.0;
```

Similarly, there is a symmetry between the auto specifier and decltype:

```
auto a = 3u;       // a is unsigned;
decltype(a) b = a; // same as auto b = a;  b is also unsigned
```

Trailing return type syntax

Since the auto specifier and decltype are complimentary operators, they work well together to help write generic functions that avoid committing to a specific type.

To declare a trailing return type for a function, use this general form:

```
auto function_name () -> return_type
{
  // function body
}
```

The -> is required to inform the compiler that a trailing return type follows.

Note that the return type is inserted after function parameters and before the function body.

```
auto f(const bool val) -> float
{
    if (val) return 123; // return widens int to float
    else return 3.14f;   // return type float
}
```

```
template<typename T, typename U>
auto add(const& T rhs, const& U lhs) -> decltype(rhs + lhs)
{
    return rhs+lhs;
}
```

Calling this add function like so:

```
auto val = numeric_limits<unsigned short>::max(); // typically 65,536
auto sum = add(val, val);
```

Even though a variable of type `unsigned short` was used in both parameters, the return type can't be unsigned short, because the returned value is too large to fit. If we had committed to a type, or used one of the generic types provided in the template, our result would overflow. Instead, the compiler used `decltype` to determine in this case, the return type should be `int`.

Do trailing return types seem like a lot of trouble? Prior to C++11, when trailing return type syntax was introduced, you could use decltype and declval instead:

```
template<typename T, typename U>
  decltype(std::declval<T>() + std::declval<U>())
  add(const& T lhs, const& U rhs) {
    return lhs + rhs;
  }
```

This get unreadable fairly quickly. For this reason, trailing return types are preferred.

More to Explore

- *Keyword: auto*
- From: cppreference.com: The auto specifier and decltype specifier.

3.4.6 General function writing guidelines

1. Write for clarity and correctness **first**
2. Avoid *premature optimization*
3. Avoid *premature "pessimization"* That is, prefer faster when **equally** clear
4. Minimize side-effects

   A function that modifies its parameters is said to have *side-effects*. Programs with too many side-effects are hard to predict and debug.

   Returning to our call-stack example. What if the function signatures were changed to accept a pass-by-reference parameter?

Side effects

```
void dig(double& x);
void deeper(double& x);
```

Given that the names of these functions provide no insight to their purpose, there is no way to know without inspecting the source if the variable x is modified when passed to these functions.

```
void dig(double& val) {
  std::cout << "Digging...\n";
  val /= 2;
  deeper(val);
  std::cout << "Done digging...\n";
}
```

Unless it's obvious from the name of the function, side effects are almost always unexpected.

Run It

```
#include <iostream>

void dig(double& val);
void deeper(double& val);

int main() {
  double pi = 3.14159;
  std::cout << "in main.\npi = " << pi << '\n';
  dig(pi);

  std::cout << "Returned to main.\npi = " << pi << '\n';
  return 0;
}

```

(continues on next page)

(continued from previous page)

```
15 void dig(double& val) {
16   std::cout << "Digging...\n";
17   val = val * 2;
18   deeper(val);
19   std::cout << "Done digging...\n";
20 }
21
22 void deeper(double& val) {
23   val -= 1;
24   std::cout << "now even deeper....\n";
25 }
```

This is one of the reasons why some programmers **only** use pass-by-reference when the parameter is `const`. Some programmers prefer passing pointers over non-`const` parameters. This requires the caller to explicitly pass in an address and clearly states that the function may modify the parameter.

5. Keep functions short
   - A function should do *one* thing well

     If you see a function doing more than one thing consider breaking it up into multiple functions
   - Is this (slightly) more work?

     In the short run, perhaps.

     In the long run, your total time spent debugging, testing, maintaining, and modifying will be far, far less than if you packed everything into one monster function
   - Note that *unit testing* is practically impossible once functions reach a certain size.
6. Strive to write a function *once* and never modify it again.
7. Check function parameters for validity. Unless you *completely* trust the caller (and their caller...)
   - It should be obvious: do not trust `argv[]`

Video

Video URL: https://www.youtube.com/watch?v=9mWWNYRHAIQ

When to write a function

As with any kind of abstraction, there are two goals to making a function:

- **Encapsulation**: If you have some task to carry out that is simple to describe from the outside, but messy to understand from the inside, then wrapping it in a function lets the caller carry out this task without having to know the details.

  This is also useful if you want to change the implementation later.
- **Code re-use**: If you find yourself writing the same lines of code in several places (or worse, are tempted to copy a block of code to several places), you should probably put this code in a function (or perhaps more than one function, if there is no succinct way to describe what this block of code is doing).

Both of these goals may be trumped by the goal of making your code **clear**. If you can't describe what a function is doing in a single, simple sentence, this is a sign that maybe you need to restructure your code. Having a function that does more than one thing (or does different things depending on its arguments) is likely to lead to confusion.

So, for example, this is not a good function definition:

```
// This code is an anti-pattern.
// It's an example of how NOT to write a function.

/**
 * If getMax is true, return maximum of x and y,
 * else return minimum.
 */
int computeMinOrMax(int x, int y, bool getMax) {
  if(x > y) {
    if(getMax) {
      return x;
    } else {
      return y;
    }
  } else {
    if(getMax) {
      return y;
    } else {
       return x;
    }
  }
}
```

This function is clearly trying to do two things and not doing either one very well. Two functions would be far simpler:

```
// return the maximum of x and y
// if x == y, return y
int maximum (int x, int y) {
  return (x < y) ? y : x;
}

// return the minimum of x and y
// if x == y, return y
int minimum (int x, int y) {
  return (y < x) ? y : x;
}

int computeMinOrMax(int x, int y, bool getMax) {
    if(getMax) {
      return maximum(x, y);
    }
    return minimum(x, y);
}

// or more compactly:
int computeMinOrMax(int x, int y, bool getMax) {
  return getMax ? maximum(x,y) : minimum(x,y);
}
```

Is this *slightly* more typing? Yes. At the end of the day, you will be far happier testing and debugging the three simpler functions than the first version. Your future co-workers will thank you.

Note

Also be aware the STL provides functions std::min and std::max, which eliminate the need for our `minimum` and `maximum` entirely and have the advantage of working on any *comparable* type.

This would transform the previous example to:

```
#include <algorithm>

int computeMinOrMax(int x, int y, bool getMax) {
  return getMax ? std::max(x,y) : std::min(x,y);
}
```

Example: number guessing

A more realistic example might help.

Original

While randomly surfing the internet I stumbled on a small program intended to help people understand C++.

Unfortunately, it is full of issues and this is the kind of program structure that will **not** help you when trying to create your own complicated projects.

Open this in Replit and run `./main` in the console tab. You may need to type `make main` first.

Try This!

How many bugs or other issues can you find in this program without looking at the 'Bugs' tab?

Bugs

This program has a few bugs.

Code like:

```
if(tries >= 20)
```

seems to imply you have 20 tries. The program actually gives you 21. Off-by-one errors like this are common.

This code looks ok, but isn't.

```
std::cout << "Enter a number between 1 and 100 (" << 20 - tries << " tries left): ";
std::cin >> guess;
std::cin.ignore();
```

There is no error checking on guess before it is used. Non-integer input causes the program to enter an infinite loop.

Related to this, there is this code:

```
int guess;

// try to get guess

if(guess > number)
```

Since guess is uninitialized, if `cin` fails to fill `guess`, then `guess` will not have any value when the if statement is evaluated, which is undefined behavior.

Issues

This is fundamentally a C program on a C++ forum.

Yes, it uses `cin` and `cout`.

That doesn't make it a C++ program.

A clue it was copied from C:

```
int main (void) . . .
```

Explicitly using `void` to declare a function take no parameters is a best practice in C. Otherwise the compiler assumes the function can take **any number** of parameters.

In C++, `main()` or any other function that takes no parameters is implicitly void.

Next problem is the way the random numbers are created:

```
int number = rand() % 99 + 2;
```

Quick!

Can we be **certain** that this correctly creates a number from 1 to 100, inclusive?

The code is just not that easy to reason about.

- We have to know how rand works
- We have to remember what modulus does.

Yes, not big hurdles, but this is where bugs hide. And for the record, the program asks the user to pick a number from 1 to 100, but this algorithm will never choose 1.

The standard library has a superior alternative to `rand`:

```
#include <iostream>
#include <random>

int main() {
  std::cout << "Random numbers: \n";
  // Seed with a real random value, if available
  std::random_device r;
  std::default_random_engine eng(r());  // make a random number generator

  for (int i = 0; i < 10; ++i) {
    // generate the next uniformly distributed integer between 0 and 999
    // using the random default engine
    std::cout << std::uniform_int_distribution<int> {0, 999} (eng) << '\n';
  }
}
```

Because there are no functions, it is necessary to repeat block of code like this:

```
std::cout << "Enter a number between 1 and 100 (" << 20 - tries << " tries left): ";
std::cin >> guess;
std::cin.ignore();
```

In general, the pattern

- Prompt
- Assign
- Validate
- Repeat (if needed) or exit

Is common. Because it's tedious to copy over and over, this program omits the error handling and the repeat.

A function is the obvious choice here.

Try This!

Run the program on the original tab, but enter a letter or other nonsense input instead of a number.

How would you fix this? Try it!

More repetition. How many times is the number `20` used in this program?

If you wanted to change the max number of guesses to 10, how many places do you need to remember? And this is just one file. These kinds of duplications can become painful to maintain as programs grow. They can quickly get out of control.

Finally, pet peeves of mine:

This code is mostly redundant. I just prefer not to see code that looks like this, even though it works.

```
if(answer == 'n' || answer == 'N' || answer == 'y' || answer == 'Y') {
```

I would rather use a function:

```
char play_again() {
  return std::tolower(
      get_entry("Would you like to play another game? [y/n] ").front());
}

// and use it like this
while ('y' == play_again());
```

instead of:

```
while(true) { // Loop to ask user is he/she would like to play again.
   // Get user response.
   std::cout << "Would you like to play again (Y/N)? ";
   std::cin >> answer;
   std::cin.ignore();

   // Check if proper response.
   if(answer == 'n' || answer == 'N' || answer == 'y' || answer == 'Y') {
      break;
   } else {
      std::cout << "Please enter \'Y\' or \'N\'...\n";
   }
}
```

Is this comment helping?

```
// Get number.

                std::cout << "Enter a number between 1 and 100 (" << 20 - tries << "␣
↪tries left): ";
```

```
// Safely exit.
std::cout << "\n\nEnter anything to exit. . . ";
std::cin.ignore();
```

Please don't ask me to enter an additional confirmation to exit, when I **just** said 'No' to the previous question.

There is no need to do this. What is *safe* about this? Just exit your program.

Final

Open this in Replit and run `./main` in the console tab. You may need to type `make main` first.

This is **NOT** the only way to improve the original program. It's merely one way.

Notice the finished program isn't shorter than the original. That was not our goal. It rarely is. We made the program *clearer* and easier to reason about. While we were at it, we fixed some bugs and made it a bit more reusable and maintainable.

More to Explore

- C++ Core Guidelines for functions
- Unit testing library list
- A very brief description of "extract method" from Martin Fowler's Refactoring site.
- ExtractMethod discussion from the PortlandPatternRepository - the very first wiki

3.5 Pointers

The pointer data type and indirect addressing are covered.

3.5.1 Pointers

People make a big deal out of pointers. They really aren't that hard to understand. We already know that a variable stores some value:

```
double euler = 2.718281828459;
```

We use the name `euler` to retrieve the value.

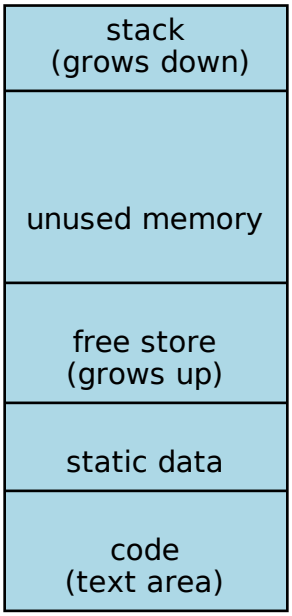
A pointer is simply a variable that stores an *address*:

```
double* e_pointer = &euler;
```

The value assigned to `e_pointer` is the *address of* the variable `euler`.

While some of the memory in a running program is stored in a small number of **registers**, these live on the CPU chip and perform specialized functions like keeping track of the location of the next machine code instruction to execute. Most of the memory of interest to programmers is **main memory**, which (mostly) lives outside the CPU chip and which

stores the code and data of a running program. Main memory is partitioned within a program into the following areas, which we have seen in the section *Introduction to functions*.



Typical program memory layout

Each part of a program: functions, variables, and objects are stored in main memory and each is assigned a unique address. When the CPU wants to fetch a value from a particular location in main memory, it must supply the address. Frequently, we don't need to concern ourselves with the address of a value. Instead, we use a variable or function name and the CPU resolves the address for us. Sometimes we need more control, and then we use **pointers** to store memory addresses and manipulate them like any other variable. In C++, they are defined as a family of types that can be passed as arguments, stored in variables, returned from functions, etc.

Declaring pointers

A pointer is defined by the operator * *and* a **type**. Both are required because a pointer can be thought of as a compound type. The * declares that the system should store an initial memory address, not a value. The type instructs the system how many bytes after the initial location need to be allocated for storage of the value pointed to. `int* int_pointer;` defines a new, uninitialized pointer to an `int`.

Both `int * int_pointer;` and `int *int_pointer;` declare the exact same variable. White space does not matter and the last variation is common. Personally, I prefer `int* p`, because it emphasizes (for me) that the *type* is *integer pointer*. Declarations of pointers to other fundamental types follow a similar pattern:

`double* dbl_pointer;` is an uninitialized pointer to a `double`.

`char* char_ptr;` is an uninitialized pointer to a `char`.

Each of these declarations creates a variable of either 4 or 8 bytes, depending on the architecture. On any given CPU all pointers are the same size, regardless of what they point to - because the **only** thing a pointer ever stores is an address. The pointer variable stores a specific memory location (the address) and the *value* associated with the pointer is stored

in one or more bytes starting at the pointer address.

Like any other variable in C++, an uninitialized pointer will initially contain garbage --- in this case, the address of a location that might or might not contain something important. To initialize a pointer, you assign it the address of something that already exists. If you already have an object, you can use the **address of operator** &:

```
int main() {
  int n = 5; // a stack int
  int* p;    // a pointer to an int
  p = &n;    // p now points to n
}
```

As you might expect, you do not need to declare pointers uninitialized. You can declare and initialize in a single step. Once you have an initialized pointer, use the **dereference operator** * to get the value stored in the pointer, or to assign a new value.

```
int main() {
   int x  = 10;
   int* p = &x;    // assign the address of x to p
   *p     = 7;     // x is now 7, p is unchanged

   int x2  = *p;   // assign the value of x to new int x2
   int* p2 = &x2;  // get a pointer to another int

   p2 = p;         // p2 and p both point to x
   p  = &x2;       // make p point to another object
}
```

The equivalent example for references is:

```
int main() {
   int y  = 10;
   int& r = y;    // the & is in the type, not in the initializer
   r      = 7;    // assign to y through reference r

   int  y2 = r;   // read y through r (no * needed)
   int& r2 = y2;  // get a reference to another int

   r2  = r;       // the value of y is assigned to y2
   //r = &y2;     // error: you can't change the value of a reference
                  //   (no assignment from int* to an int&)
}
```

More to Explore

- MyCodeSchool video: Pointers in C/C++ playlist

3.5.2 Using pointers

The simplest way to use a pointer is to get their value as with any other variable. This value will be an address, which can be stored in another pointer variable of the same type.

```
int n = 2;
int* p = &n;  // points to n
int* q = p;   // points to n also
```

Once a pointer has been dereferenced, it is treated exactly like any other variable of that type.

```
*p = *p + *q; // n = 4
```

Pointers support many of the same operations as other arithmetic types.

Operation	Result
`p == q`	`true` if and only if `p` and `q` both store the same address or the `nullptr` type
`p != q`	negation of above
`*p`	refers to the value pointed to by `p`
`*p = val`	writes `val` to the storage location pointed to by `p`
`val = *p`	reads from the location pointed to by `p` and writes to `val`
`++p p++`	increment the pointer, making it point to the next memory address after `p`
`p = p+x`	add the integral value `x` to the pointer making it point to `x` memory addresses after `p`
`--p p--`	decrement the pointer, making it point to the previous memory address before `p`
`p = p-x`	subtract `x` from `p` making it point to `x` memory addresses before `p`

The * operator binds very tightly, in other words, is it has high *precedence*. You can usually use *p anywhere you could use the variable it points to without worrying about parentheses. However, a few operators, such as the unary decrement and increment (-- and ++) operators, and the member of (.) operator used to unpack structs and classes, all have higher precedence. These require parentheses if you want the * to take precedence.

Listing 2: Using pointers

```
#include <iostream>
int main() {
  int n = 2;
  int* p = &n;  // points to n
  int* q = p;   // points to n also
  *p = *p + *q; // n = 4
  std::cout << "n = " << n << '\n';

  int* p2n = &n;  // another pointer to n
  (*p)++; // increments n
  std::cout << "n = " << n << '\n';
  std::cout << "* p = " << * p << '\n';
  *p++;   // increments p
          // p now points to next address in memory
          // Almost always an error
  std::cout << "* p = " << * p << '\n';

  return 0;
}
```

Unlike the fundamental types in C++, pointer types do not implicitly convert to other types. While we expect to be able to assign an int to a double, it is a compile error to assign an int pointer to a double pointer:

```
int    i = 5;
double d = i;     // OK.  implicit widening conversion

int*    pi = &i;
double* di = pi;  // compile error
```

More to Explore

- Operator Precedence
- From the ISO C++ FAQ: Does "Const Fred* p" mean that *p can't change?

3.5.3 Comparison with references

Recall from our earlier discussions of pass by reference that the `address of` operator `&` allows us to pass by reference:

```
1  #include <iostream>
2
3  void by_value(int x) {
4    x = 99;
5    std::cout << "in by_value the address of x is "
6              << &x << '\n';
7  }
8
9  void by_reference (int& x) {
10   std::cout << "in by_ref the address of x is   "
11             << &x << '\n';
12   x = -1;
13 }
```

In function `by_value` the statement `x = 99;` changes the copy provided. The value of `x` is printed, but is destroyed when `x` goes out of scope on line 6.

No special character is needed if you want to use a function that takes a reference:

```
#include <iostream>

int main () {
  int beta = 11;
  std::cout << "the address of beta is "
            << &beta << '\n';

  by_value(beta);

  std::cout << "beta = " << beta << '\n';

  by_reference(beta);

  std::cout << "beta is now "
            << beta << '\n';
}
```

References do have some definite advantages:

- A reference must always be initialized using an existing object. In other words, a reference can **never** be `null`.
- A reference can't be reassigned to a different object
- A `const` reference means you can't modify the thing the reference refers to
- References are simpler, more limited, and inherently safer than pointers

However, there are important things you can't do with references:

- You can't assign an address to a reference
  - This would have the effect of having a reference refer to a different object
  - The technical term for this is that references are not **assignable**
- You can't operate on a reference
  - In other words, you can't increment the referred to memory address, which, by definition, would involve having the reference refer to a different object
- You can't use a single reference to refer to more than one object
- You can't use references in containers such as `vector`
  - Containers can only hold *assignable* entities

We still need to be able to do all these kinds of memory manipulations. In C++, we achieve these goals using *pointers*.

Function passing semantics

We can pass pointers to a function that expects a reference:

```
#include <cassert>

void by_reference (int& x) {
  x = -1;
}

int main() {
  int  i = 5;
  int* p = &i;
  by_reference(p);
  assert (i == -1);
  return 0;
}
```

If we pass in only `p`, what happens?

Show Answer

The program fails to compile.

We can't pass an `int*` to a function expecting an `int&`.

Non-const references vs. pointers

Some programmers consider passing by non-const reference bad style, because the call syntax is the same as pass by value. When a variable is passed into a function by non-const reference there is no visual indication to the programmer of what to expect. Without reading additional documentation or reading the source code, there is no way to know if the function will change its parameter or not.

```
void func (int& x);

int main() {
  int x = 5;
  func(x);        // will x change?
}
```

For this reason, a function that takes a *non-owning pointer* is preferred:

```
void func (int* x);

int main() {
  int x = 5;
  func(&x);        // Caller expects x to change
}
```

A function signature is a *contract* between the function author and the function caller. A function that takes non-const references represents a poorly written contract. Callers don't know what to expect when the function is called. Even if the parameter isn't changed today, it might tomorrow. A non-owning pointer makes the intent clear. There is still no *requirement* to change the parameter, but since the caller is explicitly passing in an address, they can expect it to change.

More to Explore

- MyCodeSchool video: Pointers in C/C++ playlist

3.5.4 Pointers and arrays

Pointers are not arrays and arrays are not pointers. However, much confusion arises between them because *arrays in expressions* often behave like pointers. The term you'll often see is that *arrays decay into pointers*.

Things don't really "decay" in C or C++, but the name of an array resolves to an address. Given the following array:

```
int data[5] = {3, 5, 8, 13, 21};
```

The variable `data[0]` stores the value `3`. The variable `data` stores the address of element 0.

Pointers store addresses, so the same of an array and a pointer both store the same kinds of data.

Any array type will implicitly convert to a pointer of the type stored in the array. The pointer is constructed to point to the first element of the array. This conversion happens whenever arrays are used in an expression where arrays are not expected, but pointers are:

```
#include <iostream>

int main() {
```

(continues on next page)

(continued from previous page)

```
  int a[3] = {13, 21, 35};
  int* p = a;

  std::cout << sizeof a << '\n'  // prints size of array
            << sizeof p << '\n'; // prints size of a pointer

 for(int n: a) {          // okay: arrays can be used in range-for loops
   std::cout << n << ' '; // prints elements of the array
 }
 // for(int n: p) {       // error: no range for looping on a pointer


 // arrays and pointers share the same semantics
 std::cout << '\n'
           << *a << '\n' // prints the first element
           << *p << '\n' // same
           << *(a + 1) << ' ' << a[1] << '\n'  // prints the second element twice
           << *(p + 1) << ' ' << p[1] << '\n'; // same
}
```

This behavior applies to function calling as well:

```
#include <iostream>

// print first element of array using pointer dereference
void g(int (&a)[3]) {
  std::cout << *a << '\n';
}

// print first element of array using array semantics through pointer
void f(int* p) {
  std::cout << p[0] << '\n';
}

int main() {
  int a[3] = {13, 21, 35};
  int* p = a;

 // where arrays are acceptable, but pointers aren't, only arrays may be used
 g(a); // okay: function takes an array by reference
 // g(p); // error: pointers do not implicitly convert to arrays

 // where pointers are acceptable, but arrays aren't, both may be used:
 f(a); // okay: function takes a pointer
 f(p); // okay: function takes a pointer
}
```

Run It

This example shows many of the concepts we have discussed so far in one spot.

```cpp
#include <iostream>

int main()
{
    using std::cout;
    int data[5] = {8, 13, 21, 34, 55};

    int* p1 = data;
    int* p2 = data + 1;
    int  i1 = * data + 1;
    int  i2 = * (data + 1);
    int  i3 = * p2 + 1;

    cout << "The address of data is: " << data << ", " << &data  << ", " << p1 << '\n';
    cout << "The 1st val in data is: " << * data << ", " << * p1 << '\n';
    cout << "The address of p1 is: " << &p1 << '\n';

    cout << "The address of data[1] is: " << (data+1) << ", " << p2 << '\n';
    cout << "The 2nd val in data is: " << * (data+1) << ", " << * p2 << ", " << i2 << '\n
↪';

    cout << "8+1 equals: " << (* data +1) << ", " << i1  << ", " << * p1 + 1 << '\n';
    cout << "13+1 equals: " << * (data +1)+1 << ", " << i3  << ", " << * p2 + 1 << '\n';

    cout << "(*data + 1): \t" << (* data) << "\t\t" << (* data + 1) << '\n';
    cout << "*(data + 1): \t" << (* data) << "\t\t" << * (data + 1) << '\n';
    cout << "(&data + 1): \t" << (* (&data)) << " " << (&data + 1) << '\n';

    cout << "Print dataay address locations:\n";
    for (int i=0; i<5; ++i) {
      cout << (data+i) << ", "; // each location is 4 bytes larger
    }
    cout << std::endl;

    int array_bytes = sizeof data;
    int int_bytes = sizeof (int);
    int array_size = array_bytes / int_bytes;

    cout << "size: " << array_size
         << "\n# bytes in array: " << array_bytes
         << "\n# bytes in 1 int: " << int_bytes << '\n';

    return 0;
}
```

Array indexing pitfalls

Pitfall #1

Arrays perform absolutely no bounds checking.

Read that again.

Good.

Now consider that no compiler will complain about this code:

```
1 int* p = int[3];
2 p[0]  = 3;  // OK
3 p[2]  = 5;  // OK
4 p[99] = 8;  // oops!  where did we write this?
5 p[-7] = 8;  // or this!
```

No compiler will inform you that on line 4 we just wrote an 8 at a location 96 positions past the end of the array. Nor will it inform you that on line 5, we just wrote to a location 7 positions before the beginning of the array.

Most pointer examples you see will never attempt to use `operator[]` to index a pointer that is not an array. This is a good thing, but as you might expect, if you make a mistake, the compiler has nothing to offer:

```
int  n = 5;
int* p = &n;

int x = p[99] + 2;
```

Even with all compiler warnings enabled, most compilers will emit nothing at all. No compiler will inform you that we just accessed a piece of memory 98 `int`s past the one you own. Whatever is stored there, we then added 2 to it and assigned that value to `y`. The compiler doesn't even know `p` is a pointer to just one `int`.

Most programmers know better than to make errors this large. We're just demonstrating here that even big mistakes can be completely ignored by the compiler. What is for more common is an off by one error where your array index or pointer address is wrong only by 1. Accessing even a single byte outside your valid memory bounds is still an error and one of the most common errors in C and C++ programs.

Pitfall #2

From the standard:

> The definition of the subscript `operator[]` is that `E1[E2]` is identical to `(*((E1)+(E2)))`. Because of the conversion rules that apply to the binary `operator+`, if `E1` is an array object (equivalently, a pointer to the initial element of an array object) and `E2` is an integer, `E1[E2]` designates the `E2`-th element of `E1` (counting from zero).

Note

What the standard doesn't repeat here is that addition commutes, that is $a + b = b + a$. A side-effect of this fact is that for any array and index pair `a[i]`, then `a[i]` must be equivalent to `i[a]`.

```
1  #include <iostream>
2  using std::cout;
3
4  int main() {
5    int a[4] = {3, 5, 8, 13};
6    cout << "Print each array element 4 times:\n";
7    for (int i=0; i<4; ++i) {
8      cout << a[i]   << ' '
9           << *(a+i) << ' '
10          << *(i+a) << ' '
11          << i[a]   << '\n';
12   }
13 }
```

Although the standard does not strictly *prohibit* this syntax, doesn't mean you should use it.

This pitfall is only a problem when using arrays of type `int` with easily confused variable names. The lesson: use variables appropriate for the scope. In this case, perhaps a single letter (a) for the array was too short.

Arrays of type char

In the C language, the abstract idea of a string is implemented with an array of characters. Arrays of `char` that are null terminated are commonly called *C strings*.

In older C and C++ code using C strings, it's common to see code that uses the null terminator in the C string as a loop exit condition:

```
#include <stdio.h>

// an old C idiom to copy a 'string'
int main (int argc, char** argv) {
  char a[] = "Hello World!";
  char b[13];

  // print one char at a time
  int i;
  for (i=0; i<12;++i) putchar(a[i]);
  printf("\n");


  char* p1 = a;
  char* p2 = b;
  for (int i=0; a[i]; ++i) p2[i] = p1[i];

  printf("copy:\n");
  printf("%s\n", p2);  // print chars until '\0' detected
  return 0;
}
```

Code like this can fail if the source string contains any embedded null characters. The risk is that this code works fine 99% of the time, but fails when working with character data from an uncontrolled source (a network or socket interface, for example).

Try This!

Run the previous example, but modify it, replacing the 'Hello World' with 'Hello\0World'. What happens?

What warnings does the compiler display?

More to Explore

- Array declarations in C and C++
- MITRE Common Weakness Enumerations Off by one error
- Wikipedia Off-by-one errors

3.5.5 Pointers to pointers

A pointer can point to any memory address within the scope of the program, which includes pointers themselves. Each new pointer just adds another to the chain of pointers. The language does not impose a strict limit. The only limit is your sanity...

```
#include <iostream>

int main() {
  int x = 8;

  // all of these variables point to x
  int* p2x     = &x;
  int** p2p    = &p2x;
  int*** p2pp  = &p2p;

  return 0;
}
```

Like `int` or `char`, a pointer type is still a type. When you declare a variable of type pointer, storage still must be allocated somewhere, and this storage must have an address too.

When dealing with pointers, we have to manage the added complexity of keeping clear in our minds the difference between *the pointer variable* and *what the pointer points to*. When dealing with pointers to pointers, we have to manage the pointer, what it points to, and *what the pointer that it points to points to*.

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

using std::string;
using std::cout;

 int main() {
   string message[] = {"Alice","Bob here!","Carol checking in."};

   string *sp;   // a pointer to at least 1 string

   sp = message;
   cout << "sp:\n";
   cout << sp << '\n';
   cout << *sp << '\n';
   cout << *(sp + 1) << '\n';
   cout << *(sp + 2) << "\n\n";


   cout << "sp2:\n";
   string *sp2 = new string [3];          //create string pointer on the heap
   *sp2 = "\nAlice has left the building";
   *(sp2 + 1) = "Bob who?";
   *(sp2 + 2) = "Carol checked out.";

   cout << sp2 << '\n';
   cout << *sp2 << '\n';
   cout << *(sp2 + 1) << '\n';
   cout << *(sp2 + 2) << '\n' << '\n';
```

(continues on next page)

(continued from previous page)

```
  string **sp3;                // a pointer to a string pointer

  cout << "sp3:\n";
  sp3 = &sp2;
  cout << sp3 << '\n';
  cout << **sp3 << '\n';
}
```

You can also define a pointer to a reference variable:

Video

Video URL: https://www.youtube.com/watch?v=0QOxC7ADT80

More to Explore

- MyCodeSchool video: Pointers in C/C++ playlist

3.5.6 Other pointer characteristics

This section wraps up pointers with a discussion of some alternative constraints and special pointer types.

Constant pointers

Pointers can be declared `const`, just like any other type. Where `const` appears controls what is held constant:

```
// odd whitespace to help see where const is used
      int         x = 5;
      int*       p1 = &x;  // non-const pointer to non-const int
const int*       p2 = &x;  // non-const pointer to const int
      int* const p3 = &x;  // const pointer to non-const int
const int* const p4 = &x;  // const pointer to const int
```

You may find it helpful to read pointer declarations from right to left.

- In `p1`, nothing is constant. Either the pointer or the value pointed to can change.
- In `p2`, the pointer can change, but the value pointed to is constant. You can't use this pointer to change the value of x.
- In `p3`, the pointer is constant, but the value pointed to can change. You can use this pointer to change the value of x, but can't point to a different variable.
- In `p4`, both are held constant.

The `nullptr` type

In section *Comparison with references*, we mentioned that unlike a reference, a pointer might point to 'nothing'.

What exactly is 'nothing'?

Many languages refer to this 'nothing' as `NULL`.

Prior to C++11, there was no unambiguous definition. Typically the value 0 was used:

```
#define NULL 0LL
```

This definition carries over from standard C.

Using the value `long long 0` as an indicator for a null pointer created several problems over the years in C++ programs.

Null pointers are the same type as regular integral types.

While it is unlikely that the number 0 could ever be confused with a valid address, it creates problems regular old C never had to handle. Specifically, C++ introduces function overloads, which exposes the weakness in using an integral type for both numbers and the concept `NULL`. For example:

```
#include <cstdio>
#define NULL 0LL

// Three overloads of f
void f(int)   { puts("f(int)"); }
void f(bool)  { puts("f(bool)"); }
void f(void*) { puts("f(void*)"); }

int main() {
  f(0);     // calls f(int) overload, not f(void*)

  f(NULL);  // might not compile, typically calls
            // f(int) overload.
            // Never calls f(void*)
}
```

The overload with `f(NULL)` is never called, because `NULL` is not a pointer type.

C++ resolves this by creating a new type just to hold the null pointer. The type is `nullptr_t` and the variable of that type is `nullptr`.

```
#include <cstdio>

// Three overloads of f
void f(int)   { puts("f(int)"); }
void f(bool)  { puts("f(bool)"); }
void f(void*) { puts("f(void*)"); }

int main() {
  f(0);        // calls f(int) overload as before

  f(nullptr);  // calls f(void*) overload
}
```

The variable `nullptr` is a distinct type. It is not a pointer type, pointer to member, integral type, size type, reference type, or a member of any type group. The `nullptr` **does** implicitly convert to a pointer type.

In short, using `nullptr` improves code clarity and correctness. Using `nullptr` improves code clarity, especially when auto variables are involved. Consider the following code example, from Effective Modern C++:

```
// A function that returns a pointer
int* findRecord() {
```

(continues on next page)

(continued from previous page)

```
  return nullptr;
}

int main() {
  // If you don't happen to know (or can't easily find out) what findRecord returns,
  // it may not be clear whether result is a pointer type or an integral type.
  //
  // After all, 0 (what result is tested against) could go either way.

  {
    auto result = findRecord();

    if (result == 0) {
    }
  }

  // If you see the following, on the other hand ...
  {
    auto result = findRecord();

    if (result == nullptr) {
    }
    // there's no ambiguity: result must be a pointer type.
  }

}
```

`void` pointers

A *void pointer* is a pointer to some memory, but the compiler doesn't know the type.

It is about as close to a raw machine address as you can get in C++.

Legitimate uses are calls between functions in different languages or templates where the provided value could literally be *anything*, such as the actual implementation of `new` in C++.

Important!

`void*` is not the same as `void`

There are no objects of type void:

```
int i;          // declare an int
void x;         // error!  void is not a type
void print();   // function returns nothing
```

Any pointer can be assigned to `void*`:

```
int*    i  = new int{5};
double* x  = new double[10];
int*    j  = i;             // OK: i and j are both int*
void*   p1 = i;             // OK: assign int* to void*
void*   p2 = d;             // OK: assign double* to void*
```

(continues on next page)

(continued from previous page)

```
6
7  int*    i2 = p1;            // error
8                              // can't assign void* to int*
```

The last assignment is invalid, even though `p1` was last assigned an `int*`. A human reader knows the void pointer currently holds an int pointer, but the compiler does not.

The compiler **can't** know the size of the value pointed to. `void` isn't a type, so it has no size:

```
int*    i = new int{5};
void*   p = i;              // OK
int*    j = p;              // error
```

To resolve this error, we have to give the compiler size information. We can use one of C++ *casts* to convert `void*` to another pointer type that has a size:

```
int*    i = new int{5};
void*   p = i;                      // OK
//int*  j = p;                      // error
int*    j = static_cast<int*>(p); // OK
```

More to Explore

- From the ISO C++ FAQ: Does "Const Fred* p" mean that *p can't change?
- Effective Modern C++ by Scott Meyers Item 8: Prefer nullptr to 0 and NULL

3.5.7 Free store pointers

So far, all of our variables have been created on the *stack*. Another way to say this is our variables have *automatic storage duration*. Variables exist only as long as the scope in which they were created. Sometimes, we need to create objects with dynamic storage duration, that is, objects whose lifetime is not limited by the scope in which they were created.

One way to do this is to use the operator new to create objects on the *free store*. The free store is a system-provided memory pool for variables whose lifetime is directly managed by the programmer. Compare this to our experiences so far where variables were *automatically* managed by the system, not by the programmer.

The `new` operator takes a type and (optionally) a set of initializers for that type as its arguments. It returns a pointer to an (optionally) initialized object of its type:

```
1  struct point {
2      double x = 0;  // member values is a C++11 feature
3      double y = 0;
4  };
5
6  int main() {
7    int* p1 = new int;      // allocate 1 uninitialized int
8    int* p2 = new int[3];   // allocate 3 uninitialized ints
9    int* p3 = new int(5);   // allocate 1 int initialized to 5
10   int* p4 = new int{5};   // allocate 1 int initialized to 5, C++11 or later
```

(continues on next page)

(continued from previous page)

```
  int* p5 = new int();   // allocate 1 int initialized to 0

  point* pt1 = new point;          // allocate a default constructed point
  point* pt2 = new point();        // allocate a default constructed point
  point* pt3 = new point[3];       // allocate 3 default constructed points
}
```

In all of the above cases, since the `new` operator returns a pointer to an object of its type, the initialization could use `auto`.

```
auto pt1 = new point;
```

The operator new allocates memory. When finished with the free-store memory, we return it to the pool of available memory using the operator delete:

```
struct point {
     double x = 0;  // member values is a C++11 feature
     double y = 0;
};

int main() {
  int* p1 = new int;
  int* p2 = new int[5];

  point* pt_x = new point;
  point* points = new point[3];

  delete   p1;  // free memory allocated for a single object
  delete[] p2;  // free array memory

  delete   pt_x;  // same syntax is used for user defined types also
  delete[] points;
}
```

There should always be exactly 1 `delete` for every pointer returned by `new`.

Note

There are two forms of `delete`:

- `delete p` frees the memory for a single object allocated using `new`
- `delete[] p` frees the memory for an array of objects allocated using `new`

Mistakes over which version of delete to use is a common source of error.

Other mistakes related to delete include deleting the same pointer twice, or not deleting the pointer at all.

Deleting the same pointer twice is a problem because it leads to undefined or unpredictable behavior. The problem rarely arises in very small or short programs. However, in larger programs, strange or unpredictable events may happen long after the statements that perform the double delete are executed. Programs that free memory twice have created real-world security vulnerabilities.

Simply choosing to never delete a pointer on the theory that "well, at least my program won't crash" is not a good idea either. All computers have a finite amount of memory. Depending on how long your program needs to run, never

returning unused memory back to the memory *pool* is referred to as a *memory leak*. Also, remember that computers are *fast*. Depending on what your program does, even a short program can run out of memory before accomplishing all of its goals.

STL memory management

When memory is allocated using operator new, eventually it must be recovered using operator delete. When only a few lines of code are requesting memory, this is not a major problem. However, real world programs often request hundreds or thousands of blocks of memory. Keeping track of all this memory and when it should be freed can be labor intensive. Moreover, the consequences of an error are high: program crashes or corrupted data.

Many languages, such as Java, Python, Ruby, and JavaScript take this problem completely out of the hands of programmers. In these languages, memory is never explicitly deleted by the program. Rather it is managed by a *garbage collector*, which is responsible for cleaning up after the program (removing its *garbage*).

C++ does not provide a garbage collection mechanism by default. Given that memory management is such a problem, does the STL provide any resources to help solve it?

Yes.

The C++ Standard Template Library provides a family of classes to help solve these problems. They are all contained in the header `<memory>` and are defined as templates so that they can point to objects of any type.

Smart pointers are classes that behave like *raw* pointers but also manage objects created with `new`, so that you don't have to worry about when and whether to delete them. Smart pointers are declared on the *stack* and automatically delete the encapsulated object when the smart pointer goes out of scope. The smart pointer is defined in such a way that it can be used syntactically almost exactly like a raw pointer.

C++17 Feature

One of the earliest so-called 'smart pointers' was `auto_ptr`. Much online documentation and many text books still refer to and recommend `auto_ptr`. The `auto_ptr` function was officially deprecated in C++11 and removed in C++17. Generally, where old texts refer to `auto_ptr`, use `unique_ptr` instead.

Class `std::unique_ptr`

A unique_ptr is a so-called 'smart pointer' that owns and manages another object through a pointer and disposes of that object when the `unique_ptr` goes out of scope. A `unique_ptr` is a very lightweight wrapper around a pointer. The basic syntax is:

```
// older C++11 syntax
std::unique_ptr<int> p1 = std::unique_ptr<int>(new int);
```

C++14 Feature

make_unique simplifies initialization:

```
std::unique_ptr<int> p2 = std::make_unique<int>();
```

In each example, both `p1` and `p2` are unique pointers that 'own' an `int*`. Our earlier examples can be changed to:

```
#include <memory>
struct point {
  double x = 0;
```

(continues on next page)

(continued from previous page)

```
  double y = 0;
};

int main() {
  std::unique_ptr<int> p2 = std::make_unique<int>();
  auto                 p3 = std::make_unique<int>();       // less redundant

  // array examples
  // unique pointers to arrays of 5 elements
  std::unique_ptr<int[]> p4 = std::unique_ptr<int[]>(new int[5]);
  auto                   p5 = std::make_unique<int[]>(5);

  // user define types are no different
  auto pt_x   = std::make_unique<point>();    // one point*
  auto points = std::make_unique<point[]>(3); // array of 3 point*
}
```

Once declared, a unique pointer can be manipulated using the same syntax as a raw pointer.

```
auto p = std::make_unique<point>();
// modify point coordinates and print
p->x = 8;
p->y = 13;
std::cout << p->x << ' ' << p->y << '\n';

// this is an error
// std::cout << p.x << ' ' << p.y << '\n';
```

What makes a `unique_ptr` *unique*?

An object stored within a unique pointer **uniquely owns** its pointer. In other words, an object is 'owned' by exactly one unique_ptr. Unlike raw pointers, a unique pointer cannot be copied or assigned to another variable, even another `unique pointer`.

No two unique pointers can ever contain the same raw pointer value. This solves the 'double delete' problem if both go out of scope. The result is that some operations you **can** perform on raw pointers are not allowed on `unique_ptr`:

```
auto x = std::make_unique<point>();
std::unique_ptr<point> y = {x};     // error - copy construction not allowed

std::unique_ptr<point> z;           // new empty (nullptr)
if(!z) {                            // check if z != nullptr
  z = x;                            // error - copy assignment not allowed
}
```

Although copying unique pointers is not allowed, you can release the pointer and assign it to a raw pointer, or transfer ownership to a different `unique_ptr`.

More to Explore

- Free-store managment FAQ

- From: cppreference.com:
  - Operator new and delete:
  - unique_ptr and make_unique.

3.5.8 Pointer debugging

All the techniques described in section *Debugging* applies when debugging pointers. What can make debugging pointer errors more difficult is that when things go wrong with a pointer, often memory is corrupted as well, which limits what you can get done with `cout`.

Tools are your friends.

1. Learn how to use a debugger.

   The debugging section has a brief overview of the gdb debugger and links to more information.

2. Consider a dynamic memory checker such as `valgrind`.

Valgrind

The `valgrind` program can be used to detect some (but not all) common errors in C and C++ programs that use pointers and dynamic storage allocation. In addition to basic memory checking, `valgrind` can find many bugs related to uninitialized variables and undefinded behavior. You can run `valgrind` on your program by putting `valgrind` at the start of the command line:

```
valgrind ./my-program
```

You may pass in arguments to your program or redirect input from the shell:

```
valgrind ./my-program arg1 arg2 < test-input
```

This will run your program and produce a report of any memory allocations and de-allocations it did. Valgrind will also warn you about common errors like using uninitialized memory, dereferencing pointers to strange places, writing off the end of memory blocks allocated using `malloc` or `new`, or failing to free blocks.

You can suppress all of the output except errors using the -q option, like this:

```
valgrind -q ./my-program
```

You can also turn on more tests, e.g.:

```
valgrind -q --tool=memcheck --leak-check=yes ./my-program
```

See `valgrind --help` or `man valgrind` for more information about the (many) options, or look at the documentation at https://valgrind.org/ for detailed information about what the output means. If you want to run `valgrind` on your own machine, you may be able to find a version that works at https://valgrind.org/. Unfortunately, this is only likely to work if you are running a Unix-like operating system (which includes Linux and MacOS X, but not Windows).

A simple walk-through. Let's say we have the following code in `foo.cpp` with a simple error:

```
#include <iostream>
int main () {
  int x;
  int y = 3;
```

(continues on next page)

(continued from previous page)

```
  std::cout << "sum: " << (x + y) << '\n';
}
```

Just as with gdb, compile it with debugging symbols.

```
g++ foo.cpp -g3 -o foo
```

Valgrind provides a lot of debugging information very quickly, although on some systems, even with debug symbols compiled in, your output looks like this:

```
==1229== Conditional jump or move depends on uninitialised value(s)
==1229==    at 0x40DDBA3: std::ostreambuf_iterator<char, std::char_traits<char> >␣
↪std::num_put<char, std::ostreambuf_iterator<char, std::char_traits<char> > >::_M_
↪insert_int<long>(std::ostreambuf_iterator<char, std::char_traits<char> >, std::ios_
↪base&, char, long) const (in /usr/lib/libstdc++.so.6.0.18)
==1229==    by 0x40DE168: std::num_put<char, std::ostreambuf_iterator<char, std::char_
↪traits<char> > >::do_put(std::ostreambuf_iterator<char, std::char_traits<char> >,␣
↪std::ios_base&, char, long) const (in /usr/lib/libstdc++.so.6.0.18)
==1229==    by 0x40E9F1D: std::ostream& std::ostream::_M_insert<long>(long) (in /usr/lib/
↪libstdc++.so.6.0.18)
==1229==    by 0x40EA112: std::ostream::operator<<(int) (in /usr/lib/libstdc++.so.6.0.18)
==1229==    by 0x41CB9D2: (below main) (in /lib/libc-2.18.so)
```

In order to get `valgrind` to report line numbers related to errors in calls to library functions, you need to add the command line argument `--track-origins=yes`, like this:

```
$ valgrind --track-origins=yes ./foo

==1302== Conditional jump or move depends on uninitialised value(s)
==1302==    at 0x40DDB47: std::ostreambuf_iterator<char, std::char_traits<char> >␣
↪std::num_put<char, std::ostreambuf_iterator<char, std::char_traits<char> > >::_M_
↪insert_int<long>(std::ostreambuf_iterator<char, std::char_traits<char> >, std::ios_
↪base&, char, long) const (in /usr/lib/libstdc++.so.6.0.18)
==1302==    by 0x40DE168: std::num_put<char, std::ostreambuf_iterator<char, std::char_
↪traits<char> > >::do_put(std::ostreambuf_iterator<char, std::char_traits<char> >,␣
↪std::ios_base&, char, long) const (in /usr/lib/libstdc++.so.6.0.18)
==1302==    by 0x40E9F1D: std::ostream& std::ostream::_M_insert<long>(long) (in /usr/lib/
↪libstdc++.so.6.0.18)
==1302==    by 0x40EA112: std::ostream::operator<<(int) (in /usr/lib/libstdc++.so.6.0.18)
==1302==    by 0x41CB9D2: (below main) (in /lib/libc-2.18.so)
==1302==  Uninitialised value was created by a stack allocation
==1302==    at 0x8048717: main (foo.cpp:3)
==1302==
==1302== Use of uninitialised value of size 4
```

Show All valgrind output

```
==1302==    at 0x40DDA53: ??? (in /usr/lib/libstdc++.so.6.0.18)
==1302==    by 0x40DDB7B: std::ostreambuf_iterator<char, std::char_traits<char> >␣
↪std::num_put<char, std::ostreambuf_iterator<char, std::char_traits<char> > >::_M_
↪insert_int<long>(std::ostreambuf_iterator<char, std::char_traits<char> >, std::ios_
↪base&, char, long) const (in /usr/lib/libstdc++.so.6.0.18)
```

```
==1302==    by 0x40DE168: std::num_put<char, std::ostreambuf_iterator<char, std::char_
↪traits<char> > >::do_put(std::ostreambuf_iterator<char, std::char_traits<char> >,␣
↪std::ios_base&, char, long) const (in /usr/lib/libstdc++.so.6.0.18)
==1302==    by 0x40E9F1D: std::ostream& std::ostream::_M_insert<long>(long) (in /usr/
↪lib/libstdc++.so.6.0.18)
==1302==    by 0x40EA112: std::ostream::operator<<(int) (in /usr/lib/libstdc++.so.6.0.
↪18)
==1302==    by 0x41CB9D2: (below main) (in /lib/libc-2.18.so)
==1302==  Uninitialised value was created by a stack allocation
==1302==    at 0x8048717: main (foo.cpp:3)
==1302==
==1302== Conditional jump or move depends on uninitialised value(s)
==1302==    at 0x40DDA5C: ??? (in /usr/lib/libstdc++.so.6.0.18)
==1302==    by 0x40DDB7B: std::ostreambuf_iterator<char, std::char_traits<char> >␣
↪std::num_put<char, std::ostreambuf_iterator<char, std::char_traits<char> > >::_M_
↪insert_int<long>(std::ostreambuf_iterator<char, std::char_traits<char> >, std::ios_
↪base&, char, long) const (in /usr/lib/libstdc++.so.6.0.18)
==1302==    by 0x40DE168: std::num_put<char, std::ostreambuf_iterator<char, std::char_
↪traits<char> > >::do_put(std::ostreambuf_iterator<char, std::char_traits<char> >,␣
↪std::ios_base&, char, long) const (in /usr/lib/libstdc++.so.6.0.18)
==1302==    by 0x40E9F1D: std::ostream& std::ostream::_M_insert<long>(long) (in /usr/
↪lib/libstdc++.so.6.0.18)
==1302==    by 0x40EA112: std::ostream::operator<<(int) (in /usr/lib/libstdc++.so.6.0.
↪18)
==1302==    by 0x41CB9D2: (below main) (in /lib/libc-2.18.so)
==1302==  Uninitialised value was created by a stack allocation
==1302==    at 0x8048717: main (foo.cpp:3)
==1302==
==1302== Conditional jump or move depends on uninitialised value(s)
==1302==    at 0x40DDBA3: std::ostreambuf_iterator<char, std::char_traits<char> >␣
↪std::num_put<char, std::ostreambuf_iterator<char, std::char_traits<char> > >::_M_
↪insert_int<long>(std::ostreambuf_iterator<char, std::char_traits<char> >, std::ios_
↪base&, char, long) const (in /usr/lib/libstdc++.so.6.0.18)
==1302==    by 0x40DE168: std::num_put<char, std::ostreambuf_iterator<char, std::char_
↪traits<char> > >::do_put(std::ostreambuf_iterator<char, std::char_traits<char> >,␣
↪std::ios_base&, char, long) const (in /usr/lib/libstdc++.so.6.0.18)
==1302==    by 0x40E9F1D: std::ostream& std::ostream::_M_insert<long>(long) (in /usr/
↪lib/libstdc++.so.6.0.18)
==1302==    by 0x40EA112: std::ostream::operator<<(int) (in /usr/lib/libstdc++.so.6.0.
↪18)
==1302==    by 0x41CB9D2: (below main) (in /lib/libc-2.18.so)
==1302==  Uninitialised value was created by a stack allocation
==1302==    at 0x8048717: main (foo.cpp:3)
==1302==
sum: 134514654
==1302==
==1302== HEAP SUMMARY:
==1302==     in use at exit: 0 bytes in 0 blocks
==1302==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==1302==
==1302== All heap blocks were freed -- no leaks are possible
==1302==
==1302== For counts of detected and suppressed errors, rerun with: -v
```

```
==1302== ERROR SUMMARY: 20 errors from 4 contexts (suppressed: 0 from 0)
```

There can be a lot of noise in the output, especially when templates are involved. If you look at the last line in each group, `foo.cpp, line 3` is called out each time. And that is exactly where the error lies.

More to Explore

- The Valgrind Quick Start Guide
- Valgrind tutorial - from Barton P. Miller
- Valgrind video tutorial
- **MyCodeSchool video:**
  Memory leak in C++

3.5.9 Pointers to functions

In C++ you can point to anything with an address:

- Global variables
- Stack and free store variables
- Functions

When a function is called, an *activation record* for the function is pushed onto the *runtime stack*. This means every function has an address.

Consider the following code:

```
#include <iostream>
int foo() {
  return 5;
}
int main() {
  std::cout << foo;
}
```

What does this program print?

Hint: It doesn't call the function foo

Show

It prints the *address* of the function named `foo`

When a function is called using `operator()` (the function call operator), execution jumps to the *address* of the function being called. We can make use of this to store the address of the function.

To declare a pointer to a function that returns an `int` and takes no parameters:

```
int (*pf)();
```

The pointer variable is named `pf`. The parentheses around `(*pf)` are required due to operator precedence. Without the parentheses:

```
int *pf();     // not a pointer to a function

// same as above
int* pf();
```

Instead, this declares a function that returns an `int*` and takes no parameters.

To declare a pointer named `func` pointing to a function that returns a `double` and takes two parameters:

```
double (*func)(int x, int y);
```

Example

Once you have a valid function pointer definition, you can assign functions to it.

Given the following functions:

```
double add     (int x, int y) { return x+y;}
double multiply(int x, int y) { return x*y;}
double pi      ()             { return 3.14159265;}
```

We can define a pointer to our functions

This is legal, but not preferred, since our pointer is undefined.

```
double (*func)(int, int);
```

When you can, initialize variables with a value.

```
double (*func)(int, int) = add;
```

Run It

```
1  double add     (int x, int y) { return x+y;}
2  double multiply(int x, int y) { return x*y;}
3  double pi      ()             { return 3.14159265;}
4
5  int main () {
6    // declare func and assign add to it
7    double (* func)(int, int) = add;
8
9    cout << (* func)(2,3); // prints 4
10
11   func = multiply;
12   cout << (* func)(2,3); // prints 6
13
14   func = pi;             // error: pointer to function with
15                          // wrong number of arguments
16
17   func = add();          // error: can't assign
18                          // function return value to function pointer
19 }
```

C++11 Feature

A downside to traditional function pointer initialization is that this doesn't look like the initialization syntax we are used to. This is a legacy of the C language C++ was originally based on.

The C++11 type alias allows defining a name that refers to a previously defined type:

```
double (*func)(int, int);          // old syntax

using func = double(*)(int, int);  // since C++11
```

Try This!

Refactor the previous example to replace the traditional C syntax with the C++11 `using` type alias.

Video

Video URL: https://www.youtube.com/watch?v=p4sDgQ-jao4

Example: Caesar ciphers

A simple substitution cipher, called ROT13, short for 'rotate 13 places' can be used to obfuscate text by replacing each letter with the letter 13 letters after it in the alphabet. It is a special case of the Caesar cipher, used in ancient Rome to obscure communication between Julius Caesar and his generals. A related variation called ROT47 extends the idea of ROT13 to include numbers and common symbols.

Suppose we want to create a program that allows users to run either ROT13 or ROT47? There are many ways to implement such a program. This example demonstrates how to use function pointers to dynamically control at which function is called within a loop at runtime.

Often, when writing a program, it is useful to start at 'the top'. Suppose we want a simple command line program that takes 4 basic inputs:

-h
: A command line switch to show help.

-l
: A command line switch to transform only letters in the Latin alphabet. This switch will enable the ROT13 function.

-f
: A command line switch to transform the full set of printable letters ASCII character set. This switch will enable the ROT47 function.

standard input
: This is where the program will get the text to work on.

help.h

Once we have decided on this as our basic framework, we can create a file like `help.h`:

```
#pragma once

#include <iostream>
```

(continues on next page)

(continued from previous page)

```
static void usage(const char* name) {
    std::cerr << "Encrypt or decrypt text read from standard input\n"
        << "Usage: " << name << " [-h|l|f]\n";
}

static void help (const char* name) {
    usage(name);
    std::cerr << "Options:\n"
        << "  -h   Show this text and exit.\n"
        << "  -l   En(de)crypt using the Latin characters ([A-Z,a-z]\n"
        << "         (default)\n"
        << "  -f   En(de)crypt using the Full set of "
        << "printable ASCII characters\n"
        << "\nOnly 1 of any option can be meaningfully specified.\n"
        << "\nThe last of either '-l' or 'f' provided is used.\n"
        << "\nRunning with no input from standard in enters\n"
        << "'interactive mode':\n"
        << " - Text can be entered one message per line.\n"
        << " - The program runs until 'CTRL+C' is entered or "
        << "   EOF is reached.\n\n"
        << "Running on plain text creates cipher text\n"
        << "Running on cipher text creates plain text\n\n";
    exit(0);
}
```

ceasar.h

The 'fundamental unit' of any text input is a `char`, so it makes sense to write our transforming functions to work with a single character at a time.

We will write three functions for this program, one for ROT13 and one for ROT47. The third function takes a `std::string` and a function pointer as input, and transforms the string using the provided function, one character at a time.

First we declare our interfaces:

```
#pragma once

//
// Function to perform simple character rotations through an alphabet.
//
// !! DO NOT use to encrypt sensitive data!!
// These functions are easily decrypted, even using pencil and paper.
//

#include <string>

// A pointer to the function that will transform a character
using transform = char (*)(const unsigned char c);

// Rotate a character 13 places in the alphabet.
//
```

(continues on next page)

(continued from previous page)

```
// This function assumes a basic 26 letter Latin alphabet
//
// If the character received is not within the range [A-Z, a-z],
// then the character is returned unchanged.
char rot13(const unsigned char c);

// Rotate a character 47 places in the set of printable ASCII characters
//
// If the character received is not a 'printable' character (value < 33),
// then the character is returned unchanged.
char rot47(const unsigned char c);

// Use a 'character handler' (rot13 or rot47) to
// transform a message 1 character at a time.
void render_text(std::string message, transform handler);
```

The using declaration exists only to simplify our use of our function pointer. Any place you see the word `transform`, you can literally replace it with `char (*)(const unsigned char c)` and not change how the program behaves.

ceasar.cpp

We implemented our functions to take type `unsigned char` because they depend on the library functions `std::isdigit` and `std::isalpha`. These function have undefined behavior if the character provided is not an `unsigned char`.

With these definitions in place, we can implement them:

```
#include "caesar.h"

#include <iostream>
#include <cctype>      // isalpha, islower
#include <string>

// TODO: add locale and extend non english alphabets

char rot13(const unsigned char c) {
    if (!std::isalpha(c)) return c;   // if not a Latin letter,
                                      // then return the current char

    // in order to rotate upper or lower case
    // need to know where the alphabet 'starts'
    const char start = std::islower(c) ? 'a' : 'A';
    return (c - start + 13) % 26 + start;
}

char rot47(const unsigned char c) {
    // first printable character = 33 = '!'
    static constexpr char start = '!';
    if (c < start) return c;
    return (c - start + 47) % 94 + start;
}

void render_text(std::string message, transform handler) {
```

(continues on next page)

(continued from previous page)

```
    for (const auto& c: message) {  // extract each char from the string
        std::cout << handler(c);    // print transformed character
    }
    std::cout << std::endl;         // print newline and flush stream
}
```

main.cpp

And now we can put it all together from a small main program.

One thing to note in this main is that it uses **both** standard input and command-line arguments.

The difference is a common source of confusion.

```
#include "caesar.h"
#include "help.h"

#include <cstring>    // strcmp
#include <iostream>
#include <string>

int main(int argc, char** argv) {
    // Define a default a 'character handler'
    // the variable 'handler' provides an option to
    // change the encryption function at runtime
    transform handler = rot13;

    // loop on command line argumenats
    // call help and exit or
    // use of the 2 transforms
    // or reject the input and exit
    for (int i=1; i < argc; ++i) {
        if (!std::strcmp(argv[i], "-h")) {
            help(*argv);
        } else if (!std::strcmp(argv[i], "-l")) {
            handler = rot13;
        } else if (!std::strcmp(argv[i], "-f")) {
            handler = rot47;
        } else {
            usage(*argv);
            exit(-1);
        }
    }

    std::string message;
    while (getline(std::cin, message)) {
        render_text(message, handler);
    }
    return 0;
}
```

Note, we did not use the function call operator, `operator()` when assigning values to `handler`. The name `rot13` points to the address where the function `rot13` is stored.

Try This!

Consider compiling these files in your own environment and experimenting with variations.

Why are *13* and *47* common rotations?

Try to think of other rotations and implement them as additional program options.

More to Explore

- MyCodeSchool video: Pointers in C/C++: function pointers
- Caesar cipher from Wikipedia.
- From: cppreference.com:
  - pointers
  - isalpha and islower.
  - strcmp
  - getline

3.5.10 Lambda expressions

A *lambda expression*, or simply *lambda*, provides a means to define an anonymous function right at the location where it is invoked or passed as an argument to a function.

Often, it is more convenient to write a very short function 'in line' where the function is used rather than:

1. Write a separate function with a name
2. Store the function in a specific compilation unit
3. `#include` the compilation unit definitions where it will be used
4. Call or pass the function as an argument

While using STL facilities with lambdas is not required, it turns out that many facilities in the STL take function pointers as arguments. This fact makes lambdas particularly useful since any place a function pointer can be used, a lambda expression can be used instead.

Suppose we want to count the number of short strings in a vector. There is a 'count if' function in the STL we can use to get the job done. The std::count_if function expects 3 parameters:

- A pointer to the first element to examine in the `vector`
- A pointer to one past the last element to examine in the `vector`
- A pointer to a function that will determine if the element should be counted

The last parameter is commonly called a *predicate function*. A predicate function is a fancy name for a function that returns a `bool`.

We could implement this using the techniques we have already learned:

```
bool less_than_5(const string& str) {
  return str.size() < 5;
}

size_t num_short_strings(const vector<string>& v) {
  const string* begin = v.data();            // must be const
  const auto end = begin + v.size();
  return count_if(begin, end, less_than_5);
}
```

While it is easy to write as many functions like this as we need: `less_than_10`, `greater_than_15`, etc., it's clear this would get tedious quickly. Nor it is very flexible. Each time we want to compare a different value, or use a different comparator, we need to add a new function and recompile the program!

In addition, if `less_than_5` is never used anywhere else, it seems like a bad idea to have this special-case function elevated to the status equivalent to all other functions. Conceptually, it doesn't deserve to have the status of a full-fledged function with its own name, callable from anywhere.

Our goal is to avoid writing a new function for every little comparison we want to make. One obvious solution is to add a parameter to our `less_than_5` function:

```
bool less_than(const string& str, size_t size) {
  return str.size() < size;
}
```

Unfortunately, we can't use this function in count_if. The new version of the function is arguably more generic, but we can't use our 'improved' `less_than` in `count_if`. A predicate must be a *unary function*. In other words, it can take only 1 parameter, no more. The 'improved' function is less useful than the old, even though we made it 'generic'. We need a way to pass more than one parameter to a function that can only take 1 parameter.

Starting in C++11, a new language feature was added just for this kind of problem: lambda expressions. Depending on where you go, you'll also see these referred to as closures, lambda functions, function literals, or just lambdas.

A lambda allows us to omit the separate function definition entirely and use it within the `num_short_strings` function as an inline parameter to `count_if`:

```
1 size_t num_short_strings(const vector<string>& v) {
2   const string* begin = v.data();
3   const auto end = begin + v.size();
4
5   return count_if(begin, end,
6                   [](string x) { return (x.size() < 5); });
7 }
```

The lambda on line 6 completely replaces the old function `less_than_5`.

The general syntax for a lambda is:

```
[ captures ] (parameters) -> returnType { lambda_body; }
```

The capture block `[]` is required, even if empty. This informs the compiler that a lambda expression is beginning.

The capture block allows the lambda to **capture** variables from outside the scope of the lambda body and use them within the lambda. Without a capture, no variables outside the lambda scope are visible within the lambda.

In our previous example, it would allow us to extend num_short_strings by being able to pass in a value, rather than hard code the value 5:

```
size_t num_short_strings(const vector<string>& v, size_t sz) {
  const string* begin = v.data();
  const auto end = begin + v.size();

  return count_if(begin, end,
                  [&sz](string x) { return (x.size() < sz); });
}
```

The parameter is optional, but many useful standard algorithms that operate on containers expect to be able to pass each container element one at a time to a function that will use it. This is exactly what `count_if` does.

The return type is also optional. A lambda will ordinarily be able to deduce the correct type from the return statement, so an explicit return is not needed.

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

using std::string;
using std::vector;

// Show the number of strings in the vector whose length
// is between min and max.
//
// perhaps a better name for this function is 'show_lengths_between'
void show_lengths (const vector<string>& v, size_t min, size_t max) {
  const string* begin = v.data();
  const auto end = begin + v.size();

  std::cout << "Number of names of length \n";
  for (std::size_t i = min; i < max; ++i) {
    std::cout << i << ": \t";
    std::cout << std::count_if(begin, end,
                [&i](string x) { return (x.size() == i); });
    std::cout << '\n';
  }

}

int main () {
  vector<string> names = {
    "alice", "bob", "clarissa", "dario", "elizabeth",
    "abbi", "barnabas", "clarabelle", "daniel", "ethan"
    "farouk", "gabby", "hanh phuc", "lakshay",
    "fabrice", "gail", "habeeb", "jabir", "liza",
    "muhammad", "nora", "oscar", "pauline", "roberto",
    "scarlett", "thomas", "ubelia", "victorio",
    "wahkisha", "xan", "yacov", "zabrina"
  };

  show_lengths (names, 2, 13);
}
```

Try this!

Rewrite the rot13/rot47 program to use lambdas to perform the rotation instead of traditional function calls

Assigning a lambda to a variable

A lambda does not always need to be defined within another function or within a function parameter list. You can define a lambda any place a normal function can be defined, but if you do this, you must assign the lambda to a variable so that it has a name:

```
#include <iostream>
int main() {
  auto lambda = [] {
    std::cout << "Hello, lambda!\n";
  };
  lambda();
}
```

Once a lambda has a name, it is called like a function, using `operator()`. The previous example was about as simple as a lambda can get. No return type and no parameter.

Note the use of `auto` when defining the type of a lambda. In the case of a lambda, auto is not an option. If auto was not used, what type would we supply in its place?

```
int main () {
  vector<int> numbers { 1, 2, 3, 4, 5, 10, 15, 20, 25,
                        35, 45, 50 };
  const int* begin = numbers.data();
  const auto end = begin + numbers.size();

  auto gt_5 = count_if(begin, end,
                [](int x) { return (x > 5); });

  cout << "The # of elements > 5 is: "
       << gt_5 << ".\n";
}
```

Actually, there is no way to know. Each lambda defines a new type. Only the compiler knows exactly what the type is, so as a programmer, you must use auto and allow the compiler to deduce it.

Even two identical lambdas will become two different types when compiled.

It is possible to convert between lambda expressions and function pointers, as this video demonstrates:

Video

Video URL: https://www.youtube.com/watch?v=Cmk0Tlo1eCA

Trailing return types in lambda expressions

When the compiler cannot deduce the correct type, or does not deduce the desired type, then the return type must be specified.

Since the capture clause must be first, we have a problem: where to specify the return type. It can't come before the capture clause, which is where return types are defined for normal functions.

In a lambda, the only option is to specify a *trailing return type*. It can be used with ordinary functions also, but they are most commonly seen in lambda expressions and function templates.

A trailing return type is `operator->`, followed by the return type. The trailing return type must occur after the parameter list and before the function body. This is true for both lambdas and normal functions.

In the following function, we want to return a type other than what would normally be returned by the operations.

```
int main () {
  cout << "The return value of this odd function is: "
          [](double x, double y) -> int {
            if (x > 5) {
              return x + y;
            } else if (y < 2) {
              return x - y;
            } else {
              return x * y;
            }
          } (3.14159, 2.71828) << ".\n";
}
```

Note in the previous example we defined a lambda taking two parameters and then immediately called it using `operator()`.

More to Explore

- From: cppreference.com:
  - C++ lambda expressions.
  - std::count and std::count_if.
- *Descriptions of lambda expressions* <https://learn.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?redirectedfrom=MSDN>`_ from Microsoft's MSDN

3.5.11 The std::function template

C++11 Feature

The std::function wrapper provides a standard way to pass around lambda expressions, function pointers, and function objects.

Its facilities are accessed from `#include <functional>`. The primary goal of `std::function` is clarity: to simplify and standardize the way function pointers are moved within a program.

Starting with a simple function:

```
void print_num(int i) {
  std::cout << i << '\n';
}
```

Before `std::function`, we would declare a function pointer like this:

```
void (*print_func)(int val);
```

This defines a pointer to a function that returns `void` and take one `int` parameter. The syntax is messy:

- The pointer operator requires surrounding parentheses to resolve order of operations problems
- The return type is 'far away' from the parameters
- The syntax is generally inconsistent with modern C++.

Compare the previous declaration with:

```
std::function<void (int)>  print_func;
```

This syntax is more consistent with other declarations:

- The variable type is listed first: it's a function pointer
- The function parameters and return type are all together as template parameters and the parameters look like a traditional function declaration.
- The name of the variable follows the type

We can store any function-like entity using a consistent syntax:

```
#include <functional>
#include <iostream>

void print_num(int i) {
  std::cout << i << '\n';
}

int main() {
  // store a free function
  std::function<void(int)> print_func = print_num;
  print_func(-9);

  // store a lambda
  std::function<void()> print_42 = []() { print_num(42); };
  print_42();

  return 0;
}
```

More to Explore

- From: cppreference.com:
  - std::function

3.6 Recursion

Recursive functions and simple recursive data structures are introduced. The sections on recursive functions are expected to be review of first semester material.

3.6.1 What is Recursion?

Recursion refers to defining something in terms of itself. While recursion may appear in many forms (paintings and poems, for example), in computer science and mathematics, objects or functions exhibit recursive behavior when they can be defined by two properties:

- A *base case* - a terminating scenario that does **not** use recursion to produce an answer
- A *recursive step* - a set of rules that reduces all other cases toward the base case

Some famous sequences in mathematics, such as the Fibonacci sequence are generated recursively.

Accumulating a sum

Consider a problem you are already familiar with: computing the sum of a sequence of numbers.

Example: iterative sum

The function `accumulate` loops over each element in the vector `values`.

The variable `sum` holds the running total.

When the loop completes, the total is returned.

If the vector is empty, then 0 is returned.

```
int accumulate(const std::vector<int>& values) {
  int sum = 0;
  for (const auto& x: values) {
    sum = sum + x;
  }
  return sum;
}

int main () {
  std::vector<int> values = {1,2,3,4,5,6,7,8,9,10};

  auto sum = accumulate(values);
  std::cout << "sum is: " << sum << '\n';

  return sum;
}
```

Run It

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

// iterative sum
int accumulate(const std::vector<int>& values) {
  int sum = 0;
  for (const auto& x: values) {
    sum = sum + x;
  }
  return sum;
}

```

(continues on next page)

(continued from previous page)

```
int main () {
  std::vector<int> values = {1,2,3,4,5,6,7,8,9,10};

  auto sum = accumulate(values);
  std::cout << "sum is: " << sum << '\n';

  return sum;
}
```

The implementation details are not critical here. We could have used a traditional `for` loop or a `while` loop and computed the same value. The point is that all these solutions are *iterative*.

It is possible to solve this problem without using a loop. Actually, there is a loop of sorts, but it is the function itself that is used as the looping construct. Let's pick apart accumulating the sum to see how.

Recall that addition is a *binary operation* - an operation with two operands: a pair of numbers. We need to restate the accumulate problem from adding a vector to adding pairs of numbers.

$$1 + 2 + 3 + 4$$

Can be rewritten as:

$$(1 + (2 + (3 + 4)))$$

Notice that the innermost set of parentheses, $(3 + 4)$, is a problem that we can solve without a loop or any special constructs. In fact, we can use the following sequence of simplifications to compute a final sum.

$$\begin{aligned} total = &\ (1 + (2 + (3 + 4))) \\ total = &\ (1 + (2 + 7)) \\ total = &\ (1 + 9) \\ total = &\ 10 \end{aligned}$$

How can we take this idea and turn it into a C++ program? First, let's restate the sum problem in terms of a C++ vector. We might say the sum of the vector `values` is the sum of the first element of the vector (`values[0]`), and the sum of the numbers in the rest of the vector - the range `values.begin()+1` to `values.end()`.

Example: recursive sum

Instead of a local variable to accumulate the sum, the return value of the function stores the accumulated sum.

Each return value is the current first elment of the vector plus a *slice* of the vector composed of all the elements after the first element.

The base case occurs when the vector is empty - there is nothing left to add. We return zero since no matter what is in the vector, adding zero does not change the final result.

If the container is not empty, then it must contain at least one element. The recursive case is then the first element, plus everything after the first element (which might be nothing).

```
int accumulate(const std::vector<int>& values) {
  // base case
  if (values.empty()) return 0;

  // recursive case
  auto slice = std::vector<int>(values.begin()+1, values.end());
```

(continues on next page)

(continued from previous page)

```
  return values[0] + accumulate(slice);
}

int main () {
  std::vector<int> values = {1,2,3,4,5,6,7,8,9,10};

  auto sum = accumulate(values);
  std::cout << "sum is: " << sum << '\n';

  return sum;
}
```

Run both the iterative an recursive versions and verify they both produce the same results.

Run It

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

// recursive sum
int accumulate(const std::vector<int>& values) {
  // base case
  if (values.empty()) return 0;

  // recursive case
  auto slice = std::vector<int>(values.begin()+1, values.end());
  return values[0] + accumulate(slice);
}

int main () {
  std::vector<int> values = {1,2,3,4,5,6,7,8,9,10};

  auto sum = accumulate(values);
  std::cout << "sum is: " << sum << '\n';

  return sum;
}
```

The recursive calls to accumulate perform in code the same steps outlined when we grouped the addition sequence using parentheses.

More to Explore

- Recursion on Wikipedia
- TED: The magic of Fibonacci numbers, Arthur Benjamin

3.6.2 Properties of recursive functions

All recursive algorithms must implement 3 properties:

1. A recursive algorithm must have a **base case**.
2. A recursive algorithm must change its state and move toward the base case.
3. A recursive algorithm must call itself.

The base case is the condition that allows the algorithm to stop the recursion and begin the process of returning to the original calling function. This process is sometimes called *unwinding the stack*. A base case is typically a problem that is small enough to solve directly. In the `accumulate` algorithm the base case is an empty vector.

The second property requires modifying something in the recursive function that on subsequent calls moves the state of the program closer to the base case. A change of state means that some data that the algorithm is using is modified. Usually the data that represents our problem gets smaller in some way. In the `accumulate` algorithm our primary data structure is a vector, so we must focus our state-changing efforts on the vector. Since the base case is the empty vector, a natural progression toward the base case is to shorten the vector.

Lastly, a recursive algorithm must call itself. This is the very definition of recursion. Recursion is a confusing concept to many beginning programmers. As a novice programmer, you have learned that functions are good because you can take a large problem and break it up into smaller problems. The smaller problems can be solved by writing a function to solve each problem. When we talk about recursion it may seem that we are talking ourselves in circles. We have a problem to solve with a function, but that function solves the problem by calling itself! But the logic is not circular at all; the logic of recursion is an elegant expression of solving a problem by breaking it down into a smaller and easier problems.

In the remainder of this chapter we will look at more examples of recursion. In each case we will focus on designing a solution to a problem by using the three properties of recursive functions.

Self Check

Q1

Question

How many recursive calls are made when computing the sum of the vector {2,4,6,8,10}?

- ◯ 6
- ◯ 5
- ◯ 4
- ◯ 3

Q2

Question

Suppose you are going to write a recusive function to calculate the factorial of a number. fact(n) returns n * n-1 * n-2 * ... Where the factorial of zero is defined to be 1. What would be the most appropriate base case?

- ◯ n == 0

◯ n == 1

◯ n >= 0

◯ n <= 1

More to Explore

- TBD

3.6.3 Example: Converting an Integer to a String

Suppose you want to convert an integer to a string in some base between binary and hexadecimal. For example, convert the integer 10 to its string representation in decimal as `"10"`, or to its string representation in binary as `"1010"`. While there are many algorithms to solve this problem, the recursive formulation of the problem is elegant.

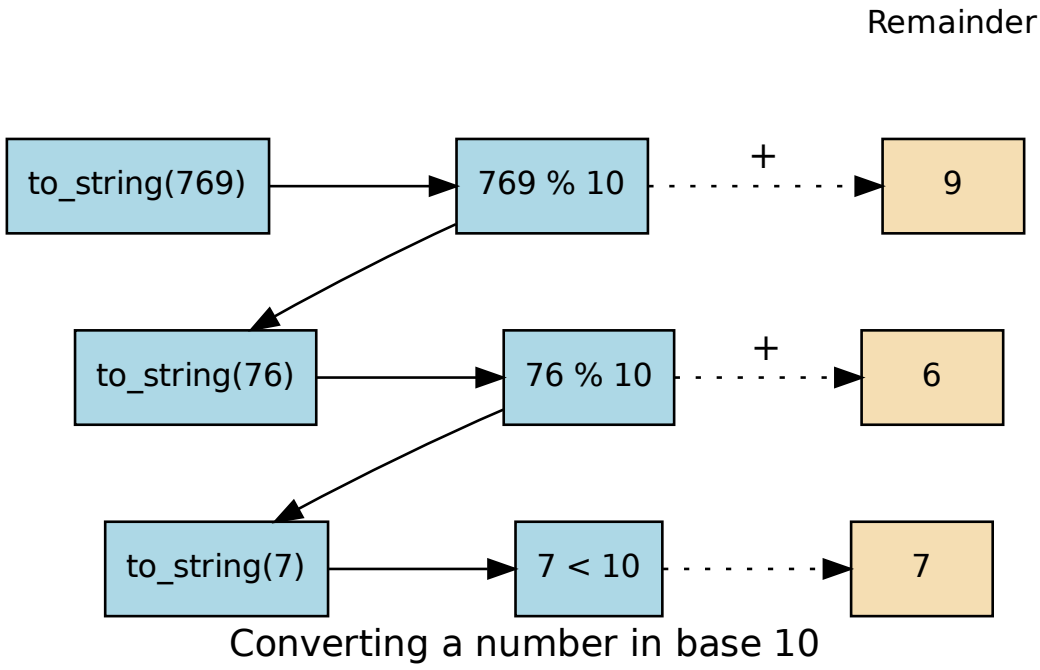
Let's look at a concrete example using base 10 and the number 769. Suppose we have a sequence of characters corresponding to the first 10 digits, like `digits = "0123456789"`. It is easy to convert a number less than 10 to its string equivalent by looking it up in the sequence. For example, if the number is 9, then the string is `digits[9]` or `"9"`. If we can arrange to break up the number 769 into three single-digit numbers, 7, 6, and 9, then converting it to a string is simple. A number less than 10 sounds like a good base case.

Knowing what our base is suggests that the overall algorithm will involve three components:

1. Reduce the original number to a series of single-digit numbers.
2. Convert the single digit-number to a string using a lookup.
3. Concatenate the single-digit strings together to form the final result.

The next step is to figure out how to change state and make progress toward the base case. Since we are working with an integer, let's consider what mathematical operations might reduce a number. The most likely candidates are division and subtraction. While subtraction might work, it is unclear what we should subtract from what. Integer division with remainders gives us a clear direction. Let's look at what happens if we divide a number by the base we are trying to convert to.

Using integer division to divide 769 by 10, we get 76 with a remainder of 9. This gives us two good results. First, the remainder is a number less than our base that can be converted to a string immediately by lookup. Second, we get a number that is smaller than our original and moves us toward the base case of having a single number less than our base. Now our job is to convert 76 to its string representation. Again we will use integer division plus remainder to get results of 7 and 6 respectively. Finally, we have reduced the problem to converting 7, which we can do easily since it satisfies the base case condition of $n < base$, where $base = 10$. The series of operations we have just performed is illustrated in the figure below. Notice that the numbers we want to remember are in the remainder boxes along the right side of the diagram.



to_string

```
1 std::string to_string(int n, int base) {
2   const char* digits = "0123456789ABCDEF";
3   if (n < base) {
4     return std::string(1, digits[n]);
5   }
6   return to_string(n/base, base).append(1, digits[n%base]);
7 }
```

Notice that in line 3 we check for the base case where `n` is less than the base we are converting to.

When we detect the base case, recursion stops and a char from `digits` is returned.

The final return on line 6 satisfies both the second and third properties of recursion - by making the recursive call and by reducing the problem size using division.

Note

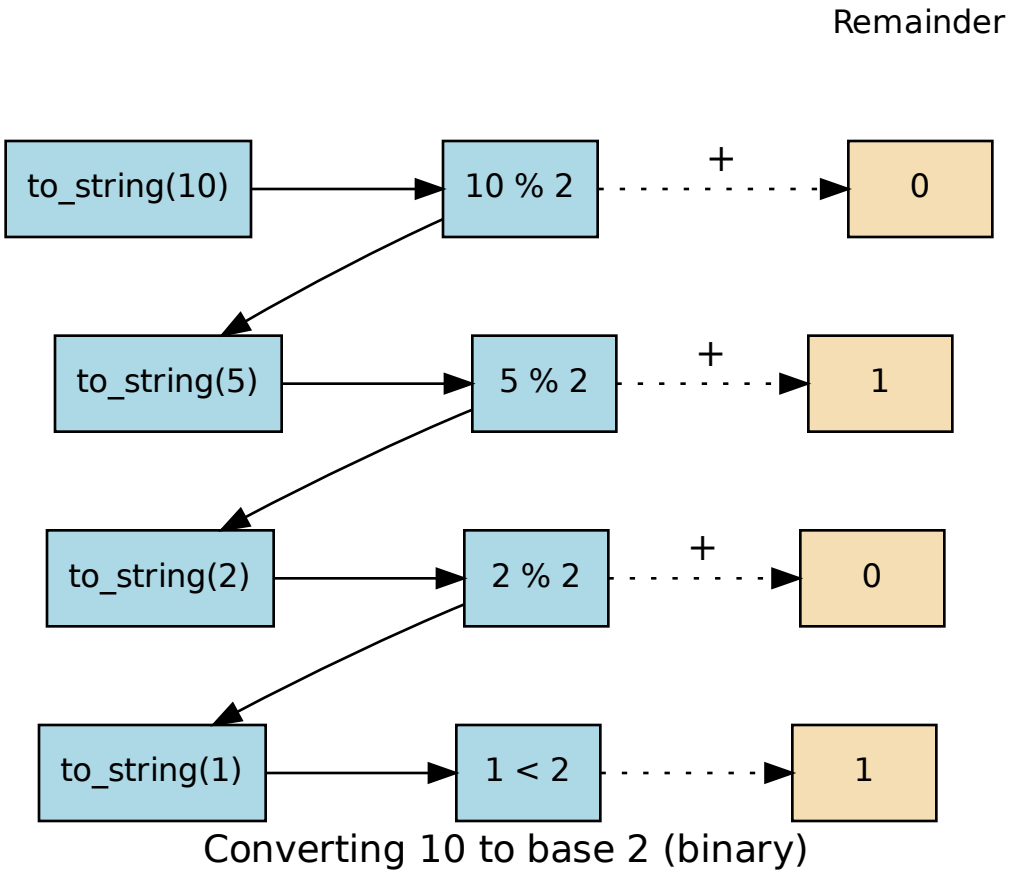
The standard library includes a set of functions called std::to_string to convert a variety of numeric types into a *std::string*, but it does not handle arbitrary change of base.

Run It

Listing 3: Recursively converting from int to string

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

std::string to_string(int n, int base) {
  const char* digits = "0123456789ABCDEF";
  if (n < base) {
    return std::string(1, digits[n]);
  }
  return to_string(n/base, base).append(1, digits[n%base]);
}

int main() {
  std::cout << to_string(314159, 16) << '\n';
}
```

Let's trace the algorithm again; this time we will convert the number 10 to its base 2 string representation (`"1010"`).



Converting 10 to base 2 (binary)

The previous figure shows that we get the results we are looking for, but it looks like the digits are in the wrong order. The algorithm works correctly because we make the recursive call first on line 6, then we add the string representation of the remainder. If we reversed returning the `digits` lookup and returning the `to_string` call, the resulting string would be backward! But by delaying the concatenation operation until after the recursive call has returned, we get the result in the proper order.

More to Explore

- std::to_string from cppreference.com

3.6.4 The Binary Tree ADT

Unlike the linear sequences covered so far, a *tree* is a *hierarchical* abstract data type. Conceptually, it can be thought of as a collection of *nodes* defined by parent-child relationships.

Refer to *Tree ADT concepts* to review the basic vocabulary and structure of binary trees.

So far, we have discussed trees only at a conceptual level. In this section we will program one.

In a binary tree, one node is the *root*. It serves as the 'trunk' of the tree and serves the same function as the *head* of a *list* The root node is the *only* node in a tree without a parent. All other nodes in a *tree* refer to exactly 1 parent. In a *binary tree*, the children are commonly referred to as the **left** and **right** children.

A simple binary tree can be implemented as a recursive data structure:

```
template <class T>
 struct tree {
    T value;
    tree<T>* left = nullptr;
    tree<T>* right = nullptr;
};
```

This tree stores a single value of type `T` and has two pointers to potential children. It is important to remember when working with a recursive tree such as this that a child might be a `nullptr`.

Binary tree traversal

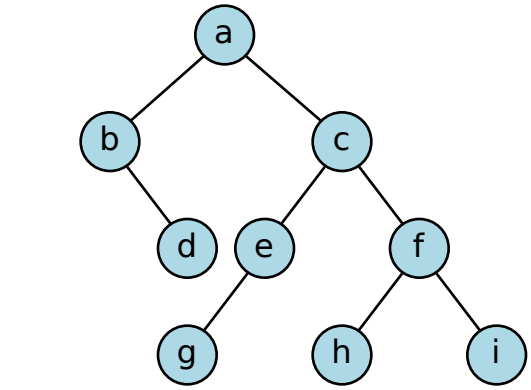
Traversal of a data structure means visit elements of the structure. It might mean visiting a subset, but can also involve visiting each element from the first to last.

For sequential containers, identifying the start and end of the container is simple. But where are the *first* and *last* elements of a tree?

The short answer is that there is no single answer. Since a tree is not a sequential data structure, a tree allows for different sequences (or traversals). There are several different types of traversals. The most common tree traversals are:

- Preorder
- Postorder
- Inorder
- Level order

Given the following binary tree, let's implement functions that traverse the tree using each of the four methods.



Subject of the next 4 traversals

Preorder traversal

A depth first traversal.

Visit all nodes **before** visiting children:

Preorder

In this generic code block, the function `visit` represents the action to take on the current node.

```
void preorder(tree<T>* node) {
  if (node == nullptr) {
    return;
  }
  visit(node);
  preorder(node->left);
  preorder(node->right);
}
```

Run It

```
#include <iostream>

template <class T>
struct tree {
  T value;
  tree<T>* left = nullptr;
  tree<T>* right = nullptr;
};

template <class T>
void print(tree<T>* node) {
```

(continues on next page)

(continued from previous page)

```
  if (node == nullptr) {
    return;
  }
  std::cout << node->value << ' ';
  print(node->left);
  print(node->right);
}

====

int main() {
  tree<char> a;
  tree<char> b;
  tree<char> c;
  tree<char> d;
  tree<char> e;
  tree<char> f;
  tree<char> g;
  tree<char> h;
  tree<char> i;
  a.value = 'a';
  b.value = 'b';
  c.value = 'c';
  d.value = 'd';
  e.value = 'e';
  f.value = 'f';
  g.value = 'g';
  h.value = 'h';
  i.value = 'i';
  a.left  = &b;
  a.right = &c;
  b.right = &d;
  c.left  = &e;
  c.right = &f;
  e.left  = &g;
  f.left  = &h;
  f.right = &i;

  std::cout << "Preorder: ";
  print (&a);

  return 0;
}
```

Postorder traversal

A depth first traversal.

Visit all nodes **after** visiting children:

Postorder

```
void postorder(tree<T>* node) {
  if (node == nullptr) {
    return;
  }
  postorder(node->left);
  postorder(node->right);
  visit(node);
}
```

Run It

The only difference between the preorder example and this is the order of the function calls in the `print` function.

```
#include <iostream>

template <class T>
struct tree {
  T value;
  tree<T>* left = nullptr;
  tree<T>* right = nullptr;
};

template <class T>
void print(tree<T>* node) {
  if (node == nullptr) {
    return;
  }
  print(node->left);
  print(node->right);
  std::cout << node->value << ' ';
}

====

int main() {
  tree<char> a;
  tree<char> b;
  tree<char> c;
  tree<char> d;
  tree<char> e;
  tree<char> f;
  tree<char> g;
  tree<char> h;
  tree<char> i;
  a.value = 'a';
  b.value = 'b';
  c.value = 'c';
  d.value = 'd';
  e.value = 'e';
  f.value = 'f';
  g.value = 'g';
```

(continues on next page)

(continued from previous page)

```
  h.value = 'h';
  i.value = 'i';
  a.left  = &b;
  a.right = &c;
  b.right = &d;
  c.left  = &e;
  c.right = &f;
  e.left  = &g;
  f.left  = &h;
  f.right = &i;

  std::cout << "Postorder: ";
  print (&a);

  return 0;
}
```

Inorder traversal

A depth first traversal.

Visit the left child (and the left child subtree), then visit the current node, then visit the right child (and the right child subtree),

Inorder

```
void inorder(tree<T>* node) {
  if (node == nullptr) {
    return;
  }
  inorder(node->left);
  visit(node);
  inorder(node->right);
}
```

Run It

```
#include <iostream>

template <class T>
struct tree {
  T value;
  tree<T>* left = nullptr;
  tree<T>* right = nullptr;
};

template <class T>
void print(tree<T>* node) {
  if (node == nullptr) {
    return;
  }
  print(node->left);
```

(continues on next page)

(continued from previous page)

```
  std::cout << node->value << ' ';
  print(node->right);
}

====

int main() {
  tree<char> a;
  tree<char> b;
  tree<char> c;
  tree<char> d;
  tree<char> e;
  tree<char> f;
  tree<char> g;
  tree<char> h;
  tree<char> i;
  a.value = 'a';
  b.value = 'b';
  c.value = 'c';
  d.value = 'd';
  e.value = 'e';
  f.value = 'f';
  g.value = 'g';
  h.value = 'h';
  i.value = 'i';
  a.left  = &b;
  a.right = &c;
  b.right = &d;
  c.left  = &e;
  c.right = &f;
  e.left  = &g;
  f.left  = &h;
  f.right = &i;

  std::cout << "Inorder: ";
  print (&a);

  return 0;
}
```

Level order traversal

Differs from the previous traversals: it is a 'breadth first' traversal. Also, this algorithm is easier to implement iteratively than recursively.

Visit each node on each level of the tree then visit the children one level deeper.

Level order

This is an iterative, not a recursive function.

```
void levelorder(tree<T>* node) {
  if (node == nullptr) {
```

(continues on next page)

(continued from previous page)

```
    return;
  }
  std::queue<tree<T>*> q;
  q.push(node);
  while (!q.empty()) {
    auto tmp = q.front();
    visit(tmp);
    q.pop();
    if(tmp->left)  q.push(tmp->left);
    if(tmp->right) q.push(tmp->right);
  }
}
```

Run It

```
#include <iostream>
#include <queue>

template <class T>
struct tree {
  T value;
  tree<T>* left = nullptr;
  tree<T>* right = nullptr;
};

template <class T>
void print(tree<T>* node) {
  if (!node) return;

  std::queue<tree<char>*> q;
  q.push(node);
  while (!q.empty()) {
    auto tmp = q.front();
    std::cout << tmp->value << ' ';
    q.pop();
    if(tmp->left)  q.push(tmp->left);
    if(tmp->right) q.push(tmp->right);
  }
}

====

int main() {
  tree<char> a;
  tree<char> b;
  tree<char> c;
  tree<char> d;
  tree<char> e;
  tree<char> f;
  tree<char> g;
  tree<char> h;
```

(continues on next page)

(continued from previous page)

```
  tree<char> i;
  a.value = 'a';
  b.value = 'b';
  c.value = 'c';
  d.value = 'd';
  e.value = 'e';
  f.value = 'f';
  g.value = 'g';
  h.value = 'h';
  i.value = 'i';
  a.left  = &b;
  a.right = &c;
  b.right = &d;
  c.left  = &e;
  c.right = &f;
  e.left  = &g;
  f.left  = &h;
  f.right = &i;

  std::cout << "Level order: ";
  print (&a);

  return 0;
}
```

More to Explore

- STL containers library
- Visualgo: binary heap

3.7 Sorting

Sorting is the process of placing elements from a collection in some kind of order. For example, a list of words could be sorted alphabetically or by length. A list of cities could be sorted by population, by area, or by zip code. We have already seen a number of algorithms that were able to benefit from having a sorted list (recall the final anagram example and the binary search).

This chapter introduces several sorting algorithms and their trade-offs. This chapter refers to relative algorithm performance, so reviewing the chapter on *algorithm analysis* may help.

3.7.1 Bubble sort

The **bubble sort** makes multiple passes through an array. It compares adjacent items and exchanges those that are out of order. Each pass through the array places the next largest value in its proper place. In essence, each item "bubbles" up to the location where it belongs.

Figure 1 shows the first pass of a bubble sort. Shaded items are being compared to see if they are out of order. If there are n items in the array, then there are $n - 1$ pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the array is part of a pair, it will continually be moved along until the pass is complete.

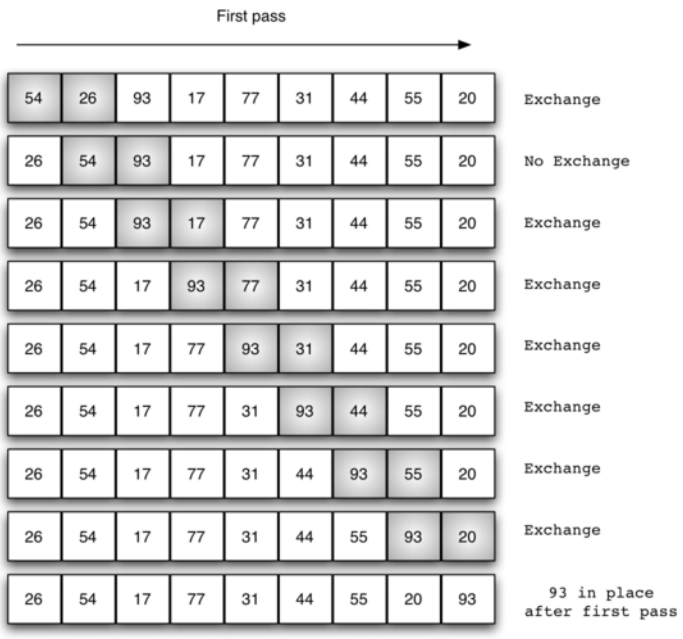


Fig. 1: Figure 1: `bubble_sort`: The First Pass

Bubble sort

At the start of the second pass, the largest value is now in place. There are $n - 1$ items left to sort, meaning that there will be $n - 2$ pairs. Since each pass places the next largest value in place, the total number of passes necessary will be $n - 1$. After completing the $n - 1$ passes, the smallest item must be in the correct position with no further processing required. The 'Run It' tab shows the complete `bubble_sort` function. It takes the array as a parameter, and modifies it by swapping items as necessary.

Typically, swapping two elements in an array requires a temporary storage location (an additional memory location). A code fragment such as

```
temp = alist[i];
alist[i] = alist[j];
alist[j] = temp;
```

will exchange the *ith* and *jth* items in the array. Without the temporary storage, one of the values would be overwritten.

Note

This exchange is referred to as the *swap idiom* and occurs frequently.

std::swap is part of the standard library and many swap specializations are defined to make swap efficient for STL types.

Run it

Listing 4: The Bubble Sort

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

(continues on next page)

(continued from previous page)

```
using std::vector;

// function goes through list sorting adjacent values as it bubbles
// the largest value to the top.
vector<int> bubble_sort(vector<int> data) {
  for (int passnum = data.size()-1; passnum > 0; --passnum) {
    for (int i = 0; i < passnum; ++i) {
      if (data[i] > data[i+1]) {
        // could be replaced with std::swap
        int temp = data[i];
        data[i] = data[i+1];
        data[i+1] = temp;
      }
    }
  }
  return data;
}

int main() {
  vector<int> data = {54,26,93,17,77,31,44,55,20};

  for (const auto& value: bubble_sort(data)) {
    std::cout << value << ' ';
  }
  return 0;
}
```

The following animation shows bubble sort in action. The values being sorted are bars of various heights. When bar colors change to red, this indicates these are the two values compared by bubble sort. Bars shown in gray have already reached their final position.

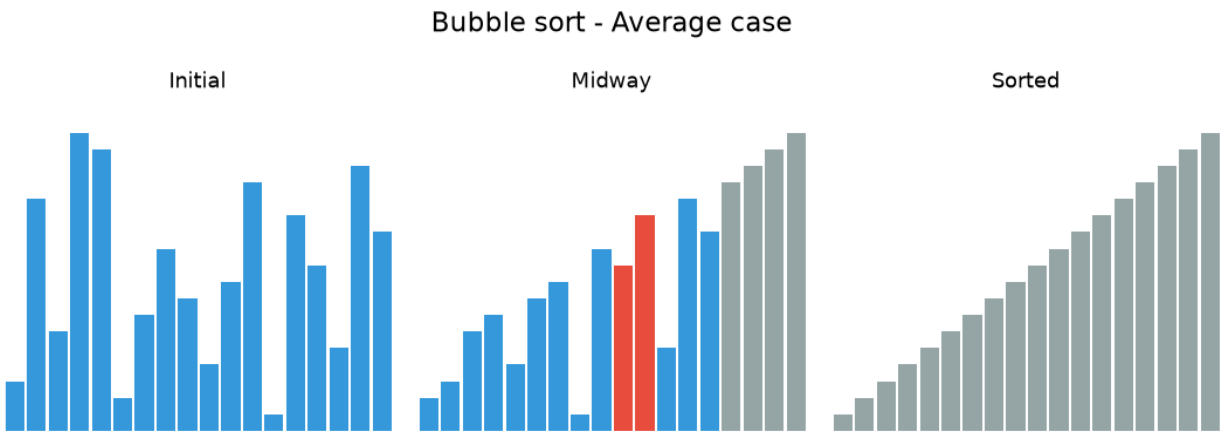


Fig. 2: Bubble sort on a deterministic shuffled input.

To analyze the bubble sort, we should note that regardless of how the items are arranged in the initial array, $n-1$ passes will be made to sort an array of size *n*. *Table 1* shows the number of comparisons for each pass. The total number of comparisons is the sum of the first $n-1$ integers. Recall that the sum of the first *n* integers is $\frac{1}{2}n^2 + \frac{1}{2}n$. The sum of the first $n-1$ integers is $\frac{1}{2}n^2 + \frac{1}{2}n - n$, which is $\frac{1}{2}n^2 - \frac{1}{2}n$. This is still $O(n^2)$ comparisons. In the best case, if the array is already ordered, no exchanges will be made. However, in the worst case, every comparison will cause an

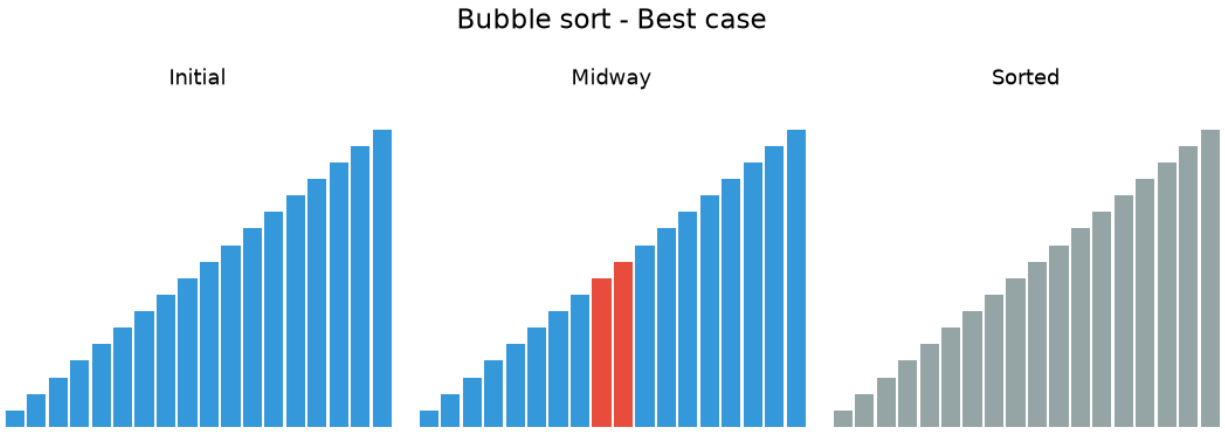


Fig. 3: Bubble sort on an already sorted input.

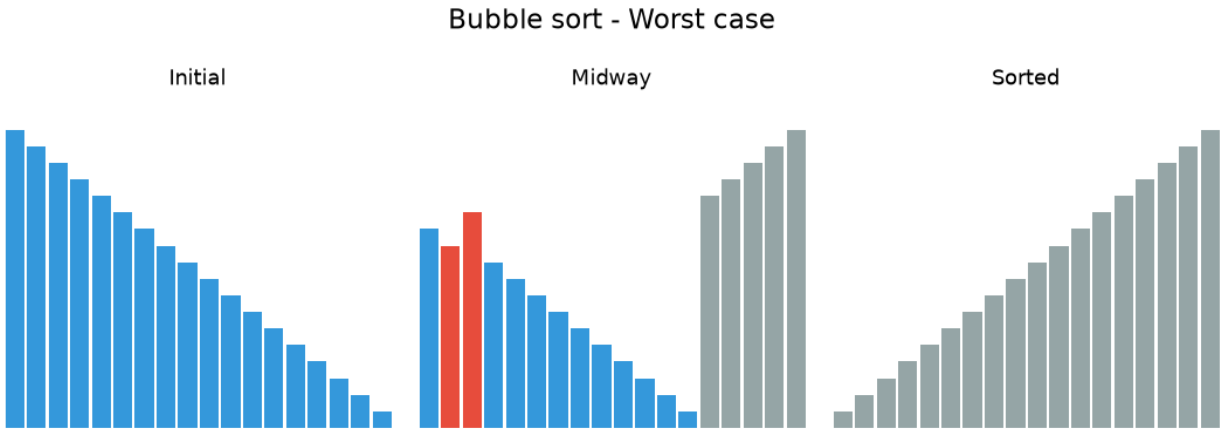


Fig. 4: Bubble sort on a reverse sorted input.

exchange. On average, we exchange half of the time.

Table 3: **Table 1: Comparisons for Each Pass of Bubble Sort**

Pass	Comparisons
1	$n-1$
2	$n-2$
3	$n-3$
...	...
$n-1$	1

Short Bubble

A bubble sort is often considered the most inefficient sorting method since it must exchange items before the final location is known. These "wasted" exchange operations are very costly. However, because the bubble sort makes passes through the entire unsorted portion of the array, it has the capability to do something most sorting algorithms cannot. In particular, if during a pass there are no exchanges, then we know that the array must be sorted. A bubble sort can be modified to stop early if it finds that the array has become sorted. This means that for arrays that require just a few passes, a bubble sort may have an advantage in that it will recognize the sorted array and stop. This modification is often referred to as the **short bubble**.

Run It

Listing 5: The 'Short' Bubble Sort

```
#include <iostream>
#include <utility>
#include <vector>
using std::vector;

vector<int> short_bubble(vector<int> data) {
  bool exchanges = true;
  int passnum = data.size();
  while (passnum > 0 && exchanges) {
    exchanges = false;
    for (int i = 0; i < passnum; ++i) {
      if (data[i] > data[i+1]) {
        std::swap(data[i], data[i+1]);
      }
    }
    // decrement passnum variable so that the next pass is one less
    // than the previous: the largest value has already 'bubbled' all the way up.
    --passnum;
  }

  return data;
}

int main() {
  vector<int> data = {20,30,40,90,50,60,70,80,110,100};

  for (const auto& value: short_bubble(data)) {
    std::cout << value << ' ';
```

(continues on next page)

(continued from previous page)

```
  }
  return 0;
}
```

Self Check

Q1

Question

Suppose you have the following array of numbers to sort: [19, 1, 9, 7, 3, 10, 13, 15, 8, 12] which array represents the partially sorted list after three complete passes of bubble sort?

- ◯ [1, 9, 19, 7, 3, 10, 13, 15, 8, 12]
- ◯ [1, 3, 7, 9, 10, 8, 12, 13, 15, 19]
- ◯ [1, 7, 3, 9, 10, 13, 8, 12, 15, 19]
- ◯ [1, 9, 19, 7, 3, 10, 13, 15, 8, 12]

More to Explore

- TBD

Acknowledgements

This section is adapted from Problem Solving with Algorithms and Data Structures using C++, by Brad Miller and David Ranum, Luther College, and Jan Pearce, Berea College released under the CC BY-NC-SA 4.0, and Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

3.7.2 Selection sort

The **selection sort** improves on the bubble sort by making only one exchange for every pass through the first part of the vector. We will call this a step. In order to do this, a selection sort looks for the largest value as it makes a partial pass and, after completing the partial pass, places it in the proper location, ending the step. As with a bubble sort, after the first step, the largest item is in the correct place. After the second step, the next largest is in place. This process continues and requires $n - 1$ steps to sort n items, since the final item must be in place after the $(n - 1)$ step.

Selection Sort

On each step, the largest remaining item is selected and then placed in its proper location. If the 3 largest values are `[55, 77, 93]`, then the first pass places 93, the second pass places 77, the third places 55, and so on.

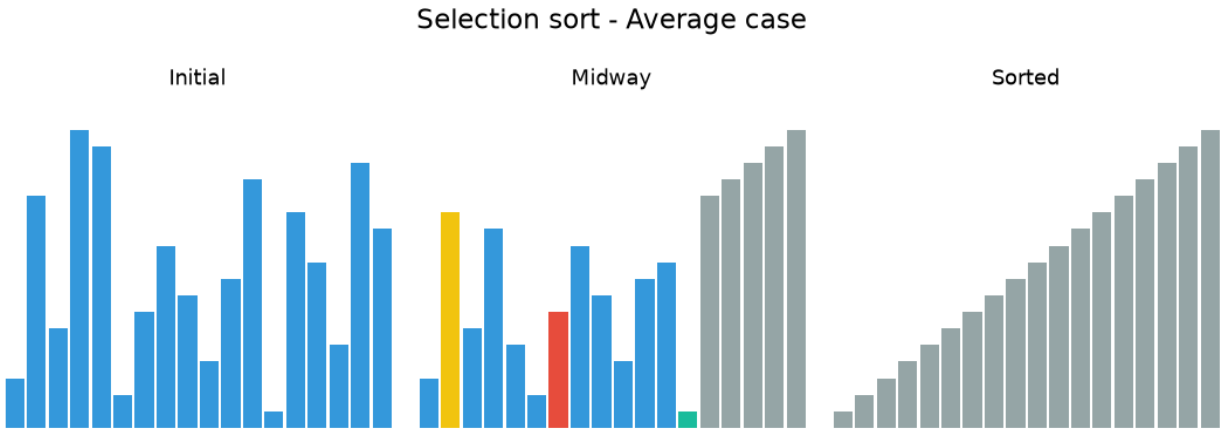
Yellow marks the current candidate, red marks the value being compared, teal marks the boundary for the current step, and gray marks values already sorted.

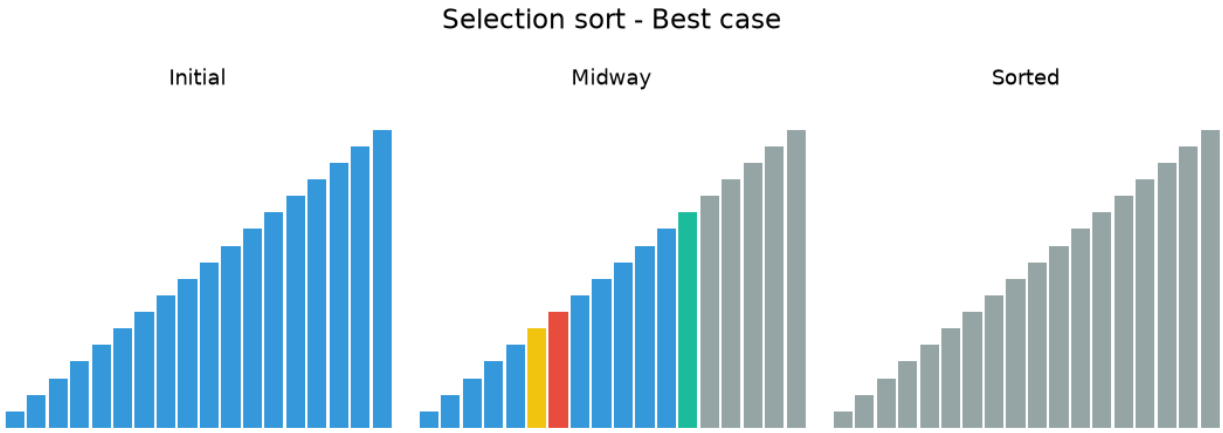
Run It

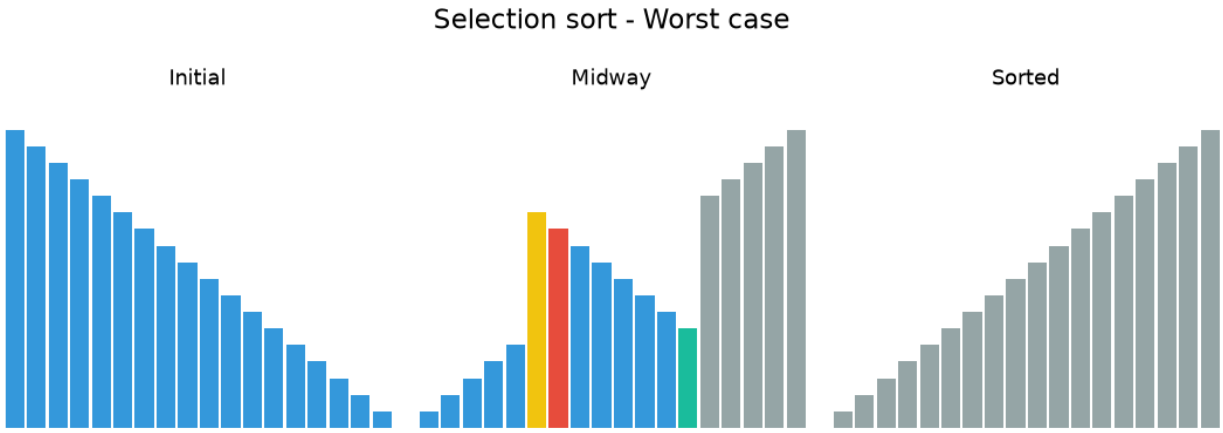
Listing 6: The Selection Sort

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

(continues on next page)


Selection sort - Average case
Initial
Midway
Sorted


Selection sort - Best case
Initial
Midway
Sorted


Selection sort - Worst case
Initial
Midway
Sorted

(continued from previous page)

```
using std::vector;

vector<int> selection_sort(vector<int> data) {
  for (int slot = data.size()-1; slot >= 0; --slot) {
    int largest = 0;  // location of the largest element
    for (int location = 1; location < slot+1; location++) {
      if (data[location] > data[largest]) {
        largest = location;
      }
    }
    std::swap(data[slot], data[largest]);
  }
  return data;
}

int main() {
  vector<int> data = {54, 26, 93, 17, 77, 31, 44, 55, 20};
  for (const auto& value: selection_sort(data)) {
    std::cout << value << ' ';
  }
  std::cout << '\n';
  return 0;
}
```

You may see that the selection sort makes the same number of comparisons as the *bubble sort* and is therefore also $O(n^2)$. Like the bubble sort, the selection sort makes no exchanges when the data is already sorted ascending. In other cases selection sort generally makes fewer exchanges than bubble sort when presented with the same data. Due to the smaller number of exchanges, the selection sort typically executes faster than bubble sort.

Self Check

Q1

Question

Suppose you have the following vector of numbers to sort: [11, 7, 12, 14, 19, 1, 6, 18, 8, 20] which vector represents the partially sorted (ascending) vector after three steps of selection sort?

- ◯ [7, 11, 12, 1, 6, 14, 8, 18, 19, 20]
- ◯ [7, 11, 12, 14, 19, 1, 6, 18, 8, 20]
- ◯ [11, 7, 12, 14, 1, 6, 8, 18, 19, 20]
- ◯ [11, 7, 12, 14, 8, 1, 6, 18, 19, 20]

More to Explore

- TBD

Acknowledgements

This section is adapted from Problem Solving with Algorithms and Data Structures using C++, by Brad Miller and David Ranum, Luther College, and Jan Pearce, Berea College released under the CC BY-NC-SA 4.0.

3.7.3 Insertion sort

What would you do if you have a stack of phone bills from the past two years and you want to order by date? A fairly natural way to handle this is to look at the first two bills and put them in order. Then take the third bill and put it into the right position with respect to the first two, and so on. As you take each bill, you would add it to the sorted pile that you have already made. This simple approach is the inspiration for our first sorting algorithm, called **Insertion Sort**.

Insertion Sort iterates through a list of records. For each iteration, the current record is inserted in turn at the correct position within a sorted list composed of those records already processed. Given an array named `data` that stores n records, one way to implement an insertion sort could be this.

```
template <class Comparable>
void insertion_sort(Comparable* data[], int n) {
  for (int i = 1; i < n; ++i) {
    for (int j = i; (j > 0) && (*data[j] < *data[j-1]); --j) {
      swap(data, j, j-1);
    }
  }
}
```

This sort is still $O(n^2)$, but works differently from the bubble sort and selection sort, since it always maintains a sorted subvector in the container. Each new item is then "inserted" back into the previous subvector such that the sorted subvector is one item larger. *Figure 4* shows the insertion sorting process. The shaded items represent the ordered subvectors as the algorithm makes each pass.

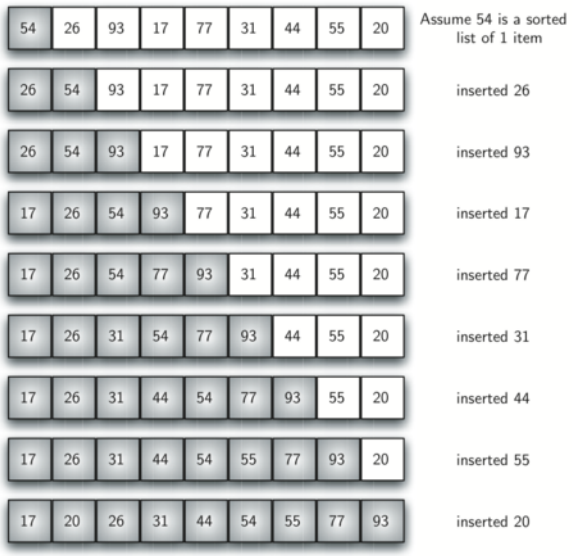


Fig. 5: Figure 4: `Insertion Sort`

We begin by assuming that a vector with one item (position 0) is already sorted. On each pass, one for each item 1 through $n - 1$, the current item is checked against those in the already sorted subvector. As we look back into the already sorted subvector, we shift those items that are greater to the right. When we reach a smaller item or the end of the subvector, the current item can be inserted.

Figure 5 shows the fifth pass in detail. At this point in the algorithm, a sorted subvector of five items consisting of 17, 26, 54, 77, and 93 exists. We want to insert 31 back into the already sorted items. The first comparison against 93 causes 93 to be shifted to the right. 77 and 54 are also shifted. When the item 26 is encountered, the shifting process stops and 31 is placed in the open position. Now we have a sorted subvector of six items.

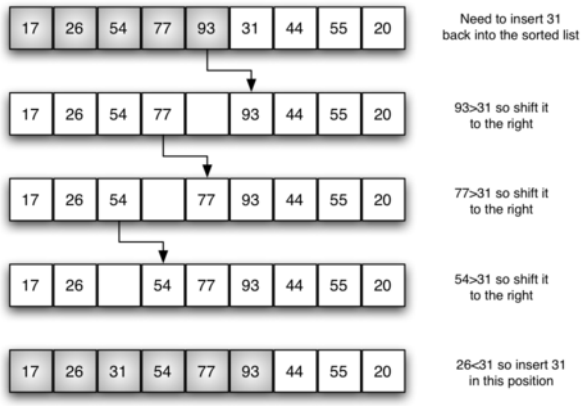
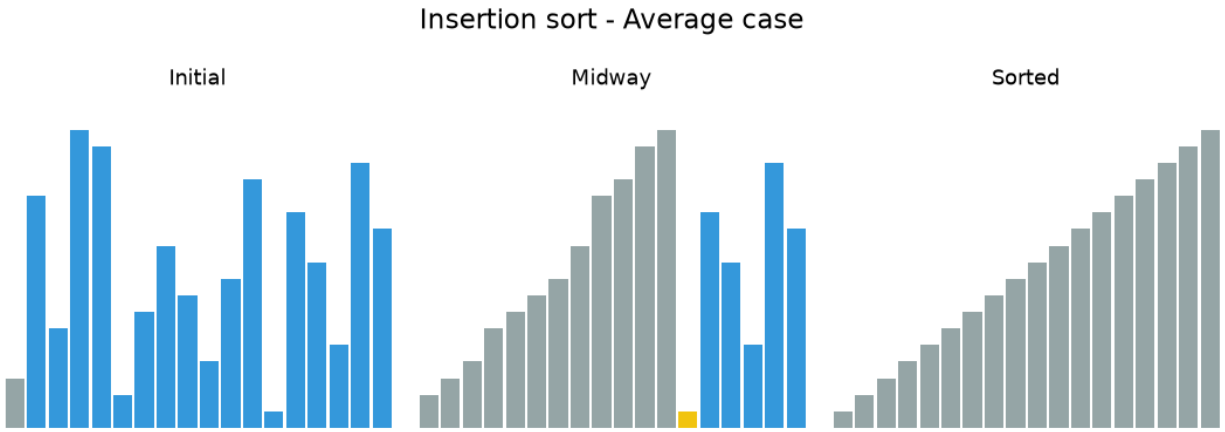


Fig. 6: Figure 5: `Insertion Sort`: Fifth Pass of the Sort

In the following animations, yellow marks the value being inserted, red marks a value being shifted, and gray marks the sorted prefix.

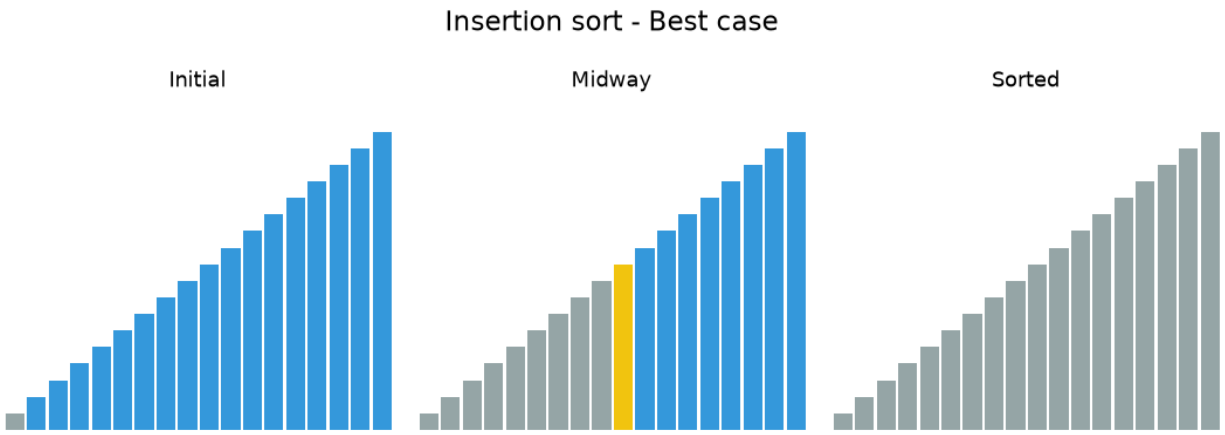


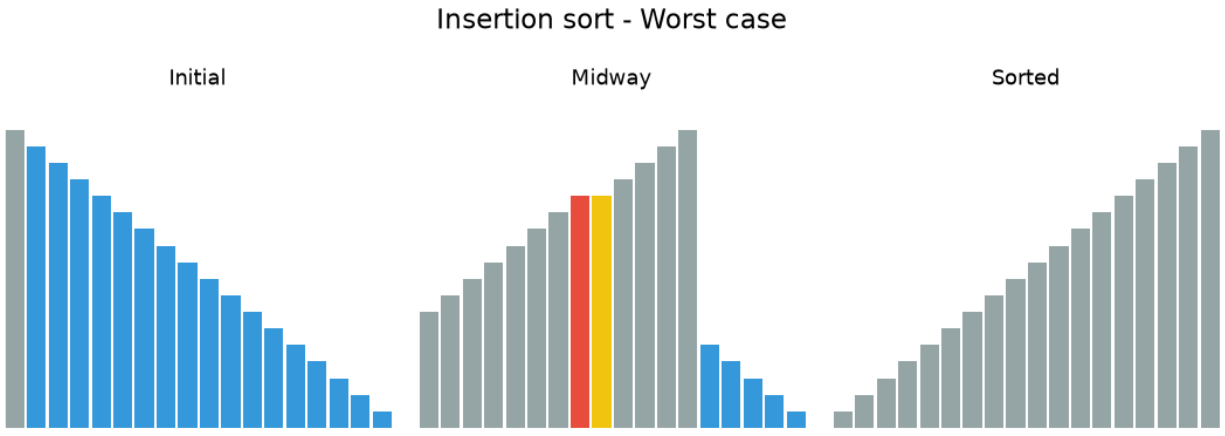
Insertion Sort

The implementation of `insertion_sort` (*Run It tab*) shows that there are again $n - 1$ passes to sort n items. The iteration starts at position 1 and moves through position $n - 1$, as these are the items that need to be inserted back into the sorted subvectors. Line 8 performs the shift operation that moves a value up one position in the vector, making room behind it for the insertion. Remember that this is not a complete exchange as was performed in the previous algorithms.

The maximum number of comparisons for an insertion sort is the sum of the first $n - 1$ integers. Again, this is $O(n^2)$. However, in the best case, only one comparison needs to be done on each pass. This would be the case for an already sorted vector.

One note about shifting versus exchanging is also important. In general, a shift operation requires approximately a third of the processing work of an exchange since only one assignment is performed. In benchmark studies, insertion sort will show very good performance.


Insertion sort - Best case
Initial
Midway
Sorted


Insertion sort - Worst case
Initial
Midway
Sorted

Run It

Listing 7: The Insertion Sort

```
#include <iostream>
#include <vector>
using std::vector;

vector<int> insertion_sort(vector<int> data) {
  for (unsigned int index=1; index<data.size(); ++index) {
    int currentvalue = data[index];
    int position = index;

    while (position>0 && data[position-1]>currentvalue) {
      data[position] = data[position-1];
      --position;
    }
    data[position] = currentvalue;
  }
  return data;
}

int main() {
  vector<int> data = {54, 26, 93, 17, 77, 31, 44, 55, 20};
  for (const auto& value: insertion_sort(data)) {
    std::cout << value << ' ';
  }
  std::cout << '\n';
  return 0;
}
```

Note

If you have not examined the previous you should. Many job interviews involve a 'whiteboard interview' that may ask you to write a bubble sort or an insertion sort.

I don't know why bubble sort in particular is such a popular question. The bubble sort:

- is not particularly easy to memorize
- is not an obvious solution to the sorting problem
- performs rather badly - it's average performance is n-squared!

STL Exchange Sorts

If sorting does get asked and you can get past these points with a future employer, you might score some points with the following examples, which:

- are far easier to memorize and explain: the actual sort is only 2 lines
- even though still n-squared, is typically better in practice than bubble sort
- demonstrates familiarity with the standard library.

Listing 8: STL Sorts

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <numeric>
#include <vector>
#include <random>

#define RandomAccessIterable typename
template <RandomAccessIterable It> void print(const It begin, const It end, const char* msg);
template <RandomAccessIterable It> void selection_sort(It begin, It end);
template <RandomAccessIterable It> void insertion_sort(It begin, It end);

int main() {
    std::vector<int> v(20);
    std::iota(v.begin(), v.end(), -10);
    auto generator = std::mt19937{std::random_device{}()};
    std::shuffle(v.begin(), v.end(), generator);

    print (v.begin(), v.end(), "before:         \t");
    selection_sort(v.begin(), v.end());
    print (v.begin(), v.end(), "selection sort: \t");

    std::shuffle(v.begin(), v.end(), generator);
    print (v.begin(), v.end(), "before:         \t");
    insertion_sort(v.begin(), v.end());
    print (v.begin(), v.end(), "insertion sort: \t");
}

template <RandomAccessIterable It>
void selection_sort(It begin, It end) {
    for (auto i = begin; i != end; ++i) {
        std::iter_swap(i, std::min_element(i, end));
        //print (begin, end, "\t");
    }
}

template <RandomAccessIterable It>
void insertion_sort(It begin, It end) {
    for (auto i = begin; i != end; ++i) {
        std::rotate(std::upper_bound(begin, i, *i), i, std::next(i));
        //print (begin, end, "\t");
    }
}

template <RandomAccessIterable It>
void print(const It begin, const It end, const char* msg) {
    // not very pretty
    // avoids hardcoded ostream_iterator<int>
    using os_iter = std::ostream_iterator<typename std::iterator_traits<It>::value_type>;

```

(continues on next page)

(continued from previous page)

```
    std::cout << msg;
    std::copy(begin, end, os_iter(std::cout, " "));
    std::cout << '\n';
}
```

One nice feature of the selection sort implemented using iter_swap and min_element is that the `min_element` function takes a custom comparator.

This means that while the default behavior is to sort ascending since the default comparator is std::less any other binary predicate operation could be passed in.

C++20 Feature

If your compiler supports C++20 ranges, then you could use the ranges versions of `min_element` and `iter_swap` and use a range-for loop.

Try This!

Modify the `selection_sort` template to take an optional comparison function that changes the sort order.

Self Check

Q1

Question

Suppose you have the following list of numbers to sort: [15, 5, 4, 18, 12, 19, 14, 10, 8, 20] which list represents the partially sorted list after three complete passes of insertion sort?

- ◯ [4, 5, 12, 15, 14, 10, 8, 18, 19, 20]
- ◯ [15, 5, 4, 10, 12, 8, 14, 18, 19, 20]
- ◯ [4, 5, 15, 18, 12, 19, 14, 10, 8, 20]
- ◯ [15, 5, 4, 18, 12, 19, 14, 8, 10, 20]

More to Explore

- Top 5 beautiful C++ algorithms
- Insertion sort

Acknowledgements

This section is adapted from Problem Solving with Algorithms and Data Structures using C++, by Brad Miller and David Ranum, Luther College, and Jan Pearce, Berea College released under the CC BY-NC-SA 4.0, and Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

3.7.4 Costs of Exchange Sorting

The running time for each of the sorts discussed so far is $\Theta(n^2)$ in the average and worst cases. The cost summary for the *Insertion Sort*, *Bubble Sort*, and *Selection Sort* in terms of their required number of comparisons and swaps in the best, average, and worst cases is shown.

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

The remaining sorting algorithms presented in this chapter are significantly better than these three under typical conditions. But before continuing on, it is instructive to investigate what makes these three sorts so slow. The crucial bottleneck is that only *adjacent* records are compared. Thus, comparisons and moves (for Insertion and Bubble Sort) are by single steps. Swapping adjacent records is called an *exchange*. Thus, these sorts are sometimes referred to as an *exchange sort*. The cost of any exchange sort can be at best the total number of steps that the records in the array must move to reach their "correct" location. Recall that this is at least the number of inversions for the record. An inversion occurs when a record with key value greater than the current record's key value appears before it.

More to Explore

- Last 3 sections:
  - *Insertion Sort*,
  - *Bubble Sort*, and
  - *Selection Sort*
- *Algorithm Analysis*

Acknowledgements

This section is adapted from Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

3.7.5 Shell sort

The **shell sort**, sometimes called the "diminishing increment sort," improves on the *insertion sort* by breaking the original vector into a number of smaller subvectors, each of which is sorted using an insertion sort. The unique way that these subvectors are chosen is the key to the shell sort. Instead of breaking the vector into subvectors of contiguous items, the shell sort uses an increment `i`, sometimes called the **gap**, to create a subvector by choosing all items that are `i` items apart.

This can be seen in *Figure 6*. This vector has nine items. If we use an increment of three, there are three subvectors, each of which can be sorted by an insertion sort. After completing these sorts, we get the vector shown in *Figure 7*. Although this vector is not completely sorted, something very interesting has happened. By sorting the subvectors, we have moved the items closer to where they actually belong.

Figure 8 shows a final insertion sort using an increment of one; in other words, a standard *insertion sort*. Note that by

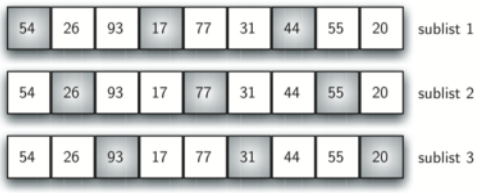


Fig. 7: Figure 6: A Shell Sort with Increments of Three

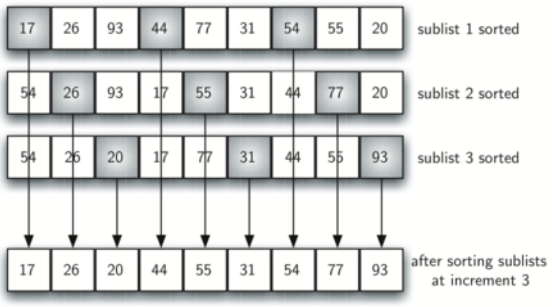


Fig. 8: Figure 7: A Shell Sort after Sorting Each subvector

performing the earlier subvector sorts, we have now reduced the total number of shifting operations necessary to put the vector in its final order. For this case, we need only four more shifts to complete the process.

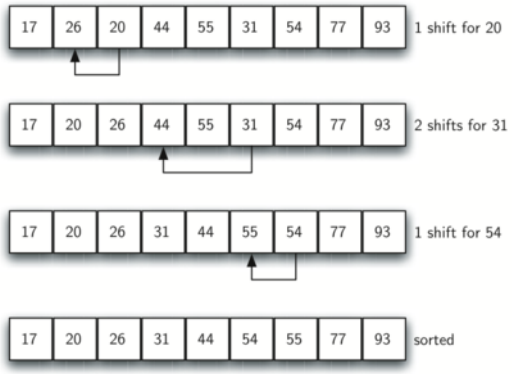


Fig. 9: Figure 8: ShellSort: A Final Insertion Sort with Increment of 1

Shell Sort

We said earlier that the way in which the increments are chosen is the unique feature of the shell sort. The function shown in *Run It tab* uses a different set of increments. In this case, we begin with $\frac{n}{2}$ subvectors. On the next pass, $\frac{n}{4}$ subvectors are sorted. Eventually, a single vector is sorted with the basic insertion sort. *Figure 9* shows the first subvectors for our example using this increment.

The following invocation of the `shell_sort` function shows the partially sorted vectors after each increment, with the final sort being an insertion sort with an increment of one.

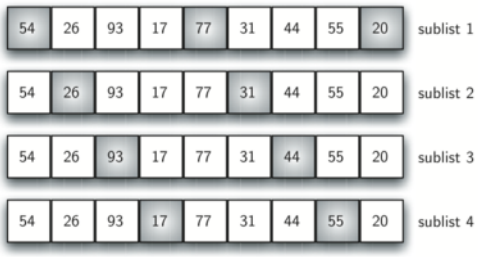


Fig. 10: Figure 9: Initial Subvectors for a Shell Sort

Run It

Listing 9: The Shell Sort

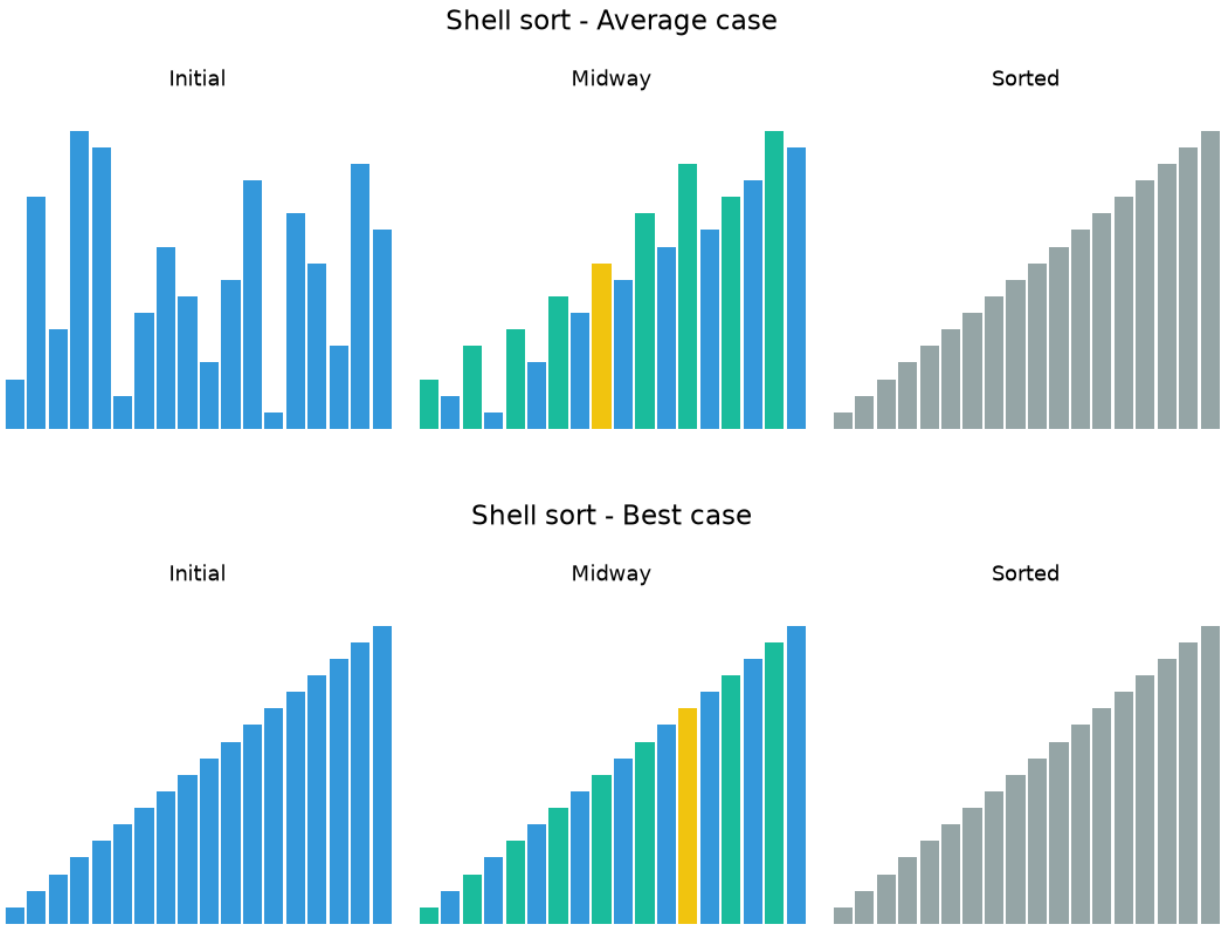
```
#include <iostream>
#include <string>
#include <vector>
using std::vector;

void print(const vector<int>& data, const std::string& comment) {
   for (const auto& value: data) {
      std::cout << value << ' ';
   }
   std::cout << comment << '\n';
}

vector<int> gap_insertion_sort(vector<int> data, int start, int gap) {
   for (int i = start+gap; i < int(data.size()); i += gap) {
      int value = data[i];
      int pos = i;

      while (pos >= gap && data[pos-gap] > value) {
         data[pos] = data[pos-gap];
         pos -= gap;
      }
      data[pos] = value;
   }
   return data;
}

vector<int> shell_sort(vector<int> data) {
   int pivot = data.size() / 2;
   while (pivot > 0) {
      for (int start = 0; start < pivot; ++start) {
         data = gap_insertion_sort(data, start, pivot);
      }
      std::string text = ": pivot = ";
      text.append(std::to_string(pivot));
      print(data, text);
      pivot /= 2;
   }
```

(continues on next page)

(continued from previous page)

```
    return data;
}

int main() {
  vector<int> data = {54, 26, 93, 17, 77, 31, 44, 55, 20};
  print(shell_sort(data), ": done");
  return 0;
}
```

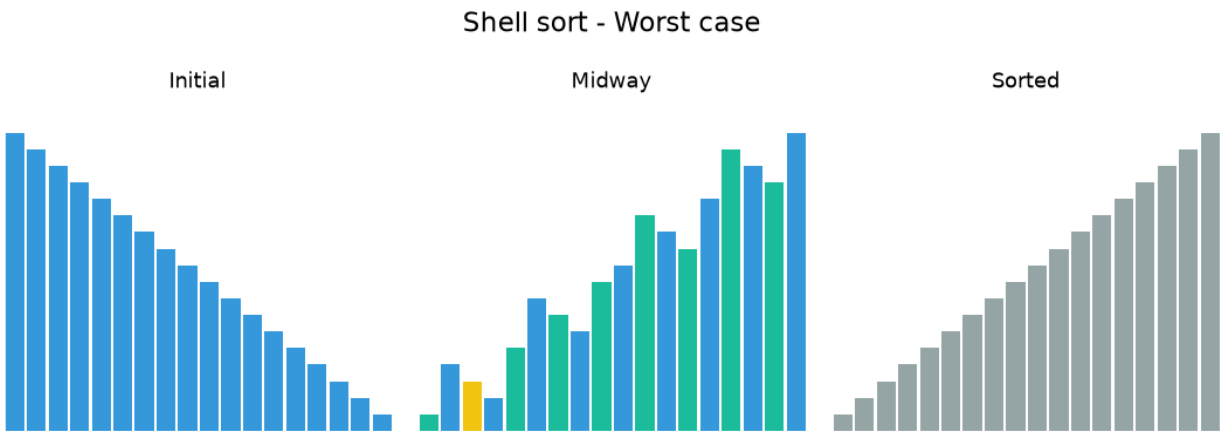
The following animations show the shell sort in action. Teal marks the current gap group, yellow marks the value being inserted, red marks a shifted comparison, and gray marks the final sorted state.



At first glance you may think that a shell sort cannot be better than an insertion sort, since it does a complete insertion sort as the last step. It turns out, however, that this final insertion sort does not need to do very many comparisons (or shifts) since the list has been pre-sorted by earlier incremental insertion sorts, as described above. In other words, each pass produces a list that is "more sorted" than the previous one. This makes the final pass very efficient.

Although a general analysis of the shell sort is well beyond the scope of this text, we can say that it tends to fall somewhere between $O(n)$ and $O(n^2)$, based on the behavior described above. For the increments shown in *Run It tab*, the performance is $O(n^2)$. By changing the increment, for example using $2^k - 1$ (1, 3, 7, 15, 31, and so on), a shell sort can perform at $O(n^{\frac{3}{2}})$.

Self Check



Q1

Question

Given the following list of numbers: [5, 16, 20, 12, 3, 8, 9, 17, 19, 7], which answer illustrates the contents of the list after all swapping is complete for a gap size of 3?

- ◯ [5, 3, 8, 7, 16, 19, 9, 17, 20, 12]
- ◯ [3, 7, 5, 8, 9, 12, 19, 16, 20, 17]
- ◯ [3, 5, 7, 8, 9, 12, 16, 17, 19, 20]
- ◯ [5, 16, 20, 3, 8, 12, 9, 17, 20, 7]

More to Explore

- TBD

Acknowledgements

This section is adapted from Problem Solving with Algorithms and Data Structures using C++, by Brad Miller and David Ranum, Luther College, and Jan Pearce, Berea College released under the CC BY-NC-SA 4.0.

3.7.6 Merge sort

We now turn our attention to using a *divide and conquer* strategy as a way to improve the performance of sorting algorithms. The first algorithm we will study is the **merge sort**. Merge sort is a recursive algorithm that continually splits a vector in half. If the vector is empty or has one item, it is sorted by definition (the base case). If the vector has more than one item, we split the vector and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a **merge**, is performed. Merging is the process of taking two smaller sorted vectors and combining them together into a single, sorted, new vector. *Figure 10* shows our familiar example vector as it is being split by `mergeSort`. *Figure 11* shows the simple vectors, now sorted, as they are merged back together.

Merge Sort

The `merge_sort` function begins by checking the base case. If the length of the vector is less than or equal to one, then we already have a sorted vector and no more processing is needed. If the length is greater than one, then we extract the left and right halves. It is important to note that the vector may not have an even number of items. That does not matter, as the lengths will differ by at most one.

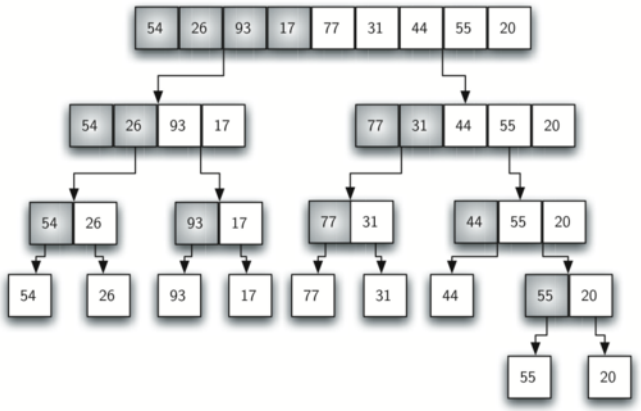


Fig. 11: Figure 10: Splitting the vector in a Merge Sort

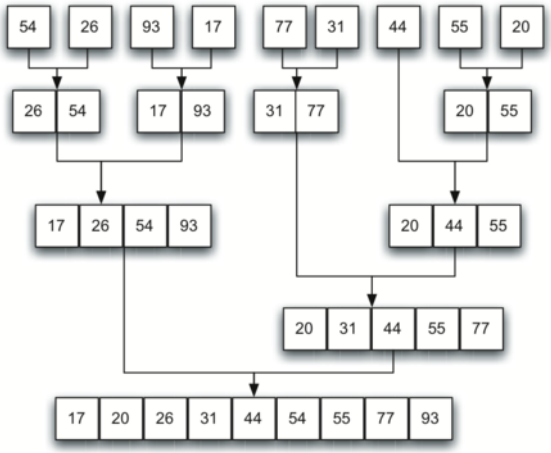


Fig. 12: Figure 11: vectors as They Are Merged Together

Once the `merge_sort` function is invoked on the left half and the right half (lines 23-24), it is assumed they are sorted. The rest of the function is responsible for merging the two smaller sorted vectors into a larger sorted vector. Notice that the merge operation places the items back into the original vector (`data`) one at a time by repeatedly taking the smallest item from the sorted vectors.

The print function shows the vector contents at the start of each invocation. There is also a print call at the end to show the merging process. The output shows the result of executing the function on our example vector.

Note that the vector with 44, 55, and 20 will not divide evenly. The first split gives [44] and the second gives [55,20]. It is easy to see how the splitting process eventually yields a vector that can be immediately merged with other sorted vectors.

Run It

Listing 10: The Merge Sort

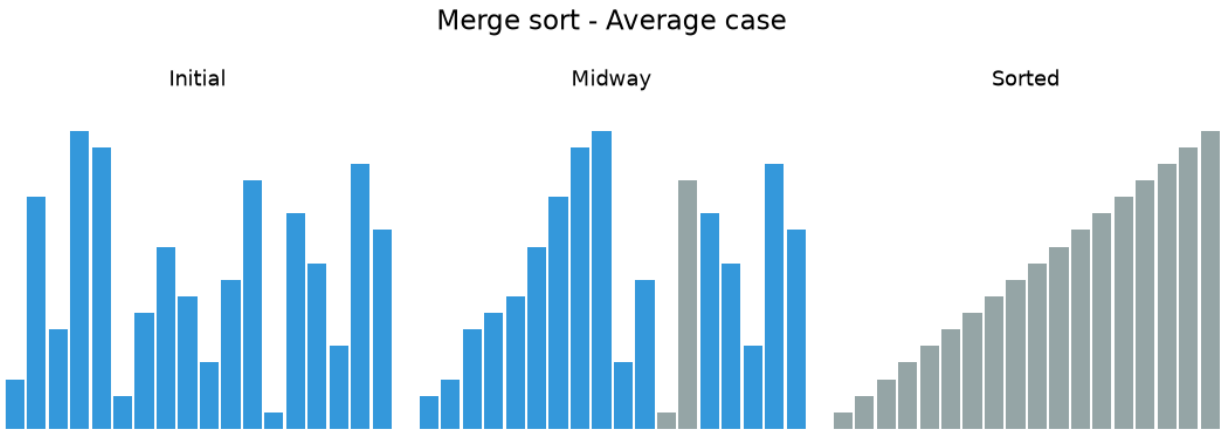
```
#include <iostream>
#include <string>
#include <vector>
using std::cout;
using std::vector;

void print(const vector<int>& data) {
   for (const auto& value: data) {
      cout << value << ' ';
   }
   cout << '\n';
}

vector<int> merge_sort(vector<int> data) {
   cout << "Splitting ";
   print(data);
   if (data.size() > 1) {
      int mid = data.size() / 2;
      // split data into 2 halves
      vector<int> left(data.begin(),data.begin()+mid);
      vector<int> right(data.begin()+mid,data.end());

      left = merge_sort(left);
      right = merge_sort(right);

      int i = 0, j = 0, k = 0;
      while (i < int(left.size()) && j < int(right.size())) {
         if(left[i] < right[j]) {
            data[k] = left[i];
            ++i;
         } else {
            data[k] = right[j];
            ++j;
         }
         ++k;
      }
      while(i < int(left.size())) {
         data[k] = left[i];
```

(continues on next page)

(continued from previous page)

```
            ++i;
            ++k;
        }
        while(j < int(right.size())) {
            data[k] = right[j];
            ++j;
            ++k;
        }
    }
    cout << "Merging ";
    print(data);

    return data;
}

int main() {
  vector<int> data = {54, 26, 93, 17, 77, 31, 44, 55, 20};
  print(merge_sort(data));
  return 0;
}
```

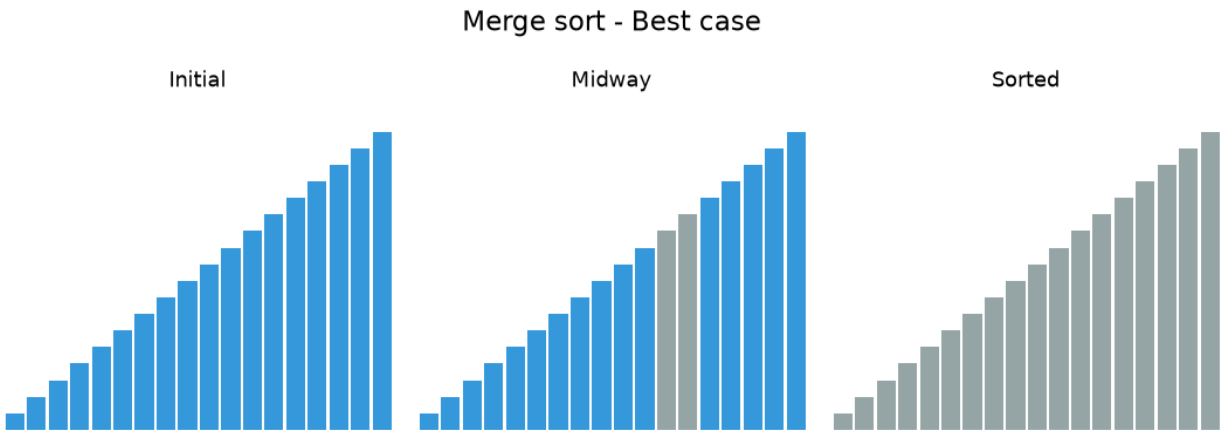
In the following animations, purple marks a range being split, teal marks the range being merged, yellow marks the target slot currently filled, and gray marks merged ranges.

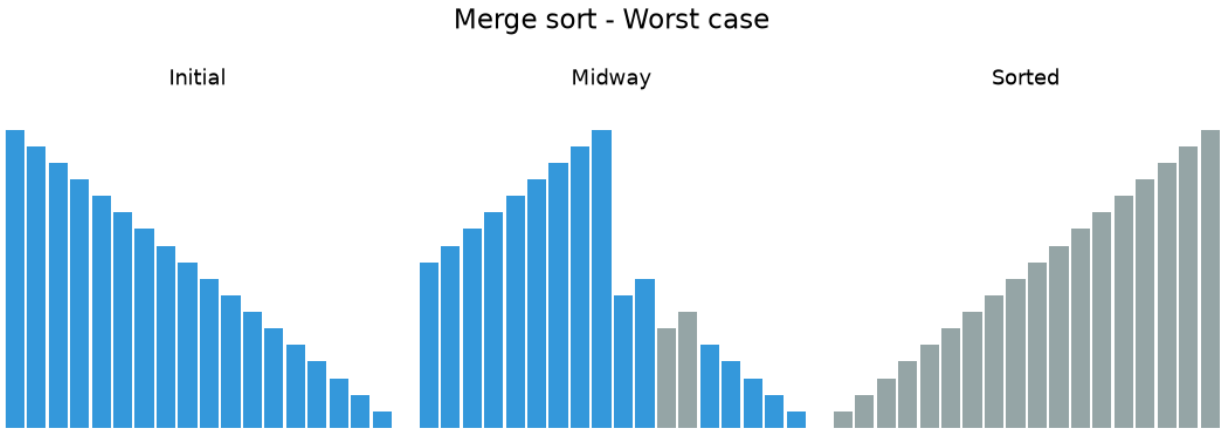


In order to analyze the `merge_sort` function, we need to consider the two distinct processes that make up its implementation. First, the vector is split into halves. We already computed (in a binary search) that we can divide a vector in half $\log n$ times where n is the length of the vector. The second process is the merge. Each item in the vector will eventually be processed and placed on the sorted vector. So the merge operation which results in a vector of size n requires n operations. The result of this analysis is that $\log n$ splits, each of which costs n for a total of $n \log n$ operations. A merge sort is an $O(n \cdot logn)$ algorithm and even better, it is also $\Omega(n \cdot logn)$ in the worst case.

Recall that the slicing operator is $O(k)$ where k is the size of the slice. In order to guarantee that `merge_sort` will be $O(n \cdot logn)$ we will need to remove the slice operator. Again, this is possible if we simply pass the starting and ending indices along with the vector when we make the recursive call. We leave this as an exercise.

It is important to notice that the `merge_sort` function requires extra space to hold the two halves as they are extracted with the slicing operations. This additional space can be a critical factor if the vector is large and can make this sort problematic when working on large data sets.


Merge sort - Best case
Initial
Midway
Sorted


Merge sort - Worst case
Initial
Midway
Sorted

Self Check

Q1

Question

Given the following list of numbers: [21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40] which answer illustrates the list to be sorted after 3 recursive calls to mergesort?

- ◯ [16, 49, 39, 27, 43, 34, 46, 40]
- ◯ [21,1]
- ◯ [21, 1, 26, 45]
- ◯ [21]

Q2

Question

Given the following list of numbers: [21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40] which answer illustrates the first two lists to be merged?

- ◯ [21, 1] and [26, 45]
- ◯ [[1, 2, 9, 21, 26, 28, 29, 45] and [16, 27, 34, 39, 40, 43, 46, 49]
- ◯ [21] and [1]
- ◯ [9] and [16]

More to Explore

- TBD

Acknowledgements

This section is adapted from Problem Solving with Algorithms and Data Structures using C++, by Brad Miller and David Ranum, Luther College, and Jan Pearce, Berea College released under the CC BY-NC-SA 4.0.

3.7.7 Quick sort

The **quick sort** uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished.

A quick sort first selects a value, which is called the **pivot value**. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort.

Figure 12 shows that 54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31. The **partition** process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.

Partitioning begins by locating two position markers - let's call them `leftmark` and `rightmark` - at the beginning and end of the remaining items in the list (positions 1 and 8 in *Figure 13*). The goal of the partition process is to move items

Fig. 13: Figure 12: The First Pivot Value for a Quick Sort

that are on the wrong side with respect to the pivot value while also converging on the split point. *Figure 13* shows this process as we locate the position of 54.

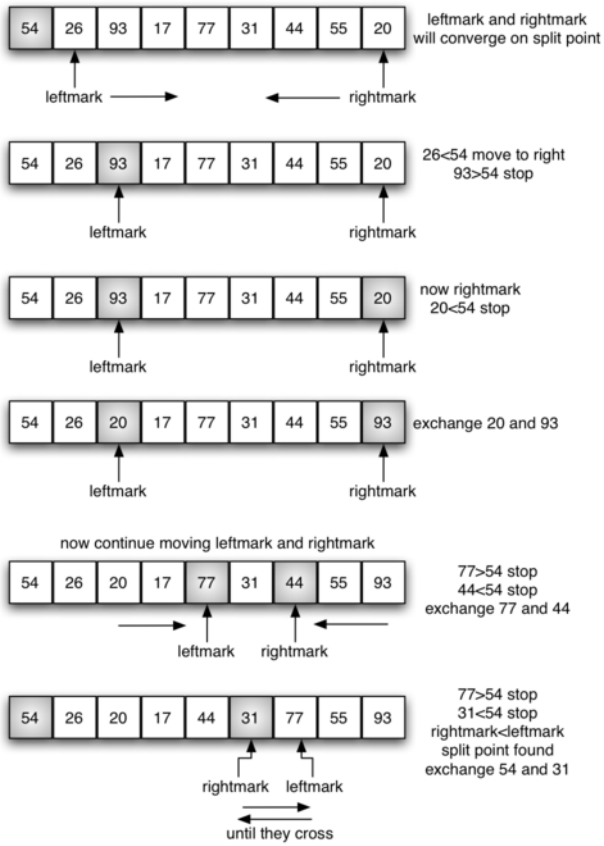


Fig. 14: Figure 13: Finding the Split Point for 54

We begin by incrementing `leftmark` until we locate a value that is greater than the pivot value. We then decrement `rightmark` until we find a value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again.

At the point where `rightmark` becomes less than `leftmark`, we stop. The position of `rightmark` is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place (*Figure 14*). In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.

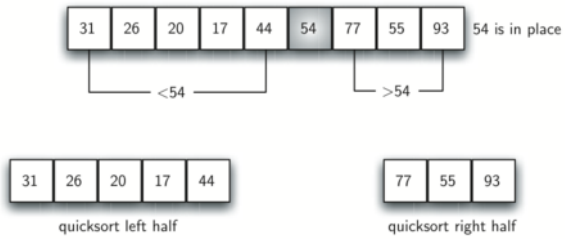


Fig. 15: Figure 14: Completing the Partition Process to Find the Split Point for 54

Quick Sort

The `partition` function implements the process described earlier. The following program sorts the list that was used in the example above.

Run It

Listing 11: The Quick Sort

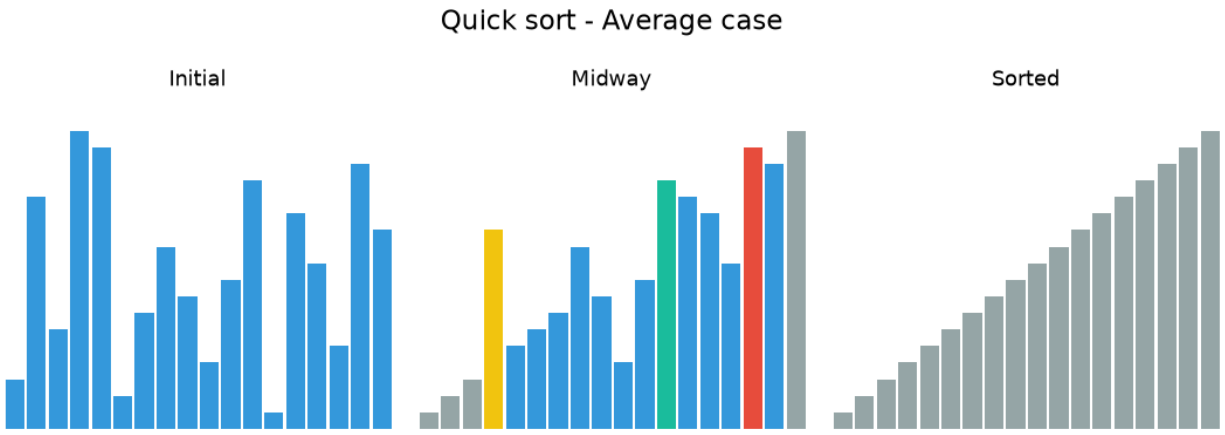
```
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using std::vector;

void print(const vector<int>& data) {
   for (const auto& value: data) {
      std::cout << value << ' ';
   }
   std::cout << '\n';
}

template <typename RandomIt>
RandomIt partition(RandomIt first, RandomIt last) {
   if (first == last) return first;

   auto pivotvalue = *first;
   RandomIt leftmark  = first + 1;
   RandomIt rightmark = last - 1;
   bool done = false;
   while (!done) {
      while(leftmark <= rightmark &&
            *leftmark <= pivotvalue) {
        ++leftmark;
      }
      while(rightmark >= leftmark &&
            *rightmark >= pivotvalue) {
        --rightmark;
      }
      if(rightmark<leftmark) {
         done = true;
      } else {
```

(continues on next page)

(continued from previous page)

```
            std::swap(*rightmark, *leftmark);
        }
    }
    std::swap(*rightmark, *first); //*
    return rightmark;
}

template <typename RandomIt>
void quick_sort(RandomIt first, RandomIt last)
{
    if (std::distance(first, last) > 1) {
        RandomIt pivot = partition(first, last);
        quick_sort(first, pivot);
        quick_sort(pivot + 1, last);
    }
}

int main() {
  vector<int> data = {54, 26, 93, 17, 77, 31, 44, 55, 20};
  quick_sort(data.begin(), data.end());
  print(data);
  return 0;
}
```

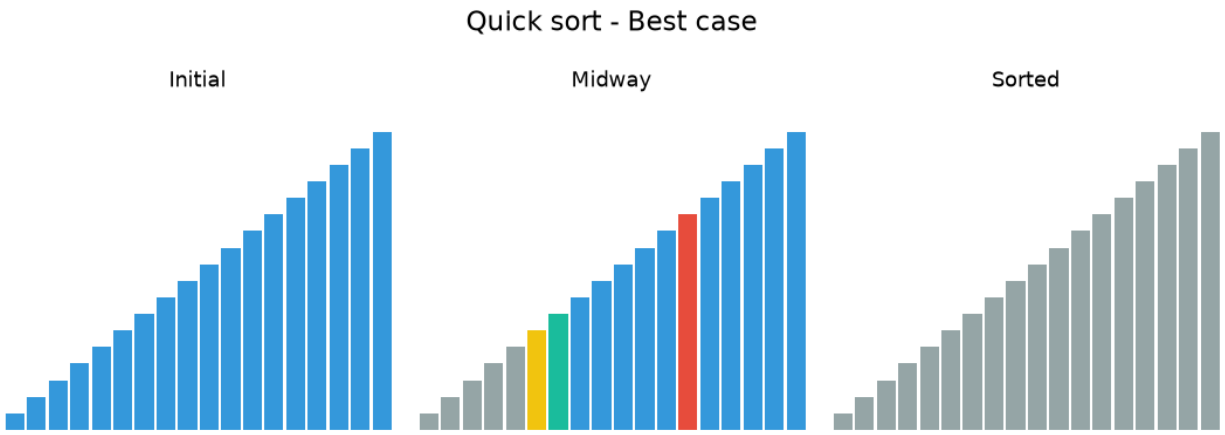
The following animations show quick sort in action. Yellow marks the current pivot, red marks values being compared or exchanged, teal marks a scan boundary, and gray marks pivot values that have been placed.

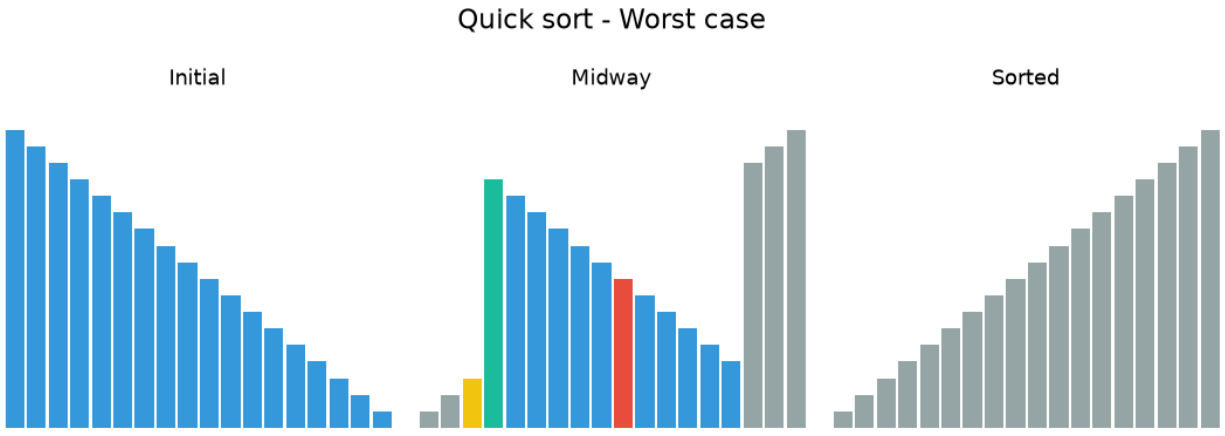


To analyze the `quick_sort` function, note that for a list of length n, if the partition always occurs in the middle of the list, there will again be $\log n$ divisions. In order to find the split point, each of the n items needs to be checked against the pivot value. Therefore, the average case complexity is $n \cdot \log n$. In addition, there is no copying of list data as in the merge sort process.

The ideal pivot selection point is the median **value** in the data set, however the cost to find the median often exceeds the benefits for many typical data sets.

Unfortunately, in the worst case, the split points may not be in the middle and can be very skewed to the left or the right, leaving a very uneven division. In this case, sorting a list of n items divides into sorting a list of 0 items and a list of $n-1$ items. Then sorting a list of $n-1$ divides into a list of size 0 and a list of size $n-2$, and so on. The result is an $O(n^2)$ sort with all of the overhead that recursion requires. This worst case example is shown in the above code


Quick sort - Best case
Initial
Midway
Sorted


Quick sort - Worst case
Initial
Midway
Sorted

example.

A recursive $O(n^2)$ algorithm makes quick sort susceptible to stack overflow errors on very large data sets.

Quick sort therefore poses an interesting dilemma. The quick sort average case is very fast. It tends to be the fastest, on average, of the known $O(n \cdot logn)$ average case sorting algorithms in actual clock time. But its worst case is just dreadful.

So the suitability of quick sort winds up coming down to a question of how often we would actually expect to encounter the worst case or *nearly* worst case behavior. That, in turn, depends upon the choice of pivot.

We mentioned earlier that there are different ways to choose the pivot value. In particular, we can attempt to alleviate some of the potential for an uneven division by using a technique called **median of three**. To choose the pivot value, we will consider the first, the middle, and the last element in the list. In our example, those are 54, 77, and 20. Now pick the median value, in our case 54, and use it for the pivot value (of course, that was the pivot value we used originally). The idea is that in the case where the first item in the list does not belong toward the middle of the list, the median of three will choose a better "middle" value. This will be particularly useful when the original list is somewhat sorted to begin with. We leave the implementation of this pivot value selection as an exercise.

Self Check

Q1

Question

Given the following list of numbers [14, 17, 13, 15, 19, 10, 3, 16, 9, 12] which answer shows the contents of the list after the second partitioning according to the quicksort algorithm?

- ◯ [9, 3, 10, 13, 12]
- ◯ [9, 3, 10, 13, 12, 14]
- ◯ [9, 3, 10, 13, 12, 14, 17, 16, 15, 19]
- ◯ [9, 3, 10, 13, 12, 14, 19, 16, 15, 17]

Q2

Question

Given the following list of numbers [1, 20, 11, 5, 2, 9, 16, 14, 13, 19] what would be the first pivot value using the median of 3 method?

- ◯ 1
- ◯ 9
- ◯ 16
- ◯ 19

Q3

Question

Which of the following sort algorithms are guaranteed to be O(n log n) even in the worst case?

- ◯ Shell Sort
- ◯ Quick Sort
- ◯ Merge Sort
- ◯ Insertion Sort

Q4

Matching question

Match each sorting method with its appropriate estimated comparisons.

A. Insertion/Bubble/Merge	___ between O(n) and O(n^2)
B. Merge Sort	___ O(n log n)
C. Quick Sort	___ O(n log n) or O(n^2)
D. Shell Sort	___ O(n^2)

Q5

Question

Which sort should you use for best efficiency if you need to sort through 100,000 random items in a list?

- ◯ Merge
- ◯ Selection
- ◯ Bubble
- ◯ Insertion

More to Explore

- TBD

Acknowledgements

This section is adapted from Problem Solving with Algorithms and Data Structures using C++, by Brad Miller and David Ranum, Luther College, and Jan Pearce, Berea College released under the CC BY-NC-SA 4.0.

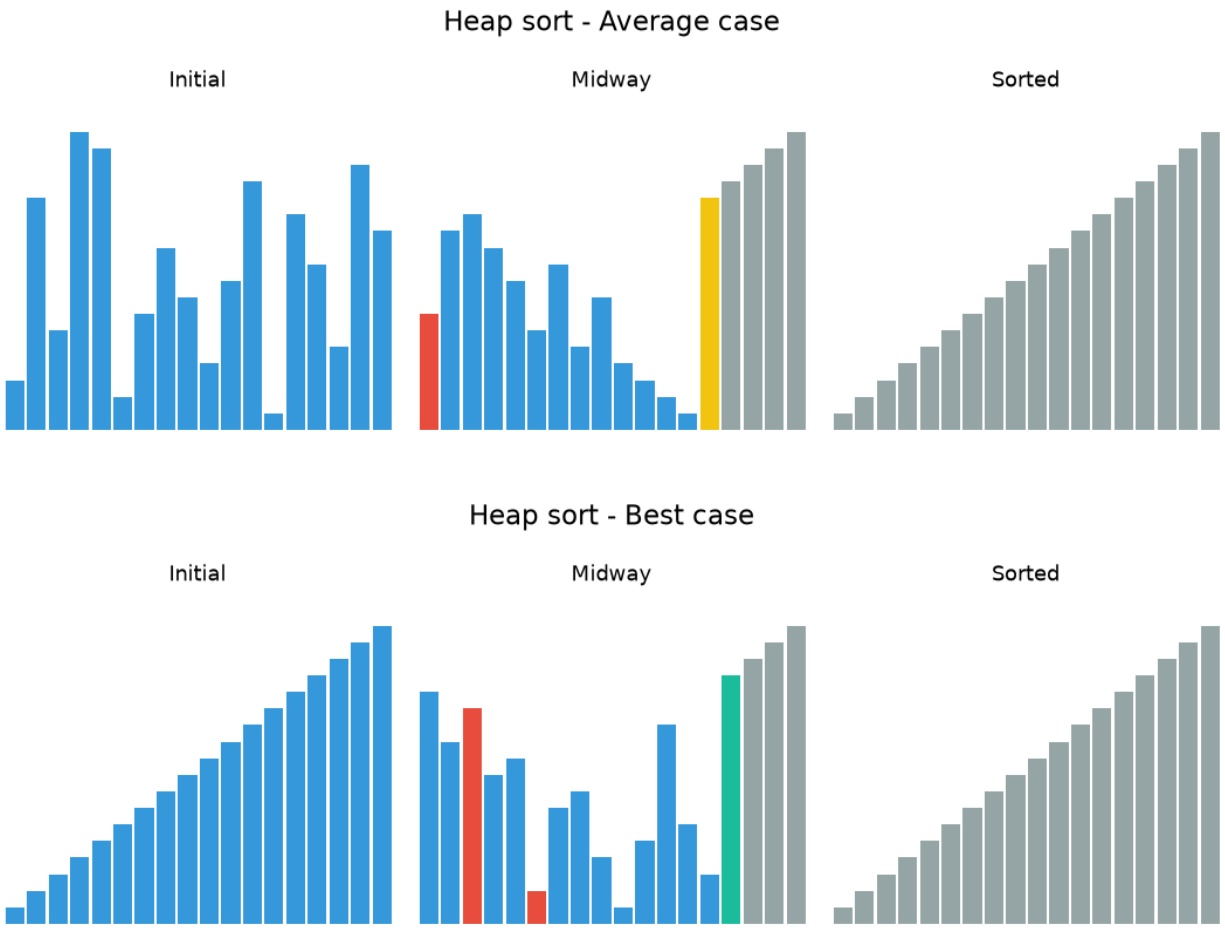
3.7.8 Heap sort

Our discussion of Quicksort began by considering the practicality of using a BST for sorting. The BST requires more space than the other sorting methods and will be slower than Quicksort or Mergesort due to the relative expense of inserting values into the tree. There is also the possibility that the BST might be unbalanced, leading to a $\Theta(n^2)$ worst-case running time. Subtree balance in the BST is closely related to Quicksort's partition step. Quicksort's pivot serves roughly the same purpose as the BST root value in that the left partition (subtree) stores values less than the pivot (root) value, while the right partition (subtree) stores values greater than or equal to the pivot (root).

A good sorting algorithm can be devised based on a tree structure more suited to the purpose. In particular, we would like the tree to be balanced, space efficient, and fast. The algorithm should take advantage of the fact that sorting is a special-purpose application in that all of the values to be stored are available at the start. This means that we do not necessarily need to insert one value at a time into the tree structure.

Heapsort is based on the heap data structure. Heapsort has all of the advantages just listed. The complete binary tree is balanced, its array representation is space efficient, and we can load all values into the tree at once, taking advantage of the efficient `buildheap` function. The asymptotic performance of Heapsort when all of the records have unique key values is $\Theta(n \log n)$ in the best, average, and worst cases. It is not as fast as Quicksort in the average case (by a constant factor), but Heapsort has special properties that will make it particularly useful for external sorting algorithms - used when sorting data sets too large to fit in main memory.

The following animations show heap sort in action. Yellow marks the active root or subtree parent, red marks compared or exchanged values, teal marks the active heap boundary, purple marks the completed max heap, and gray marks the sorted suffix.



More to Explore

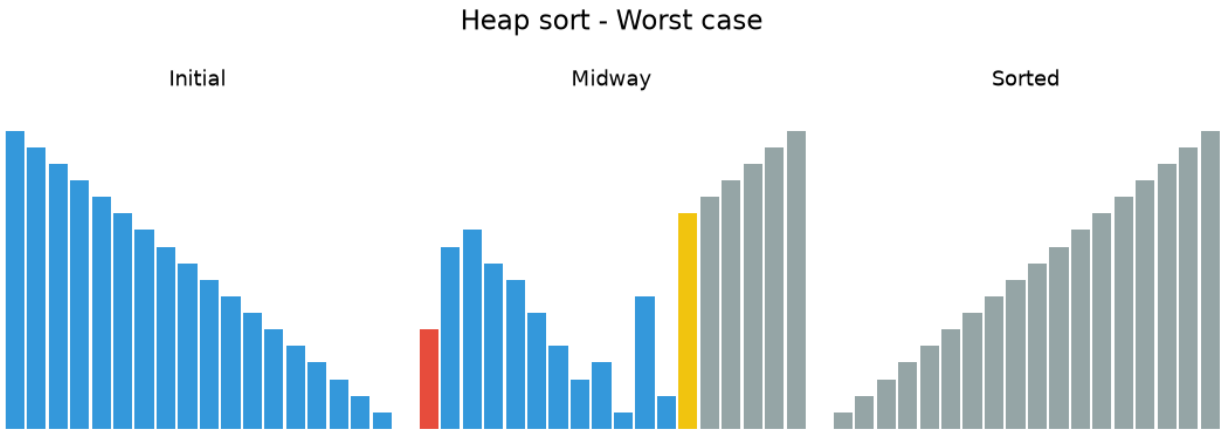
- TBD

Acknowledgements

This section is adapted from Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

3.7.9 Radix sort

Consider a sequence of records with keys in the range 0 to 99. If we have ten bins available, we can first assign records to bins by taking their key value modulo 10. Thus, every key will be assigned to the bin matching its rightmost decimal digit. We can then take these records from the bins **in order**, and reassign them to the bins on the basis of their leftmost (10's place) digit. We will define values in the range 0 to 9 to have a leftmost digit of 0. In other words, assign the i'th record from array `A` to a bin using the formula `A[i]/10`. If we now gather the values from the bins **in order**, the result is a sorted list.



In this example, we have $r = 10$ bins and key values in the range 0 to $r^2 - 1$. The total computation is $\Theta(n)$, because we look at each record and each bin a constant number of times. This is a great improvement over the simple Binsort where the number of bins must be as large as the key range. Note that the example uses $r = 10$ so as to make the bin computations easy to visualize: Records were placed into bins based on the value of first the rightmost and then the leftmost decimal digits. Any number of bins would have worked if we interpret the key values in terms of the corresponding base. This is an example of a *Radix Sort*, so called because the bin computations are based on the *radix* or the *base* of the key values. This sorting algorithm can be extended to any number of keys in any key range. We simply assign records to bins based on the keys' digit values working from the rightmost digit to the leftmost. If there are k digits, then this requires that we assign keys to bins k times.

Array-based Radix Sort

As with *Merge sort*, an efficient implementation of Radix Sort is somewhat difficult to achieve. In particular, we would prefer to sort an array of values and avoid processing linked lists. If we knew how many values would be in each bin, then an auxiliary array of size r can be used to define these lengths and guide us to were each one starts in the output array. For example, if during the first pass the 0 bin will receive three records and the 1 bin will receive five records, then we could simply reserve the first three array positions for the 0 bin and the next five array positions for the 1 bin. Exactly this approach is taken by the following implementation. At the end of each pass, the records are copied back to the original array.

```
static void radixsort(int A[], int k, int r, int n) {
  int B[n];
  int count[r];
  int i, j, rtok;

  for (i = 0, rtok = 1; i < k; i++, rtok *= r) {  // For k digits
    for (j = 0; j < r; j++) count[j] = 0;  // Initialize count

    // Count the number of records for each bin on this pass
    for (j = 0; j < n; j++) count[(A[j]/rtok)%r]++;

    // count[j] will be index in B for last slot of bin j.
    // First, reduce count[0] because indexing starts at 0, not 1
    count[0] = count[0] - 1;
    for (j = 1; j < r; j++) count[j] = count[j-1] + count[j];

    // Put records into bins, working from bottom of bin
    // Since bins fill from bottom, j counts downwards
```

(continues on next page)

(continued from previous page)

```
    for (j  = n-1; j >= 0; j--) {
      B[count[(A[j]/rtok)%r]] = A[j];
      count[(A[j]/rtok)%r] = count[(A[j]/rtok)%r] - 1;
    }
    for (j = 0; j < n; j++) A[j] = B[j];  // Copy B back
  }
}
```

The first inner `for` loop initializes array `count`. The second loop counts the number of records to be assigned to each bin. The third loop sets the values in `count` to their proper indices within array `B`. Note that the index stored in `count[j]` is the *last* index for bin `j`; bins are filled from high index to low index. The fourth loop assigns the records to the bins (within array `B`). The final loop simply copies the records back to array `A` to be ready for the next pass. Variable `rtoi` stores r^i for use in bin computation on the i'th iteration.

Radix Sort Analysis

Is it really a reasonable assumption to treat k as a constant? Or is there some relationship between k and n? If the key range is limited and duplicate key values are common, there might be no relationship between k and n. To make this distinction more clear, use N to denote the number of distinct key values used by the n records. Thus, $N \leq n$. Because it takes a minimum of $\log_r N$ base r digits to represent N distinct key values, we know that $k \geq \log_r N$.

Now, consider the situation in which no keys are duplicated. If there are n unique keys then $n = N$. It would require n distinct values to represent them. So now it takes a minimum of $\log_r n$ base r digits to represent the n distinct key values. This means that $k \geq \log_r n$. Because it requires *at least* $\log n$ digits to distinguish between the n distinct keys (within a constant factor -- meaning, the number of digits is $\Omega(\log n)$), k is in $\Omega(\log n)$. **This means that Radix Sort requires $\Omega(n \log n)$ time to process n distinct key values.**

Of course the key range could be much bigger $\log_r n$ bits is merely the best case possible for n distinct values. Thus, the $\log_r n$ estimate for k could be overly optimistic. The bottom line of this analysis is that, for the general case of n distinct key values, Radix Sort is at best a $\Omega(n \log n)$ sorting algorithm.

Radix Sort's running time can be much improved (by a constant factor) if we make base r be as large as possible. This is simplest if we think about integer key values. Set $r = 2^i$ for some i. In other words, the value of r is related to the number of bits of the key processed on each pass. Each time the number of bits is doubled, the number of passes is cut in half. When processing an integer key value, setting $r = 256$ allows the key to be processed one byte at a time. Processing a 32-bit integer key requires only four passes. It is not unreasonable on most computers to use $r = 2^{16} = 64$K, resulting in only two passes for a 32-bit key. Of course, this requires a `count` array of size 64K. Performance will be good only if the number of records is about 64K or greater. In other words, the number of records must be large compared to the key size for Radix Sort to be efficient. In many sorting applications, Radix Sort can be tuned in this way to give better performance.

Radix Sort depends on the ability to make a fixed number of multiway choices based on a digit value, as well as random access to the bins. Thus, Radix Sort might be difficult to implement for certain key types. For example, if the keys are real numbers or arbitrary length strings, then some care will be necessary in implementation. In particular, Radix Sort will need to be careful about deciding when the "last digit" has been found to distinguish among real numbers, or the last character in variable length strings.

More to Explore

- TBD

Acknowledgements

This section is adapted from Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License.

3.7.10 Summary of sorting algorithms

As a convenience, a summary of the key characteristics of the sorts discussed in this chapter are presented in the following table.

Sort	Best Case	Average Case	Worst Case
Comparisons:			
Bubble	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Heap	$O(n \log(n))$	$O(n \log(n))$	$2n \log(n) + O(n)$
Insertion	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Merge	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$\Omega(n \log(n))$
Quick	$\Theta(n)$	$\Theta(n \log(n))$	$\Theta(n^2)$
Radix	$\Omega(n \log(n))$	$\cdots$	$\Omega(kn)$
Selection	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Shell	$O(n \log(n))$	$\cdots$	$O(n^2)$
Swaps:			
Bubble	0	$\Theta(n^2)$	$\Theta(n^2)$
Heap	$\cdots$	$\cdots$	$2n \log(n) + O(nn))$
Insertion	0	$\Theta(n^2)$	$\Theta(n^2)$
Quick	0	$\Theta(n \log(n))$	$\Theta(n^2)$
Radix	k	$\Theta(d(n+k))$	$\Theta(d(n+k))$
Selection	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

More to Explore

- TBD

There are many, many sorting algorithms that have been developed and analyzed. This suggests that sorting is an important area of study in computer science. Sorting a large number of items can take a substantial amount of computing resources. Like searching, the efficiency of a sorting algorithm is related to the number of items being processed. For small collections, a complex sorting method may be more trouble than it's worth. The overhead may be too high. On the other hand, for larger collections, we want to take advantage of as many improvements as possible. In this section we will discuss several sorting techniques and compare them with respect to their running time.

Before getting into specific algorithms, we should think about the operations that can be used to analyze a sorting process. First, it will be necessary to compare two values to see which is smaller (or larger). In order to sort a collection, it will be necessary to have some systematic way to compare values to see if they are out of order. The total number of comparisons will be the most common way to measure a sort procedure. Second, when values are not in the correct position with respect to one another, it may be necessary to exchange them. This exchange is a costly operation and the total number of exchanges will also be important for evaluating the overall efficiency of the algorithm.

Although there are many kinds of sorts, many are *comparison sorts*. As the name implies, comparison sorts compare values and reorders them until the data is sorted. The details on how comparisons are made and how elements are reordered vary. Most fall into one of two categories:

- Exchange sorts: find elements out of order and swap them.
- Partition sorts: split a large set into smaller sets and then reassemble them in order.

Now that we have explored both iterative and recursive functions we have the tools we need to analyze and understand sort algorithms.

First up: the exchange sorts.

3.8 Introduction to classes

This chapter introduces the class type and the basic concepts of object oriented programming. Discussion of classes will be spread over several weeks and this chapter is only an introduction.

3.8.1 Classes

Up until now, we have emphasized *procedural* programming; in which programs are composed of *procedures*. The program executes procedures (functions) one at a time, placing unfinished functions on the call stack, if needed, working towards a desired end state. Functions usually exist independently from each other. C++ does provide some tools to make working with free functions easier. Functions can be grouped in a namespace declaration, or in a single compilation unit. Function overloads allow functions that perform the same task, but on different types to be given the same name.

When procedural programs run, data is passed around or returned from functions. In the late 1960's, concerns about the cost, reliability, and maintainability of large software systems manifested in what was called a 'software crisis'.

Very large programs comprised of many thousands of functions are inherently complex. Simply managing the names of functions in a very large program can be burdensome. When every function is effectively public, Then every function has access to every other function, even if there is no good reason for public visibility.

How to solve this problem? *Abstraction*. Humans deal with complexity by abstracting details away. For example: driving a car doesn't require knowledge of the internal combustion engine; it's enough to think of a car as simple transport.

There is nothing special about forming abstractions, people do it every day. However, in object-oriented programming this is a main focus of activity and organization. Object-oriented (OO) programming takes advantage of the natural human tendency to work with real world abstractions. In OO:

- We define abstractions and call them *classes*
- When we use our classes, then we call them *objects*

In contrast to *procedural* programming, *object-oriented programming* allows programmers to hide data and functions within a *class*.

A class also effectively defines a *type*. In other words, when you create a new class, You are creating a new *type*. Most classes define both:

- A meaningful representation of data

and

- The operations associated with the data

Together, the data and operations form an *abstract data type*. Since it is you inventing the type, you also see the term *user-defined type*. Class, abstract data type, and user-defined type are all synonyms for the same thing.

Don't let the new terms scare you. You have been working with classes since chapter 2. The types string, and all the sequential and associative containers, are classes. When designed correctly, a class can be manipulated with the same operations as the built-in types. Any class you create allows you to define a new type that works with the rest of C++ and the STL as if it were built into C++.

While many classes manage their own data, just as many classes store no data at all. This is normal. Because the data in a class is normally hidden, there is no way to know if a class even **has** data without looking at the source code.

Part of the beauty of the object-oriented abstraction is that you, the user of the class do not need to care. We use classes in the same way we use any built-in type: to get something specific done in a clear, efficient manner.

The basic structure of a class

Create a new class using either the `class` or `struct` keywords. The general (simplified) format is:

```
class class-name {
  public:
     // publicly visible class members
  private:
     // hidden class members
};
```

We will be adding to this basic structure over the next several chapters.

In C++, a `struct` **is a** `class` with default public access:

Example

This program works because everything in a `struct` is publicly visible to any other component in the program.

```
#include <cstdio>

struct Talk {
  void hello() {
    std::puts("Hello, world!");
  }
};

int main() {
  Talk say;     // Create an object from a class
  say.hello();  // Call a function in the object
}
```

Note

Two things:

1. Notice the trailing semi-colon (`;`) after the definition of `struct Talk`?

   This is required for a `class` or `struct` to compile and forgetting to include a trailing `;` is a common source of error.

2. The function `main` accesses the function `hello` using the member access operator.

   The general format is *object name* . *member name*.

Run It

```
#include <cstdio>

struct Talk {
  void hello() {
    std::puts("Hello, world!");
  }
};

```

(continues on next page)

(continued from previous page)

```
int main() {
  Talk say;     // Create an object from a class
  say.hello();  // Call a function in the object
}
```

Here we have a very similar program, but with one important change.

Example

This program does not compile because everything in a `class` is **private** by default. Only objects of type `Talk` may use or even know about its private data.

```
#include <cstdio>

class Talk {
  void hello() {
    std::puts("Hello, world!");
  }
};

int main() {
  Talk say;     // Create an object from a class
  say.hello();  // Call a function in the object
}
```

Run It

```
#include <cstdio>

class Talk {
  void hello() {
    std::puts("Hello, world!");
  }
};

int main() {
  Talk say;     // Create an object from a class
  say.hello();  // Call a function in the object
}
```

We can fix our broken `class Talk` by adding `public:` to the class. The access specifiers *public* and *private* are used in a class or a struct to control what parts of the class may be accessed from outside the class.

```
class Example {
  public:             // all declarations after this point are public
    void add(int x) { // member "add" has public access
      n += x;         // OK: private Example::n can be accessed from Example::add
    }
  private:            // all declarations after this point are private
    int n = 0;        // member "n" is private
};
```

Try this!

- Add an access specifier to `class Talk` so that it compiles and runs.
- Add an access specifier to `struct Talk` so that it fails with a similar error as `class Talk` did before you modified it.

More to Explore

- Motivation for OO
- From cppreference.com
  - Access specifiers.
- From C++ Core Guidelines
  - C.1: Organize related data into structures
  - C.2: Use class if the class has an invariant
  - C.3: Represent the distinction between an interface and an implementation using a class
  - C.8: Use class rather than struct if any member is non-public

3.8.2 Constructors

When we allocate storage for a built-in type, we use the name of the type and a name for the variable:

```
int x;
```

As we saw in the previous section, the syntax for user-defined types is the same:

```
Talk say;
```

User-defined types use special functions to *construct* an object using the class definition. We call these special functions *class constructors*. Constructors are very similar to regular free functions, but there are a few special rules:

- A constructor is a **class member** function
- The function name is the **same name** as the class name
- A constructor has **no return type**

All classes must have at least 1 constructor. If you do not write one, then compiler will try to create it automatically. The `Talk` class works because it used an automatically defined default constructor generated by the compiler.

Example

```
#include <cstdio>

struct Talk {
  void hello() {
    std::puts("Hello, world!");
  }
};
```

(continues on next page)

(continued from previous page)

```
int main() {
  Talk say;     // Create an object from a class
  say.hello();  // Call a function in the object
}
```

Run It

```
#include <cstdio>

struct Talk {
  void hello() {
    std::puts("Hello, world!");
  }
};

int main() {
  Talk say;     // Create an object from a class
  say.hello();  // Call a function in the object
}
```

Although we did not need to create a constructor for the `Talk` class, we could have:

Example

This constructor doesn't change how our code functions in any way. The compiler would have generated the code `Talk() {};` for us.

```
#include <cstdio>

struct Talk {
  Talk() {}        // default constructor

  void hello() {
    std::puts("Hello, world!");
  }
};

int main() {
  Talk say;     // Create an object from a class
  say.hello();  // Call a function in the object
}
```

In C++11, we can instruct the compiler to create the default constructor for us with

```
Talk() = default;
```

Run It

```
#include <cstdio>

struct Talk {
```

(continues on next page)

(continued from previous page)

```
  Talk() {}        // default constructor

  void hello() {
    std::puts("Hello, world!");
  }
};

int main() {
  Talk say;     // Create an object from a class
  say.hello();  // Call a function in the object
}
```

Object life cycle

In C++, objects get created, used, and destroyed. Constructors, assignment functions, and destructors control the life cycle of objects: creation, copy, move, and destruction.

Like the default constructor, C++ defines default operations for other parts of an object life cycle. The default operations are a set of related operations that together implement the life cycle semantics of an object.

The list of default operations since C++11 is:

Operation	Function Signature
default constructor	`X()`
copy constructor	`X(const X&)`
copy assignment	`operator=(const X&)`
move constructor	`X(X&&)`
move assignment	`operator=(X&&)`
destructor	`~X()`

We will be discussing these operations more over the next several chapters.

Let's improve our `Talk` class by making it possible to say more than one thing and to set the default text in a constructor:

Example

```
#include <cstdio>
#include <string>

using std::string;

struct Talk {
  string text_;               // a variable to store what we want to say

  Talk()                      // a new default constructor
    : text_ ("Hello, world!")
  { }

  Talk(const string& value)    // a one argument constructor
    : text_ (value)
  { }
```

(continues on next page)

(continued from previous page)

```
  void text() {
    std::puts(text_.c_str());
  }
};

int main() {
  Talk say;     // Create the default object
  say.text();

  say.text_ = "Something else";
  say.text();

  // Create a non-default object
  Talk talk("The 80's were a long time ago.");
  talk.text();
}
```

Note that we also changed the name of our function from `hello` to `text`. Our old function name is no longer very appropriate since we can say more things than just *Hello, world!*.

Run It

```
#include <cstdio>
#include <string>

using std::string;

struct Talk {
  string text_;               // a variable to store what we want to say

  Talk()                      // a new default constructor
    : text_ ("Hello, world!")
  { }

  Talk(const string& value)    // a one argument constructor
    : text_ (value)
  { }

  void text() {
    std::puts(text_.c_str());
  }
};

int main() {
  Talk say;     // Create the default object
  say.text();

  say.text_ = "Something else";
  say.text();

  // Create a non-default object
  Talk talk("The 80's were a long time ago.");
```

(continues on next page)

(continued from previous page)

```
  talk.text();
}
```

Since C++11, this syntax for constructors:

```
Talk(const string& value)
  : text_ (value)
{ }
```

is preferred.

This is called *initializer list syntax*. The general format is:

```
ClassName(arguments)
  : class_member1 {expression1},
    class_member2 {expression2},
    class_memberN {expressionN} . . .
{ }
```

Initializer list expressions can be surrounded by `( )` or `{ }`.

Prior to C++11, standard function syntax was used. It is still allowed, but initializer list syntax is preferred.

```
ClassName(arguments) {
  class_member1 = expression1;
  class_member2 = expression2;
  class_memberN = expressionN;
}
```

In C++11, it is also permissible to initialize class members with constants directly in the class when declared. In-class initialization is preferred because it makes it explicit that the same value is expected to be used in all constructors, avoids repetition, and avoids maintenance problems. It leads to the shortest and most efficient code. Consider the following:

```
class BadInit {
    int i;
    string s;
    int j;
  public:
    BadInit() :i{666}, s{"nothing"} { }  // j is uninitialized
    BadInit(int value) :i{value} {}      // s is "" and j is uninitialized
    // ...
};
```

How would a maintainer know whether `j` was deliberately uninitialized (probably a poor idea anyway) and whether it was intentional to give `s` the default value `""` in one case and `nothing` in another? This is almost always a bug. Forgetting to initialize a member often happens when a new member is added to an existing class.

All these problems are easily fixed with in-class initializers:

```
class OkInit {
    int i {666};
    string s {"ok"};
    int j {0};
```

(continues on next page)

(continued from previous page)

```
  public:
    OkInit() = default;              // all members initialized to default values
    OkInit(int value) :i{value} {}   // s and j initialized to their defaults
    // ...
};
```

A common error is to confuse constructors with other functions.

Question

Given:

```
class date {
  int y, m, d;

  public:
    date ();
    date (int y, int m, int d);
    date (const date& d);
    date get_date ();
};
```

Which lines are valid constructor declarations? (Choose all that apply)

- ☐ `date ();`
- ☐ `date (int y, int m, int d);`
- ☐ `date (const date& d);`
- ☐ `date get_date ();`
- ☐ `int y, m, d;`

Class invariants

A struct is acceptable to define a type as long as *every* struct member may be assigned **any** value at any time. If this is not true for your type, then we say that your type has *invariants*. Class invariants are guarantees made by your type. Invariants represent things that must hold true for your class to be valid.

Let say we need to prevent `Talk` from allowing zero length strings in the member `text_`. Currently, since everything is public, but it is easily fixed:

Example

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

using std::string;

class Talk {
  public:

    Talk() : text_ ("Hello, world!") { }

    Talk(const string& value)
```

(continues on next page)

(continued from previous page)

```
      : text_ {value.empty()? "default text": value}
    { }

    string text() {
      return text_;
    }
    void text(string value) {
      if (value.empty()) return;
      text_ = value;
    }

  private:
    string text_;
};

int main() {
  Talk say;
  std::cout << say.text() << '\n';

  say.text("Something else");
  std::cout << say.text() << '\n';
  say.text("");
  std::cout << say.text() << '\n';
}
```

Without running this program, can you predict it's output?

Run It

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

using std::string;

class Talk {
  public:

    Talk() : text_ ("Hello, world!") { }

    Talk(const string& value)
      : text_ {value.empty()? "default text": value}
    { }

    string text() {
      return text_;
    }
    void text(string value) {
      if (value.empty()) return;
      text_ = value;
    }

  private:
```

(continues on next page)

(continued from previous page)

```
    string text_;
};

int main() {
  Talk say;
  std::cout << say.text() << '\n';

  say.text("Something else");
  std::cout << say.text() << '\n';
  say.text("");
  std::cout << say.text() << '\n';
}
```

Our class now enforces its *invariants* that it will never allow the `Talk` text to be empty (we're a talkative Talk class).

We made several changes:

- The member `text_` is now private.
- A function `void text(string value)` was added. This was needed because we made `text_` private. Without this function, the only way to set the text was in the constructor.
- Added an additional check to our one argument constructor to enforce the "can't be empty" invariant.

Try This!!

Modify the last example so that it accepts an additional class member variable to repeat what `Talk` says more than once.

More to Explore

- From cppreference.com
  - Classes
  - Default constructors
- From C++ Core Guidelines
  - Constructors
  - Prefer in-class iniitalizers
  - Default operations: rule of zero and rule of five
  - Concrete types
- CplusPlus.com classes tutorial.
  - And if you read the tutorial, then also review Avoid protected data

3.8.3 Pointers to objects

As we have previously discussed, pointers can point to anything. We create pointers to objects much like any other type.

Raw pointers where we have to manage the memory ourselves:

```
int* p = new int{5};
dog* d = new dog{"Fido"};
```

And smart pointers that manage the memory for us:

```
auto p = std::unique_ptr<int>(new int{5});
auto d = std::unique_ptr<dog>(new dog{"Fido"});
```

In both cases, the initialization essentially identical.

Given a simple *POD* for a dog:

```
struct dog {
  std::string name;
  double age;
};
```

Access to members of any objects created uses the *member access operator* `operator .`:

```
// create a dog with initial values
dog buddy = {"Andy", 12.6};

// use member access operator to get values
std::cout << "My dog's name and age is: "
  << buddy.name << " and "
  << buddy.age << ".\n";
```

When you need to access members through a pointer, the operator precedence rules for pointer dereference and member access are a common source of error. When `buddy` is a pointer:

```
auto buddy = std::unique_ptr<dog>(new dog{"Andy", 12.6});
```

It seems that if `buddy.name` works when not a pointer, then given a pointer to a `buddy`, that `*buddy.name` should work, but it does not. The member access operator has higher precedence than the dereference operator. The code `*buddy.name` is equivalent to `*(buddy.name)`. This is almost always a bug. In this case, `name` is not a pointer type and cannot be dereferenced.

Explicit use of parentheses is one way to fix this problem:

```
(*buddy).name
```

This works, but the syntax is ugly. For this reason, the `operator ->` is used to access members of a pointer to an object. The code `buddy->name` is easier to read than `(*buddy).name`.

Putting it all together:

```
#include <iostream>
#include <memory>
struct dog {
  std::string name;
```

(continues on next page)

(continued from previous page)

```
  double age;
};

int main() {
  using std::cout;
  auto buddy = std::unique_ptr<dog>(new dog{"Andy", 12.6});

  cout << "name using dereference and member access: " << (*buddy).name
       << '\n'
       << "name using pointer to member: " << buddy->name;
}
```

The last version is the most commonly used because it is less error prone and easier to read.

More to Explore

- From cppreference.com
  - C++ Operator precedence and member access operators.

3.8.4 'this' pointer

The this pointer is a value that stores the address of the current object. Every non-static class member function is passed the `this` pointer. Whenever a member function or variable is used in a function, `this->` is automatically added before the name, resulting in a member access expression.

```
class example
{
    int x;
    int y;

    public:

       void foo()
       {
           x = 6;       // same as this->x = 6;
           this->x = 5; // explicit use of this->
       }

       void foo() const
       {
         // x = 7; // Error: *this is constant
       }

       void foo(int x) // parameter x shadows the member with the same name
       {
           // x = x;       makes no sense in this function
           this->x = x; // unqualified x refers to the parameter
                        // 'this->' required for disambiguation
       }
```

(continues on next page)

(continued from previous page)

```
      // No ambiguity in C++11 initializer syntax
      example (int x)
        : x(x),      // use parameter x to initialize member x
          y(this->x) // use member x to initialize member y
      {}

};
```

The this pointer can only be used in a member function. It is a compile error to use `this` outside of a member function.

Although `this` is a pointer, it is *immutable*. You can't change it and have it point to some other object. It *always* points to the current object.

That is why the this pointer can't be used in a static member function. Static members are members of the **class**. There is only a single copy that all objects of a class share. A static member has no way to determine which object instance the this pointer refers to.

More to Explore

- this pointer

3.8.5 Interfaces and implementation

A class defines a *type*. When you design a type, some parts are hidden (private), while others are visible (public). The public parts of your type represent an *interface*. The interface to your class determine how other functions will interact with it.

Consider a simple `date` class. This class does not allow direct access to y, m, and d. These class members can only be set using the 3 arg constructor provided and can only be retrieved using the functions provided.

```
class date {
  int y, m, d;

  public:
    date ()
      : y{1776}, m{7}, d{4}
    { }
    // Declare other constructors & functions
    date (int y, int m, int d);
    std::tuple<int,int,int> as_tuple ();
    int year ();
    int month ();
    int day ();
};
```

Public member functions define the class interface. Everything hidden are implementation details. No one needs to know (or care) how things are actually implemented.

Keeping interface specifications separate from the implementation is considered a best practice:

- As programs grow large, the time it takes to compile grows proportionally. On large projects, this can be a serious impact on your productivity. It's even the source of jokes.

- It makes your code easier to maintain. You don't have to find *where* in a large file your code is when you can find the named file instead.

Declare interfaces in your header files (*date.h*):

```
#pragma once

#include <tuple>

class date {
    int y {1776};
    int m {7};
    int d {4};
  public:
    date () = default;
    date (int y, int m, int d);
    std::tuple<int,int,int> as_tuple ();
    int year ();
    int month ();
    int day ();
};
```

Implement interfaces in your cpp files (*date.cpp*):

```
#include "date.h"
#include <tuple>

date::date (int year, int mon, int day)
  : y{year}, m{mon}, d{day}
{ }

std::tuple<int,int,int> date::as_tuple () {
  return std::make_tuple(y, m,d);
}

int date::year ()  { return y; }
int date::month () { return m; }
int date::day ()   { return d; }
```

File extensions

All source files are simply text files, however, by convention, different kinds of files have different extensions:

- Header files should end in `.h`, `.hpp`, or `.hxx`.

  The extension you use is mostly a matter of preference, but some organizations define explicit guidelines.

  Some code editors may assume `.h` headers are for C code and that `.hpp` headers are C++ code and may apply different syntax highlighting rules.

  It won't matter to your compiler.

- C++ Source files should end in `.cpp` and
- C Source files should end with `.c`.

C and C++ source files *are* handled differently by your compiler. That is when you compile your code, you need to compile the C files differently from the C++ files. Having a simple convention to distinguish them is important.

More to Explore

- Include guards from wikipedia.
- From cppreference.com
  - Source file inclusion
  - #pragma directives
- From C++ Core Guidelines
  - Interfaces

3.8.6 `const` member functions

We have *previously mentioned* uses for `const`, and classes add another situation where the const keyword can be used.

An object can be created with the `const` *cv qualifier*, just like any other type:

```
const int x = 3;

const Fibonacci foo = {5, 8, 13};
```

When created `const`, then no changes are allowed to the object. It is OK to call a *non-modifying member function*.

How does the compiler know that a function is a non-modifying member function? When the function is declared as `const`.

Create a const member function by inserting the const keyword after the parameter list and before the function body:

```
bool verbose() const {
  return true;
}
```

Here `const` tells the compiler this function **will not change** the object state.

It is a *promise*. If a `const` function attempts to change **any** class member, then a compile error results.

Note

Only class member functions can be marked `const`.

Attempting to mark a free function as `const` is a compile error.

Question

Given the following:

```
#include <iostream>

class Foo {
  int value_ = 0;
  public:
```

(continues on next page)

(continued from previous page)

```
    Foo() {}
    void value (const int x) {
      value_ = x;
    }
    int value() {
      return value_;
    }
};
int main () {
  const Foo a;
  a.value(13);
  std::cout << a.value() << '\n';
}
```

What is the output? Choose the **one** correct answer.

- ◯ Program prints 13
- ◯ Does not compile
- ◯ Program runs, but does not reach the end of main
- ◯ Program behavior is undefined

More to Explore

- From cppreference.com
  - Const and volatile type qualifiers.
  - Const member functions.

3.8.7 Abstraction

Most programming constructs are *abstractions*. For example, even the idea of **number** on a computer is an abstraction.

Programmers need to define, manipulate, and store numbers. Each language and compiler implements specific properties and operations. C++ defines Integral types and Floating point types. However, these numbers are **not** the numbers you learned in your math classes. In math, you learned that:

- The set of integers is infinite
- There is only 1 value for 0
- $0.1 * 10$ exactly equals 1

None of these things are guaranteed true on a computer.

It depends on the abstractions used in your language. In C++, no numbers are infinite. Some numbers can only store positive values. The floating point numbers have two different representations for the value 0. Decimal numbers that have exact values in mathematics can only be approximated in C++.

Even with all these limitations, numeric types in C++ are quite useful. They achieve the goal of allowing programmers to work with numbers easily without having to worry much about how they are actually represented in hardware.

Abstraction is all about hiding implementation details.

Consider a car as an example of abstraction in design. Here's an outline for starting a Model-T in 1915[1] :

1. On front of car, pull chock near right fender and engage crank lever under radiator.
    - Turn slowly to prime carburetor.
2. Get into car, insert key in ignition.
    - Turn start setting to either magneto or battery.
    - Adjust timing stalk and throttle stalk.
    - Pull back on handbrake to place car in neutral.
3. Return to front of car.
    - Use left hand to crank lever. (If the engine back-fires, using your left hand results in fewer broken arms)
4. If car starts, jump in!

The Model-T was one of the most popular cars ever made. It had no frills and few abstractions. You had to understand how it was *made* even to get it started.

This is not true with most cars today. Compare the previous steps with starting a car today:

1. Step on brake pedal and push button.

Over the last 100 years manufacturers have hidden most implementation details even though cars today are far more complex than a Model T.

Class abstractions

The design principles that apply to *everyday objects* also apply to *software objects*.

- Hide as many details as possible
- Don't hide too many

You, as the designer of a class are responsible for getting the abstractions correct in the classes you design. Keep in mind that every class defines a new *type*. In general we want to create classes that are easy to use and easy to modify.

Let's examine a poorly abstracted car class:

```
class Car {
  private:
    double speed_;
    double heading_;
    int    x_;
    int    y_;
  public:
    double speed ();
    double heading ();
    int    x ();
    int    y ();
    void speed (double speed);
    void heading (double direction);
    void x (int new_x);
    void y (int new_y);
};
```

What is wrong with this design?

[1] From https://www.caranddriver.com/features/a15142307/how-to-drive-a-ford-model-t

- We have created a class and dutifully made all of the class variables private.
- We then (also dutifully) created functions to set and get each of the class member variables.

Isn't this is what object-oriented programming is all about?

No, it is not.

What is wrong with this design? This list is long, but here are some of the major problems:

1. By default, no class members are initialized. A default constructed car is invalid.
2. It is still functionally a `struct`, even though this class has invariants.
    - The x and y values can be modified independently of speed and heading.
    - The speed could be set < 0.
    - The heading could be anything. It's not even clear what values are valid ($0 - 360$? $0 - 2\pi$?)
3. The car position is maintained in two separate `int` members, not a location object.
4. If users try to use this `Car`, then **they** are responsible for calculating correct x and y from the current heading and speed.
5. Fundamentally, this is not how cars are operated.
    - A car has a steering wheel and at least 1 pedal to accelerate
    - Cars only change position when moving
    - Cars only change direction when moving AND when the steering wheel is turned
    - The steering wheel has limits to keep the tires aligned with the forward momentum.

How can we improve the abstractions in our car class?

First, group the x and y coordinates into a single type. We could do just this:

```
// a location on a Cartesian grid
struct Point {
  double x = 0.0;
  double y = 0.0;
};
```

This works, but it's not much fun to use. When using `Point` objects, we will often want to initialize both the x and y values at once:

```
Point r {3.0,3.0};
```

The default constructors won't do this automatically. We need a 2 argument constructor:

```
Point (double x_value, double y_value)
    : x{x_value}, y{y_value}
{}
```

Once a non-default constructor is added to a class, the compiler will **not** automatically generate the default constructor for us, so we need to be explicit:

```
Point () = default;
```

And putting them together we have very minimal class:

```
// a location on a Cartesian grid
struct Point {
  double x = 0.0;
  double y = 0.0;
  Point (double x_value, double y_value)
      : x{x_value}, y{y_value}
  {}
  Point () = default;
};
```

We can't consider this class complete, but we will stop with Point for now and move onto the Car class. When we use Point, we should prevent `Car` users from changing it directly:

```
class Car {
  private:
    Point  location_;
    double speed_    = 0.0;
    double heading_  = 0.0;

  public:
    Point  location() const;
    double speed()    const;
    double heading()  const;
};
```

It is OK to keep the access functions for speed and heading. `Car` users are likely to need this information, but they should never set them directly. We should also promise that these functions will never change the state of a `car` by making all of the access functions `const`.

In order to make our car more 'real', we should add two more private member variables, one for the current steer angle and one for the current change in speed. Users are never given access to these, but internally a Car can use them to compute a new location:

```
class Car {
  private:
    Point  location_;
    double speed_    = 0;
    double heading_  = 0;
    double angle_    = 0;  // current steering angle
    double rate_     = 0;  // current change in speed

  public:
    Point  location() const;
    double speed()    const;
    double heading()  const;
};
```

Since users can't change steer angle directly, there needs to be come way to influence that value. One way is to define an enumerated type to set the angle on the steering wheel:

```
enum class Direction { CENTER, LEFT, RIGHT };
```

With these parts added, we can add public functions that use them:

```
//
// Users steer the car by choosing a steer direction.
// As long as a direction is applied, the steer angle will increase (or decrease)
// towards the indicated direction until the max steering angle
// for the car is reached.
//
// The max steer angle might be Car make/model dependent.
//
double steer (Direction dir);

//
// Change the car speed by (de)accelerating.
// Positive values will increase car speed.
// Negative values will reduce car speed.
// Zero values leave the car speed unchanged.
double accelerate (double rate);
```

And finally, an `update` function a car simulator might call to modify the state of the car every time step. Putting it all together:

```
// a location on a Cartesian grid
struct Point {
  double x = 0.0;
  double y = 0.0;
  Point (double x_value, double y_value)
      : x{x_value}, y{y_value}
  {}
  Point () = default;
};

enum class Direction { CENTER, LEFT, RIGHT };

class Car {
  private:
    Point  location_;
    double speed_    = 0;
    double heading_  = 0;
    double angle_    = 0;
    double rate_     = 0;

  public:
    Point  location() const;
    double speed()    const;
    double heading()  const;

    double steer (Direction dir);
    double accelerate (double rate);
    void   update();
};
```

Actually implementing these functions is a lab assignment.

While this class is far from complete, at least now we have a design that we can extend without needing to completely change it later once we realize it did not control any of its invariants.

More to Explore

- C++ enumerations
- Default constructors
- Abstraction in OOP

Footnotes

3.8.8 Enumerated types

An enum (enumerated type) is a distinct type whose value is restricted to a range of values (see below for details), which may include several explicitly named constants ("enumerators"). The values of the constants are values of an integral type known as the *underlying type* of the enumeration.

There are two distinct kinds of enumerations:

unscoped enumeration
: declared with the keyword `enum`

scoped enumeration
: declared with the enum-key `enum class` or `enum struct`

Unscoped enumerations

Each enumerator becomes a named constant of the enumeration's type (that is, name), visible in the enclosing scope, and can be used whenever constants are required.

```
enum color { red, green, blue };
color r = red;
switch(r)
{
  case red  : std::cout << "red\n";   break;
  case green: std::cout << "green\n"; break;
  case blue : std::cout << "blue\n";  break;
}
```

Each enumerator is associated with a value of the underlying type. When initializers are provided in the enumerator-list, the values of enumerators are defined by those initializers. If the first enumerator does not have an initializer, the associated value is zero. For any other enumerator whose definition does not have an initializer, the associated value is the value of the previous enumerator plus one.

Values of unscoped enumeration type are implicitly-convertible to integral types. If the underlying type is not fixed, the value is convertible to the first type from the following list able to hold their entire value range: `int`, `unsigned int`, `long`, `unsigned long`, `long long`, or `unsigned long long`. If the underlying type is fixed, the values can be converted to their promoted underlying type.

```
enum color { red, yellow, green = 20, blue };
color c = red; // c  implicitly converts to 0
int n = blue;  // n == 21
```

Consider the following program:

```cpp
#include <iostream>
enum Direction { north, south, east, west };

void show_direction(int direction) {
  std::cout << "Direction: " << direction << '\n';
}

int main() {
  Direction dir = west;
  show_direction(dir);
  int num = dir;
  show_direction(num);

  for (int i = north; i < 8; ++i) {
    show_direction(i);
  }
  return 0;
}
```

What problems does this program have?

1. The unscoped enum `Direction` is not type safe
    - Any integral type can be assigned to it
2. The line `int num = dir;` assigns a direction to an int
3. The for loop is unaware that there are only 4 directions, not 8.
4. Function `show_direction` implies it takes a direction as an argument, but any integral type will be processed without complaint
5. The function `show_direction` can't print a human readable direction. It can only print a number.

Scoped enums were introduced to address these specific shortcomings.

Scoped enumerations

Like unscoped enumerations, each enumerator becomes a named constant of the enumeration's type visible in the enclosing scope, and can be used whenever constants are required.

Unlike unscoped enums, scoped enums do **not** implicitly convert to integral types, but can be converted with a static or explicit cast.

```cpp
enum class Color { red, green = 20, blue };
Color r = Color::blue;
switch(r)
{
  case Color::red  : std::cout << "red\n";   break;
  case Color::green: std::cout << "green\n"; break;
  case Color::blue : std::cout << "blue\n";  break;
}
// int n = r;                // error: no implicit int conversion
int n = static_cast<int>(r); // OK, n = 21
```

All the C++ enumerated types are very simple compared to the same types in other languages. However, it is easy to add features as needed. Given the following scoped enum:

```
// If this were declared enum struct, nothing in this example changes...
//
enum class Direction { NORTH, SOUTH, EAST, WEST };
```

We can implement a stream overload to allow printing enum members with human readable strings:

```
std::ostream& operator<<(std::ostream& os, const Direction& rhs) {
  std::string dir;
  switch (rhs) {
    case Direction::NORTH: dir = "NORTH"; break;
    case Direction::EAST:  dir = "EAST";  break;
    case Direction::SOUTH: dir = "SOUTH"; break;
    case Direction::WEST:  dir = "WEST";  break;
  }
  return os << dir;
}
```

And an array allows the enum to be used in a range for loop:

```
const std::array<Direction,4> directions =
{
  {
    Direction::NORTH,
    Direction::EAST,
    Direction::SOUTH,
    Direction::WEST
  }
};
```

Now the show directions program looks like this:

```
#include "Direction.h"
#include <iostream>

void show_direction(const Direction d) {
  std::cout << "Direction: " << d << std::endl;
}

int main() {
  Direction dir = Direction::WEST;
  std::cout << "Show one direction: " << std::endl;
  show_direction(dir);

  std::cout << "Loop through all directions: " << std::endl;
  for (const auto& d: directions) {
    show_direction(d);
  }
  return 0;
}
```

Another example uses the same techniques to build a deck of cards.

Source

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

enum class Rank {
  Ace = 1, Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen, King
};

// Define an array of Rank to allow range-for looping
const std::array<Rank,13> ranks =
{
  {
    Rank::Ace, Rank::Two, Rank::Three, Rank::Four, Rank::Five,
    Rank::Six, Rank::Seven, Rank::Eight, Rank::Nine, Rank::Ten,
    Rank::Jack, Rank::Queen, Rank::King
  }
};

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

// Define an array of Suit to allow range-for looping
const std::array<Suit,4> suits =
{
  {
    Suit::Clubs, Suit::Diamonds, Suit::Hearts, Suit::Spades
  }
};

struct Card {
  Rank rank;
  Suit suit;
};

std::ostream& operator<<(std::ostream& os, const Rank& rhs );
std::ostream& operator<<(std::ostream& os, const Suit& rhs );
std::ostream& operator<<(std::ostream& os, const Card& rhs );


// short, but take care . . .
// a change in the order of the enum could break this,
// Assumes order: A,2,3,4,5,6,7,8,9,10,J,Q,K
// A more verbose switch would be better
std::ostream& operator<<(std::ostream& os, const Rank& rhs ) {
  if (rhs > Rank::Ace && rhs < Rank::Jack) {
    os << static_cast<int>(rhs);
  } else if (rhs == Rank::Ace) {
    os << "Ace";
  } else if (rhs == Rank::Jack) {
    os << "Jack";
```

(continues on next page)

(continued from previous page)

```
  } else if (rhs == Rank::Queen) {
    os << "Queen";
  } else {
    os << "King";
  }

  return os;
}

std::ostream& operator<<(std::ostream& os, const Suit& rhs ) {
  switch(rhs) {
    case Suit::Clubs:    os << "Clubs"; break;
    case Suit::Diamonds: os << "Diamonds"; break;
    case Suit::Hearts:   os << "Hearts"; break;
    case Suit::Spades:   os << "Spades"; break;
  }
  return os;
}

std::ostream& operator<<(std::ostream& os, const Card& rhs ) {
  return os << rhs.rank << " of " << rhs.suit;
}

int main() {
  std::vector<Card> deck;
  for (const auto& s: suits) {
    for (const auto& r: ranks) {
      deck.push_back(Card{r,s});
    }
  }
  for(const auto& c : deck ) {
    std::cout << c << '\n';
  }
  return 0;
}
```

Run it

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

enum class Rank {
  Ace = 1, Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen, King
};

// Define an array of Rank to allow range-for looping
const std::array<Rank,13> ranks =
{
  {
    Rank::Ace, Rank::Two, Rank::Three, Rank::Four, Rank::Five,
```

(continues on next page)

(continued from previous page)

```
    Rank::Six, Rank::Seven, Rank::Eight, Rank::Nine, Rank::Ten,
    Rank::Jack, Rank::Queen, Rank::King
  }
};

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

// Define an array of Suit to allow range-for looping
const std::array<Suit,4> suits =
{
  {
    Suit::Clubs, Suit::Diamonds, Suit::Hearts, Suit::Spades
  }
};

struct Card {
  Rank rank;
  Suit suit;
};

std::ostream& operator<<(std::ostream& os, const Rank& rhs );
std::ostream& operator<<(std::ostream& os, const Suit& rhs );
std::ostream& operator<<(std::ostream& os, const Card& rhs );


// short, but take care . . .
// a change in the order of the enum could break this,
// Assumes order: A,2,3,4,5,6,7,8,9,10,J,Q,K
// A more verbose switch would be better
std::ostream& operator<<(std::ostream& os, const Rank& rhs ) {
  if (rhs > Rank::Ace && rhs < Rank::Jack) {
    os << static_cast<int>(rhs);
  } else if (rhs == Rank::Ace) {
    os << "Ace";
  } else if (rhs == Rank::Jack) {
    os << "Jack";
  } else if (rhs == Rank::Queen) {
    os << "Queen";
  } else {
    os << "King";
  }

  return os;
}

std::ostream& operator<<(std::ostream& os, const Suit& rhs ) {
  switch(rhs) {
    case Suit::Clubs:    os << "Clubs"; break;
    case Suit::Diamonds: os << "Diamonds"; break;
    case Suit::Hearts:   os << "Hearts"; break;
```

(continues on next page)

(continued from previous page)

```
    case Suit::Spades:   os << "Spades"; break;
  }
  return os;
}

std::ostream& operator<<(std::ostream& os, const Card& rhs ) {
  return os << rhs.rank << " of " << rhs.suit;
}

int main() {
  std::vector<Card> deck;
  for (const auto& s: suits) {
    for (const auto& r: ranks) {
      deck.push_back(Card{r,s});
    }
  }
  for(const auto& c : deck ) {
    std::cout << c << '\n';
  }
  return 0;
}
```

More to Explore

- Enumeration declaration
- Enumeration (Microsoft)

3.9 Class constructors and overloads

This chapter introduces the different kinds of functions that are found only within classes.

3.9.1 Constructors

All user defined types include a constructor. Even if you do not write one explicitly, the compiler will write one for you. For most of the classes and structs seen so far, we let the compiler generate the default constructors for us.

Operation	Function Signature
default constructor	`X()`
copy constructor	`X(const X&)`
copy assignment	`operator=(const X&)`
move constructor	`X(X&&)`
move assignment	`operator=(X&&)`
destructor	`~X()`

default constructor
: Create a new object in 'the default way'. If you do not write one, the compiler will unless you write your own overloaded constructor.

The default constructor can be overloaded like any other function. When you design your types, you may create constructors with one, two, or more arguments as appropriate.

copy constructor
Create a new object by copying an existing object.

copy assignment
Replace the state of an existing object using another object.

move constructor
Added in C++11. Like copy construction, but the existing source object is invalidated. The state is *moved* from the source object when the new object is constructed.

move assignment
Added in C++11. Like copy assignment, but the existing source object is invalidated. The state is *moved* from the source object when the object state is transferred.

Move assignment and move constructors are optional. If you do not define them, the compiler will fall back to the copy constructor and copy assignment instead. However, move semantics may make your code more efficient by replacing potentially expensive copy operations with faster alternatives.

destructor
A *destructor* is a special function that is **always** called when an object goes out of scope. The purpose of a destructor is to reclaim any memory or perform any other cleanup tasks that need doing before the object is finally destroyed. A few points about destructors:

- Like constructors, destructors have no return type.
- Destructors take no arguments, so they never have a parameter list. There is no way to overload a destructor.
- We don't call this function ourselves, we leave that to a program as it runs.

We will explore destructors more when we explore containers. For now, just remember that a destructor always exists for every class whether you write one or not and it is always called when the class is destroyed.

The compiler will **not** generate a default constructor if we write any non-default ones. The compiler may still create other default operations.

So while this is fine:

```
struct point {
  int x = 0;
  int y = 0;
};

int main () {
  point p;        // implicitly defined constructor
  point q = p;    // implicitly defined copy constructor
}
```

We can only initialize our point using the default constructor.

```
// this works, but it's tedious
point p;
p.x = 8;
p.y = 13;


// This syntax is preferred, but won't compile
point e {3,5};
```

If we want to initialize class members at construction time, then we need to add custom constructors.

```
struct point {
  int x;
  int y;
  point (int x, int y) : x(x), y(y) {}
};

int main () {
  point p {3,5};  // 2 arg constructor OK
  point q = p;    // default copy constructor
  point e;        // compile error
}
```

Now we can use our 2 argument constructor, but now our old default invocation is broken. You'll see and error like this:

```
foo.cpp:11:11: error: no matching constructor for initialization of 'point'
```

This is fixed by either:

- Writing our own default implementation

  ```
  point ()
    : x(3), y(5)
  {}
  ```

- Telling the compiler to write it

  ```
  point () = default;
  ```

- Telling the compiler to delete it

  ```
  point () = delete;
  ```

  In this case, attempting the use the default constructor is still a compile error, but the error is more explicit: you can't use it because it was deleted.

Note

If you write a non-default constructor, then you should *always* write your own default constructor, or explicitly instruct the compiler to make it for you, or delete it.

In general, take care deleting the default constructor. Delete it only when you are sure objects of the class will **never** need to be default constructed or that there is simply no sensible set of defaults for the class.

Initialization syntax

Some programmers coming to C++ from other OO languages sometimes feel as if they have to initialize objects like this:

```
point p();
```

Even though all semester, you been writing:

```
std::string s;
std::vector<int> v;
std::random_device r;
```

When it comes to user defined types, sometimes it feels 'incomplete' if you don't include the `()`. Usually, those parentheses create more problems than they resolve. This is because of an inherent ambiguity in the C++ language. Although it seems obvious to us the statement `point p();` is a call to the default constructor and the results should be a new variable `p`, the compiler interprets it differently.

The basic rule is:

```
If it looks like a function call, it's a function call.
```

This means that in the code above, the compiler instead looks for:

- a function named `p`
- that takes no arguments
- and returns an object of type `point`

Since in this case, there is no such function, it returns an error. Some compilers, like *clang*, will try to tell you:

```
point.cpp:7:12: warning: empty parentheses interpreted as a function declaration [-
↪Wvexing-parse]
    point p();
           ^~
point.cpp:7:12: note: replace parentheses with an initializer to declare a variable
    point p();
           ^~
```

C++11 Feature

C++ resolves this ambiguity in C++11 using the *uniform initializer syntax*. You can use curly braces: `{}` instead of parentheses to initialize objects. Braces are an extension of the initializer list syntax for containers and can be used even for default constructed objects.

```
string s{};
vector<int> v{};
point p{};
string s{"hello, world!"};
vector<int> v{1,2,3,4,5};
```

While the above works every time, omitting the braces entirely when not needed is preferred:

```
string s;
vector<int> v;
point p;
```

Initializer syntax works within constructors as well.

```
vector<point> redundant {
   point {2,3},
   point {3,5},
   point {8,13},
```

(continues on next page)

(continued from previous page)

```
   point {21,34}
};

// The compiler can deduce the type in the container,
// so we don't have to repeat the type every time
vector<point> points {
   {2,3}, {3,5}, {8,13}, {21,34}
};
```

Recall that for containers, there is a difference between `vector<int>(5)` and `vector<int>{5}`. What's the difference?

Show

The first version creates a vector of size 5 with no initialized values.

The second version creates a vector of size 1 with a single value equal to 5.

Overloaded constructors

The same guidelines that apply to writing good functions apply to writing good overloaded constructors. A good class is built around good functions. Just as with regular functions, avoid confusing parameter lists. Consider the following:

```
date (int, int, int);
```

It's seems likely that the three parameters represent the year, month, and day, but without reading the code, there is no way to know what order.

```
// is this correct?
date d = {1776, 7, 4};

// or this?
date d = {4, 1776, 7};
```

Even if we read the code and learn the order, it is still probable that we will forget the order and transpose a month and day at some point.

Instead of resigning ourselves to hoping we remember or having debugging problems at runtime, simply defining appropriate types improves clarity and utility:

```
class year {
  int y;
  public:
    year() = default;
    year(int value) : y {value} {}

    int year() { return y;}
};
enum class month {
  jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};
```

(continues on next page)

(continued from previous page)

```
// now a new date can be constructed like:
date d = {year{1776}, month::jul, 4};
```

This version is easier for programmers to remember and any errors are compile errors instead of runtime errors.

Telescoping constructors

The original `date` class suffered from a common design problem: too many parameters of the same type. A closely related problem is how to provide flexibility when constructing new objects. A common solution is to provide constructors with different numbers of arguments:

```
struct date {
   date();                          // make a default date
   date(year y);                    // use a default month and day
   date(year y, month m);           // use a default day
   date(year y, month m, int day);  // specify the entire date
};
```

What about the possibility of specifying `month` and `day`? How many different constructors should be allowed? The number of permutations get unmaintainable even for a relatively few number of parameters.

This is called a *telescoping constructor*, and is generally considered an *anti-pattern*. That is, there are better solutions to this problem.

The easiest solution in C++ is to use default values for function parameters. This works best when the default values are different types and there is no need to allow every possible combination of parameters.

```
struct date {
   date(year y = date::current_year(),
        month m = date::current_month(),
        int day = date::current_day());
};
```

This solution is still limited by the fact that defaults are still evaluated left to right. A date declaration of the form

```
date d {15};
```

won't create a date for the 15th day of the current month and year. In addition, the solution does not work well when all (or most) of the parameters are the same type. Consider this example:

```
class NutritionFacts {
  private:
    // variables in need of initialization to make valid object
    const double serving_size_;  // mL
    const int servings_;         // per container
    const double calories_;      // Kcal
    const double fat_;           // g
    const double sodium_;        // mg
    const double carbs_;         // g

  public:
    // How about this as a solution?
   NutritionFacts(double, int, double, double, double, double);
};
```

Is the proper order `calories`, `fat`, `carbs`, or `fat`, `calories`, `carbs`, or something else? Even if we give these parameters meaningful names, there is no runtime enforcement. It's easy to make a mistake when too many parameters are the same type.

When confronted with many optional parameters, a builder is an effective alternative. Basic ideas:

- Use constructor parameters to accept mandatory parameters.
- Use a helper class (Builder) to default initialize optional parameters.
- A `Builder::build()` function creates a NutritionFacts object from a builder.
- The builder makes the class it helps a friend.

  This is used only avoid creating builder accessor functions.
- A conversion constructor is used to copy builder state into the enclosing class.

```
#pragma once

#include <iostream>

class NutritionFacts {
  private:
    // variables in need of initialization to make valid object
    const double serving_size_;  // mL
    const int servings_;         // per container
    const double calories_;      // Kcal
    const double fat_;           // g
    const double sodium_;        // mg
    const double carbs_;         // g


  public:
    // Only one simple constructor for mandatory parameters
    // - rest is handled by Builder
    NutritionFacts( const double serving_size, const int servings)
      : serving_size_{serving_size}, servings_{servings},
      calories_{0}, fat_{0}, sodium_{0}, carbs_{0}
    {}

    // use this class to construct Nutritionfacts
    class Builder {
      private:
        friend NutritionFacts;
        double serving_size_ = 15;  // mL
        int servings_ = 10;         // per container
        double calories_ = 0;       // Kcal
        double fat_ = 0;            // g
        double sodium_ = 0;         // mg
        double carbs_ = 0;          // g

      public:
        Builder() = default;

        // create a NutritionFacts object from a builder
        NutritionFacts build() {
```

(continues on next page)

(continued from previous page)

```
        return NutritionFacts (*this);
      }

      Builder& serving_size(const double size) {
        serving_size_ = size;
        return *this;
      }
      Builder& servings(const int s) {
        servings_ = s;
        return *this;
      }
      Builder& calories(const double c) {
        calories_ = c;
        return *this;
      }
      Builder& fat(const double f) {
        fat_ = f;
        return *this;
      }
      Builder& sodium(const double s) {
        sodium_ = s;
        return *this;
      }
      Builder& carbohydrates(const double c) {
        carbs_ = c;
        return *this;
      }

  };

  explicit NutritionFacts(const Builder& builder)
    : serving_size_{builder.serving_size_},
    servings_{builder.servings_},
    calories_{builder.calories_},
    fat_{builder.fat_},
    sodium_{builder.sodium_},
    carbs_{builder.carbs_}
  {}

  double serving_size() const { return serving_size_; }
  int servings() const { return servings_; }
  double calories() const { return calories_; }
  double fat() const { return fat_; }
  double sodium() const { return sodium_; }
  double carbohydrates() const { return carbs_; }
};


std::ostream& operator<<(std::ostream& os, const NutritionFacts& rhs) {
  return os << "Serving size: " << rhs.serving_size()
            << "\tServings: " << rhs.servings()
            << "\tCal: " << rhs.calories()
```

(continues on next page)

(continued from previous page)

```
            << "\tFat: " << rhs.fat()
            << "\tSodium: " << rhs.sodium()
            << "\tCarbs: " << rhs.carbohydrates();
}
```

When complete, the classes can be used like this:

```
#include "NutritionFacts.h"

#include <iostream>

int main() {
  // make facts without any optional parts
  NutritionFacts cake = {75, 8};

  // create a builder
  NutritionFacts::Builder b;

  // change the state
  b.serving_size(28.4).servings(1);
  b.fat(10).sodium(2).calories(150).carbohydrates(15);

  // create a set of nutrition facts using the builder
  auto chips = b.build();

  // create nutrition facts without creating a (named)
  // temporary builder object
  NutritionFacts soda = NutritionFacts::Builder()
                        .serving_size(368).servings(1)
                        .carbohydrates(40).calories(150).sodium(15).build();

  std::cout << "Cake:\t" << cake << '\n';
  std::cout << "Chips:\t" << chips << '\n';
  std::cout << "Soda:\t" << soda << '\n';

  return 0;
}
```

While not the most idiomatic C++ solution, it is something we can create and use with only the knowledge of classes we have so far. We will revisit the builder pattern later after we cover inheritance.

More to Explore

- Most vexing parse (wikipedia)
- Item #6 "Most Vexing Parse" from 'Effective STL' by Scott Meyers (Addison-Wesley Professional). Copyright 2001 Scott Meyers, 978-0-201-74962-5.
- Builder design pattern:
  - Builder Design Pattern on oodesign.com
  - Builder Design Pattern on Wikipedia

- Example telescoping constructor
- Effective Java, by Joshua Bloch. Item #2: Consider a builder when faced with many constructor parameters

3.9.2 Static members

It is possible to define classes that have static member functions and variables. Static members are **not** stored in any instance of a class. In fact, you don't need to ever create a class instance in order to access static members.

Like global variables, static member variables persist as long as the entire program. For global variables refresher, refer to the section *Scope*.

The static keyword is only used with the declaration of a static member, inside the class definition, but not with the definition of that static member:

```
class X { static int n; }; // declaration (uses 'static')
int X::n = 1;              // definition (does not use 'static')
```

An example of a static member:

```
#include <iostream>

// a class that counts how many live objects
// currently exist
class counter {
   static int instance_count;   // declaration, but no definition
   public:
      // increase the count when the counter object is created
      counter() {
         ++instance_count;
      }
      // decrease the count when the counter object is destroyed
      ~counter() {
         --instance_count;
      }

      int count() const noexcept {
         return instance_count;
      }
};

// must be defined before use
int counter::instance_count = 0;    // definition. Now the type is complete

void print(const counter& c) {
   std::cout << "There are " << c.count()
             << " counter objects in memory\n";
}

int main() {
   counter a;
   print(a);
   {
```

(continues on next page)

(continued from previous page)

```
      counter a;
      counter b;
      counter c;
      print(a);
      print(b);
      print(c);
   }
   print(a);
   return 0;
}
```

We could have written our counter so that we did not need an instance member to determine the count.

```
#include <iostream>

// a class that counts how many live objects
// currently exist
class counter {
   static inline int instance_count = 0;   // declaration and definition
   public:
      counter() {
         ++instance_count;
      }
      ~counter() {
         --instance_count;
      }

      // can only access static member data
      static int count() noexcept {
        return instance_count;
      }

};

void print() {
   std::cout << "There are " << counter::count()
             << " counter objects in memory\n";
}

int main() {
   print();
   {
      counter a;
      print();
      {
         counter a;
         counter b;
         counter c;
         print();
      }
   }
   print();
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

A static member function allows us to get the count even if no instances of a counter class have ever been created.

C++20 Feature

Starting in C++20, our counter could be initialized constinit:

```
constinit static inline int instance_count = 0;
```

constinit is useful for variables that must be initialized at compile time but are still mutable.

More to Explore

- **static members from cppreference.com.**
  Compare to static storage duration, which is different.

3.9.3 Operator overloads

There are many reasons to want to overload the builtin operator functions for user defined types.

Case 1

Operator overloads are *required* for some containers in the STL. Notably set, which requires any type used in a set overloads `operator<`, or the operator defined for the set if a custom comparison function is defined. Similarly, map requires `operator<` and unordered_map requires both that and `operator==`.

Classes that do not overload these comparison operators can't be used in these containers.

Overloading these operators is useful in any conditional expression, but only makes sense when creating a class with data members that are *comparable*.

Case 2

C++ provides many builtin types and operators to manipulate them. We want new user defined types to be as simple and intuitive to use as the builtin types.

For example, finding the roots of a quadratic equation:

```
double root1 = (-b + sqrt(b*b - (4 * a * c))) / (2 * a);
```

When all the variables are builtin types, this looks familiar. But without operator overloads, a rational number class would be a pain to read and use.

The following example might be a way to use a Rational Java class.

```
// Static constructors are a common feature of Java value classes.
Rational a = Rational.of(2, 3);
Rational b = Rational.of(6);
Rational c = Rational.of(5, 3);

Rational fourA = a.multiply(Rational.of(4));
```

(continues on next page)

(continued from previous page)

```
Rational discriminant = b.multiply(b).minus(c.multiply(fourA));
Complex tmp = Complex.sqrt(discriminant);
Rational twoA = a.multiply(Rational.of(2));
Complex root = b.invSign().plus(tmp).divide(twoA);
```

When *everything* has to be a named function, and the standard operators can only be used on builtin types, the result is not as clean as we would like.

What we want here is the ability to use familiar semantics on user defined types:

```
rational a {2,3}, b{6,1}, c{5,3};
complex root1 = (-b + sqrt(b*b - (4 * a * c))) / (2 * a);
```

Overloading the standard operator functions make this possible.

In C++, operators are overloaded in the form of functions with special names. For example, `a+b` and `operator+(a,b)` both call the same function.

As with other functions, overloaded operators can generally be implemented either as a member function of their left operand's type or as non-member functions.

Most of the work in overloading operators is boiler-plate code. Not surprising, since most operators defer their actual work to plain functions. The programming community has already thought long and hard on techniques to make overloads efficient and easy to maintain.

Basic overloading guidelines

Most C++ operators can be overloaded. You cannot change the meaning of operators for built-in types in C++. Operators can only be overloaded when at least 1 operand is a user-defined type. Other rules of overloads still apply: overloads for a specific function signature can only be used once.

Not all operators can be overloaded in C++. Those that cannot be overloaded are: `operator .`, `operator ::`, `operator sizeof`, `operator typeid`, `operator .*`, and `operator ? :`.

When it comes to operator overloading in C++, there are basic guidelines you should follow. As with all such guidelines, there are exceptions. The goal of operator overloading is to make classes *easier* to use and to make them behave more like builtin types. Keep that in mind when overloading.

1. Whenever the meaning of an operator is not obviously clear and undisputed, it should not be overloaded.

   Instead, provide a function with a well-chosen name. It is hard to understand the semantics behind the operator unless the use of the operator in the application domain is well known and undisputed. Contrary to popular belief, this is hardly ever the case.

2. Always stick to the well-known semantics for the operator.

   C++ poses no limitations on the semantics of overloaded operators. Your compiler will happily accept code that implements the binary + operator to subtract from its right operand. However, the users of such an operator would never suspect the expression a + b to subtract a from b. Of course, this supposes that the semantics of the operator in the application domain is undisputed.

3. Always provide all out of a set of related operations.

   Operators are related to each other and to other operations.

   - If your type supports `a + b`, then users will expect to be able to call `a += b`, too.
   - If it supports prefix increment `++a`, then `a++` is likely expected also.
   - If they can check whether `a < b`, then most users expect to also to be able to check `a > b`.

- If copy-construction is allowed, then assignment is expected.

The remaining sections describe the specific techniques and function signatures for some of the most common operator overloads.

Assignment operator

The copy assignment operator is called when an object appears on the left side of an assignment expression.

The canonical copy-assignment operator is expected to perform no action on self-assignment, and to return the `lhs` by reference:

```
T& T::operator=(const T& other)
{
  // copy data from other to
  // current instance
  return *this;
}
```

Note

If a class requires a user-defined copy assignment operator, it almost certainly requires a user-defined copy constructor as well, and vice versa. This principle is often referred to as part of the rule of zero and rule of five

The fundamental reason for this is that both the copy constructor and the copy assignment operator deal with the process of creating a copy of an object's state. If a class manages resources (like dynamically allocated memory, file handles, network connections, etc.) that require special handling during copying, the default-generated versions of these functions need the same special handling. Handling that the default generated functions will not provide.

In those situations where copy assignment cannot benefit from resource reuse (it does not manage a heap-allocated array and does not have a member that does, such as a member std::vector or std::string), there is a popular convenient shorthand: the copy-and-swap assignment operator, which takes its parameter by value swaps with the parameter, and lets the destructor clean it up.

```
// copy/move constructor is called to construct other
T& T::operator=(T other) noexcept
{
    using std::swap;
    swap(*this, other); // exchange resources between *this and other
    return *this;
}
```

When the function ends, the destructor of `other` is called to release the resources formerly held by `*this`. A custom swap function for the user defined type `T` is required for this technique to work:

```
friend void swap(T& first, T& second)
{
  using std::swap;

  // by swapping the members of two objects,
  // the two objects are effectively swapped
  swap(first.size_, second.size_);
  swap(first.value_, second.value_);
```

(continues on next page)

(continued from previous page)

```
  // repeat for each member
}
```

Insertion and extraction operators

The bitshift operators << and >>, although still used in hardware interfacing for the bit-manipulation functions they inherit from C, have become more prevalent as formatted stream operators in C++.

The overloads of `operator >>` and `operator <<` that take a std::istream reference or std::ostream reference as the left hand argument are known as insertion and extraction operators.

Since these operators change their left argument (they alter the stream), they should, according to the rules of thumb, be implemented as members of their left operand's type. However, their left operands are streams from the standard library, and while most of the stream output and input operators defined by the standard library are indeed defined as members of the stream classes, when you implement output and input operations for your own types, you cannot change the standard library's stream types. So clearly, these overloads cannot be stream member functions.

It is common, however, to see C++ examples posted on the internet that define these overloads as friends. The advantage of making these functions friends is they have access to the private data of the class, if needed. The disadvantage of making these functions friends is that you may decide to stream data out of a class that is otherwise not accessible.

Why is that bad?

Josh Bloch in *Effective Java*, dedicates an entire section to this topic. Effective Java discusses the `toString()` overload, but it serves a similar purpose to `operator <<` in the Java language.

When you create an output function that streams data out of your class that is not available through any other function, then some users will use your stream function just to get their hands on your private data.

- If your data needs to be part of the output stream, then make a function to access it.
- If you have functions that provide access to every private member that is part of the stream, then you don't need it to be a member function or a `friend`.

That's why you need to implement these operators for your own types as non-member non-friend functions. The canonical forms are:

```
std::ostream& operator<<(std::ostream& os, const T& rhs)
{
  // write rhs to stream

  return os;
}

std::istream& operator>>(std::istream& is, T& rhs)
{
  // read rhs from stream

  if( /* could not construct T from stream */ ) {
    is.setstate(std::ios::failbit);
  }

  return is;
}
```

Function call operator

When a user-defined class overloads the function call operator, `operator()`, it becomes a FunctionObject type. A function object is a class that can be called as if it was a function. Many standard algorithms, from sort to accumulate accept objects of such types to customize behavior. Prior to the C++11 additions of function and lambda expressions, function objects were an important way to pass functions to algorithms.

There are no particularly notable canonical forms of `operator()`, but to illustrate the usage:

```
struct Sum {
    int sum;
    Sum() : sum(0) { }
    void operator()(int n) { sum += n; }
};
Sum s = std::for_each(v.begin(), v.end(), Sum());
```

Note

Unlike lambda expressions and function pointers, when passing a function object to an algorithm, the function call operator **must** be included.

A function call overload can be overloaded to take any number of additional arguments, including zero. A single class can contain more than 1 function call operator overload, subject to the other rules of function overloading.

Comparison operators

Standard algorithms such as std::sort and containers such as set expect `operator <` to be defined, by default, for the user-provided types, and expect it to implement strict weak ordering. Strict weak ordering defines members of a set as *comparable* to each other. The general signatures for these non-member functions is:

```
inline bool operator<(const T& lhs, const T& rhs)
{
   // compare the data in left-hand side and right-hand side objects
   // for less than
}

inline bool operator==(const T& lhs, const T& rhs)
{
   // compare the data in left-hand side and right-hand side objects
   // for equality
}
```

An idiomatic way to implement strict weak ordering for a structure is to use lexicographical comparison provided by std::tie:

```
struct package
{
    std::string name;
    unsigned int floor;
    double weight;
};

inline bool operator<(const package& lhs, const package& rhs)
{
```

(continues on next page)

(continued from previous page)

```
    // parameters passed to each tie must be in the same order
    // or this will always return false
    return std::tie(lhs.name, lhs.floor, lhs.weight)
         < std::tie(rhs.name, rhs.floor, rhs.weight);
}
```

If some of the data required for the comparison is private and has no function to access the data members, then you may need to make your relational operators friends.

If you do need to define `operator<` as a member function, then the left-hand side of the operator will be the this pointer. The signature of the operator overload changes:

```
bool operator<(const T& rhs) const
{
    /* do actual comparison with *this */
}
```

Note that this form of the overload must be `const` in order to compile as a member function.

Once you have defined `operator<` and `operator==`, there is no need to rewrite the comparison logic again. It is much better to implement the remaining comparison functions in terms of < and ==.

```
// note the operands swapped inside the function body
inline bool operator> (const T& lhs, const T& rhs){ return   rhs < lhs; }

inline bool operator<=(const T& lhs, const T& rhs){ return !(lhs > rhs); }
inline bool operator>=(const T& lhs, const T& rhs){ return !(lhs < rhs); }

inline bool operator!=(const T& lhs, const T& rhs){ return !(lhs == rhs); }
```

C++20 Feature

Since C++20, all 6 comparison operators are defined if the three-way comparison operator `operator<=>` is defined, and *that* operator, in turn, is generated by the compiler if it is defined as defaulted:

```
struct package
{
    std::string name;
    unsigned int floor;
    double weight;
    auto operator<=>(const package&) const = default;
};
// packages can now be compared with ==, !=, <, <=, >, and >=
```

The three-way comparison operator is also called the 'spaceship operator' because of the way it looks in an expression: `a <=> b`.

Unlike the 'two-way' comparison operators, `operator<=>` does not return `bool`. With a single operation, it determines the entire ordering relationship between two values, returning one of three possible outcomes: less than, equal to/equivalent, or greater than:

- A negative result (or a value representing less) means the left operand is less than the right operand (`a < b`).
- A result of zero (or a value representing equal/equivalent) means the operands are equal (`a == b`) or that they are equivalent (`!(a < b) && !(b < a)`).

- A positive result (or a value representing greater) means the left operand is greater than the right operand (`a > b`).

If the three-way comparison is **not** defaulted, then you must overload `operator==` in addition to `operator<=>`. Neither of these overloads need to be member functions. As with the earlier guidance for comparison overloads, non-friend non-member functions are preferred.

Once these two overloads have been implemented, all the other comparison operations can be generated automatically.

One thing you may have noticed is the number of classes that removed the `!=`, `<`, `<=`, `>`, and `>=` operators starting with C++20.

Binary arithmetic operators

Binary operators are typically implemented as non-members to maintain symmetry. For example, when adding a complex number and an integer, if `operator+` is a member function of the complex type, then addition doesn't behave in a way most people expect:

```
complex a = {1,1};
int b = 3;
complex c = a+b;  // compiles
complex d = b+a;  // error
```

As a member function, only complex+integer would compile, not integer+complex. Since for every binary arithmetic operator there exists a corresponding compound assignment operator, canonical forms of binary operators are implemented in terms of their compound assignments:

```
class T
{
 public:
  T& operator+=(const T& rhs) // compound assignment (does not need to be a member,
  {                           // but often is, to modify the private members)
    // add rhs to *this
    return *this;             // return the result by reference
  }
};
```

The normal addition is often implemented as a non-friend non-member:

```
T operator+(      T lhs,    // passing lhs by value helps optimize chained a+b+c
            const T& rhs)   // passing lhs by non-const reference is acceptable
{
  lhs += rhs; // reuse compound assignment
  return lhs; // return the result by value (uses move constructor)
}
```

The remaining binary arithmetic operators are implemented using the same pattern.

Increment and decrement operators

Unlike many of the operator overloads in this section, the increment and decrement operators are *unary* operators -- only one operand, the current object, is involved.

The postfix increment and decrement operator is usually implemented in terms of the prefix version:

```
struct T
{
    // prefix increment
    T& operator++()
    {
        // do the actual increment here
        return *this;
    }

    // postfix increment
    const T operator++(int)
    {
        T tmp(*this); // copy
        operator++(); // pre-increment
        return tmp;   // return old value
    }
};
```

There are a couple of things to notice about postfix increment:

- It returns a constant copy of the previous object value.

  Any modifications made to the returned object after calling `operator++(int)` would be modifying a temporary object. This is always bad. Returning a `const` object prevents accidental changes to a temporary object. This overload should never return a reference.

- It takes a 'dummy parameter' of type int.

  The 'dummy' parameter simply allows two functions with the same name -- `operator++` to have different overloads and therefore different behaviors. When the postfix increment and decrement appear in an expression, the corresponding user-defined function (`operator++` or `operator--`) is called with an integer argument 0.

Conversion operators

C++ allows you to create operators to convert between your type and other ADT's. A conversion function is declared like a non-static member function or member function template with

- no parameters,
- no explicit return type, and
- with the name of the form:

```
// implcit conversion
operator int() const { /* return int version of type */ }

// explicit conversion
explicit operator int() const { /* return int version of type */ }
```

Suppose we want to concatenate a `rational` to a string?

```
rational a {2,3};
std::string s = {"A = "};
s += a;                    // will not compile
```

Your compiler may present something like:

```
error: no viable overloaded '+='
candidate function not viable:
no known conversion from 'rational' to
'const std::__1::basic_string<char>' for 1st argument
_LIBCPP_INLINE_VISIBILITY basic_string& operator+=(const basic_string...
```

Defining a conversion operator for `std::string` allows this conversion to happen:

```
rational::operator std::string() const {
   std::stringstream ss;
   ss << numerator << '/' << denominator;
   return ss.str();
}
```

A class constructor taking a single argument can also be seen as converting its argument into the type:

```
rational (int x) {
   numerator = x;
   denominator = 1;
}
```

And an expression like this works:

```
rational r = 3;
```

The conversion both of these previous cases, for the `string` and the `int` happened implicitly. Often, we don't want types to **always** implicitly convert to a user-defined type:

```
void func (rational r);

int main () {
   func(3);   // this works
}
```

The call to `func()` works because rational has a conversion constructor that converts int values to rational ones. C++ provides a keyword `explicit` that requires a cast - it inhibits the implicit conversion of a user defined type.

```
explicit rational (int x) { . . .

explicit operator std::string() const { . . .
```

And the previous expressions need casts or an explicit constructor call:

```
rational r = rational(3);       // constructor
rational r = rational{3};       // constructor - no implicit conversion
func(rational(3));              // constructor
func(static_cast<rational>(3)); // cast
func((rational)3);              // c-style cast

// explicitly convert a rational to string
// using functional conversion syntax
std::string s = string(rational(3));
```

More to Explore

- Operator overloading in C++ from stackoverflow. Much of the content in this section was taken from there.
- Converting constructors
- C++ Core Guidelines for overloading
- Comparison operators from cppreference.com.
- Named requirements: Compare. This page also includes a list of the parts of the STL that expect types that satisfy this requirement.
- Effective Java, 3rd edition, Joshua Bloch. Addison-Wesley Professional, Jan 2018.

  The section regarding overloading toString is in section 12 of the third edition and section 10 of the second.

3.9.4 Friend vs non-friend functions

Some operators must be implemented as member functions, `operator=`, `operator[]`, and member access - both `operator.` and `operator->`, because the language requires it. We have choices where we define the others.

Some are commonly implemented as non-member functions, because their left operand cannot be modified by you. The most prominent of these are the stream insertion and extraction operators. The left operands are stream classes from the standard library which you cannot change.

For operators where you have to choose to either implement them as a member function or a non-member function, use the following guidelines:

1. If it is a **unary operator**, then implement it as a **member** function. For example, `operator++`.
2. If a binary operator treats both operands equally then implement as a **non-member** function.

   Generally, neither operand is modified in this situation. The relational operators all fall into this category.
3. If a binary operator does not treat both of its operands equally then consider making it a member function.

   If the left-hand side operator is modified in the operation, or the function returns the `this` pointer, then it should be a member function of the left hand operand type.

   Otherwise, it can be implemented as a non-member function.

In the previous section, the relational operators were all declared as *non-friend non-member* functions. This is considered best practice by many programmers.

> Prefer writing non-friend non-member functions
>
> —Item 44 of *C++ Coding Standards*, by Herb Sutter and Andrei Alexandrescu

Compare to the functionally similar friend, member overload for `operator==`:

```
friend bool operator==(const item& x, const item& y) {
  return x.value == y.value;
}
```

- A non-friend function does not automatically know that a function is part of a class template unless told.

  This is why the non-friend functions repeat the template declaration from the `struct`.
- The friend functions declared in the class are implicitly *inlined*. The compiler *may* replace function calls to these functions with in-line copies of the function body. The compiler is not obligated to do so, but usually does.

  To get the same behavior from non-member functions, the inline keyword is used.

- The `friend` keyword is often used to provide private member access to non-member functions. In the case of the `item` struct, this wasn't needed.

  The use of friend here prevents the `this` pointer from being passed to functions declared (and in this case defined) in the data structure body.

More to Explore

- friend specifier
- Item 44 from C++ Coding Standards, Sutter and Alexandrescu, 2004.

3.10 Class design

This chapter focuses on some of the design considerations that classes and object oriented programming make possible.

3.10.1 Object-oriented concepts

Object-oriented programming (OOP) is a programming paradigm based on the concept of *objects*, which are *data structures* that contain data, in the form of fields (or attributes) and code, in the form of procedures, (or functions, or methods). A distinguishing feature of objects is that an object's procedures provide access to and modify its fields.

In object-oriented programming, computer programs are created out of objects that interact with one another. There is significant diversity in object-oriented programming, but most popular languages are class-based, meaning that objects are instances of classes, which typically also determines their type.

Object orientation is an outgrowth of procedural programming. Procedural programming is based upon the concept of the procedure call. Procedures, also known as functions, subroutines, or methods define the computational steps to be carried out.

Any given procedure might be called at any point during a program's execution, including by other procedures or itself. Procedural programming is a list or set of instructions telling a computer what to do step by step and how to perform from the first code to the second code. Procedural programming languages include C, Fortran, Pascal, and BASIC.

The focus of procedural programming is to break down a programming task into a collection of variables, data structures, and functions. In object-oriented programming the primary effort involves breaking down a programming task into **classes**. Each class defines data that is commonly private. The private data is manipulated using the publicly available functions. The public functions define the class **interface**.

The most important distinction is that while procedural programming uses procedures to operate on data structures, object-oriented programming bundles the two together, so an **object**, which is an **instance of a class**, manipulates its own data.

Object-oriented programming principles

There are many views on the main features and motivations for object oriented programming[1][2]. There are 4 principles that apply to most:

Encapsulation
: Encapsulation refers to the creation of self-contained modules (classes) that bind processing functions to its data members. The data within each class is kept private. Each class defines rules for what is publicly visible and what modifications are allowed.

[1] Wikipedia OO fundamental concepts
[2] SOLID Object oriented design principles

Inheritance
Classes may be created in hierarchies, and inheritance lets the structure and methods in one class pass down the *class hierarchy*. By *inheriting* code, complex behaviors emerge through the reuse of code in a parent class. If a step is added at the bottom of a hierarchy, only the processing and data associated with that unique step must be added. Everything else above that step may be inherited. Reuse is considered a major advantage of object orientation.

Polymorphism
Object oriented programming lets programmers create procedures for objects whose exact type is not known until runtime. For example, a screen cursor may change its shape from an arrow to a line depending on the program mode. The routine to move the cursor on screen in response to mouse movement can be written for "cursor", and polymorphism lets the right version for the given shape be called.

Abstraction
An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

Abstraction denotes a model, a view, or some other focused representation for an actual item. It's the development of a software object to represent an object we can find in the real world. Encapsulation hides the details of that implementation.

Encapsulation

Consider the following example:

```
#include <string>

// A class with no encapsulation
struct BadShipping {
  int weight;      // in pounds
  int distance;    // in miles
  std::string address;
};

int main()
{
    BadShipping bad;
    bad.weight = -3;  // Nothing prevents me from doing this
}
```

It's clearly a bad idea to allow people to set the shipping weight to a negative value. In this case, we say our class has *invariants*: constraints on data that must be preserved for the class to remain valid. In this case, the shipping weight of items must be > 0.

How can you change this class to prevent problems like this from happening? One solution is to make the `weight` private and write a method that allows the class to set limits on weight. Unfortunately, if we had already delivered our BadShipping code, then as you make this 'fix', you break every class that currently uses it, including those that already obey the class invariant. It would have been much better to deliver code that could have been more easily changed in the first place.

The ability to change your code without breaking every class that uses it is one of the key benefits of encapsulation. By limiting access and hiding the implementation details of your class to the maximum extent possible, you make it possible to change, fix, extend, or rework your class without requiring changes in any of the code that uses your class.

How do we ensure our code remains flexible and maintainable?

- Keep fields hidden using a *private* access modifier
- Make *public accessor methods* and force callers to use them by hiding your fields.
- Encourage programming to *interfaces* instead of *implementation*. More on this later.

Compare our first example with the following:

```
// A class with simple encapsulation
class BetterShipping {
  public:
    unsigned weight()   { return weight_; }
    void weight(int value) {
      weight_ = value;
    }

    unsigned distance() { return distance_; }
    std::string address() {
      return address_;
    }
    // other mutators ommitted . . .
  private:
    unsigned weight_;      // in pounds
    unsigned distance_;    // in miles
    std::string address_;
};
```

You might be thinking "Hey! How is this any better than the first example?" We added methods to set and get the weight, but added no new capability. What have we gained?

We have gained quite a bit. Now we are free to change our minds about how weight values are set and retrieved. Even though we aren't doing anything now, we are free to change the implementation later and no calling class will know.

Good OO design demands thinking about the future. Which brings us to our final example. No classes would need to be modified to add the new capability below.

```
#include <algorithm>
#include <string>

static constexpr int min_weight = 1;

class EvenBetterShipping {
  public:
    EvenBetterShipping() = default;
    EvenBetterShipping(int w, int d, std::string a) :
      weight_{std::max(min_weight, w)}, distance_{d}, address_{a}
    {}

    int  weight()          { return weight_; }
    void weight(int value) {
      weight_ = std::max(min_weight, value);  // no upper limit on weight
    }


    int distance() { return distance_; }
    std::string address() {
```

(continues on next page)

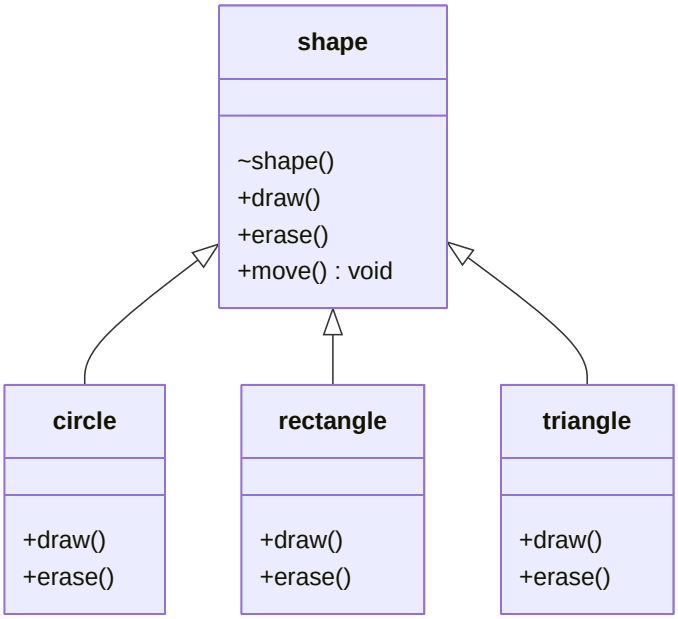
(continued from previous page)

```
    return address_;
  }
private:
  int weight_ = 2;         // in pounds
  int distance_ = 100;     // in miles
  std::string address_ = "My mom's house";
};
```

Inheritance

Inheritance enables new classes to receive --- or inherit --- the properties and methods of existing classes. Inheritance is a programming strategy used to increase the flexibility of your objects. In particular, inheritance is **not** a code reuse strategy. The purpose of inheritance in C++ is to express interface compliance (creating a subtype), not to reuse code. In C++, code reuse usually comes via composition rather than via inheritance. In other words, inheritance is mainly a specification technique rather than an implementation technique.

It is common to draw inheritance relationships like this:



This is different from extending classes through *composition*. More comparisons between inheritance and composition will be made in later sections.

Polymorphism

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped" and polymorphism itself comes in two distinct forms:

- *Runtime polymorphism*

  Base classes may define and implement abstract, or virtual methods, and derived classes can override them, which means they provide their own definition and implementation. At runtime, when client code calls the method, the type is resolved and invokes that override of the virtual method. Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

  At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this occurs, the object's declared type is no longer identical to its runtime type.

  Note that a derived class may be treated as any type in its inheritance hierarchy. Also, it is perfectly valid for an overloaded method to be overridden.

- *Compile-time polymorphism*

  Compile-time polymorphism is simply method overloading. **Overloaded** methods have the same method name but different number of arguments or different types of arguments or both.

Abstraction

One of the key advantages of object oriented languages over *procedural* languages is that objects act as metaphors for the real-world. In other words, objects *model* the real world. In a procedural language, tasks are executed in functions or procedures and the data that the functions operate on is stored elsewhere. A better way to manage the complexity of large programs is to keep the data in a program and the operations allowed on that data in a cohesive logical unit. A program describing a car might perform basic tasks: steer, speed up, slow down, but also needs to store information about the car: current speed, direction, cruise control setting, etc.

If you wrote your car driving program in a procedural language, you would likely require different functions to control each of the car behaviors. You might create functions for `turnCarOn()`, `turnCarOff()`, `accelerate()`, `steer()`, and others. You would also need variables to store the current state of the car. Although it's perfectly valid to construct such a car in a procedural language, these functions and variables we have created only exist as a whole entity, a *car* in the mind of the programmer who created it. The idea that individual units within a program each have a specific role or responsibility is called *cohesion* and is difficult to achieve in procedural programs.

For very large programs, which might contain hundreds or even thousands of entities, lack of cohesion can introduce errors, make programs more difficult to understand and maintain, and complicate the development of very large programs.

More to Explore

- SOLID Principles and the Arts of Finding the Beach.
- Slides: 'Why Every Element of SOLID is Wrong', by Dan North, 2017. With some additional backstory written in 2021.

Footnotes

3.10.2 Unified modeling language

The Unified Modeling Language, or UML, is an industry standard graphical notation for describing and analysing software designs. The symbols and graphs used in the UML are an outgrowth of efforts in the 1980's and early 1990's to devise standards for Computer-Aided Software Engineering (CASE). UML represents a **unification** of these efforts. In 1994 - 1995 several leaders in the development of modeling languages, Grady Booch, Ivar Jacobson, and James Rumbaugh, attempted to unify their work. To eliminate the method fragmentation that they concluded was impeding commercial adoption of modeling tools, they developed UML, which provided a level playing field for all tool vendors.

UML has been accepted as a standard by the Object Management Group[1] (OMG). The OMG is a non-profit organization with about 700 members that sets standards for distributed object-oriented computing.

UML was initially largely funded by the employer of Booch, Jacobson & Rumbaugh, aka *the three amigos*, Rational Software, which was sold to IBM in 2002.

A software model is any textual or graphic representation of an aspect of a software system. This could include requirements, behavior, states or how the system is installed. The model is **not** the actual system, rather it describes different aspects of the system to be developed. UML defines a set of diagrams and corresponding rules that can be used to model a system. The diagrams in the UML are generally divided into two broad categories or *views*, **static** and **dynamic**.

This course does not provide anywhere near a comprehensive review of the UML. The intent is to introduce you to the basics you need to understand the designs presented in this course. Since there is an excellent chance you will encounter the UML or something very similar to it in your professional career and the diagrams used in this course are used not only in the UML, but in other modeling systems as well[2][3][4].

Static and Dynamic diagrams

Static diagrams emphasize the static structure of the system, its objects' attributes, methods, and relationships. Static views include:

- Class diagrams and
- Deployment diagrams

In this course we are primarily interested in class diagrams.

Dynamic diagrams emphasize the dynamic behavior of a system, its states or modes and the collaborations between objects. Dynamic views include:

- Sequence diagrams
- State diagrams
- Use Case diagrams

Class diagrams

A simple class

[1] OMG Homepage
[2] Data Flow Diagrams
[3] The Integration DEFinition (IDEF) model family
[4] The Open Group Architecture Framework

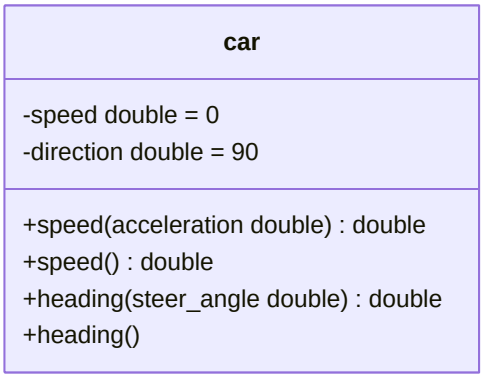
The **class diagram** is one of the most commonly encountered diagrams. It describes the types of objects in a system and the kinds of static relationships that exist among them.

In UML, a class is represented by a rectangle with one or more horizontal compartments. By convention, the class

name starts with a capital letter. Another convention is to italicize the class name if the class is an *AbstractClass*. The top compartment holds the name of the class. The name of the class is the only required field in a class diagram. The middle compartment of the class rectangle holds the list of the class attributes. The bottom compartment holds the list of methods.

Some editors will omit the middle and lower compartments if they are blank.

Attribute and method visibility is indicated using a single character before the class member. Static members are indicated by underlining the member name. The UML term for static members is *classifier members*.



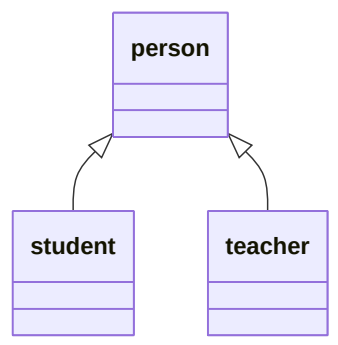
The UML syntax for an attribute is: *visibility name : type = defaultValue*

Symbol	Visibility
+	Public
-	Private
#	Protected
/	Derived

Class diagrams use different notational standards to display class inheritance, class composition, and other associations.

Inheritance relationships

Inheritance



Generalization in action:

Students and Teachers are both People

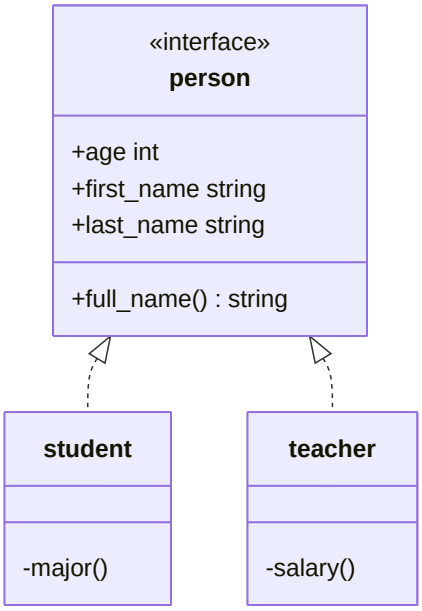
In the UML, the Inheritance relationship is referred to as a *generalization*.

Inheritance is drawn as an empty arrow, pointing from the derived class to the base class. The base class is considered a *generalization* of the derived class, so it makes sense that the arrow should point to the base class. The arrow is trying to say that the derived class **IS A** type of the base class.

In the example diagram, two classes inherit from the more general base class. Some UML drawing tools draw each inheritance line as a separate straight line to the base class, other merge lines when possible. This has no impact on the meaning of the relationship. A merged line showing relationships does not imply that the two derived classes are in any way interdependent, other than they share a common ancestor.

Realization relationships

A *realization* is a relationship between two model elements, in which one model element (the client) realizes (implements or executes) the behavior that the other model element (the supplier) specifies.



The UML graphical representation of a realization is a hollow triangle shape on the interface end of the dashed line (or tree of lines) that connects it to one or more implementers.

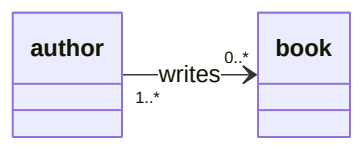
In this course, we are primarily concerned with relationships between classes. Note the addition at the top of the **Person** class: `<<interface>>`. The angle brackets define a *stereotype*. The stereotype allows UML modelers to extend the vocabulary of a model element or to be more specific about the role or purpose of a model element. In this case, the stereotype `<<interface>>` tells us this is not just any old class, but this class defines an *interface*.

Notice the similarity between the **generalization** relationship and the **realization** relationship. **Generalization** always models **inheritance** relationships between classes. **Realization** always models **interface implementation** relationships between classes.

Association

An association represents a relationship between two classes.

Association



An association between two classes is shown by a line joining the two classes. Association indicates that one class uses an attribute or calls methods of another class. If there is no arrow on the line, the association is taken to be bi-directional, that is, both classes hold information about the other class. A unidirectional association is indicated by an

arrow pointing from the object which holds to the object that is held.

Association is the least specific type of association. It is used when the classes each have their own life cycle and are independent of each other. For example, two classes might be related because one or both takes the other as a parameter to a method.

```
struct Author {
  void write(Book b) {
    // do something with the Book
  }
};
```

Multiplicity

Associations have a multiplicity (sometimes called cardinality) that indicates how many objects of each class can legitimately be involved in a given relationship. Multiplicity is expressed using an `n..m` notation near one end of the association line, close to the class whose multiplicity in the association we want to show.

Here `n` refers to the minimum number of class instances that may be involved in the association, and `m` to the maximum number of such instances. If `n = m` only the `n` value is shown. An optional relationship is expressed by writing `0` as the minimum number. The wildcard character * is used to represent the concept *zero or more*.

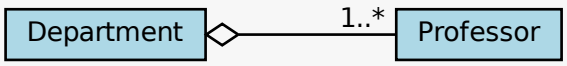
Example multiplicity values

Cardinality and modality	Multiplicity Values
One-to-one and mandatory	`1`
One-to-one and optional	`0..1`
One-to-many and mandatory	`1..*`
One-to-many and optional	`*`
With lower bound `l` and upper bound `u`	`l..u`
With lower bound `l` and no upper bound	`l..*`

Aggregation

If an association conveys information that one object is part of another object, but their lifetimes are independent (they could exist independently), then this relationship is called aggregation.

Aggregation



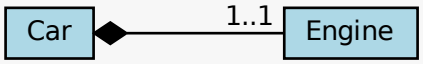
For example, a university owns various departments (e.g., chemistry), and each department has a number of professors. If the university closes, the departments will no longer exist, but the professors in those departments will continue to exist. Therefore, a University can be seen as a composition of departments, whereas departments have an aggregation of professors. In addition, a Professor could work in more than one department, but a department could not be part of more than one university. For example:

```
struct Department {
  Professor* prof;    // non-owning pointer to a professor
};
```

A `Department` **has-a** `Professor`, but the professor exists independently of any department.

Compositon

Composiiton



A car not only *has* an engine, it *owns* it.

Composition is even more specific than aggregation. Like aggregation, one class *has an* instance of another class, but the child class's instance life cycle is dependent on the parent class's instance life cycle. In other words, when the parent dies, the child dies.

Tip

Use aggregation judiciously

> Few things in the UML cause more consternation than aggregation and composition, in particular how they vary from regular association.
>
> The full story is muddled by history. In the pre-UML methods there was a common notation of defining some form of part --- whole relationships. The trouble was that each method defined different semantics for these relationships (although to be fair, some of these were pretty semantics free).
>
> So when the time came to standardize, lots of people wanted part --- whole relationships, but they couldn't agree on what they meant. So the UML introduced two relationships.
>
> **aggregation** (*white diamond*) has no semantics beyond a regular association. It is, as Jim Rumbaugh puts it, a modeling placebo. People can, and do, use it --- but there are no standard meanings for it. I would advise not using it yourself without some form of explanation.
>
> **composition** (*black diamond*) does carry semantics. The most particular is that an object can only be part of one composition relationship. So even if both windows and panels can hold menu bars, any instance of menu bar must be held by only one whole. This is a constraint you can't easily express with the regular multiplicity markers.
>
> —Martin Fowler, AggregationAndComposition blog post 17 May 2003.

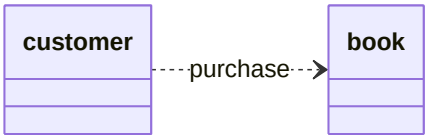
Dependency

Dependency is the weakest relationship. It represents a reference to class passed in as a method parameter to a function in another class. For example, an instance of class book is passed as a parameter to a function in class customer:

```
struct book {};

struct customer {
   void purchase(book b) {}
};
```

The *customer* class requires the *book* class to function, but doesn't own it. The caller of the purchase method is required to supply a *book*.

This type of relationship is represented with a dashed line:



As discussed in the introduction to this section, the UML is much more involved than simple class diagrams. For our purposes, which currently are limited to visualizing inheritance and composition, this is enough.

More to Explore

- Scott Ambler has a good introduction to object oriented programming in general and UML diagrams
- More example diagrams and explanations can be viewed at uml-diagrams.org.

Footnotes

3.10.3 Composition

Every class definition also defines a new *type* that becomes available for use anywhere a built in type could be used.

Composition is also fundamental to *every* object oriented language. It is natural to think about things in terms of parts and components. It would be difficult to break down complex problems into solvable chunks without composition.

The simplest way to incorporate a class is to just use the type directly in another class. A class can be made up of any number and type of other objects, in any combination needed to implement the capabilities desired in the new class. Because this results in composing a new class from existing classes, this concept is called *composition*. Composition is often described as a "has-a" relationship, as in "a car has an engine."

Composition models the relation where two object lifetimes are linked:

- When a Car is created, it comes with an Engine.
- The Engine can exist only as long as the Car exists.
- The Engine exists solely for the benefit of the Car that contains the Engine
- No other car can use this engine.
- When the Car is destroyed, the Engine is destroyed.

```
struct Engine { . . . };
struct Tire   { . . . };
struct Stereo { . . . };

struct Car {
  Engine e;
  Stereo s;
  std::array<Tire, 4> tires;
};

int main() {
  {  // create a temporary scope

    // create a car (and all its components)
    Car c;
  } // when the Car goes out of scope,
    // both the car and its components are destroyed
}
```

Composition provides a great deal of flexibility. Member objects are usually private, making them inaccessible to the users of a type. This enables changing those members without disturbing existing client code. You can also change the member objects at runtime, to dynamically change the behavior of your program. Inheritance, which is described next, does not have this flexibility since the compiler must place compile-time restrictions on classes created with inheritance.

Guideline

Prefer composition over inheritance.

Because inheritance is so important in object-oriented programming it is often highly emphasized, and a new programmer might think that inheritance should be used everywhere. This can result in awkward and overly-complicated designs. Instead, prefer composition when creating new classes, since it is simpler and more flexible. If you take this approach, your designs will stay cleaner.

More to Explore

- Effective C++ Item #40: Model HAS-A using composition Note that Meyers uses the term *layering* as a synonym for composition.
- McLaughlin, Pollice, and West, *Head First Object-Oriented Analysis and Design*. Chapters 2-3, covers "has-a" relationships and class collaboration.
- Aggregation and Composition

3.10.4 Inheritance

Inheritance enables new classes to receive --- or inherit --- the properties and methods of existing classes. In C++, we can extend a class, adding data members or behavior in a new type. Consider the following example:

```
enum class Color {RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET};

class shape {
    Color color_ = Color::BLUE;
  public:
    virtual ~shape() = default;
    void   color (Color new_color) { color_ = new_color; }
    Color  color ()  const         { return color_; }
    virtual void   move();
};
```

The class `shape` defines common behaviors that can be shared among all classes. This shape class provides some definitions and requires derived classes to provide others.

In the base class `shape`, the `virtual` keyword instructs the compiler that the marked functions can be *overridden* in derived classes. The shape example illustrates different ways base class functions may be implemented.

The `color` functions are not marked virtual. These functions are inherited by all derived classes, and cannot be overridden. These functions represent a *mandatory implementation*. In this design, **every** shape must have a color, and it is changed using these functions.

The `move` function is marked virtual. A default implementation is defined in the `shape` class, but derived classes are free to override it if needed.

The shape class destructor: `~shape()` is also marked virtual. Even if this class manages no resources, it is a good idea to define a virtual destructor for every base class. This allows derived constructors to be called if the base class is used in a container.

Derived classes declare their bases immediately after the derived class name. The general format is:

```
class derived_name: {access_modifier} base_name, {access_modifier} base2_name, . . .
```

Applying this to our base class shape we can define a circle like this:

```
class circle: public shape {
   double radius = 1;
  public:
    void   move() override;
};
```

Which creates a by default a blue circle with radius = 1. A circle inherits its ability to change color from its parent: shape. This circle implements its own version of the `move()` function.

C++11 Feature

The keyword `override` tells the compiler that this function intends to *override* a virtual function in a base class. Although this keyword not required, it is very highly recommended since it provides the compiler more information about your intent and can flag functions with incorrect signatures.

Note that a class may inherit from more than one base class.

Base class references and pointers

One of the primary benefits of inheritance become apparent when passed as parameters or when used in container classes. Object references and pointers will call the correct derived class member function in an inheritance hierarchy. For example, some part of our drawing program need to draw **any** shape, without having special case code to determine *what* the shape type is:

```
draw_shape (const shape& s) {
  s.draw();
}

int main() {
  Circle c;
  draw_shape(c);
}
```

Although we can't instantiate a `shape`, we *can* pass a derived class instance (circle, triangle, etc.) to a function that takes a *reference to a shape*. This works because a `circle` **is a** `shape`. A `circle` is both a `circle` **and** a `shape`.

Passing a pointer would work as well as a reference:

```
draw_shape (const shape* s) {
  s->draw();
}

int main() {
  Circle c;
  draw_shape(&c);
}
```

The polymorphism achieved by assigning derived classes **only** works when assignment is through a reference or a pointer.

Recall that containers are limited to values of a single type and that references are not *assignable*. How do we create a vector of `shape` objects? Through a pointer:

```
#include <memory>
#include <vector>

using std::unique_ptr;
using std::make_unique;

draw_all (const std::vector<unique_ptr<shape>>& shapes) {
  for (const auto& s: shapes) {
    s->draw();
  }
}

int main() {
  std::vector<unique_ptr<shape>> shapes;
  shapes.push_back(make_unique<circle>());
  shapes.push_back(make_unique<rectangle>());
  shapes.push_back(make_unique<triangle>());

  draw_all(shapes);
}
```

The vector of unique pointers could have been implemented with a vector of raw pointers:

```
std::vector<shape*> shapes;
shapes.push_back(new circle());
```

This version works essentially the same as the previous version, but requires a bit more code to manage our own memory.

When to use inheritance

Adapted from *Composition vs. Inheritance: How to Choose?*.

The most common --- and beneficial --- use of inheritance is to incrementally extend types. If we need a widget that is just like an existing Widget class, but with a few tweaks and enhancements, then inheritance is suitable. Inheritance is the right choice because our derived class is still a widget. We want to reuse the entire interface and implementation from the base class and our changes are primarily **additive**. That is, the derived class adds capabilities to base. If you find that a derived class is removing things provided by the base class, then question inheriting from that base.

Inheritance is most useful for grouping related sets of concepts, identifying families of classes, and in general organizing the names and concepts that describe the domain. As we delve deeper into the implementation of a system, we may find that our original generalizations about the domain concepts, captured in our inheritance hierarchies, are incorrect. Don't be afraid to disassemble inheritance hierarchies into sets of complementary cooperating interfaces and components when the code leads you in that direction.

There is no substitute for object modeling and critical design thinking. But if you must have some guidelines, consider these.

Inheritance should only be used when:

- Both classes are in the same logical domain
- The derived class is a proper subtype of the base class: think **is a**
- The base class implementation is necessary or appropriate for the derived class

- The enhancements made by the derived class are primarily additive

Private inheritance

In the classes derived from `shape`, we declare `public` members of the shape class to also have `public` access the derived classes. Compare:

```
class circle: public shape {} // case #1: public inheritance

class circle: shape {}        // case #2: private inheritance
```

In the second case, the public members of shape are treated as `private` members of class circle. This is almost always a bug for new programmers and a common source of error.

The default inheritance model in C++ is *private inheritance* for classes and public for structs. In private inheritance **all** of the base class members: data and functions, public, protected, and private, are treated as **private members** of the derived class.

A common question is "Why would we ever do this?"

If a derived class wants to reuse all of the code from a base class, but *not* conform to the interface, then private inheritance is how to achieve that.

Consider std::stack. It is a container that *adapts* the capabilities of an underlying container. Although the default container for a stack is a std::deque, we don't want to expose all of the functions of a `deque` in a `stack`.

Non-virtual base class functions

Every non-virtual base class function defines a **mandatory interface** for all derived classes. The language allows a derived class to implement its own version. For example:

```
struct B {
  void foo();
};
struct D: B {
  void foo();        // derived class D has its own version
};
```

If class `D` implements its own version of `foo`, then this is **not** an `override`. This is called *shadowing* and is often a bug.

The problem is this:

- An instance of `B` will always call `B::foo()`
- An instance of `D` will always call `D::foo()`
- An instance of `D` in a container of pointers to `B` will call `B::foo()`.

  New programmers are often caught off guard by this behavior.

  Most modern compilers will warn about this.

Even if the programmer is careful to ensure the contract defined by `B::foo()` is also met by struct D, there is no guarantee this can't change in the future. There is no way to know what else may depend on the contract defined by `B::foo` or any if its invariants.

In general, if a derived class can't use the existing mandatory interface defined by a base class, then it probably shouldn't be a derived class.

Multiple inheritance

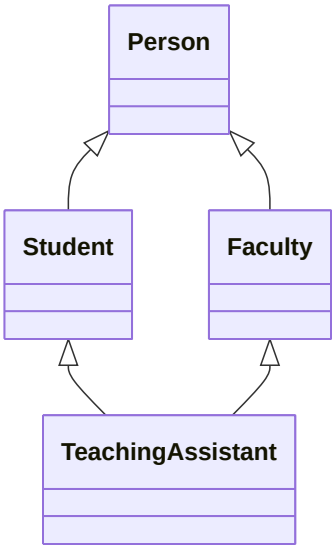
C++ allows for a single class to inherit capabilities from more than 1 class. The constructors of inherited classes are called in the same order in which they are inherited. For example, in the following program, B's constructor is called before A's constructor.

```
#include<iostream>
using std::cout;

struct A {
  A() { cout << "construct A\n"; }
};

struct B {
  B() { cout << "construct B\n"; }
};

struct C: public B, public A {     // Note the order
  C() { cout << "construct C\n"; }
};

int main() {
  C c;
}
```

The destructors are called in reverse order of constructors.

The Diamond of Death

Since C++ allows multiple inheritance, the following relationships are valid:



The `TeachingAssistant` class is both a `Faculty` and a `Student` and inherits two copies of the `Person` base class data. When a TA is created, the Person constructor is called *twice*. Once for each copy of the Person stored. This is both wasteful and creates ambiguities.

The C++ solution to this problem is to inherit *virtual base* classes. For each distinct base class that is specified virtual, the most derived object contains only one base class sub-object of that type, even if the class appears many times in the inheritance hierarchy (as long as it is inherited virtual every time). For example:

```
#include<iostream>
#include<string>
using std::cout;
using std::string;

struct Person {
    explicit
    Person(string n)  { cout << "Person(" << n << ") called\n";   }
    Person()          { cout << "Person() called\n";   }
};

struct Faculty : virtual public Person {
    explicit
    Faculty(string n) : Person(n)   {
       cout<<"Faculty(" << n << ") called\n";
    }
};

struct Student : virtual public Person {
    explicit
    Student(string n) : Person(n) {
        cout<<"Student(" << n << ") called\n";
    }
};

struct TeachingAssistant : public Faculty, public Student  {
    explicit
    TeachingAssistant(string n)
      : Faculty(n), Student(n)   {
        cout<<"TA(" << n << ") called\n";
    }
};

int main()  {
    TeachingAssistant ta("Alice");
}
```

This solves the 'multiple grandparent problem' for the teaching assistant class, but note that the **default** Person constructor is called. If the name is stored in the Person class, then we need to call the non-default constructor.

The `Person(string)` constructor can be explicitly called in the `TeachingAssistant` initializer. In order for `Faculty` and `Student` to initialize correctly, the Person class must be constructed first:

```
explicit
    TeachingAssistant(string n)
      : Person(n), Faculty(n), Student(n)   { . . . }
```

Try This!

Change the TA signature in the previous active code example to call the 1 argument Person constructor.

What about the situation where `Person` defines a virtual function, which is overridden by `Faculty` and `Student`?

Which version of the function is invoked?

There is no way to know. Technically, any version could be called. The standard doesn't specify anything in this situation Most compilers will essentially bail and not call **any** of the functions.

The TA class can resolve the ambiguity by explicitly calling a specific base class function. The derived class must call the fully qualified name of the function like this:

```
TeachingAssistant::foo() {
   if (weekday) {
      Faculty::foo();
   } else {
      Student::foo();
   }
}
```

There is no obligation to always call all implementing functions, but in practice, this is often needed.

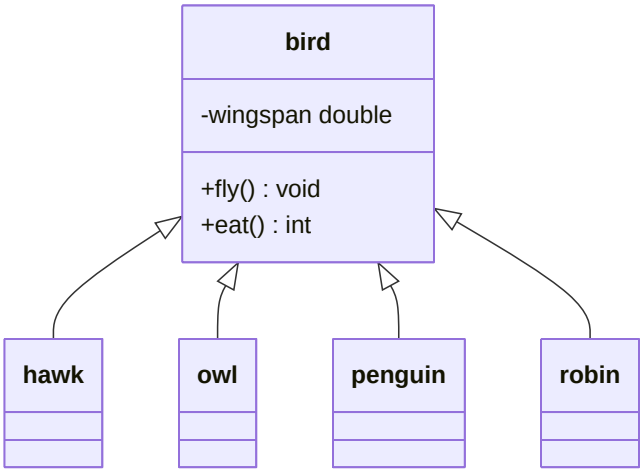
Note that this defeats the entire purpose of having runtime polymorphism. The derived class at the end of the inheritance chain might need code containing 'knowledge' about **all** of its ancestor classes. This is partly why the diamond is considered 'deadly'.

> **Do we really need multiple inheritance?**
>
> Not really. We can do without multiple inheritance by using workarounds, exactly as we can do without single inheritance by using workarounds. We can even do without classes by using workarounds. C is a proof of that contention. However, every modern language with static type checking and inheritance provides some form of multiple inheritance. In C++, abstract classes often serve as interfaces and a class can have many interfaces. Other languages -- often deemed "not MI" -- simply has a separate name for their equivalent to a pure abstract class: an interface. The reason languages provide inheritance (both single and multiple) is that language-supported inheritance is typically superior to workarounds (e.g. use of forwarding functions to sub-objects or separately allocated objects) for ease of programming, for detecting logical problems, for maintainability, and often for performance.
>
> —Bjarne Stroustrup's C++ Style and Technique FAQ

Design problems

New programmers are generally eager to "do things the OO way" and tend to overuse inheritance relationships. This is especially true if starting with UML diagrams: many diagram look 'too simple' without a lot of boxes connected by generalization and dependency relations.

Consider the following classes.


bird
-wingspan double
+fly() : void
+eat() : int
hawk
owl
penguin
robin

Is this OK?

No.

We have asserted that a penguin can fly.

We might choose to implement `fly()` in our penguin class and simply do nothing, but generally when we do that we are coding our way around a basic design problem.

We will explore solutions for fixing these types of design problems in the next section.

Guideline

Prefer composition over inheritance.

It is very important when creating a class hierarchy using inheritance that *every* derived class passes the **is a** test for **all** of its bases. For example:

```
struct oven: public kitchen { . . . };
```

This is not a proper relationship. An oven is a thing commonly *found* in a kitchen, but that does not mean an oven *is a* kitchen. Because it fails this basic test, it is likely that variables and functions that apply to the base: `cupboards`, `sink`, `enter_room()`, etc will fail to make sense when applied to the derived class.

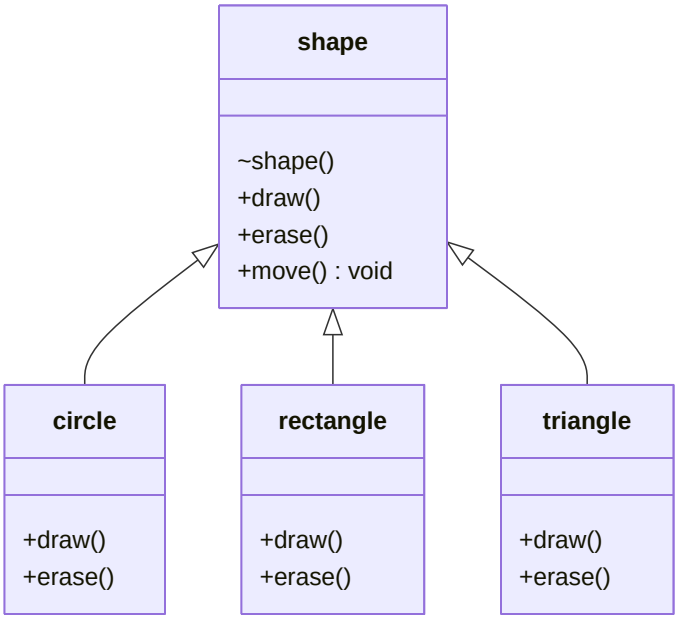
This is an example better modelled through composition. A kitchen **has a** sink in it.

More to Explore

- Derived classes from cppreference.com
- Ineritance basics from the C++ FAQ
- Effective C++ #35: Make sure public inheritance models "IS-A"
- Effective C++ #36: Never redefine inherited non-virtual functions
- Composition vs. Inheritance: How to Choose?

3.10.5 Abstract base classes

Often the functions declared in a base class cannot be defined in the base class, but instead can only be implemented in the derived classes. Consider the shape example from the section *Inheritance*. How might we add `draw` and `erase` functions?



There is no situation where drawing a 'generic shape' makes sense. A shape might define general behaviors like `draw()`, but there is no all-purpose draw function. The shape class *can't* know the implementation ahead of time. There is a way to code this in C++: by creating an *abstract base class*. In C++, an abstract class is any class with at least one

unimplemented function:

```
class enum Color {RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET};

class shape {
    Color color_ = Color::BLUE;
  public:
    virtual ~shape() = default;
    void   color (Color new_color) { color_ = new_color; }
    Color  color ()  const         { return color_; }
    virtual void   move();      // implemented in shape.cpp
    virtual void   draw() = 0;
    virtual void   erase() = 0;
};
```

The functions `draw` and `erase` are marked virtual. Note the `= 0;` at the end of the declaration. Referred to as the *pure specifier*, this marks these functions as *pure virtual*. It is the presence of at least 1 pure virtual function that make a class *abstract*. In this case, a `shape` does not know how to draw itself. Code that can only properly be implemented in the class that properly 'owns' the behavior (`draw` and `erase`) should be implemented in the *derived* classes.

Because a class the defines a pure virtual function cannot implement it, that means any class containing a pure virtual function can never be instantiated. Given the shape class defined here, this code:

```
shape s;
```

will not compile.

A class containing at least one pure virtual function can **only** be used as a base class.

```
class circle: public shape {
   double radius = 1;
  public:
    void   draw()  const override;
    void   erase() override;
};

class rectangle: public shape {
   double ht = 1;
   double wd = 1;
  public:
    void   draw()  const override;
    void   erase() override;
};
```

There is no requirement that pure virtual functions be implemented in the first derived class that inherits from a base. A derived class *could* implement it, but can itself remain abstract. A non-abstract derived class can make itself abstract by declaring a new pure virtual function, or by declaring an existing virtual function as pure virtual.

```
struct Base {
    virtual void f() = 0; // pure virtual
};

struct X : Base {
    void f() override {} // non-pure virtual
    virtual void g();     // non-pure virtual
```

(continues on next page)

(continued from previous page)

```
};

struct Y : X {
    void g() override = 0; // pure virtual overrider
};

struct Z : Y {
    void g() override {}; // non-pure virtual
};

int main()
{
    Base b;       // Error: abstract class
    X x;          // OK

    Base& b = x; // OK to reference abstract base
    b.f();       // virtual dispatch to X::f()
    Y y;         // Error: abstract class (final overrider of g() is pure)
    Z z;         // OK: final overrider of g() is non-pure
}
```

Interfaces

When an abstract base class declares no member variables and declares **only** pure virtual functions, then the class is referred to as an *interface*. Technically, *every* class with at least 1 member function defines an interface, however, some languages give interface classes special treatment, so the term has fallen into use in C++ also.

We can rewrite the abstract class `shape` and convert it to behave like an interface:

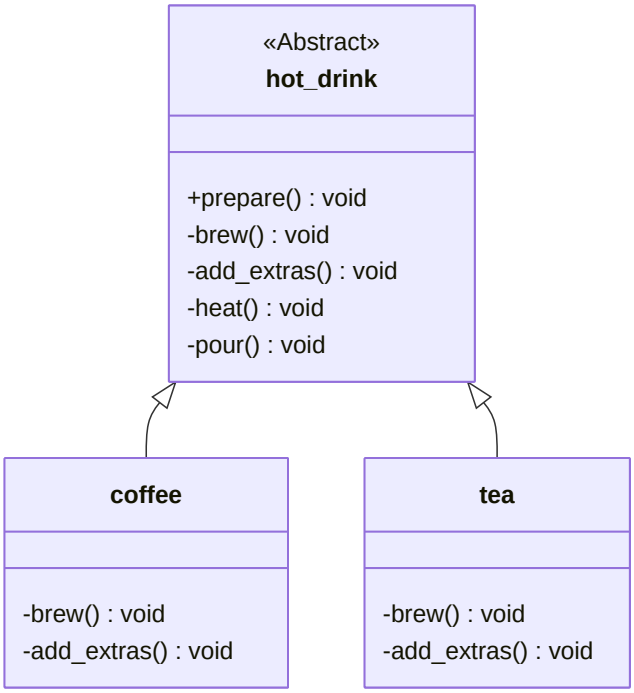
```
struct shape {
    virtual ~shape() = default;
    virtual void   move() = 0;      // implemented in shape.cpp
    virtual void   draw() = 0;
    virtual void   erase() = 0;
};
```

Notice we still have a default implementation for move, even though we declared it to be pure virtual.

Some experts (for example Herb Sutter in his article *Virtuality*, advocate it as a best practice to always define virtual methods private, unless there is a good reason to make them protected. Virtual methods, in their view, should never be public, because they define the class interface, which must remain consistent in all derived classes. Protected and private virtual functions define the class customizable behavior, and there is no need to make them public. A public virtual method would define both interface and a customization point, a duality that could reflect weak design.

The primary problem with this type of public interface is that it doesn't hold up well to changing requirements. If `draw` or `erase` need to return values, or if we need to add more pure virtual functions, every derived class is affected.

We can protect ourselves from future changes using the *Template Method* design pattern. The Template Method defines the steps of an algorithm and allows derived classes to provide implementations for one or more steps.



We now have a generic framework for making all kinds of brewed hot drinks, assuming they follow this basic recipe such as coffees or teas.

```
class hot_drink {
   public:
     ~hot_drink() = default;
     void prepare();       // the template method

   private:
     // ALL implementation steps are private
     virtual void brew() = 0;
     virtual void add_extras();

     // non-virtual mandatory parts of the recipe
     void heat();
     void pour();
};

void hot_drink::prepare()
{
  heat();
  brew();        // call to the derived class
  pour();
  add_extras();  // optionally call derived class
}
void hot_drink::add_extras()
{
  // this default implementation could just be a stub
  // it can optionally be customized by a derived class
}
```

It may be surprising that private virtual functions can be overridden, let alone are valid. You have likely been taught that private members in a base class are not accessible in classes derived from it, which is correct. However this inaccessibility **by** the derived class does not have anything to do with the virtual call mechanism, which is **to** the derived class.

What good is a method that the derived class can't call? Even though the derived class can't call it in the base class, the base class can call it --- down to the appropriate derived class. And that's what the Template Method pattern is all about.

In our coffee shop, we want to write the base class once, but still want the flexibility to think of new drink recipes today. The base class code we wrote a year ago will call the private virtual methods in its recipe. This might result in the base class calling code that did not exist when the base class was originally written.

Once our base class is done, we can implement derived classes:

```
class coffee : public hot_drink {
   public:
     ~coffee() = default;
   private:
     virtual void brew() = 0;
     virtual void add_extras();
};

void coffee::brew()
{
  // drip water through grinds
```

(continues on next page)

(continued from previous page)

```
}
void coffee::add_extras()
{
  // add milk and sugar
}

class tea : public hot_drink {
   public:
     ~tea() = default;
   private:
     virtual void brew() = 0;
     virtual void add_extras();
};

void tea::brew()
{
  // steep tea in water
}
void tea::add_extras()
{
  // add lemon
}
```

And use it:

```
int main() {
  coffee c;
  c.prepare();

  tea t;
  t.prepare();
}
```

The final specifier

Added in C++11, this keyword specifies that a virtual function cannot be overridden in a derived class or that a class cannot be inherited from.

When used in a virtual function declaration or definition, final ensures that the function is virtual and specifies that it may not be overridden by derived classes. Attempting to override a final function is a compile error.

```
struct Base
{
    virtual void foo();
};

struct A : Base
{
    void foo() final; // A::foo is overridden and it is the final override
    void bar() final; // Error: non-virtual function cannot be overridden or be final
};

struct B final : A // struct B is final
```

(continues on next page)

(continued from previous page)

```
{
    void foo() override; // Error: foo cannot be overridden as it's final in A
};

struct C : B // Error: B is final
{
};
```

More to Explore

- From cppreference.com:
  - virtual function specifier
  - Abstract classes
  - The Template Method design pattern
  - final specifier
- Virtuality by Herb Sutter

3.10.6 Design patterns

A software design pattern is a general, reusable solution to a common software problem. It is not an actual implementation, but rather an idea.

Design patterns gained popularity in computer science after the book *Design Patterns: Elements of Reusable Object-Oriented Software* was published in 1994 by (Gamma et al.). The book had 4 authors and together they became known as the "Gang of Four", or even more briefly the "GoF".

Design patterns are useful for several reasons:

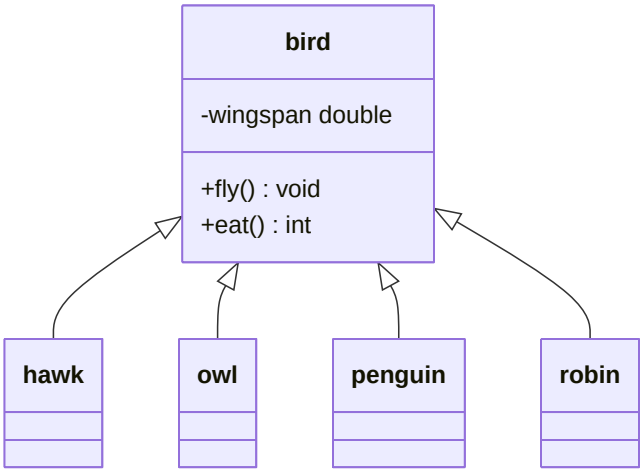
- They provide useful solutions to common problems
- They provide a common vocabulary between programmers when discussing a design
- They exploit lessons learned by other developers

  ... with patterns you get *experience* reuse.

  —Head First Design Patterns, Freeman and Robson, 2014, p. 1.

Fixing an inheritance problem

Recall our flying penguin problem from the section on *Inheritance*:



This type of design error is actually common.

It's easy to overlook a design error, in this case flying penguins. Part of what makes these errors problematic is that

design errors that actually get delivered in the final product are among the most expensive software errors to fix.

So how do we fix our bird design?

We **could** just override the `fly` method for penguins and have the override do nothing. This might solve our immediate problem, but it's not a general solution to this problem:

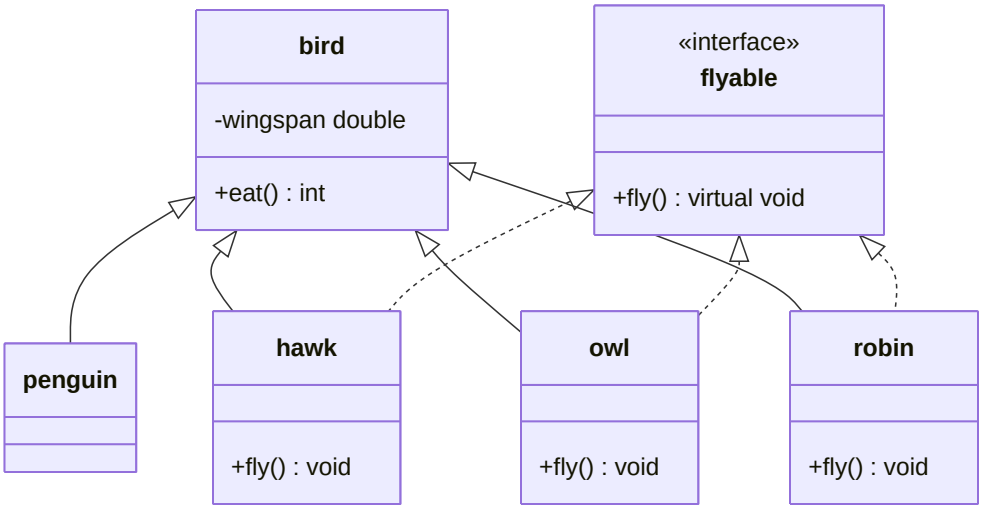
- It doesn't work as well for non-void functions. In this case, `fly` -- a void function -- really could do nothing A non-void function **must** return something. If the function returns something, but that 'something' actually means 'nothing', then we have another design problem. This value now represents a marker that every caller of the function might need to handle and understand that no action was taken.
- What happens when we need behaviors other than `fly`? If every time we have an exception case for a derived type and we create overrides that do nothing, then it can become difficult to understand the true capabilities of any object in the hierarchy just by looking that the interface.

  You have to examine the *code* to see how each (or if) an interface was *implemented*.

  Any design that requires us to look at the source code before we can completely understand it is not a design we want to use.

Another problem with the existing design is that it encourages duplication in derived classes. For example, let's assume we need to model bird flying behavior. Most birds don't just 'fly'. They soar, flock, flit about, dive, or don't fly at all. There are many thousands of different kinds of birds, but relatively few ways of flying.

We *could* decide to solve this problem using an interface:



This does allow limiting the flying behavior to those birds that actually fly, but at a high cost. Now every bird that *does* fly needs to reimplement the code for `fly()`. Derived classes cannot inherit code from each other. Future maintenance of all the duplicated code could be expensive. Our current situation:

- Not all birds fly, so inheritance is not the right choice
- A simple interface solves the inheritance problem, but creates an unacceptable maintenance burden.

There must be a better option.

Note

Accommodating change

How can we isolate the parts of a system that change from the parts that do not change?

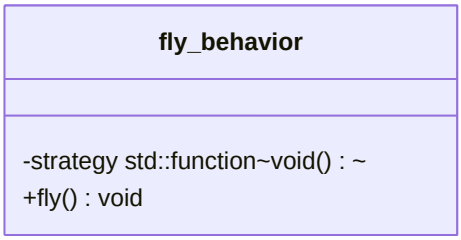
More to Explore

- Software_design_pattern from Wikipedia.
- Design Patterns Are Missing Language Features from the PortlandPatternRepository.
- Revenge of the Nerds an excellent article written by Paul Graham in 2002. About the evolution of language and how modern languages are becoming more like Lisp -- which was discovered in 1958. The end has a short criticism of patterns.

  Keep this date in mind when he uses phrases like 'recently invented': that's still over 20 years ago!

3.10.7 Strategy pattern

We can think of flying as a *strategy* different birds employ to move around. Birds don't *inherit* a fly behavior, they *have it*. The centerpiece of the solution is to isolate many different types of flying behavior behind a single class that stores a pointer to a function implementing the behavior. Each implementation defines a different **strategy** for the interface.



Any callable entity (function, function object, or lambda) is a potential strategy that derived classes of a bird can now use.

```
#include <iostream>
#include <functional>
#include <utility>

#define FunctionObject typename

// an alias to avoid copying std::function<void()> everywhere
using fly_strategy = std::function<void()>;

class fly_behavior {
  public:
    template<FunctionObject F>
    explicit fly_behavior(F strategy)
      : strategy {strategy}
    { }

    void fly() { strategy(); }

  private:
    fly_strategy strategy;
};

// a function object that implements a strategy
struct soar
{
  void operator() () {
   std::cout << "fly by soaring.\n";
  }
};

// a free function can also implement a strategy
void no_flying_allowed() {
  std::cout << "I don't fly.\n";
}
```

The base class now *delgates* the fly behavior to the strategy instead of either defining a single fixed behavior or forcing every derived class to create one. In languages without lambda expressions, each implemented strategy is usually implemented as a separate class, each inheriting from the base strategy class. In C++, an inheritance based solution is possible, but not required. There is no 'best' solution - your needs must drive the final design decision. In general, if the strategy also needs to store state information, then implement as a class or function object. If the strategy is stateless, then implement a functional solution.

```
class bird {
  fly_strategy strategy = soar();

  public:
  bird () = default;
  explicit bird(fly_strategy strategy)
   : strategy(strategy)
  {}
  ~bird () = default;

  // change strategy mid-stream
```

(continues on next page)

(continued from previous page)

```
  void fly_behavior (fly_strategy new_strategy) {
   strategy = new_strategy;
  }

  void fly() {
   strategy();
  }
};
```

In this example, a bird may

- default construct the default soaring strategy, or
- set a strategy when constructed, or
- change the strategy at some point after construction.

An example of birds using the strategy:

```
// a hawk can use the default soar behavior
class hawk : public bird {
  public:
   hawk() = default;
};

// this penguin defines its fly behavior using a free function
class penguin : public bird {
  public:
   penguin()
     : bird(no_flying_allowed)
   {}
};

int main() {
  hawk h;
  h.fly();

  penguin p;
  p.fly();

  // change the behavior for just this penguin
  p.fly_behavior(
     [](){
      std::cout << "With a rocket pack, now I can fly!!\n";
     }
  );
  p.fly();

  return 0;
}
```

Notice that we fixed our inheritance problem by using *composition*. Not only did composition allow us to encapsulate a family of behaviors, it also allowed a simple hook to enable changing the behavior at runtime.

More to Explore

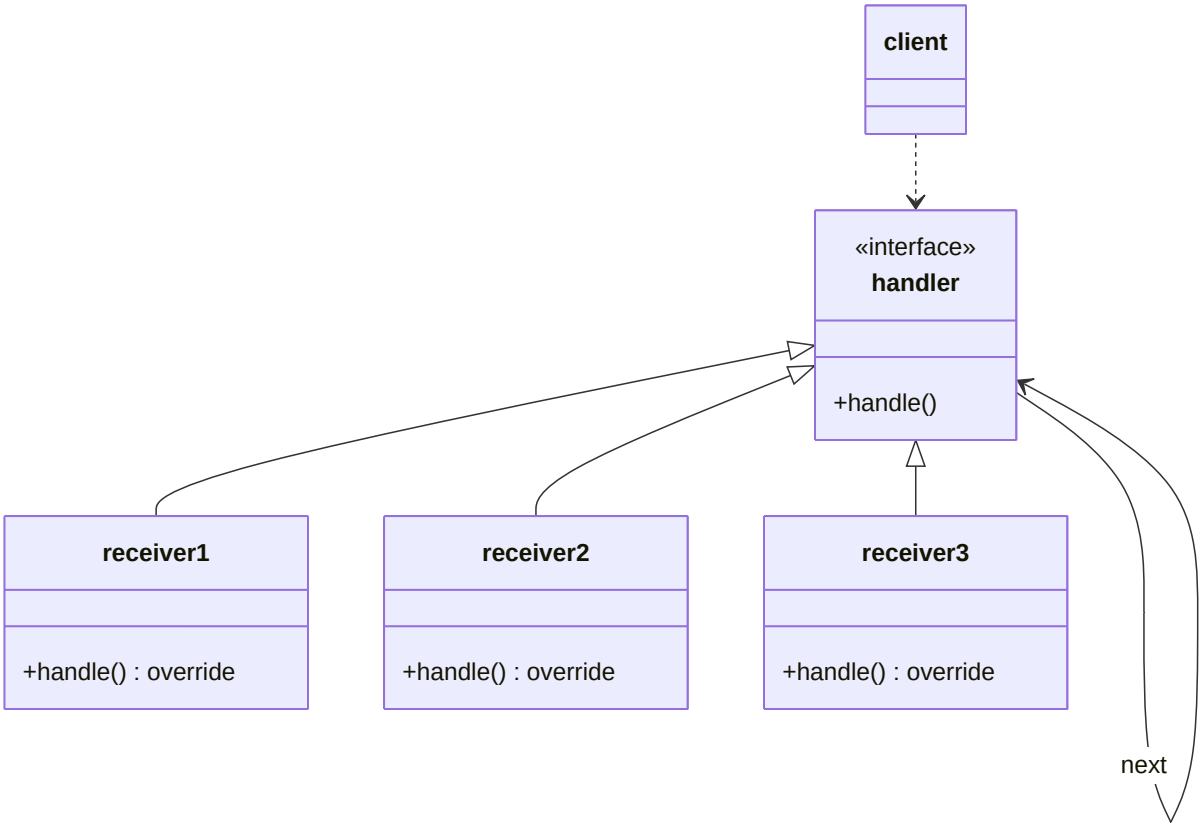
- Strategy Design Pattern on oodesign.com and on Wikipedia.
- Design Patterns Are Missing Language Features from the PortlandPatternRepository.

3.10.8 Chain of Responsibility pattern

Sometimes we need to give more than one object a chance to handle a request. The Chain of Responsibility pattern is a 'behavioral' software design pattern. The goal of the pattern is to separate request senders and request handlers, while giving more than one object a chance to handle the request.

The Chain of Responsibility design pattern allows an object to send a command without knowing what object will receive and handle it. The request is sent from one object to another making them parts of a chain and each object in this chain can handle the command, pass it on or do both. The most usual example of a machine using the Chain of Responsibility is the vending machine coin slot: rather than having a slot for each type of coin, the machine has only one slot for all of them. The dropped coin is routed to the appropriate storage place that is determined by the receiver of the command.

Instead of calling a single function to satisfy a request, multiple functions in the chain have a chance to satisfy the request. Since the chain is effectively a list, it can be dynamically created, so you could also think of it as a more general, dynamically-built switch statement.



The 'classic' implementation of the chain of responsibility implements the design shown in the UML diagram with a handler interface and an abstract handler to encapsulate maintenance of the chain.

Interface

The Handler interface declares methods for building the chain of handlers and for executing a request.

```
struct handler {
    virtual handler* next(handler* handler) = 0;
    virtual std::string handle(std::string request) = 0;
    virtual ~handler() = default;
};
```

The AbstractHandler manages the chain maintenance common to all handlers.

```
class abstract_handler : public handler {
  private:
    handler* next_handler_ = nullptr;

  public:
    abstract_handler() : next_handler_(nullptr) {
    }
    handler* next(handler* link) override {
      next_handler_ = link;
      return link;
    }
    std::string handle(std::string request) override {
      if (next_handler_ != nullptr) {
        return this->next_handler_->handle(request);
      }
      return {};
    }
};
```

Implementing classes

All Concrete Handlers either handle a request or pass it to the next handler in the chain.

```
struct monkey_handler : public abstract_handler {
    std::string handle(std::string request) override {
      if (request == "Banana") {
        return "Monkey: I'll eat the " + request + ".\n";
      } else {
        return abstract_handler::handle(request);
      }
    }
};

struct squirrel_handler : public abstract_handler {
    std::string handle(std::string request) override {
      if (request == "Nut") {
        return "Squirrel: I'll eat the " + request + ".\n";
      } else {
        return abstract_handler::handle(request);
      }
    }
};
```

(continues on next page)

(continued from previous page)

```
struct dog_handler : public abstract_handler {
    std::string handle(std::string request) override {
      if (request == "Meatball") {
        return "Dog: I'll eat the " + request + ".\n";
      } else {
        return abstract_handler::handle(request);
      }
    }
};
```

Client

A client uses the handler to process its data. The client passes each item to be process to the handler one at a time, but it unaware that anything other than the handler is involved.

```
void client(handler& food_handler) {
  std::vector<std::string> food = {"Nut", "Banana", "Cup of coffee"};
  for (const std::string &snack : food) {
    std::cout << "Client: Who wants a " << snack << "?\n";
    const std::string result = food_handler.handle(snack);
    if (!result.empty()) {
      std::cout << "  " << result;
    } else {
      std::cout << "  " << snack << " was left untouched.\n";
    }
  }
}

int main() {
  auto monkey = new monkey_handler;
  auto squirrel = new squirrel_handler;
  auto dog = new dog_handler;
  monkey->next(squirrel)->next(dog);

  client(* monkey);

  delete monkey;
  delete squirrel;
  delete dog;

  return 0;

}
```

However, the linked list of handlers needs to be manually constructed somewhere before it can be used. This gives users flexibility in what is included in the list, but requires knowledge of the internal workings of the chain which are better kept private.

And the memory is the responsibility of users to clean up.

Run It

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

struct handler {
    virtual handler* next(handler* handler) = 0;
    virtual std::string handle(std::string request) = 0;
    virtual ~handler() = default;
};

class abstract_handler : public handler {
  private:
    handler* next_handler_ = nullptr;

  public:
    abstract_handler() : next_handler_(nullptr) {
    }
    handler* next(handler* link) override {
      next_handler_ = link;
      return link;
    }
    std::string handle(std::string request) override {
      if (next_handler_ != nullptr) {
        return this->next_handler_->handle(request);
      }
      return {};
    }
};

struct monkey_handler : public abstract_handler {
    std::string handle(std::string request) override {
      if (request == "Banana") {
        return "Monkey: I'll eat the " + request + ".\n";
      } else {
        return abstract_handler::handle(request);
      }
    }
};

struct squirrel_handler : public abstract_handler {
    std::string handle(std::string request) override {
      if (request == "Nut") {
        return "Squirrel: I'll eat the " + request + ".\n";
      } else {
        return abstract_handler::handle(request);
      }
    }
};

struct dog_handler : public abstract_handler {
    std::string handle(std::string request) override {
```

(continues on next page)

(continued from previous page)

```
      if (request == "Meatball") {
        return "Dog: I'll eat the " + request + ".\n";
      } else {
        return abstract_handler::handle(request);
      }
    }
};


void client(handler& food_handler) {
  std::vector<std::string> food = {"Nut", "Banana", "Cup of coffee"};
  for (const std::string &snack : food) {
    std::cout << "Client: Who wants a " << snack << "?\n";
    const std::string result = food_handler.handle(snack);
    if (!result.empty()) {
      std::cout << "  " << result;
    } else {
      std::cout << "  " << snack << " was left untouched.\n";
    }
  }
}

int main() {
  auto monkey = new monkey_handler;
  auto squirrel = new squirrel_handler;
  auto dog = new dog_handler;

  monkey->next(squirrel)->next(dog);

  client(* monkey);

  delete monkey;
  delete squirrel;
  delete dog;

  return 0;

}
```

The 'classic' Chain of Responsibility in the 'Gang of Four' Patterns book does some work that is not needed - specifically the code related to creating and managing a linked list from scratch. The standard library provides several containers that will work just as well.

Keeping in mind that the essence of Chain of Responsibility is to try a number of solutions until you find one that works, you'll realize that the implementation of the sequencing mechanism is not an essential part of the pattern.

Interface

The interface declares methods for building the chain of handlers and for executing a request.

```
struct food_handler {
    const std::string empty;
    virtual std::string handle(std::string request) const = 0;
```

(continues on next page)

(continued from previous page)

```
    virtual ~handler() = default;
};
```

Notice the code to visit the next node has been removed and the abstract handler that had been used to encapsulate the linked list is no longer needed.

Implementing classes

The implementing classes are mostly the same. Differences:

- Classes now inherit directly from the interface type.
- Instead of returning the value from the next link in the chain, classes return an empty string if they do not handle anything.

  The string is used as an early exit condition from the chain: as soon as some link in the change returns a non-empty string the chain can return the result to the client.

```
struct monkey_food_handler : food_handler {
  std::string handle(std::string request) const override {
    if (request == "Banana") {
      return "Monkey: I'll eat the " + request + ".\n";
    }
    return empty;
  }
};
struct squirrel_food_handler : food_handler {
  std::string handle(std::string request) const override {
    if (request == "Nut") {
      return "Squirrel: I'll eat the " + request + ".\n";
    }
    return empty;
  }
};
struct dog_food_handler : food_handler {
  std::string handle(std::string request) const override {
    if (request == "MeatBall") {
      return "Dog: I'll eat the " + request + ".\n";
    }
    return empty;
  }
};
```

The abstract handler class is no longer abstract and uses a vector to manage the handlers. Any iterable standard library container could be used here, but vector, array, and list would be typical choices.

The constructor builds the chain, which is now completely private.

The big improvement here is no the chain is responsible for managing it's own memory. It defers the task to unique_ptr, but in contrast wi the earlier example, all the memory needed to be managed by everyt *user* of the chain.

```
class food_handlers : food_handler
{
  std::vector<std::unique_ptr<food_handler>> chain;
  public:
```

(continues on next page)

(continued from previous page)

```
    food_handlers() {
      chain.push_back(std::unique_ptr<food_handler>(new monkey_food_handler));
      chain.push_back(std::unique_ptr<food_handler>(new squirrel_food_handler));
      chain.push_back(std::unique_ptr<food_handler>(new dog_food_handler));
    }
    std::string handle(std::string food_item) const override {
      std::string reply;
      for(const auto& link: chain) {
        reply = link->handle(food_item);
        if(!reply.empty()) { return reply; }
      }
      return reply;
    }
};
```

No other code needs to know what classes are actually in the chain.

Client

The signature of the client function has changed slightly to reflect passing in a different type, but the client is essentially untouched. It still loops on each data item and sends each one to the handler for processessing.

```
void client(const food_handlers& eaters) {
  std::vector<std::string> food = {"Nut", "Banana", "Cup of coffee"};
  for (const auto& snack : food) {
    std::cout << "Client: Who wants a " << snack << "?\n";
    const std::string result = eaters.handle(snack);
    if (result.empty()) {
      std::cout << ' ' << snack << " was left untouched.\n";
    } else {
      std::cout << ' ' << result;
    }
  }
}
```

The big change is in main, which no longer contains any boilerplate code to build individual links in the chain, or clean up after the chain.

```
int main() {
  food_handlers eaters;
  client(eaters);
  return 0;
}
```

Run It

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

struct food_handler {
```

(continues on next page)

(continued from previous page)

```
    const std::string empty;
    virtual std::string handle(std::string request) const = 0;
    virtual ~food_handler() = default;
};

struct monkey_food_handler : food_handler {
    std::string handle(std::string request) const override {
      if (request == "Banana") {
        return "Monkey: I'll eat the " + request + ".\n";
      }
      return empty;
    }
};
struct squirrel_food_handler : food_handler {
    std::string handle(std::string request) const override {
      if (request == "Nut") {
        return "Squirrel: I'll eat the " + request + ".\n";
      }
      return empty;
    }
};
struct dog_food_handler : food_handler {
    std::string handle(std::string request) const override {
      if (request == "MeatBall") {
        return "Dog: I'll eat the " + request + ".\n";
      }
      return empty;
    }
};
class food_handlers : food_handler
{
  std::vector<std::unique_ptr<food_handler>> chain;
  public:
    food_handlers() {
      chain.push_back(std::unique_ptr<food_handler>(new monkey_food_handler));
      chain.push_back(std::unique_ptr<food_handler>(new squirrel_food_handler));
      chain.push_back(std::unique_ptr<food_handler>(new dog_food_handler));
    }
    std::string handle(std::string food_item) const override {
      std::string reply;
      for(const auto& dude: chain) {
        reply = dude->handle(food_item);
        if(!reply.empty()) { return reply; }
      }
      return reply;
    }
};

void client(const food_handlers& eaters) {
  std::vector<std::string> food = {"Nut", "Banana", "Cup of coffee"};
  for (const auto& item : food) {
    std::cout << "Client: Who wants a " << item << "?\n";
```

(continues on next page)

(continued from previous page)

```
    const std::string result = eaters.handle(item);
    if (result.empty()) {
      std::cout << ' ' << item << " was left untouched.\n";
    } else {
      std::cout << ' ' << result;
    }
  }
}

int main() {
  food_handlers eaters;
  client(eaters);
  return 0;
}
```

This fun example is adapted from Thinking in C++, Vol 2. It uses some non-standard vocabulary to define the basic elements of the chain, but it is still a chain of responsibility.

Interface

First we define an interface each handler in the chain of responsibility must implement.

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

enum class Answer { NO, YES };

// This is our handler interface.
// Every class that inherits from this
// must implement the canIHave function
struct GimmeStrategy {
  virtual Answer canIHave() = 0;
  virtual ~GimmeStrategy() = default;
};
```

Rather than a `bool`, in this case, our early termination criteria is an enumerated type.

Implementing classes

For a chain to be a chain, at least two classes must implement the interface. (It's not much of a chain with only 1 link).

```
struct AskMom : public GimmeStrategy {
  Answer canIHave() {
    std::cout << "Mommy? Can I have this?\n";
    return Answer::NO;
  }
};

struct AskDad : public GimmeStrategy {
  Answer canIHave() {
    std::cout << "Dad, I really need this!\n";
    return Answer::NO;
```

(continues on next page)

(continued from previous page)

```
  }
};

struct AskGrandpa : public GimmeStrategy {
  Answer canIHave() {
    std::cout << "Grandpa, is it my birthday yet?\n";
    return Answer::NO;
  }
};

struct AskGrandma : public GimmeStrategy {
  Answer canIHave() {
    std::cout << "Grandma, I really love you!\n";
    return Answer::YES;
  }
};
```

Building the chain

Much discussion related to this pattern is about how to create the chain of responsibility as a linked list. However, when you look at the pattern it really shouldn't matter how the chain is created: that's an implementation detail. The only important part is that *some* kind of *iterable* type is used to visit each handler. How that is implemented should be invisible to users.

While the `Gimme` class also is derived from the `GimmeStrategy` it is used to construct the chain of all the other strategies used.

```
class Gimme : public GimmeStrategy {
   private:
      std::vector<std::unique_ptr<GimmeStrategy>> chain;
   public:
     Gimme() {
       chain.push_back(std::unique_ptr<GimmeStrategy>(new AskMom));
       chain.push_back(std::unique_ptr<GimmeStrategy>(new AskDad));
       chain.push_back(std::unique_ptr<GimmeStrategy>(new AskGrandpa));
       chain.push_back(std::unique_ptr<GimmeStrategy>(new AskGrandma));
     }
     Answer canIHave() {
       for (const auto& it: chain) {
         if (it->canIHave() == Answer::YES) {
            return Answer::YES;
         }
       }
       // Reached end without success...
       std::cout << "Waaaaaahh!!\n";
       return Answer::NO;
     }
};
```

Run It

Once the abstract and implementing classes have been defined, then calling the chain is easy:

```
int main() {
  Gimme chain;
  chain.canIHave();
}
```

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

enum class Answer { NO, YES };

struct GimmeStrategy {
  virtual Answer canIHave() = 0;
  virtual ~GimmeStrategy() = default;
};

struct AskMom : public GimmeStrategy {
  Answer canIHave() {
    std::cout << "Mommy? Can I have this?\n";
    return Answer::NO;
  }
};

struct AskDad : public GimmeStrategy {
  Answer canIHave() {
    std::cout << "Dad, I really need this!\n";
    return Answer::NO;
  }
};

struct AskGrandpa : public GimmeStrategy {
  Answer canIHave() {
    std::cout << "Grandpa, is it my birthday yet?\n";
    return Answer::NO;
  }
};

struct AskGrandma : public GimmeStrategy {
  Answer canIHave() {
    std::cout << "Grandma, I really love you!\n";
    return Answer::YES;
  }
};

class Gimme : public GimmeStrategy {
   private:
      std::vector<std::unique_ptr<GimmeStrategy>> chain;
   public:
     Gimme() {
       chain.push_back(std::unique_ptr<GimmeStrategy>(new AskMom));
```

(continues on next page)

(continued from previous page)

```
      chain.push_back(std::unique_ptr<GimmeStrategy>(new AskDad));
      chain.push_back(std::unique_ptr<GimmeStrategy>(new AskGrandpa));
      chain.push_back(std::unique_ptr<GimmeStrategy>(new AskGrandma));
    }
    Answer canIHave() {
      for (const auto& it: chain) {
        if (it->canIHave() == Answer::YES) {
          return Answer::YES;
        }
      }
      // Reached end without success...
      std::cout << "Waaaaaahh!!\n";
      return Answer::NO;
    }
};

int main() {
  Gimme chain;
  chain.canIHave();
}
```

More to Explore

- Chain of Responsibility Design Pattern on oodesign.com and on Wikipedia.
- Bruce Eckel. Thinking in C++, Vol 2., section 3.3.10.

3.11 Class templates and `std::array`

This chapter discusses how to create class templates and introduces the first of many Standard Template Library (STL) containers explored between now and the end of the course.

3.11.1 Class templates

C++ allows writing class templates using the same syntax used to write function templates. We use templates in classes for the same reason we use templates in functions: when we want to use variables without making a commitment to a fixed type. A class template is **not** a class. It defines instructions the compiler uses to create classes as needed. No code is generated from a source file that contains only template definitions. In order for any code to appear, a template must be instantiated: the template arguments must be provided so that the compiler can generate an actual class. Class template type cannot be deduced from the passed parameters in the way that function templates may.

```
template <class T, class U>
struct pair {
  T first;
  U second;
};

int main() {
  pair<char, int> symbol = {'A', 65};
```

(continues on next page)

(continued from previous page)

```
  return symbol.second;
}
```

A class template allows us to define a family of classes that performs the same behaviors on many different types. Container classes are the canonical examples of how templates can make code maintenance easier. The following class is a 'bag' - a general purpose container.

```
template <class T>
class bag {
    std::vector<T> data;

  public:
    void add(T item);
    void erase(bag<T>::iterator item);
    void swap(bag<T> item);
};
```

As this class demonstrates, a std::vector is a *bag*: a general purpose container for (nearly) any type.

Recall the reason why we wanted to create a function template. We were writing multiple copies of the same functions over and over. The function bodies were identical except for the types used. The same applies to class templates: we want to avoid writing multiple classes that are the same except for the data stored and types passed to functions.

Value class templates

Using the information we have so far, we can create a complete class template that supports most of the basic operations we are generally interested in for any value type:

- Default construct
- Copy and assign values
- Use any relational operation

The following example uses some common conventions for user defined types:

- Constructors are defined in the class body if they are short (5 lines of code or less).
- The name *other* is used as the parameter name for another object of the same type when passed as the only parameter to a function.
- Binary operator overloads use *lhs* and *rhs* to denote the right-hand side and left-hand side operands of the binary operation.
- The default keyword is used to instruct the compiler to create the default implementation for the marked special member function.

  Only special member functions may be marked `default`.

```
#pragma once

template <typename T>
struct item
{
  T value;
  item() {}
  ~item() = default;
```

(continues on next page)

(continued from previous page)

```
  // Copy construct and assignment
  item(const item& other) : value(other.value) {}
  item& operator=(const item& other) {
    value = other.value;
    return *this;
  }
  // Conversions from T and to T:
  explicit item(const T& x) : value(x) {}
  explicit operator T() const { return value; }
};

// Non-member functions

// Relational operators
template <typename T>
  inline bool operator==(const item<T>& lhs, const item<T>& rhs) {
    return lhs.value == rhs.value;
  }
template <typename T>
  inline bool operator!=(const item<T>& lhs, const item<T>& rhs) {
    return !(lhs == rhs);
  }
template <typename T>
  inline bool operator<(const item<T>& lhs, const item<T>& rhs) {
    return lhs.value < rhs.value;
  }
// Note we call <, but swap the operands!
template <typename T>
  inline bool operator>(const item<T>& lhs, const item<T>& rhs) {
    return rhs < lhs;
  }
template <typename T>
  inline bool operator<=(const item<T>& lhs, const item<T>& rhs) {
    return !(lhs > rhs);
  }
template <typename T>
  inline bool operator>=(const item<T>& lhs, const item<T>& rhs) {
    return !(lhs < rhs);
  }

// stream extraction
  template <typename T>
inline std::ostream& operator<<(std::ostream& os, const item<T>& rhs) {
  return os << rhs.value;
}
```

Friend or non-friend?

Some operators must be implemented as member functions, `operator=`, `operator[]`, and member access: both `operator.` and `operator->`, because the language requires it. We have choices where we define the others.

Some are commonly implemented as non-member functions, because their left operand cannot be modified by you. The most prominent of these are the stream insertion and extraction operators. The left operands are stream classes

from the standard library which you cannot change.

You can choose to implement them as you like, however, the following guidelines are highly recommended.

For operators where you have to choose to either implement them as a member function or a non-member function, use the following guidelines:

1. If it is a **unary operator**, then implement it as a **member** function. For example, `operator++`.
2. If a binary operator treats both operands equally then implement as a **non-member** function.

   Generally, neither operand is modified in this situation. The relational operators all fall into this category.
3. If a binary operator does not treat both of its operands equally then consider making it a member function.

   If the left-hand side operator is modified in the operation, or the function returns the `this` pointer, then it should be a member function of the left hand operand type.

   Otherwise, it can be implemented as a non-member function.

In the previous section, the relational operators were all declared as *non-friend non-member* functions. This is considered best practice by many programmers.

> Prefer writing non-friend non-member functions
>
> —Item 44 of *C++ Coding Standards*, by Herb Sutter and Andrei Alexandrescu

Compare to the functionally similar friend, member overload for `operator==`:

```
template <typename U>
friend bool operator==(const item<U>& lhs, const item<U>& rhs) {
  return lhs.value == rhs.value;
}
```

- A non-friend function does not automatically know that a function is part of a class template unless told.

  This is why the non-friend functions repeat the template declaration from the `struct`.
- The friend functions declared in the class are implicitly *inlined*. The compiler *may* replace function calls to these functions with in-line copies of the function body. The compiler is not obligated to do so, but usually does.

  To get the same behavior from non-member functions, the inline keyword is used.
- The `friend` keyword is often used to provide private member access to non-member functions. In the case of the `item` struct, this wasn't needed.

  The use of friend here prevents the `this` pointer from being passed to functions declared (and in this case defined) in the data structure body.

More to Explore

- friend declaration
- Item 44 from C++ Coding Standards, Sutter and Alexandrescu, 2004.
- C++ Core guidelines for overloads

3.11.2 Overloading `operator[]`

User-defined types that provide array-like access that allows both reading and writing often overload `operator[]`.

One of the rules of the language is that `operator[]` must be implemented as a member function. Generally `const` and non-const versions of the overload are implemented.

```
struct T
{
          value_t& operator[](size_t index)       { return data[index]; }
    const value_t& operator[](size_t index) const { return data[index]; }
};
```

In addition to the member-only rule that the C++ language requires, there are a few best practice considerations for this operator. They mostly center around the fact that *operator[]* ` can only accept a single value. What if you want to use this operator in a user defined type that behaves like a multi-dimensional array?

Often people attempt to overload *operator[][]* - but it does not exist.

In general, if you have a type with single dimension access, then it's OK to overload `operator[]`. If your type has data in multiple dimensions, then prefer overloading `operator()` instead. For a detailed description of the *why*, refer to the following subsection.

Multi-dimension array access

To provide multidimensional array access semantics, e.g. to implement a 3D array access `a[i][j][k] = x;`, the overload for `operator[]` must return a reference to a 2D object, which has to have its own `operator[]` which returns a reference to a 1D object, which has to have `operator[]` which returns a reference to the element. To avoid this complexity, some libraries opt for overloading `operator()` instead.

Because `operator()` does not have the one parameter restriction that `operator[]` does, functions taking multiple parameters can be implemented directly without any excessive complicated function call chaining. And for users, the syntax is cleaner:

```
int main() {

  matrix a;

  // operator[] syntax
  a[i][j][k] = value;

  // operator() syntax
  a(i,j,k) = value;

}
```

In the following example, notice that our matrix class uses a simple one dimensional array as its *backing store*. So even though our class exposes a two-dimensional interface, our data is stored differently.

```
data_ = new T[rows * cols];
```

An array is simple and efficient. The class does not expose this implementation detail and if we wanted to replace the array with something else later, no matrix class users would be affected.

operator() Example

When using the `operator()` overload, only a single pair of functions is required, one function to return the value and other to return a value that can be assigned to a `const`.

This solution is general and scales up and down as needed, easily accommodating more or fewer dimensions.

```
T& matrix<T>::operator() (size_t row, size_t col);

const T& matrix<T>::operator() (size_t row, size_t col) const;
```

One note about the above examples. If you know the type `T` is a primitive type, or you have a non-templated matrix and you define your value type to be a built in type (int, double, etc), then you should return by value instead of const reference.

```
double& matrix::operator() (size_t row, size_t col);

const double matrix::operator() (size_t row, size_t col) const;
```

The non-const version should still be a reference to the value in the *backing store*, so that it can be modified.

If you don't want users modifying your data at all, then only provide a `const` version of the operator overload.

Run It

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

template <class T>
class matrix {
public:
  matrix(size_t rows, size_t cols);
  T& operator() (size_t row, size_t col);
  const T&  operator() (size_t row, size_t col) const;
  // ...
 ~matrix();
  explicit matrix(const matrix& m);
  // matrix& operator= (const matrix& m);
  // many other useful functions not implemented
private:
  size_t rows_;
  size_t cols_;
  T* data_;
};

template <class T>
inline
matrix<T>::matrix(size_t rows, size_t cols)
  : rows_ (rows)
  , cols_ (cols)
{
  if (rows == 0 || cols == 0)
    throw std::out_of_range("Matrix constructor has 0 size");
```

(continues on next page)

(continued from previous page)

```
  data_ = new T[rows * cols];
  for (size_t i = 0; i < rows_*cols_; ++i) {
      data_[i] = 0;
  }
}

template <class T>
inline
matrix<T>::matrix(const matrix<T>& m)
  : rows_ (m.rows_)
  , cols_ (m.cols_)
{
  std::copy(m.data_, m.data_+rows_*cols_, data_);
}

template <class T>
inline
matrix<T>::~matrix()
{
  delete[] data_;
}

template <class T>
inline
T& matrix<T>::operator() (size_t row, size_t col)
{
  if (row >= rows_ || col >= cols_)
    throw std::out_of_range("Matrix subscript out of bounds");
  return data_[cols_*row + col];
}

template <class T>
inline
const T& matrix<T>::operator() (size_t row, size_t col) const
{
  if (row >= rows_ || col >= cols_)
    throw std::out_of_range("const Matrix subscript out of bounds");
  return data_[cols_*row + col];
}

int main()
{
    matrix<double>a {3,5};
    a(0,0) = -1;
    a(1,1) = 1;
    a(1,2) = 2;
    a(1,3) = 3;
    a(2,4) = 5;

    for (size_t i = 0; i<3; ++i) {
      for (size_t j = 0; j<5; ++j) {
         std::cout << a(i,j) << ' ';
```

(continues on next page)

(continued from previous page)

```
      }
        std::cout << '\n';
    }
}
```

In the interest of completeness, the following example shows one way to implement a 2D matrix class that provides an interface for `my_matrix[i][j]`. This example uses a vector of vectors, although other solutions are possible.

operator[] Example

What are the main differences from the preceding implementation?

The backing store is a vector of vectors:

```
std::vector<std::vector<T>> data_;
```

As previously discussed, the `operator[]` takes at most a single parameter. This means we must return a `vector<T>&` from our operator.

```
std::vector<T>& operator[] (size_t row);
```

How does the second dimension work?

Recall that the vector class has its own `operator[]` overload. The final dimension with the element value is retrieved from the index into the column vector of the matrix.

A destructor is no longer needed because this version of the matrix class does not manage its own memory. All the memory management is handled by the vector class.

Run It

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

template <class T>
class matrix {
  public:
    matrix(size_t rows, size_t cols);
    explicit matrix(const matrix& m);

    std::vector<T>& operator[] (size_t row);
    const std::vector<T>& operator[] (size_t row) const;

    size_t rows() const {return data_.size(); }
    size_t cols() const
    {
        return rows()? data_[0].size(): 0;
    }
  private:
    std::vector<std::vector<T>> data_;
};

template <class T>
```

(continues on next page)

(continued from previous page)

```
inline
matrix<T>::matrix(size_t rows, size_t cols)
  : data_ (rows)
{
  for (auto& row: data_) {
    row.resize(cols);
  }
}

template <class T>
inline
matrix<T>::matrix(const matrix<T>& m)
  : data_ (m.data_)
{ }

template <class T>
inline
std::vector<T>& matrix<T>::operator[] (size_t row)
{
    return data_[row];
}

template <class T>
inline
const std::vector<T>& matrix<T>::operator[] (size_t row) const
{
  return data_[row];
}

int main()
{
    matrix<double>a {3,5};
    a[0][0] = -1;
    a[1][1] = 1;
    a[1][2] = 2;
    a[1][3] = 3;
    a[2][4] = 5;

    for (size_t i = 0; i<3; ++i) {
      for (size_t j = 0; j<5; ++j) {
         std::cout << a[i][j] << ' ';
      }
        std::cout << '\n';
    }
}
```

More to Explore

- Operator overloading in C++ from stackoverflow.
- ISO C++ FAQ: 'How do I create a subscript operator for a Matrix class?'

3.11.3 Container classes

Container classes are simply user defined types that provide easy access to a collection of data that is the same type. You are already familiar with several container classes:

- `std::string`
- `std::vector`
- `std::bitset`
- `std::array`

There are more in the STL, but they all have one thing in common. They store a collection of data of **a single type**.

Containers are like arrays in that regard, however, container classes have many features that raw arrays lack.

C++11 adds a *list initialization* as a feature. List initialization provides 2 key benefits:

1. No implicit narrowing conversions
2. No floating point to integral type conversions
3. Ability to initialize a container using a list syntax

Consider the following examples.

```
double d = 3.14;
int pi = d;      // compiles, but probably a bug

int  i  {d};     // compile error
char c1 {i};     // compile error
char c2 {128};   // ok - 128 fits in type char
int  x  {c2};    // ok - widening conversion

double z {x};    // compile error
                 // yes, a widening conversion, but
                 // no floating point to int conversions allowed
                 // in either direction
```

What about 'initializer list' syntax?

An initializer list is a type constructed when using list initialization **followed by an assignment operator**.

```
auto a = 1;       // a is an int
auto b {2};       // b is an int
auto c = {3};     // c is an initializer_list<int>
```

The type of `c` can be unexpected when the type is not a container type. It is important to note the initializer_list constructor is preferred above all others if it exists in a class.

Initializer list constructors

The initializer_list was introduced in C++11 to simplify container initialization and make it more uniform. Prior to C++11, you could initialize an array with a value list:

```
int fib[] = {1, 1, 2, 3, 5, 8, 13};
```

However, trying the same style for other standard library containers would result in a compile error:

```
// compile error in C++03 and prior
std::vector<int> fib = {1, 1, 2, 3, 5, 8, 13};
```

Prior to C++11, if you wanted to add a known set of values to a vector you would have to push them back one at a time using `push_back`, either one line at a time, in a loop, or using an algorithm from the STL.

An **initializer_list<T>** is a lightweight wrapper that that creates a temporary array of type **T**.

```
std::initializer_list<int> fib = {1, 1, 2, 3, 5, 8, 13};
```

Any class can define a constructor that takes a initializer_list type as a parameter. Obviously, it makes sense to define an `initializer_list` constructor only for a type that can be correctly constructed from a list of values. Containers are the most obvious example and are the motivation behind initializer lists.

Example

The following example defines the absolute minimum you would need to define a simple class that can be initialized like std::array.

It defines a stack array of specified size using the non-type parameter `N` and allows a new **array** to be constructed using (only) an initializer list. No other constructors are included in this example.

```
template <class T, size_t N>
struct array {
    T data_[N];
    array(const std::initializer_list<T>& list) {
      std::copy(list.begin(), list.end(), data_);
    }
};
```

An array instance can now be created from an initializer list using standard C++ syntax.

```
array<int,7> fib = {1,1,2,3,5,8,13};
```

The assignment from the temporary list to the array object does require an implicit conversion between the two types.

Run It

This example converts the `struct` into a `class` and makes the one argument constructor `explicit`.

Notice how the declaration of variable `fib` changes in main when the initializer_list constructor is marked as `explicit`.

```
#include <algorithm>
#include <cstddef>
#include <initializer_list>
#include <iostream>

template <class T, size_t N>
```

(continues on next page)

(continued from previous page)

```
  class array {
    private:
      T data_[N];
    public:
      explicit
      array(const std::initializer_list<T>& list) {
        std::copy(list.begin(), list.end(), data_);
      }
      constexpr
        const T& operator[](const size_t& index) const
        { return data_[index]; }
      T& operator[](const size_t& index)
        { return data_[index]; }
  };

int main()
{
  auto fib = array<int, 7>{1,1,2,3,5,8,13};

  for (size_t i = 0; i<7; ++i) {
    std::cout << fib[i] << ' ';
  }
  std::cout << '\n';
}
```

Note: Initializer lists will always favor a matching initializer_list constructor over other potentially matching constructors. So, if our array had a one argument constructor that took a single value of type `T`:

```
array<int,1> fib = {3};
```

Then this declaration would invoke `array(const std::initializer_list<T>&)` and not `array(int)`. If you want to match to `int` constructor once a list constructor has been defined, you'll need to use copy initialization or direct initialization. The rule applies to `std::vector` and other standard library container classes that define both a list constructor and another one argument conversion constructor.

```
// Calls std::vector::vector(std::size_type)
// creates 3 value-initialized elements: 0 0 0
std::vector<int> data( 3 );

// Calls std::vector::vector(std::initializer_list<int>)
// creates 1 element with value = 3
std::vector<int> data{ 3 };
```

More to Explore

- C++11 list initialization
- Initializer list constructors

3.11.4 Standard Template Library Containers

If a program needs to manage a collection of closely related things, numbers, bank accounts, students, or even fruit, then the simplest approach is to put them in a container. You should already be familiar with raw arrays, which are part of the language and are not considered part of the STL. There are two broad categories of STL containers: *sequence containers* and *associative containers*. In an *earlier section*, we briefly introduced `std::vector`. A vector is an example of a sequence container.

A *sequence container* stores a sequence of elements of a given type. The sequence containers can be further divided into two 'flavors':

list-like sequences
: Things stored in a sequence

stacks and queues
: Things listed in order to be processed

As you might imagine, *associative containers* do not store elements in sequential order, but rather use a container element to determine where in the container data should be stored. The associative containers can also be further divided into two 'flavors':

sets
: Unique things

maps
: Things stored with a unique ID

All the STL containers provide similar advantages over arrays:

- Add and remove data dynamically
- Bounds checking
- Work with data at a higher level of abstraction
- Manage subsets of data as a unit

Although all of the containers in the STL share some core characteristics, the different containers have different designs, and have different trade-offs or costs. This allows them to excel in specific situations, where an 'all-purpose' container might fall short.

All containers support the same basic operations:

- Add objects to the container
  - And remove objects from it
- Find out if an object is in the container (or a group of objects)
- Retrieve an object without removing it
- Walk through the container, looking at each object in turn

The container manages the storage space that is allocated for its elements and provides member functions to access them, either directly or through *iterators* (objects with properties similar to pointers).

Ordered and sorted containers

STL containers may be ordered, unordered, sorted or unsorted. An **ordered** container simply means that you can iterate through the container in a specific order.

We normally do not think of the sequence containers as having an order. For example, a vector may contain the values `{3, 1, 4, 1, 5, 9}`, which clearly are not in the *natural order* for integers. A `vector` *is* ordered, however: by its index position. As long as items are not added or removed from an ordered container, two iterations over the same container will visit the same elements in the same order. So in our previous example, the vector containing `{3, 1, 4, 1, 5, 9}` is *ordered*, but *unsorted*.

A **sorted** container stores elements using rules defined when the container is created: the sort order. All of the sorted containers define a default sort order, determined by the `operator<` for the element type.

Containers can *never* be both sorted and unordered, because sorting of any kind is by definition an ordering.

More to Explore

- C++ Container Named Requirements:
  - Container
  - SequenceContainer
  - AssociativeContainer

3.11.5 The `std::array` class

The std::array is a container that encapsulates fixed size arrays. Since it is literally a wrapper around a raw array, the size of a `std::array` must be defined when declared.

```
std::array <int, 12> days_per_month;
```

The array class is very lightweight and has very little costs over a raw array. Additionally, `std::array` provides convenience functions such as:

at() and operator[]
: range checked access and unchecked access

front() and back()
: access to the first and last elements

size()
: return the number of elements

empty()
: check if the container is empty

Unlike a raw array, `std::array` cannot infer its size if declared with an initializer list:

```
#include <array>
#include <iostream>
using std::cout;

int main() {
  // compile error: array template parameter missing:
  //std::array<char> letters = {{'h', 'o', 'w', 'd', 'y', '!'}};
```

(continues on next page)

(continued from previous page)

```
  std::array<char, 6> letters = {{'h', 'o', 'w', 'd', 'y', '!'}};

  if (!letters.empty()) {
    cout << "The first character is: " << letters.front() << '\n';
    cout << "The last character is: " << letters.back() << '\n';

    for (const auto& c: letters) {
      cout << c;
    }
    cout << std::endl;
  }
}
```

More to Explore

- STL containers library

3.12 Memory management and `std::vector`

This section discusses class-based memory management in C++.

3.12.1 The `std::vector` class

The std::vector is a sequence container that simulates a dynamically sized array. If you have taken a class in linear algebra, the vector ADT has nothing to do with mathematics, but is simply sequential data structure.

It is a flexible and frequently used container. Refer to *The vector class* for an introduction to the basic functions available.

Like an array, elements are stored in adjacent memory slots. This means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements. The benefit of this is that a pointer to an element of a vector may be passed to any function that expects a pointer to an element of an array.

Unlike an array, vector storage is handled automatically, being expanded and contracted as needed. Vectors usually occupy more space than static arrays, because more memory is allocated to handle future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted. The total amount of allocated memory can be queried using the capacity function. Starting in C++11, extra memory can be returned to the system with a call to shrink_to_fit.

Size vs Capacity

A vector declared as:

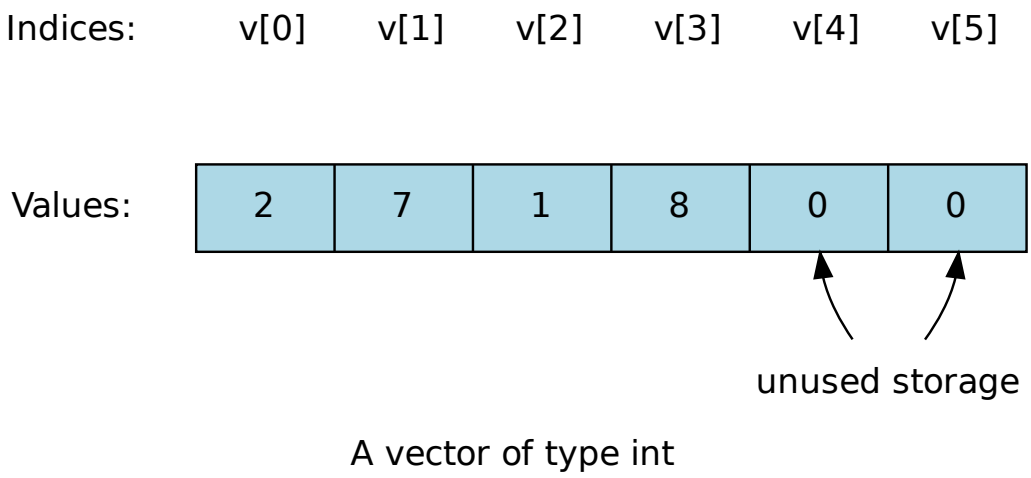
```
std::vector<int> v(6);
```

Creates a vector of type `int`, with size 6 and all values initialized to 0. At this point, the size and capacity of the vector are the same.

If we assign new values to the vector:

```
v = {2, 7, 1, 8};
```

The new values replace the old. Although the **size** of the vector lowers from 6 to 4, the **capacity** of the vector is not changed.



The data at `v[4]` and `v[5]` were originally default constructed and accessible when the vector was initially created with the explicit size = 6. Once the new data is assigned and the size of the vector is reduced, the `end()` also shrinks. A loop over the elements in the vector will not include the data at `v[4]` and `v[5]`.

Run It

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

using std::cout;
using std::vector;

void print(const vector<int>& container)
{
  cout << "size: " << container.size()
       << ", capacity: " << container.capacity()
       << ", values: ";
  for (const auto& v: container) {
      cout << v << ' ';
  }
  cout << '\n';
}


int main()
{
  vector<int> v(6);
  print (v);
```

(continues on next page)

(continued from previous page)

```
  v = {2, 7, 1, 8};
  print (v);
  v.pop_back();
  print (v);
  v.clear();
  print (v);
  v = {5,13};
  v.shrink_to_fit();
  print(v);
}
```

Try This!

Change the previous example using different ways of initializing and modifying the vector. Try to predict the `print` output *before* running the program.

Some things to try:

- Default constructed vector
- Push back 1 value onto an empty vector.
- Write a loop to push back 10 values onto a vector 1 at a time.
- Replace clear with empty
- Calling vector::reserve with values larger and smaller than the current capacity.

This is a new and important distinction.

size
: Refers to the number of elements in the container.

capacity
: Refers to the total size of the underlying storage.

The vector ADT makes the distinction primarily for performance reasons. The backing store of a vector is an array and an array cannot be resized. Adding even one element to a completely full array involves several steps:

- making a new array with a larger capacity
- copying the old array into the new array
- destroying the old array storage

That is a lot of work and a potentially expensive operation. For this reason, vectors normally never reduce the capacity of and array unless explicitly instructed to do so.

It also explains why tools like `clang-tidy` will complain if it finds calls to vector::push_back in a loop after a default constructed vector is declared.

Passing vectors to C functions

The data layout of a vector makes it easy to pass a vector to a legacy C function that expects a raw array. This is something that comes up more often than you might expect. The book *Effective STL* has a good discussion of passing string and vector objects to legacy C functions[1].

Given a legacy C function that expects a raw array:

[1] Effective STL (Item #16) by Scott Meyers (Addison-Wesley Professional). Copyright 2001 Scott Meyers, 978-0-201-74962-5.

```
void print_sum (const int* values, size_t array_size) {
  int sum = 0;
  for (size_t i = 0; i < array_size; ++i) {
    sum += values[i];
  }
  printf("Sum of ints in the array is %d\n", sum);
}
```

We expect to be able to pass in an array and print the sum:

```
int main() {

  int data[] = { -30, 102, 55, -19, 0, 222, -3000, 4000, 8, -2 };
  const int numValues = sizeof data / sizeof(int);

  print_sum (data, num_values);

  return 0;
}
```

We can pass a `vector` to this same legacy function:

```
int main() {
  int data[] = { -30, 102, 55, -19, 0, 222, -3000, 4000, 8, -2 };
  const int num_values = sizeof data / sizeof(int);

  print_sum (data, num_values);

  std::vector<int> v;
  v.insert (v.begin(), data, data + num_values);     // insert the ints in data
                                                     // into v at the front

  print_sum (&v[0], v.size());   // ok, unless v is empty

  if (!v.empty()) {              // safer
    print_sum (&v[0], v.size()); //&v[0] is better than v.begin()
  }

  return 0;
}
```

More to Explore

- STL containers library
- Clang-tidy vector performance checks

Footnotes

3.12.2 Copying objects

C++ is one of the few languages that provides precise control over how memory is managed. In C++11, every class is created with 5 special functions:

- destructor
- copy constructor
- move constructor
- copy assignment operator
- move assignment operator

In many classes, such as most of those written so far, the compiler generated default versions work fine. As we will see, sometimes you have to write them yourself.

Programmers have choices on how (or if) objects are copied and moved. Whenever an object is passed by value to a function, or returned by value from a function, a copy is implicitly performed:

```
std::vector<int> scores;
auto avg = average (scores);  // a copy of scores is passed to average
```

Copy operations also occur in range-for loops:

```
for (const int value: scores)
```

Each member of `scores` is copied into `value` on each iteration.

Explicit copying can also be performed. Whenever you have an existing object and use it to initialize a new or existing object, the copy constructor is called:

```
std::vector<std::string> words;
std::vector<std::string> w2 = words; // copy words into w2
```

Both explicit and implicit copies are controlled by a special constructor called the *copy constructor*. Like other constructors, the copy constructor is a member function with the same name as the class name. The signature **must** be able to evaluate to this:

```
class_name ( const class_name & );
```

A copy constructor *may* take other parameters, but that is uncommon. If there are other parameters, they **must all have defaults values defined**. In general, the default copy constructor generated by the compiler will suffice. If the default creation is inhibited for any reason, it is acceptable to explicitly declare the default constructor:

```
class_name ( const class_name & ) = default;
```

However the default copy constructor is created, the behavior is the same: each class member is copied, in initialization order.

Copy constructor

A simple class with a default and a copy constructor.

```
struct A
{
  int n;
  double d;
```

(continues on next page)

(continued from previous page)

```
  // user defined default constructor
  A(int n = 0, double d = 1)
    : n{n}
    , d{d}
  { }

  // user defined copy constructor
  A(const A& other)
    : n{other.n}
    , d{other.d}
  { }
};
```

In this case, the user defined copy constructor does what the default constructors would do. When that is the case, it's best not to redo the work of the compiler.

Run It

```
#include <iostream>

struct A
{
  int n;
  double d;

  // user defined default constructor
  A(int n = 0, double d = 1)
    : n{n}
    , d{d}
  {
    std::cout << "default A\n";
  }

  // user defined copy constructor
  A(const A& other)
    : n{other.n}
    , d{other.d}
  {
    std::cout << "copy into A\n";
  }
};

int main() {
  A a;
  A b = a;
  return b.n;
}
```

When objects manage their own resources, simple member-wise assignment cannot be used. Consider the following:

```
#include <cstddef>
```

(continues on next page)

(continued from previous page)

```
#include <cstring>
#include <iostream>

namespace mesa {
  class string {
    private:
      char* data = nullptr;
      size_t sz = 0;

    public:
      string() = default;
      explicit string(const char* source) {
        // if source not null terminated, strlen behavior is undefined
        sz = std::strlen(source) + 1; // +1 for null terminator
        data = new char[sz];
        std::strncpy(data, source, sz);
      }

      void upper_case() {
        for (size_t i=0; i < sz; ++i) {
          data[i] = std::toupper(data[i], std::locale());
        }
      }

      ~string() {
        delete[] data;
      }

      char* c_str() const noexcept { return data; }
  };
} // namespace mesa
```

This class encapsulates an array of characters, providing 5 functions:

- A default constructor.
- A one arg constructor that converts a cstring into a `mesa::string` The constructor allocates memory for the `char` array and copies the provided string
- A destructor to clean up memory allocated by the constructor
- A function `upper_case` to transform the entire string to upper case.

Using mesa::string

What happens when we use this class?

```
int main() {
  mesa::string hello("Hello, world!");
  mesa::string copy = hello;
  copy.upper_case();
  std::cout << hello.c_str() << '\n';
  std::cout << copy.c_str() << '\n';
```

(continues on next page)

(continued from previous page)

```
  return 0;
}
```

Run It

```
#include <cstddef>
#include <cstring>
#include <iostream>
#include <locale>

namespace mesa {
  class string {
    private:
      char* data = nullptr;
      size_t sz = 0;

    public:
      string() = default;
      explicit string(const char* source) {
        // if source not null terminated, strlen behavior is undefined
        sz = std::strlen(source) + 1; // +1 for null terminator
        data = new char[sz];
        std::strncpy(data, source, sz);
      }

      void upper_case() {
        for (size_t i=0; i < sz; ++i) {
          data[i] = std::toupper(data[i], std::locale());
        }
      }

      ~string() {
        // commented out to prevent double delete
        // delete[] data;
      }

      char* c_str() { return data; }
  };

} // namespace mesa

int main() {
  mesa::string hello("Hello, world!");
  mesa::string copy = hello;
  copy.upper_case();
  std::cout << hello.c_str() << '\n';
  std::cout << copy.c_str() << '\n';

  return 0;
}
```

Even though we copied `hello`, changing the case of copy also resulted in changes to the original. In this case, we only

copied the pointer, not the data pointed to.

When we copy a value, we expect a *cloned object*. A object that in all respects has the same attributes, but that is separate and distinct. We **don't** want changes in one to affect the other.

Code Lens

The default copy behavior is a *shallow copy*: a literal copying of the bytes of each member variable. In the case of our string class, the `char*` is faithfully copied. When the copy is made, both variables point to the same memory.

```
#include <cstddef>
#include <cstring>
#include <iostream>
#include <locale>

namespace mesa {
  class string {
    private:
      char* data = nullptr;
      size_t sz = 0;

    public:
      string() = default;
      explicit string(const char* source) {
        // if source not null terminated, strlen behavior is undefined
        sz = std::strlen(source) + 1; // +1 for null terminator
        data = new char[sz];
        std::strncpy(data, source, sz);
      }

      void upper_case() {
        for (size_t i=0; i < sz; ++i) {
          data[i] = std::toupper(data[i], std::locale());
        }
      }

      // In this broken class, the destructor delete the same
      // memory multiple times, which is a crash
      // ~string() {
      //   delete[] data;
      // }

      char* c_str() { return data; }
  };

} // namespace mesa

int main() {
  mesa::string hello("Hello, world!");
  mesa::string copy = hello;
  copy.upper_case();
  std::cout << hello.c_str() << '\n';
  std::cout << copy.c_str() << '\n';

```

(continues on next page)

(continued from previous page)

```
  return 0;
}
```

Because there are two pointers to the same data on the free store, when either is deleted, the free-store memory is recovered. Consider this:

```
mesa::string hello("Hello, world!");

// create a new scope
{
  mesa::string copy = hello;
} // local variable copy destroyed

std::cout << hello.c_str() << '\n';
```

What does the last line print?

Show

There is no way to know for sure.

When `copy` goes out of scope and its destructor is called, it deletes the memory `copy::data` points to, but this is the same array `hello` is using. When `hello.c_str()` is called, undefined behavior is the result.

Fixing these problems requires writing a custom copy constructor.

```
string (const string& other) {
  sz = other.sz;
  data = new char[sz];
  std::strncpy(data, other.data, sz);
}
```

Each class member needs to be copied. The member `sz` can simply be default copied. It's the pointer member that needs special treatment:

- Initialize a new memory block large enough to hold the copy
- Copy each element of the source array into the destination. This is what std::strncpy does.

In contrast to a *shallow* copy, this copy is a **deep copy**. It doesn't copy the pointer at all. It makes an entirely new pointer and (deeply) copies all of the data pointed to by the source pointer to the destination.

Try This!

Take the copy constructor provided and implement in the previous `mesa::string` examples in this section.

Copy assignment

The copy assignment operator is similar to the copy constructor. The key difference to remember is that a copy **constructor** is only called when the left hand side object does not yet exist: it is in the process of being constructed.

```
X& X::operator=(const X& other)
{
```

(continues on next page)

(continued from previous page)

```
  // copy other content into this
  return *this;
}
```

Copy assignment

Copy **assignment** is called when both *already exist* and you want to copy the right hand side object into the left hand side object.

```
struct A
{
  int n;
  double d;

  // user defined default constructor
  A(int n = 0, double d = 1)
    : n{n}
    , d{d}
  { }

  // user defined copy constructor
  A(const A& other)
    : n{other.n}
    , d{other.d}
  { }

  A& operator=(const A& other)
  {
    if (this == &other) { return * this; }
    n = other.n;
    d = other.d;
    return * this;
  }

};
```

Run It

```
#include <iostream>

struct A
{
  int n;
  double d;

  // user defined default constructor
  A(int n = 0, double d = 1)
    : n{n}
    , d{d}
  {
    std::cout << "default A\n";
  }
```

(continues on next page)

(continued from previous page)

```
15
16    // user defined copy constructor
17    A(const A& other)
18      : n{other.n}
19      , d{other.d}
20    {
21      std::cout << "copy into A\n";
22    }
23    A& operator=(const A& other)
24    {
25      if (this == &other) { return * this; }
26      std::cout << "copy assign A\n";
27      n = other.n;
28      d = other.d;
29      return * this;
30    }
31
32  };
33
34  int main() {
35    A a;
36    A b = a;
37    a = b;
38    return b.n;
39  }
```

Try This!

Write a copy assignment function for the `mesa::string` class.

More to Explore

- From cppreference.com:
  - Copy constructors
  - Null-terminated byte strings

3.12.3 Lvalues, rvalues, and references

In addition to the builtin types and pointers, C++ adds a new type - the reference type. A major addition to C++11 is the addition of the *rvalue reference*. It is easier to understand rvalue references if we discuss lthe others first.

The original definition of lvalues and rvalues from C is as follows:

- An **lvalue** is an expression that may appear on the left or on the right side of an assignment
- An **rvalue** is an expression that can only appear on the right hand side of an assignment. For example,

```
int a = 42;
int b = 43;

// a and b are both l-values:
// these are all OK
a = b;
b = a;
a = a * b;

// a * b is an rvalue:
int c = a * b; // ok, rvalue on right hand side of assignment
a * b = 42;    // error, rvalue on left hand side of assignment
```

In C++, this is still useful as an intuitive view to lvalues and rvalues. However, C++ with its user-defined types introduces some subtleties regarding modifiability and assignability that cause this definition to be incorrect.

An **lvalue** is an expression that identifies a non-temporary object. For example:

```
// These are all valid assignments
int i = 42;
i = 43;              // i is an lvalue
int* p = &i;         // i is an lvalue
int& foo();
foo() = 42;          // foo() is an lvalue
int* p1 = &foo();    // foo() is an lvalue
```

An **rvalue** is an expression that identifies a temporary object or a literal not assigned a memory address. For example:

```
// valid assignments
int bar();
int j = 0;
j = bar();     // bar() is an rvalue
j = 42;        // ok, 42 is an rvalue

// invalid
int* p2 = &bar(); // cannot take the address of an rvalue
```

An **rvalue reference** is a reference to an *rvalue*. That is, a reference to a *temporary object* or a reference to a literal not assigned a memory address. As we will see in the next section, rvalue references are a key component of move operations and forwarding data to another location without copying it.

If `X` is any type, then `X&&` is called an *rvalue reference to X*. To distinguish from ordinary references, the reference `X&` is now also called an *lvalue reference*. An rvalue reference is a type that behaves like the ordinary reference `X&`, but there are several thing to look out for. During function overload resolution, lvalues will prefer function signatures containing lvalue references, and rvalues prefer the new rvalue references:

```
void foo(X& x);  // lvalue reference overload
void foo(X&& x); // rvalue reference overload

X x;
X foobar();

foo(x); // argument is lvalue: calls foo(X&)
foo(foobar()); // argument is rvalue: calls foo(X&&)
```

More to Explore

- Reference declarations

The content in this section was adapted from *Rvalue References Explained*, by Thomas Becker.

3.12.4 Moving memory

In the section on *Operator overloads* we briefly covered the std::swap algorithm. The implementation of swap for built in types is trivially implemented:

```
void swap(int& a, int& b)
{
   auto temp = a;
   a = b;
   b = temp;
}
```

However, even though a and b are both passed by non-const reference, this is still an expensive function to call for any type that is large or expensive to copy.

C++11 adds facilities that give programmers tools to replace expensive copies with moves. In C++11, if the right hand side of an expression is an *rvalue* and if the object supports moving, then moving memory is performed instead of copying memory. Given a potentially large type, such as vector, we can re-write the swap algorithm in terms of moves

```
void swap(vector<string>& a, vector<string>& b)
{
   auto temp = static_cast<vector<string>&&>(a); // cast to rvalue reference
   a = static_cast<vector<string>&&>(b);
   b = static_cast<vector<string>&&>(temp);
}
```

Casting manually to a rvalue reference is ugly and awkward. Simplifying this expression is the motivation behind move:

```
void swap(vector<string>& a, vector<string>& b)
{
   auto temp = std::move(a); // cast to rvalue reference
   a = std::move(b);
   b = std::move(temp);
}
```

The `move` function simply converts it's parameter into rvalue reference, and marks the object as being ready for a 'move'. Using `std::move` is exactly the same as using a static cast to an rvalue reference.

Just a moment ago, we mentioned a big 'if' regarding moves. We said

> ... if the object supports moving ...

How do we create objects that support moving?

With a move constructor.

Move constructors and move assignment

A move constructor is a constructor of the form:

```
class_name (class_name&&);
```

Note that the parameter to the constructor is not a constant. This is done for the same reasons swap functions take non-const references. We pass non-constant rvalue references to our move constructors so that we can exchange our current (empty) object for the one provided.

```
X::X (X&& other)
{
  // exchange content between other and this
}
```

The move assignment operator is similar to copy assignment, but with the now familiar rvalue reference parameter:

```
X& X::operator=(X&& rhs)
{
  // exchange content between other and this
  return *this;
}

// Given 2 objects
X a,b;
// do something to b

// We can copy them
a = b;


// Or force a move
a = std::move(b);
```

As always we need to be concerned with what to do if our object manages its own resources. If class `X` has, for example, data on the free store, then we need to ensure any resources that might create side effects are addressed when we use move assignment.

If move semantics are implemented as a simple swap, then the effect of this is that the objects held by `a` and `b` are being exchanged between `a` and `b`. Nothing is being destructed yet. The object formerly held by `b` will of course be destructed eventually - when `b` goes out of scope. But if `a` also becomes the target of a move, then the object formerly held by `a` gets passed on again. As far as the implementer of the assignment operator is concerned, it is not known when the object will be destructed.

So we have a small problem that needs to be fixed. A variable has been assigned to, but the object formerly held by that variable is still out there somewhere. Any part of an object's destruction that has side effects should be performed explicitly in the rvalue reference overload of the assignment operator:

```
X& X::operator=(X&& rhs)
{
  // Perform a cleanup that takes care of at least those parts of the
  // destructor that have side effects. Be sure to leave the object
  // in a destructible and assignable state.

  // exchange content between other and this
}
```

More to Explore

- Move constructors
- The std::swap algorithm
- C++ Rvalue references explained The content in this section was adapted from *Rvalue References Explained*, by Thomas Becker.
- Copy and Swap, 20 years later a deeper dive into some of the tradeoffs of different implementations of copy and move assignment.

3.12.5 Allocators

Those of you who have been paying attention may have noticed most containers in the standard library have declarations like:

```
template< class T,
          class Allocator = std::allocator<T>>
class vector;
```

What is the second template type parameter for?

An allocator manages the memory of each element stored in the container. The job of an allocator is similar to what the operators new and delete do, but in a more generic an extensible way. An allocator can allocate and deallocate memory for its elements and initialize and destroy element memory. And it can perform each of these actions as separate steps.

Why bother?

Allocators were originally conceived during the initial development of the standard library when library developers realized that some classes that took a template parameter of type `T` could break if `T` did not have a default constructor. An allocator can create default values for types that don't have their own default values. Similarly, an allocator can ensure memory is destroyed when we are finished with it. In short, the original `new` and `delete` operators did not handle all the cases required for advanced memory management.

- Designers realized the container should be independent of the underlying memory model. For example, the Intel Memory Model on x86 architectures use six different variants:

  tiny, small, medium, compact, large, and huge.

  Allocators allow container users to define exactly the memory allocation scheme appropriate for their runtime environment.

- Containers separate the memory allocation and deallocation from the initialization and destruction of their elements.

  A call to vector::reserve(n) allocates memory for at least n elements. The constructor for each element will **not** be called.

We can handle types without a default value by giving users the option to specify the value to be used when we need a default:

```
void resize(size_t new_capacity, T default_value = T())
{
  reserve(new_capacity);
  std::fill(begin()+size_, begin()+capacity_, default_value);
```

(continues on next page)

(continued from previous page)

```
  size_ = new_capacity;
}
```

This form of resize() will use `T()`, unless the user provides an alternative. For example:

```
bag<double> stuff;
stuff.resize(100);        // add 100 doubles all == 0.0
stuff.resize(200, 3.14);  // add 200 copies of 3.14
stuff.resize(300, 0.0);   // add 200 copies of 0.0 (redundant)
```

The destructor problem is harder to address. We need to deal with a data structure that may contain a mix of some initialized data and some uninitialized data. Typically, we are very careful to avoid uninitialized data and the associated programming errors. Now as the developers of generic containers we have to handle this problem so that users of these containers don't have to.

First we need a way to get an manipulate uninitialized storage. **This** is where allocator comes in. A simplified allocator looks like this:

```
template<class T>
  class allocator {

    using value_type    = T;
    // other types and constructors omitted

    T*   allocate  (size_t n);        // allocate space to n objects of T
    void deallocate(T* p, size_t n);  // deallocate n objects of T
                                      // starting at location p

    void construct(T* p, const T& value);  // construct a value of T in p
    void destroy  (T* p);                  // destroy the T in p

  };
```

These 4 functions provide the core capabilities of an allocator:

- Allocate memory of a size suitable to hold an object of type `T` *without initializing it*.
- Construct an object of type `T` in uninitialized space.
- Destroy an object of type `T` - returning the memory space to the uninitialized state.
- Deallocate uninitialized memory of size suitable for an object of type `T`.

Using std::allocator_traits

An allocator is exactly what a container to separate memory allocation from object construction and memory deallocation from object destruction.

How does a container use an allocator? First, as in the standard library, we need a allocator type parameter and a local variable to store an instance of the allocator. Although you can use an allocator directly, the allocator interfaces are deprecated, and we are going to use the allocator_traits interface. The allocator_traits class template provides the standardized way to access various properties of Allocators. The standard containers and other standard library components access allocators through this template, which makes it possible to use any class type as an allocator.

```
template<class T, class Allocator = std::allocator<T>>
 class bag {
```

(continues on next page)

(continued from previous page)

```
  Allocator allocator_;

  // . . .
};
```

The allocator_traits interface consists entirely of static members - no object instance exists and it is completely stateless, however the syntax is a bit verbose, which is why I frequently alias it in a class:

```
template<class T, class Allocator = std::allocator<T>>
 class bag {

  Allocator allocator_;
  using memory = std::allocator_traits<Allocator>;

  // . . .
};
```

Now except for using our allocator object, the class is unchanged. Container users can ignore the allocator unless they need a `bag` that manages memory for its elements in some unusual way.

The only class functions that require modification are those that deal directly with memory:

- object construction and destruction
- memory allocation and deallocation

```
void reserve(size_t new_capacity)
{
  // never decrease allocation
  if (new_capacity <= capacity_) {
    return;
  }
  // allocate new space
  T* new_data = new T[new_capacity];

  // copy into new space
  std::copy(begin(), end(), new_data);

  // delete old memory
  delete[] data_;

  // point to the new data
  data_ = new_data;
  capacity_ = new_capacity;
}
```

Refactoring the original version of `reserve` to use an allocator involves several steps.

```
void reserve(size_t new_capacity)
{
  // never decrease allocation
  if (new_capacity <= capacity_) {
    return;
```

(continues on next page)

(continued from previous page)

```
  }

  // allocate new space
  T* new_data = memory::allocate(allocator_, new_capacity);

  // copy into new space
  for (size_t i = 0; i < new_capacity; ++i) {
    memory::construct(allocator_, &new_data[i], data_[i]);
  }

  // delete old memory
  for (size_t i = 0; i < capacity_; ++i) {
    memory::destroy(allocator_, &data_[i]);
  }

  // deallocate old space
  memory::deallocate(allocator_, data_, capacity_);
  data_ = new_data;
  capacity_ = new_capacity;
}
```

More to Explore

- From cppreference.com
  - Named requirement Allocator
  - allocator_traits
  - new
- Writing allocators
  - Memory Management with std::allocator
  - Allocator Boilerplate

3.12.6 `constexpr` classes

The C++ Core guidelines generally prefers constant data and objects over mutable objects and data when possible. Previously, when we have used `const` and `constexpr` it has generally been limited to variables and functions. But what about an entire class? Can we design a class that can only store a single value? Even if we can, should we?

Let's answer that last question first.

Immutable object provide several important benefits:

- Objects that are immutable are easier to reason about. They don't surprise you with unexpected behaviors.
- It can prevent errors when an object changes its state unexpectedly.
- Interfaces that accept constant objects are easier to work with and debug.
- Although we don't discuss multi-threaded programming in this course, constant objects are inherently thread safe - that is they can safely be accessed simultaneously by concurrent independent execution threads.

- Gives the compiler more tools to:
  - Report errors in usage
  - Make optimizations (more on this in a moment)

Awesome. So how can we make an immutable class?

Simply by declaring all of the class functions as `constexpr`!

Let's examine a value class for a distance in meters.

```
namespace length{

  class distance{
    public:
      explicit constexpr distance(double value = 0)
        :m{value}
      {}

    private:
      double m;       // meters
  };

} // end namespace length
```

All member functions, including any constructors must be `const` or `constexpr`. The private data is not constant. That's OK, because we won't allow any function to change it.

Recall that `constexpr` member functions are implicitly `const` member functions - that is, they cannot change the state of the object.

After we define our constructor, we can add other functions as appropriate. In our case we want to perform basic math operations on distances. And in keeping our use of the 'standard pattern' for arithmetic overloads, we want to create member functions like this:

```
constexpr distance operator+=(const distance& other);
```

Normally when we implement these functions we modify the current object and return `*this`. But we can't do that if our objects are immutable. What we do instead is construct a new distance object by combining the two objects on either side of the operand:

```
constexpr distance operator+=(const distance& other) {
  return distance(m + other.m);
}
```

The current object is still involved - it is the object on the left-hand side of the expression, but we do not modify it.

An important implication of this implementation is that every change in state creates a new object to store it.

Adding the overloads for addition, subtraction, multiplication, and division yields the following:

```
namespace length{

  class distance{
    public:
      constexpr distance(double i)
        :m{i}
```

(continues on next page)

(continued from previous page)

```
      {}

      constexpr distance operator+=(const distance& other) {
        return distance(m + other.m);
      }
      constexpr distance operator-=(const distance& other) {
        return distance(m - other.m);
      }
      constexpr distance operator*=(double scalar) {
        return distance(m*scalar);
      }
      constexpr distance operator/=(int scalar) {
        return distance(m/scalar);
      }

    private:
      double m;       // meters
  };

  constexpr distance operator+(distance lhs, const distance& rhs){
    return distance(lhs+= rhs);
  }
  constexpr distance operator-(distance lhs,const distance& rhs){
    return distance(lhs-= rhs);
  }
  constexpr distance operator*(int scalar, distance a){
    return distance(a*=scalar);
  }
  constexpr distance operator/(distance a, size_t denominator){
    return distance(a/=denominator);
  }

} // end namespace length
```

We might choose to add more, but these 4 demonstrate the basic idea.

We should also implement the complete set of relational overloads, since there is no reason to treat distances as anything other than completely regular types.

Working exclusively in meters is not always convenient, so we can also add distance literals so that we can easily work with numbers that are either meters or kilometers:

```
namespace length{
  namespace unit{
    constexpr distance operator "" _km(long double d){
      return distance(1000*d);
    }
    constexpr distance operator "" _m(long double m){
      return distance(m);
    }
  } // end namespace unit
} // end namespace length
```

Notice that these overloads are non-friend non-member functions. Each simply constructs a new distance based on the

units implied by the literal used.

Using distance

Finally we can write some functions that use our constexpr class.

Here we add a free function tthat takes a list of distances and accumulates an average. We could have used std::accumulate, or in C++17 and later, we could use std::reduce to achive the same outcome.

Once we have that, we can define some distances, generate a few weeks works of values and compute the final result.

```
constexpr length::distance average_distance(std::initializer_list<length::distance>␣
↪distances){
  auto sum = length::distance{0.0};
  for (auto d: distances) sum = sum + d;
  return sum/distances.size();
}

int main(){
  using namespace length::unit;

  constexpr auto work = 63.0_km;
  constexpr auto commute = 2 * work;
  constexpr auto gym = 2 * 1600.0_m;
  constexpr auto shopping = 2 * 1200.0_m;

  constexpr auto week1 = 4*commute + gym + shopping;
  constexpr auto week2 = 4*commute + 2*gym;
  constexpr auto week3 = 4*gym     + 2*shopping;
  constexpr auto week4 = 5*gym     + shopping;

  constexpr auto avg_travel = average_distance({week1,week2,week3,week4});

  return int(avg_travel); // 264000m
}
```

Run It

This example does not print a value, but merely returns the final value from main. If you're curious as to why, copy this code into the online Compiler explorer

```
#include <cstdlib>
#include <initializer_list>

namespace length{

  class distance{
    public:
      explicit constexpr distance(double i)
        :m{i}
      {}

      constexpr distance operator+=(const distance& other) {
        return distance(m + other.m);
      }
```

(continues on next page)

(continued from previous page)

```
      constexpr distance operator-=(const distance& other) {
        return distance(m - other.m);
      }
      constexpr distance operator*=(double scalar) {
        return distance(m*scalar);
      }
      constexpr distance operator/=(int scalar) {
        return distance(m/scalar);
      }

      constexpr operator int() const { return static_cast<int>(m);}

    private:
      double m;         // meters
  };

  constexpr distance operator+(distance lhs, const distance& rhs){
    return distance(lhs+= rhs);
  }
  constexpr distance operator-(distance lhs,const distance& rhs){
    return distance(lhs-= rhs);
  }
  constexpr distance operator*(int scalar, distance a){
    return distance(a*=scalar);
  }
  constexpr distance operator/(distance a, size_t denominator){
    return distance(a/=denominator);
  }

  namespace unit{
    constexpr distance operator "" _km(long double d){
      return distance(1000*d);
    }
    constexpr distance operator "" _m(long double m){
      return distance(m);
    }
  }

} // end namespace length

constexpr length::distance average_distance(std::initializer_list<length::distance>␣
↪distances){
  auto sum = length::distance{0.0};
  for (auto d: distances) sum = sum + d;
  return sum/distances.size();
}

int main(){
  using namespace length::unit;

  constexpr auto work = 63.0_km;
  constexpr auto commute = 2 * work;
```

(continues on next page)

(continued from previous page)

```
  constexpr auto gym = 2 * 1600.0_m;
  constexpr auto shopping = 2 * 1200.0_m;

  constexpr auto week1 = 4*commute + gym + shopping;
  constexpr auto week2 = 4*commute + 2*gym;
  constexpr auto week3 = 4*gym     + 2*shopping;
  constexpr auto week4 = 5*gym     + shopping;

  constexpr auto avg_travel = average_distance({week1,week2,week3,week4});

  return int(avg_travel); // 264000m
}
```

Declaring all variables as `constexpr` means all instances of distance and all functions are constant expressions. The compiler performs all of these operations at compile time. That means the *entire program will be executed at compile time* and all the program variables and instances are immutable.

Try This!

Copy this code into the online Compiler explorer and see what the generated code looks like.

Try setting the compiler optimiztion in the explorer "compiler options" text box: *-O2* - does anything change? It should!

Is the final symbol code what you expected?

What do you think is going on here?

More to Explore

- The content on this page was adapted from Rainer Grimm's blog *MODERNES C++*: Immutable data
- From cppreference.com
  - constexpr
- C++ Core Guidelines
  - Con: Constants and immutability
  - Con.5: Use constexpr for values that can be computed at compile time

3.13 Stacks and Queues

The Adapter design pattern is explored in the context of the STL containers `stack` and `queue`.

3.13.1 The Adapter pattern

Design patterns provide a reliable and easy way to follow proven design principles and to write well-structured and maintainable code. One of the popular and often used patterns in object-oriented software development is the adapter pattern. It follows Robert C. Martin's Dependency Inversion Principle and enables you to reuse an existing class even so it doesn't implement an expected interface.

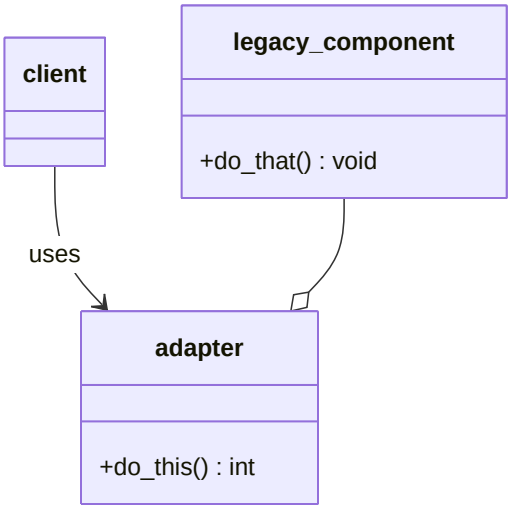
If you do some research on the adapter pattern, you will find two different versions of it:

1. The **class** adapter pattern that implements the adapter using *inheritance*.
2. The **object** adapter pattern that uses *composition* to reference an instance of the wrapped class within the adapter.

The object adapter pattern is generally more popular and the one used within the STL to implement stack and queue.

The general idea of an adapter in software development is identical to the one in the physical world. If you have travelled to foreign countries, you probably noticed that electrical outlet vary from country to country. Outlet shapes vary such that the plug of your electrical device doesn't fit. How do you connect the charger of your mobile phone or laptop to these power outlets?

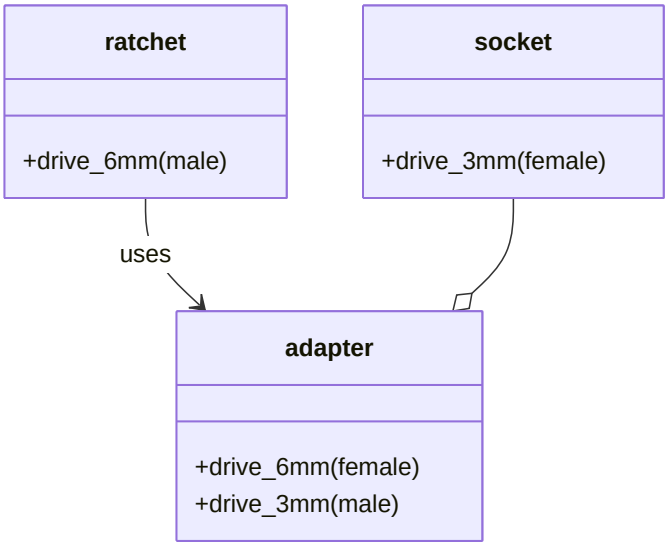
The answer is simple. You get an adapter which you can put into the power outlet and then you put your plug into the other end of the adapter. The adapter changes the form of your plug so that you can use it with the power outlet. In that example and in most other situations, the adapter doesn't provide any additional features. It just enables you to connect your plug to a different outlet.



Often when programming you have a class that does *almost* what you need it to, or it contains a lot of capability you would like to reuse, but the class interface is not a good fit. A class adapter is a simple design pattern that helps solve problems like this. We can use an adapter when we want to convert the interface of a class into another interface clients

expect. An adapter lets classes work together that couldn't otherwise because of incompatible interfaces. An adapter also allows us to wrap an existing class with a new interface without making any changes to the original class - the class being adapted.

Socket wrenches provide an example of the Adapter. A socket attaches to a ratchet, provided that the size of the drive is the same. Typical drive sizes in the United States are 1/2" and 1/4". Obviously, a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used. A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket.



The data structures in this chapter stack and queue both use the adapter patter to achieve their design goals.

More to Explore

- Sourcemaking - Adapter Design Pattern
- DIP in the Wild (The Dependency Inversion Principle)
- Design Patterns Explained - Adapter Pattern with Code Examples

3.13.2 The stack class

Sometimes we want to limit access to all parts of a sequential data structure. In other words, we want to store as much data as we want to, but we want to restrict the ability to access any element at random. One way to limit access to only one end of a container is to use a stack.

Imagine creating a pile by adding items one at a time on top of each other:

- plates
- pancakes
- sheets of paper
- stones

Any of these visual analogies you prefer will work. Each of them is a *stack*. In each case, adding items to the top of the stack makes other items deeper in the stack inaccessible. The only way to observe or remove an item from the stack is to remove all of the items above it first.

Because the first item added to a stack is also the item farthest from the top of the stack, we refer to a stack as a Last-In-First-Out (LIFO) data structure.

The std::stack is a container adapter that gives the programmer the functionality of a stack.

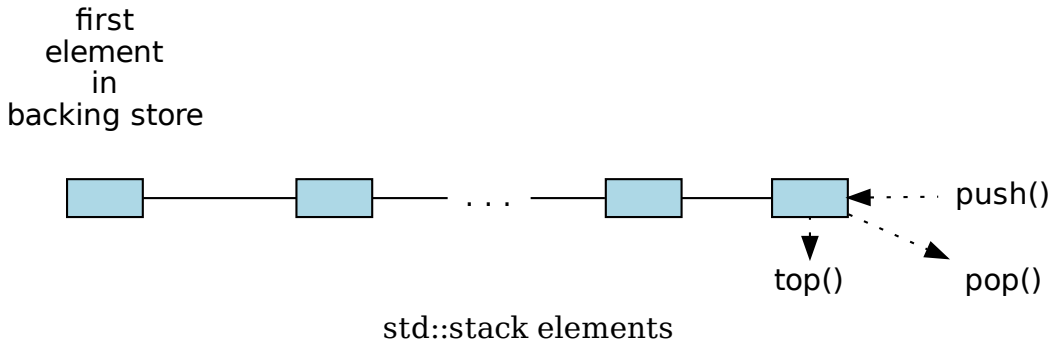
The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

Given a `std::stack<T>`, the defining operations of a stack are:

void push (T value)
: Add a new element to the top of the stack.

void pop()
: Remove an element from the top of the stack.

T top()
: Get the value of the element at the top of the stack.



Using std::stack

Before running the following example, predict the output, then check yourself.

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

using std::cout;
using std::stack;

// remove all elements from a stack and print them out
template <class Container>
void pop_all(Container& s) {
  while(!s.empty()) {
    cout << s.top() << " ";
    s.pop();
  }
  cout << "\npopped all from stack\n";
}

int main () {
  stack<std::string> strings;
  cout << "push strings onto stack...\n";
  strings.push("one");
  strings.push("two");
  strings.push("three");
  strings.push("four");
  strings.push("five");

  cout << "size of stack before: " << strings.size() << '\n';
  pop_all (strings);
  cout << "size of stack after: " << strings.size() << '\n';
  if (strings.empty()) {
    cout << "stack is empty.\n";
  }
```

(continues on next page)

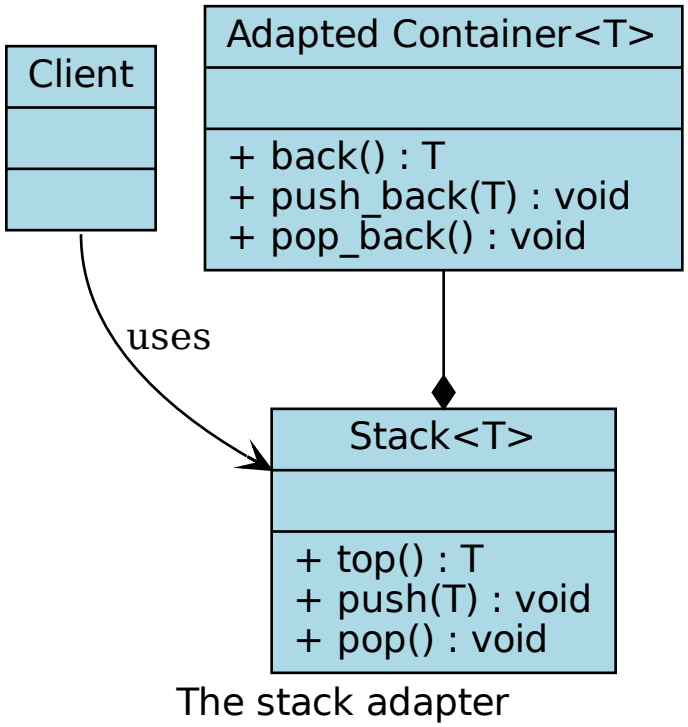
(continued from previous page)

```
33
34
35   return 0;
36 }
```

It is also possible to initialize a stack from another container. The initializing container must match the container adapted for the stack instance. By default, deque is used, but any container that provides

- `back()`
- `push_back()`
- `pop_back()`

can be used as a stack adapter. In the STL, besides deque, vector and list also can be adapted by a stack.



The stack adapter

Initializers

Because the default backing store for a stack is a deque, a container adapter does not need to be specified.

```
// initialize stack from deque
std::deque<int> x = { 1, 2, 3, 4, 5 };
stack<int>> numbers(x);
```

To copy a list into a stack will only work if the stack instance uses a list as its backing store.

```
// initialize stack from list
std::list<int> y = { 1, 2, 3, 4, 5 };
stack<int, std::list<int>> numbers(y);
```

Run It

Before running the following example, predict the output, then check yourself.

```
#include <iostream>
#include <stack>
#include <list>

using std::cout;
using std::stack;

// remove all elements from a stack and print them out
template <class Container>
void pop_all(Container& s) {
  while(!s.empty()) {
    cout << s.top() << " ";
    s.pop();
  }
  cout << "\npopped all from stack\n";
}

int main () {
  cout << "initialize stack from list:\n";
  std::list<int> tmp = { 1, 2, 3, 4, 5 };
  stack<int, std::list<int>> numbers(tmp);

  cout << "list has " << tmp.size() << " entries\n";
  pop_all (numbers);
  if (numbers.empty()) {
    cout << "stack is empty.\n";
  }

  return 0;
}
```

Notice the elements from the list are pushed onto the stack in the order they are retrieved from the list. The number 1 is pushed first, so when initialization is complete, it is on the bottom of the stack.

Stack elements **cannot** be accessed directly in the way you are used to with other sequential containers like arrays, vectors, and lists. To 'visit' each element in a `stack`, the items need to be popped off.

If you think you need to visit all the elements in a `stack`, then you probably should not be using a `stack`.

Postfix Notation

A compiler generates machine instructions required to carry out the statements of a source program written in a high-level language. One part of this task is to generate instructions for evaluating arithmetic expressions such as

```
value = a * (b + c);
```

In most programming languages, arithmetic expressions are written using **infix notation** as in the above example. The symbol for each binary operation is placed between the operands. Many compilers first transform infix expressions into **postfix notation**, and then generates machine instructions to evaluate these postfix expressions. This two-step process is used because transformations from infix to postfix is straightforward, and postfix expressions are generally easier to evaluate than infix expressions.

In postfix notation the operator follows the operands and parentheses are not needed. In the earlier example, the infix expression:

```
2 * (3 + 5);
```

can be re-written as a postfix expression:

```
2 3 5 + *
```

Evaluation this expression proceeds left to right:

```
// Scan numbers until the first operator is encountered
// operate on the operands immediately to the left
// of the operator
2 3 5 + *

// becomes
2 8 *

// which becomes
16
```

This method of evaluating a postfix expression requires that the operands be stored until an operator is encountered in the left-to-right scan. Once an operator is found, the last two operands must be retrieved and combined using the operation encountered. This suggests that a stack should be used to store the operands.

Each time an operand is encountered, it is pushed onto the stack. Then, when an operator is encountered, the top two values are popped from the stack; the operation is applied to them, and the result is pushed back onto the stack.

More to Explore

- STL containers library
- STL stack class
- MyCodeSchool video: Data structures: introduction to stack

3.13.3 The queue class

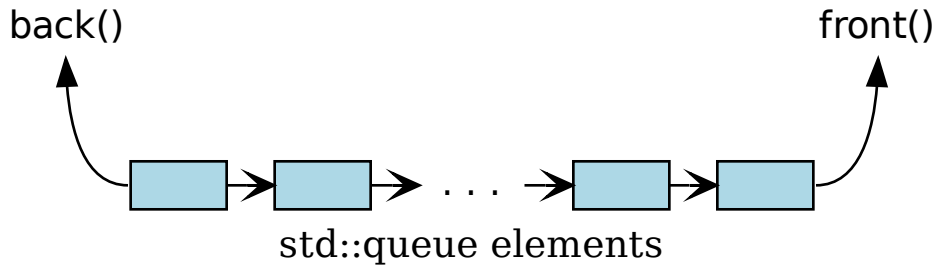
A queue is another special purpose container adapter that limits random element access to all parts of the storage. A *queue* is just another word for a line. Like a stack, a queue restricts element access to the ends. *Unlike* a stack, a queue allows access to both ends:

- New elements can only be added to the "back" of the line
- Elements can only be retrieved from the "front" of the line.

Imagine a line at the bank or a store. An orderly queue means that the people who get in line first are the first customers called. This is the guarantee queue enforces. A queue is a FIFO (first-in, first-out) data structure.

The std::queue is a container adapter that gives the programmer the functionality of a queue.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The queue pushes elements on the back of the underlying container, and pops them from the front.



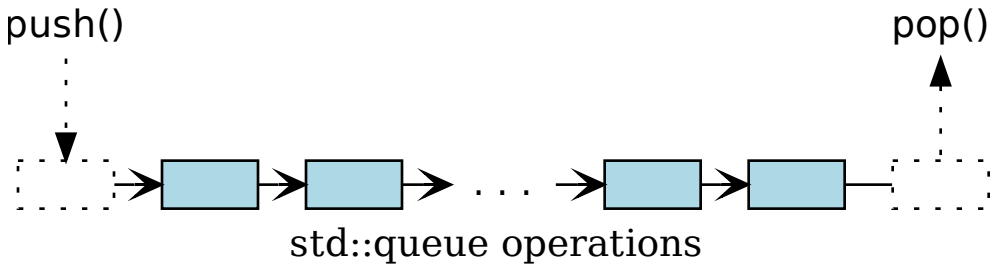
The defining operations of a queue are:

push
: Add a new element to the back (end) of the queue.

pop
: Remove an element from the front (beginning) of the queue.

front
: Get the value of the element at the beginning of the queue.

back
: Get the value of the element at the end of the queue.



Minor modifications change `pop_all()` from a function performing `stack` operations into one performing `queue` operations:

```
#include <iostream>
#include <queue>
```

(continues on next page)

(continued from previous page)

```
#define QueueContainer typename

template <QueueContainer C>
void pop_all(C& q) {
  while(!q.empty()) {
    std::cout << q.front() << " ";
    q.pop();
  }
  std::cout << "\npopped all from queue\n";
}
```

The STL containers `std::list` and `std::deque` can be adapted to create a queue.

Circular queues

A circular queue, cyclic buffer, or ring buffer is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end. A ring buffer is a good choice when you need the behavior of a queue and the buffer size can be fixed.

There are many ways to implement this data structure and the following is just an example of one.

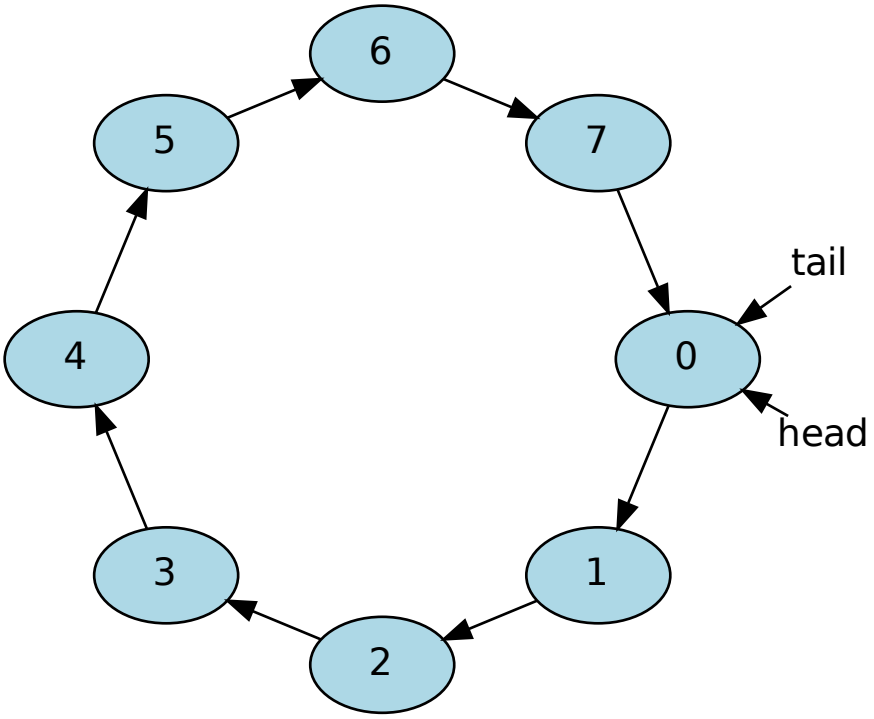
Empty

Conceptually, a circular buffer is a closed ring of data.

- One element needs to be chosen as the start of the data. The terms `start`, `begin`, or `head` are all reasonable.
- One element needs to be chosen as the end of the data. The terms `last`, `end`, or `tail` are all reasonable.

The start locations of `head` and `tail` are arbitrary.

A ring buffer is initially empty and of some predefined length. For example, this is an 8-element buffer conceptually:

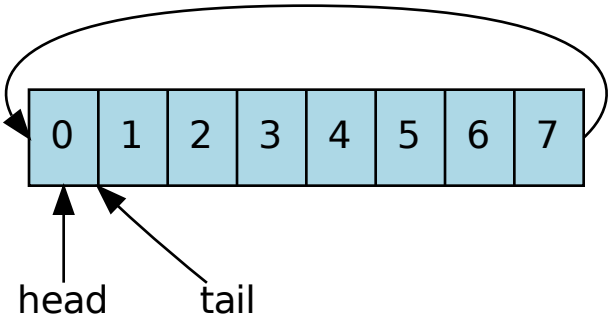


The `tail` node always refers to the element just past the end of our data (as always). So when the head and tail are equal, the buffer is empty.

The `capacity` is the maximum number of elements that can be stored in the buffer. In this example, the capcacity is `8`.

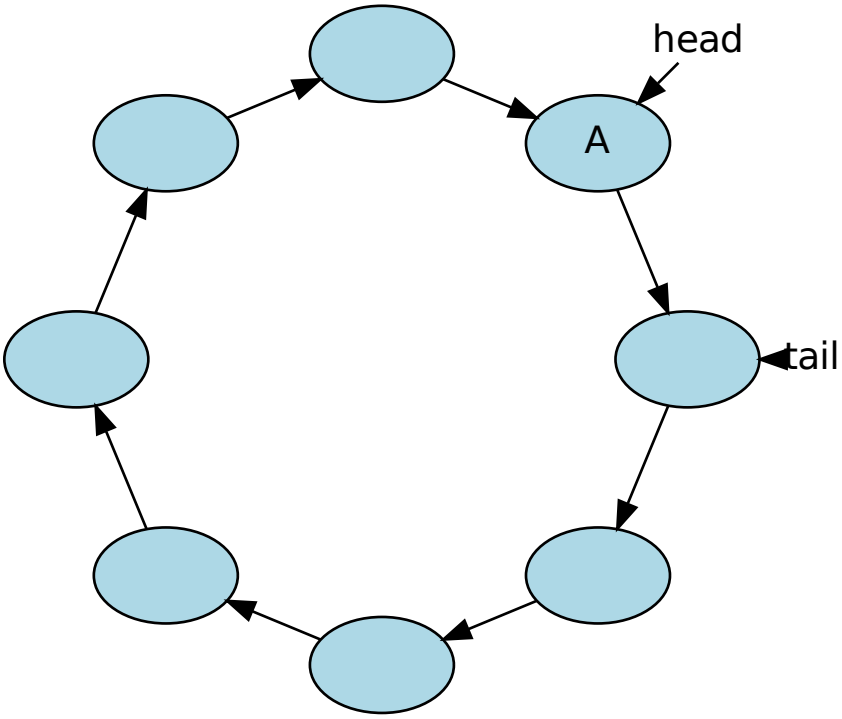
The `size` is the current number of elements used in the buffer. In our initially empty buffer, the size is `0`.

Since there are no true circular sections of memory, it is normal to represent a circular buffer in a normal contiguous linear memory block. An array is a good choice.

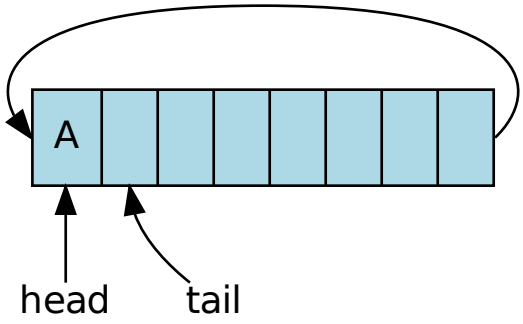


Add

Adding one element to the buffer involves storing a new value at the tail location and moving the tail.

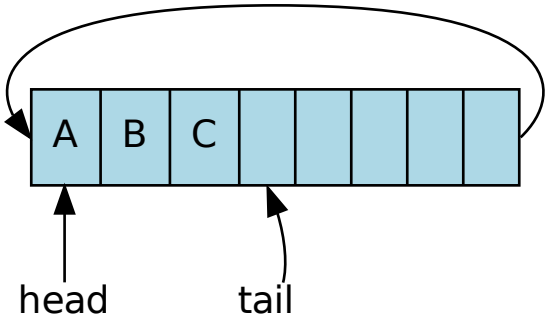


And in the array:



The buffer size is now 1.

If two more items are added, the tail moves accordingly. The head does not move as items are added.



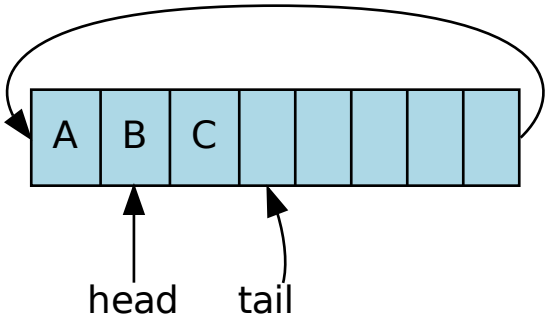
The buffer size is now 3.

Remove

Removing an element from the buffer involves

- returning the oldest element from the buffer
- moving the head
- decrease buffer size

As with earlier containers, there is no need to erase the oldest element, since after the head has moved, we can no longer access it.



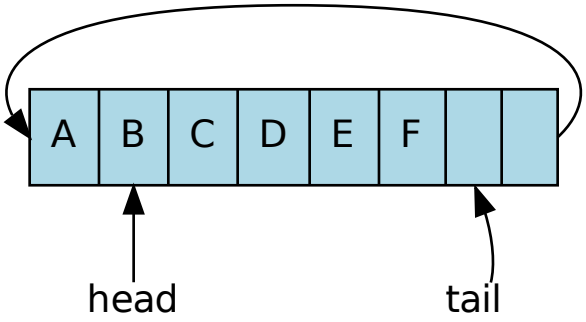
The buffer size is now 2.

Full Buffer

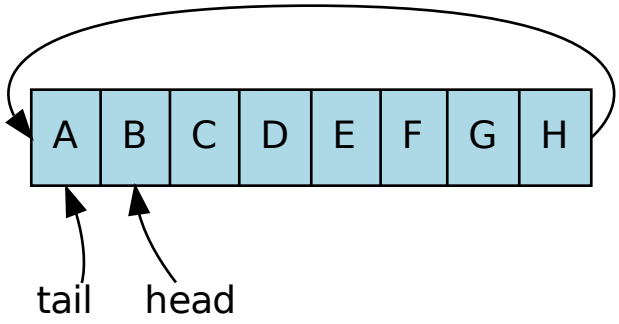
Starting with our buffer containing [B,C], we can add elements until it is completely full.

Recall we popped `A` from this buffer earlier.

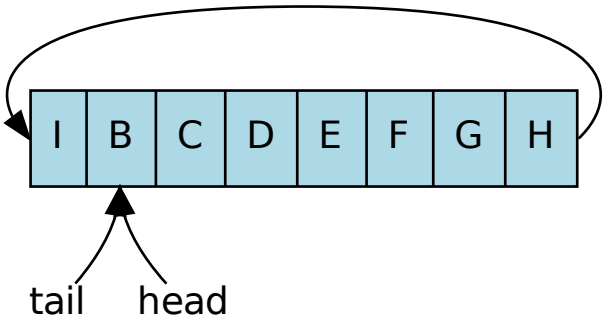
Adding a few elements moves the tail and increases the size.



At this point the buffer is almost full. The tail now refers to the first element in the array. It needed to wrap around to avoid potentially allowing a write outside the array bounds. The slot containing 'A' is available for writing, since it was removed earlier.



One more write to the element at position 0 and now the buffer is completely full.



The buffer size is now `8`. It is important to note that in this implementation `head == tail` does not represent the idea of an empty buffer. In this implementation the `head == tail` state can mean either a completely empty or a full queue.

An extra variable `size` is used to distinguish empty from full, since we know the array size ahead of time. If we chose to not keep track of size and use only the head and tail then the maximum size of the would be $capacity - 1$.

Different designs could result in different outcomes, there are no hard and fast rules here. I chose this implementation because it is easy to reason about and does not waste a storage slot, at the cost of an additional variable.

What do we do when our buffer is full? At this point, we have choices:

- Allow no writes to the buffer until elements are removed.

  This is common when it is important to never lose any information, such as when processing keystrokes from the user, or managing a print queue.

- Allow writes to overwrite the oldest elements. The oldest values are lost in favor of new values.

  This implementation is used when the oldest information may no longer be important enough by the time the buffer is full.

More to Explore

- STL containers library
- STL queue class
- MyCodeSchool video: Data structures: introduction to queue
- Circular buffer on wikipedia

3.13.4 The deque class

The deque (double-ended queue) is an indexed sequence container that allows fast insertion and deletion at both its beginning and its end. In addition, insertion and deletion at either end of a `deque` never invalidates pointers or references to the rest of the elements.

It's primary role in the standard library is to function as the default container underlying `std::stack` and `std::queue`.

std::deque

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

using std::cout;
using std::deque;
using std::string;

void print (const deque<string>& container)
{
  cout << "\n\nItems in the Deque: \n";
  for (const auto& value: container) {
    cout << value << " ";
  }
  cout << '\n';

}

int main() {
  deque<string> d;
  cout << "Deque Empty? " << d.empty() << endl;
  d.push_back("Zebra");
  cout << "Deque Empty? " << d.empty() << endl;

  d.push_front("Turtle");
  d.push_front("Panda");
  d.push_back("Catfish");
  d.push_back("Giraffe");

  cout << "Deque Size: " << d.size() << endl;
  cout << "Item at the front: " << d.front() << endl;
```

(continues on next page)

(continued from previous page)

```
  cout << "Item at the back: " << d.back() << endl;

  print (d);

  d.pop_back();
  d.pop_front();

  cout << endl << "\nItem at the front: " << d.front();
  cout << "\nItem at the back: " << d.back();
  cout << "\nDeque Size: " << d.size();

  print (d);

  return 0;
}
```

Run It

Listing 12: Basic std::deque usage

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

using std::cout;
using std::deque;
using std::string;

void print (const deque<string>& container)
{
  cout << "\n\nItems in the Deque: \n";
  for (const auto& value: container) {
    cout << value << " ";
  }
  cout << '\n';

}

int main() {
  deque<string> d;
  cout << "Deque Empty? " << d.empty() << endl;
  d.push_back("Zebra");
  cout << "Deque Empty? " << d.empty() << endl;

  d.push_front("Turtle");
  d.push_front("Panda");
  d.push_back("Catfish");
  d.push_back("Giraffe");

  cout << "Deque Size: " << d.size() << endl;
  cout << "Item at the front: " << d.front() << endl;
  cout << "Item at the back: " << d.back() << endl;
```

(continues on next page)

(continued from previous page)

```
  print (d);

  d.pop_back();
  d.pop_front();

  cout << endl << "\nItem at the front: " << d.front();
  cout << "\nItem at the back: " << d.back();
  cout << "\nDeque Size: " << d.size();

  print (d);

  return 0;
}
```

An interesting problem that can be easily solved using the deque data structure is the classic palindrome problem. A **palindrome** is a string that reads the same forward and backward, for example, *radar*, *toot*, and *madam*. We would like to construct an algorithm to input a string of characters and check whether it is a palindrome.

One solution to this problem uses a deque to store the characters of the string. First store each character in the string into a new deque. Using the properties of the deque, we can process the characters from both ends and compare them to each other.

Since we can remove both of them directly, we can compare them and continue only if they match. If we can keep matching first and the last items, we will eventually either run out of characters or be left with a deque of size 1 depending on whether the length of the original string was even or odd. In either case, the string must be a palindrome.

Check Palindrome

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

using std::deque;
using std::string;

bool check_palindrome(const string& value) {
  if (value.size() < 2) return true;
  deque<char> letters (value.begin(), value.end());

  while (letters.size() > 1) {
    char first = letters.front();  // could omit these temporaries
    char last = letters.back();
    if (first != last) {
       return false;
    }
    letters.pop_front();
    letters.pop_back();
  }
  return true;
}
```

Run It

Listing 13: Palindrome checker using std::deque

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

using std::deque;
using std::string;

bool check_palindrome(const string& value) {
  if (value.size() < 2) return true;
  deque<char> letters (value.begin(), value.end());

  while (letters.size() > 1) {
    char first = letters.front();  // could omit these temporaries
    char last = letters.back();
    if (first != last) {
       return false;
    }
    letters.pop_front();
    letters.pop_back();
  }
  return true;
}

int main() {
  std::cout << std::boolalpha
            << check_palindrome("not a palindrome") << '\n';
  std::cout << check_palindrome("radar") << '\n';
  return 0;
}
```

A moment of full disclosure: even though it is possible to use a deque to determine if a string is a palindrome or not, it's far from the simplest or most efficient solution tot he the problem. Simply checking the string characters directly is better:

Is Palindrome

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

bool is_palindrome(const std::string& value) {
 return equal(value.begin(),
              value.begin() + value.size()/2,
              value.rbegin());
}
```

The STL provides the equal template which allows comparing the values in a pair ranges.

The first *begin* and *end* define a range of values to be compared. The second *begin* defines the start of the second range of values. The second *end* does not need to be specified, because the comparison stops once the first end is reached.

Notice that in this example, the second begin is *rbegin*. This means the second iterator starts at the **reverse beginning**, which is the *end* of the string and each iteration moves one step closer to the begining.

Run It

Listing 14: Palindrome checker using std::equal

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

bool is_palindrome(const std::string& value) {
 return equal(value.begin(),
              value.begin() + value.size()/2,
              value.rbegin());
}

int main() {
  std::cout << std::boolalpha
            << is_palindrome("not a palindrome") << '\n';
  std::cout << is_palindrome("radar") << '\n';
  return 0;
}
```

While this solution does require more familiarity with the standard library, it avoids copying the string into the container, removing elements from the container, and is generally simpler.

More to Explore

- STL containers library
- STL deque class

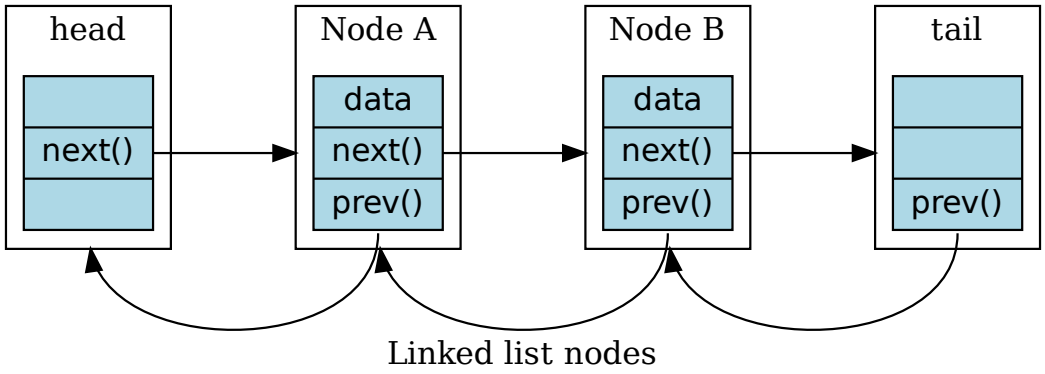
3.14 Linked lists

This section focuses on a sequential data structure in which data elements do not occupy a continuous block of memory. While linked list are useful, they do present a problem that will need to be solved: visiting the next linked list item in a generic way.

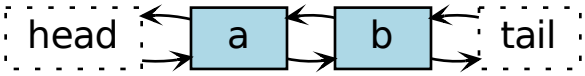
We use this problem as our springboard for iterators.

3.14.1 The list class

The std::list is a sequence container that stores data in *nodes*. Each *node* in a *list* points to the next (and previous) node in the list. Each node is a separate object that exists to encapsulate a piece of data and to allow navigation to adjacent nodes.



A more compact way to graphically represent our *doubly linked list* is like this:



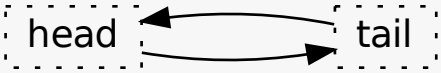
A *linked list* that stores a sequence of `int`s can be trivially implemented using a `struct`:

```
struct node {
   int value;
   node* next;
   node* prev;
};
```

The `struct node` contains a single value it 'owns', plus pointers to adjacent nodes.

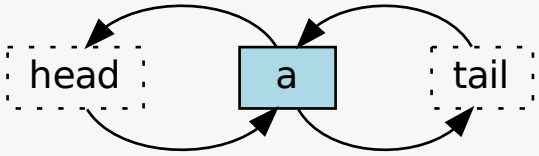
Creating a *linked list* from such a 'home grown' `struct` is not complicated, but it isn't pretty either:

An empty list



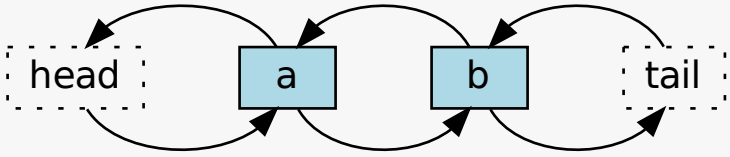
```
// create an empty list
node* head = new node;
node* tail = new node;
head->next = tail;
tail->prev = head;
```

Insert node 'a' to our list




```
// insert node a into the list
node* a = new node;
a->value = 61;
a->next = tail;
a->prev = head;
head->next = a;
tail->prev = a;
```

Insert node 'b' after a




```
// insert node b after node a
node* b = new node;
b->value = 62;
b->next = tail;
b->prev = a;
a->next = b;
tail->prev = b;
```

At this point, we have created the basic structure shown in the first list diagram. Once we have such a list, we can access all of the elements, if we have a pointer to any one of them. For example, to print all of the elements, we could:

```
node* p = head->next;
while (p->next != nullptr) {
  std::cout << p->value << ' ';
  p = p->next;
}
```

Which, given the list we created, will print `61 62\0`.

Obviously, no one would want to use such a list. Every trivial detail needs to be managed, and any program using it would be more likely to leak memory or fail suddenly due to some programming error.

The std::list class hides all the implementation details and provides a list with many convenient features:

```
#include <iostream>
#include <list>
using std::cout;

void print_list(const list<int>&);

int main () {
  std::list<int> list = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  cout << "size: "  << list.size();
  cout << "\nfront: " << list.front();
  cout << "\nback: "  << list.back();

  cout << "\n\npush_back 13: ";
  list.push_back(13);
  cout << "\nsize: "  << list.size();
  cout << "\nback() " << list.back();

  print_list(list);

  return 0;
}

void print_list(const std::list<int>& list) {
  if (list.empty()) {
    cout << "list is empty.\n";
  } else {
    cout << "list contains:\n";
  }
  for(const int i: list) {
    cout << i << " ";
  }
  cout << "\n\n";
}
```

The defining operations of a list are:

push_back
: Add a new element to the end of the list.

pop_back

Remove an element from the end of the list.

back
Get the value of the element at the end of the list.

push_front
Add a new element to the beginning of the list.

pop_front
Remove an element from the beginning of the list.

front
Get the value of the element at the beginning of the list.

Note

empty() vs. size() == 0

In most containers, calling `size()` is constant time.

That is it takes the same amount of time regardless of the number items in the container.

Not so for lists.

There are situations where a list cannot determine the size without traversing the range and counting them.

In general, never assume `size()` is as efficient as `empty()`.

If you **really** want to know if a container is empty (or not), then call `empty()`.

If you **really** want to know the number of elements in a container, then call `size()`.

See *Effective STL*, for more details[1].

Underneath, the standard library `list` is not very different from the `struct node` above. The primary characteristics are:

- All data is stored on the heap
- Node traversal is accomplished by following pointers from one node to the next
- Access based on an index is not allowed. This kind of access, called *random access* describes the ability to compute a location in memory using a starting address and an offset. Arrays and vectors support random access. Linked lists do not.

More to Explore

- STL containers library
- STL iterator library
- Visualgo: lists

Footnotes

[1] Effective STL (Item #4) by Scott Meyers (Addison-Wesley Professional). Copyright 2001 Scott Meyers, 978-0-201-74962-5.

3.14.2 Analysis of list operators

Conventional wisdom states that a list is faster than a vector when operating on elements other than the back. We know `push_back()` is $O(1)$ for vector and we know that vector doesn't have a 'push_front', because it's something we intuitively feel we should discourage in a vector. Every push front would involve shifting all the elements in the vector down one position.

The table below shows the average complexity efficiency of some basic list operations. Note that many are constant time. Note that many list operations such as `insert` and `erase` take an iterator as a parameter. Once you have the iterator, these operations take constant time, however, getting the correct iterator can often take $O(n)$, if you have not saved the iterator from a previous operation.

Table 4: **Complexity of C++ List Operators**

Operation	Complexity
assignment =	O(n)
push_back()	O(1)
pop_back()	O(1)
erase(i)	O(1)
insert(i, item)	O(1)
insert(i, b, e)	O(n) in the range from b, e
splice()	O(1)
begin()	O(1)
end()	O(1)
size()	O(1) or O(n) C++11
size()	O(1) after C++11

Both `vector` and `list` support an `insert()` method. There are multiple overloads for each and both support inserting a range of elements at an arbitrary location in the container. The *following code* shows the code for inserting a range into a container.

```
template<class Container>
void test_insert(Container data, Container new_data){
    data.insert(data.begin(), new_data.begin(), new_data.end());
}
```

The ref:*test_insert code* <*lst_test_insert*> inserts the range at the beginning of the current data set. This situation should benefit the linked list and handicap the vector. This is one of the classic situations where linked lists are said to outperform vectors. Let's insert chunks of data onto the front of both a list and a vector. *The following code* shows what happens when an `int` is stored in the containers.

In this example, we take increasingly large containers and insert increasingly large containers to their fronts.

Each insert operation creates a new vector with an initial size. The test function then inserts a second vector of equal size at position 0.

```
#include <chrono>
#include <iostream>
#include <iomanip>
#include <list>
#include <vector>

using std::list;
using std::vector;

```

(continues on next page)

(continued from previous page)

```
template<class Container>
void test_insert(Container data, Container other){
    data.insert(data.begin(), other.begin(), other.end());
}


int main(){
    using std::cout;
    using clock = std::chrono::high_resolution_clock;
    using msec_t = std::chrono::duration<double, std::milli>;

    cout << std::setw(6) << "size\t"
         << std::setw(8) << "vector::insert\t"
         << std::setw(8) << "list::insert\n";

    for(int size = 10'000; size < 100'001; size += 10'000) {
      vector<int> vector_data (size);
      vector<int> new_vector_data (size);
      auto begin = clock::now();
      test_insert(vector_data, new_vector_data);
      auto end = clock::now();
      msec_t elapsed_1 = end - begin;

      list<int> list_data (size);
      list<int> new_list_data (size);
      auto begin2 = clock::now();
      test_insert(list_data, new_list_data);
      auto end2 = clock::now();
      msec_t elapsed_2 = end2 - begin2;

      cout << std::setprecision(6) << std::fixed
           << size << '\t'
           << std::setw(8) << elapsed_1.count() << '\t'
           << std::setw(8) << elapsed_2.count() << '\n';
    }
    return 0;
}
```

Both list and vector have similar complexity for this form of insert. Both are linear in `std::distance(first, last)` - and vector has an additional linear term in the distance between the insert position and end of the container. (Vector has all those moves to perform). Since we chose to insert at the first element location and force the destination vector to resize on every insert, we really expect lists to outperform vector.

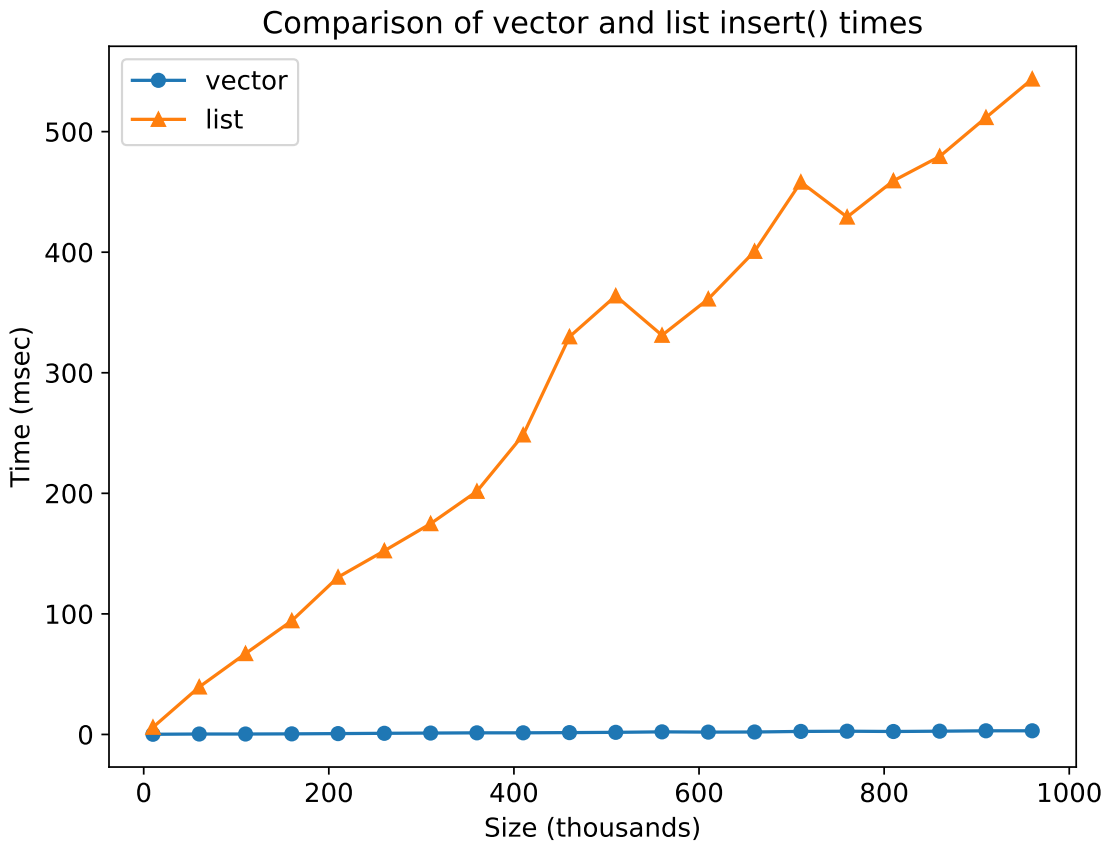
But it's not even close. Running the previous code on values up to 1,000,000 should produce results similar to this:

Try This!

The online compiler is limited in both memory and time allowed.

Run this example on your own computer with larger values and compare.

It may not look like it, but both of these lines are both $O(n)$ on the distance over the size of the range inserted. It's just that for small types like `int`, the vector is on average 150 times faster than the linked list.


Comparison of vector and list insert() times
vector
list
Time (msec)
0
100
200
300
400
500
0
200
400
600
800
1000
Size (thousands)

How can this be?

In short: memory.

Recall that for a vector all the data resides in a single chunk of data. For a linked list, each new member lives in a separate location.

Computers have a feature called cache memory and it turns out the vector is able to exploit this resource better than a list.

What is Cache Memory?

Cache memory is a small amount of computer memory that provides high-speed data access to a processor and stores frequently used computer programs, applications and data. Cache memory is the fastest memory available and acts as a buffer between RAM and the CPU. When a processor requests data that already has an instance in cache memory, it does not need to go to the main memory or the hard disk to fetch the data. The processor checks whether a corresponding entry is available in the cache every time it needs to read or write a location which reduces the time required to access information.

Cache memory is relatively small - it is intended to speed access to frequently used data, not serve as a replacement for RAM. When the cache is full and something needs to be written, the least frequently used data is overwritten.

Although both keep their data on the free store, because the vector is a single chunk, the CPU has a better chance of keeping more of the data in cache memory.

In addition, it turns out that modern CPU's are just very good at creating, copying, and moving chunks of memory.

There are situations where a list does outperform vector, we just have to work harder to see it.

To force our test containers to work harder, instead of a vector of `int`, we create a type that is a simple array wrapper:

```
// a bloated class to make insert work harder
template<int N>
struct junk {
    std::array<unsigned char, N> data;
};
```

Other than the change in type stored in the vector, nothing is different from the `int` timing example.

```
vector<junk<2048>> vector_data (size);
```

```
#include <array>
#include <chrono>
#include <iostream>
#include <iomanip>
#include <list>
#include <vector>

using std::list;
using std::vector;

template<class Container>
void test_insert(Container data, Container other){
    data.insert(data.begin(), other.begin(), other.end());
}

```

(continues on next page)

(continued from previous page)

```
// a bloated class to make insert work harder
template<int N>
struct junk {
    std::array<unsigned char, N> data;
};


int main(){
    using std::cout;
    using clock = std::chrono::high_resolution_clock;
    using msec_t = std::chrono::duration<double, std::milli>;

    cout << std::setw(6) << "size\t"
         << std::setw(8) << "vector::insert\t"
         << std::setw(8) << "list::insert\n";

    constexpr int JUNK_SIZE = 2048;

    for(int size = 1'000; size < 10'001; size += 1'000) {
      vector<junk<JUNK_SIZE>> vector_data (size);
      vector<junk<JUNK_SIZE>> new_vector_data (size);
      auto begin = clock::now();
      test_insert(vector_data, new_vector_data);
      auto end = clock::now();
      msec_t elapsed_1 = end - begin;

      list<junk<JUNK_SIZE>> list_data (size);
      list<junk<JUNK_SIZE>> new_list_data (size);
      auto begin2 = clock::now();
      test_insert(list_data, new_list_data);
      auto end2 = clock::now();
      msec_t elapsed_2 = end2 - begin2;

      cout << std::setprecision(6) << std::fixed
           << size << '\t'
           << std::setw(8) << elapsed_1.count() << '\t'
           << std::setw(8) << elapsed_2.count() << '\n';
    }
    return 0;
}
```

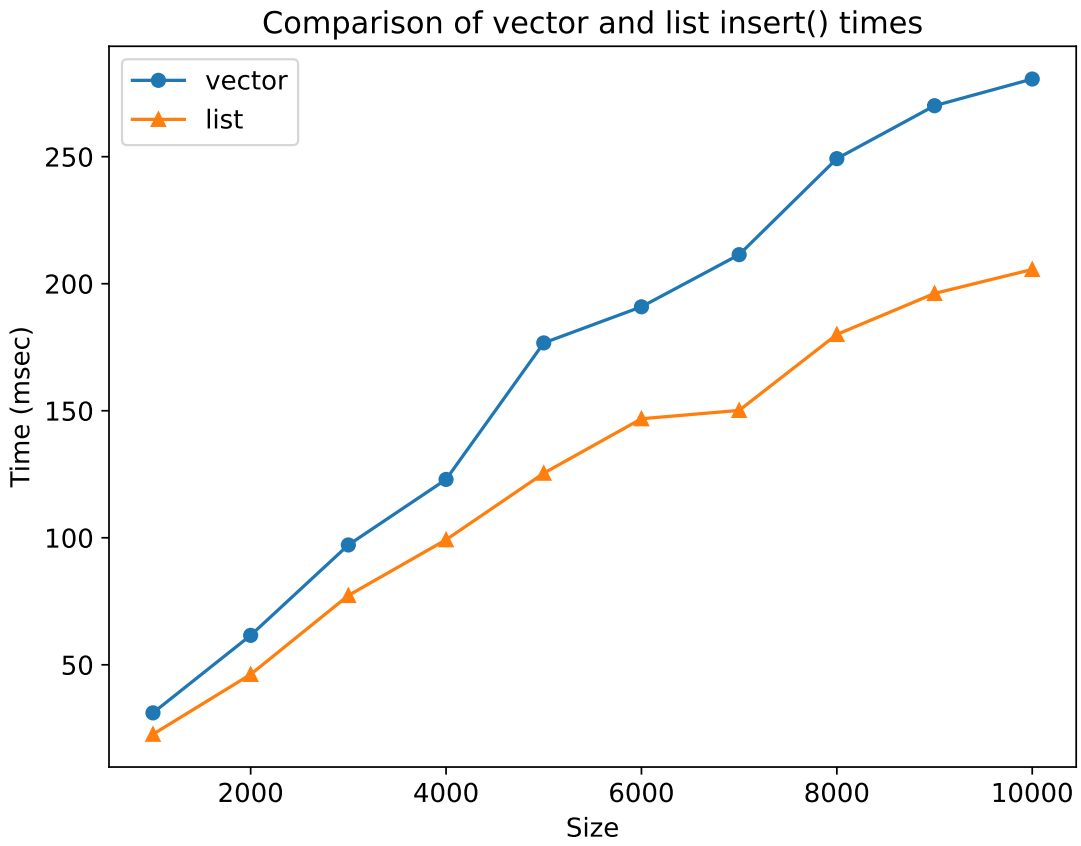
A change in data type stored in the vector produces different results.

The online compiler is limited in both memory and time allowed. The 2MB value for `JUNK_SIZE` is roughly the 'break even' point on this compiler. The graph below shows what running with and 8MB size looks like.

Run this example on your own computer with larger values and compare.

The sheer size of the data in each vector element increases the likelihood of a *cache miss*. In this case, the data is too large to fit much, if anything, in cache memory. The CPU fails to find it in cache, so it must retrieve it from RAM every time.

Both the vector and list are clearly $O(n)$ and the list is outperforming the vector.


Comparison of vector and list insert() times
vector
list
Time (msec)
250
200
150
100
50
2000
4000
6000
8000
10000
Size

Try This!

What other situations might a list outperform a vector. Try some of the following with data types of different sizes:

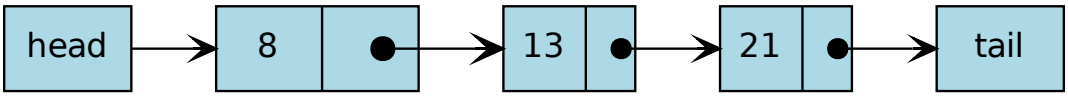
- Reversing data
- Sorting data
- Filling or constructing data
- Removing data

More to Explore

- C++ benchmark - std::vector VS std::list VS std::deque

3.14.3 std::forward_list

Like list, the forward_list is a container that stores elements in *nodes* A *forward list* only defines pointers to the next node in the list. This means that a forward list can only be traversed in the direction of the tail.



The defining operations of a forward_list are:

push_front
: Add a new element to the beginning of the list.

pop_front
: Remove an element from the beginning of the list.

front
: Get the value of the element at the beginning of the list.

Compared to list this container provides more space efficient storage when bidirectional iteration is not needed. A very light-weight container, it does not have any overhead compared to its implementation in C.

More to Explore

- STL containers library
- STL iterator library
- Visualgo: lists

3.14.4 Iterable ADT's

How can we visit each *element* in a *container* and remain ignorant of the underlying container implementation details? For example, given:

```
array<string, 3> names = {"Alice", "Bob", "Clara"};
std::list<int>   ages  = {27, 3, 1};
```

What options do we have for operating on each *element* in `names` and `ages`? A traditional for or while loop works for `names`:

```
for (unsigned i=0; i < names.size(); ++i) {
  cout << names[i] << '\n';
}

unsigned i = 0;
while(i < names.size()) {
  cout << names[i++] << '\n';
}
```

However, this code does not compile:

```
for (unsigned i=0; i < ages.size(); ++i) {
  cout << ages[i] << '\n';
}

unsigned i = 0;
while(i < ages.size()) {
  cout << ages[i++] << '\n';
}
```

Traditional loops using an `int` index do not work with containers that do not overload `operator[]`. Containers in this category include list, set, and map.

We solve this problem by avoiding explicit indexing altogether. The range-based for loop provides a more readable equivalent to the traditional for loop:

```
for (string s: names) {
  cout << s << '\n';
}

// better: avoids copying
for (const auto& s: names) {
  cout << s << '\n';
}
```

The same syntax can be used for any STL container:

```
std::list<int>  ages  = {27, 3, 1};

for (const auto& a: ages) {
  cout << a << '\n';
}
```

When you need to loop over each element in a collection, the range-for loop has completely abstracted away the idea of moving from one element to the next.

We say these containers are *iterable*.

More to Explore

- Iterator Library at cppreference.com
- C++ Concepts: Iterator

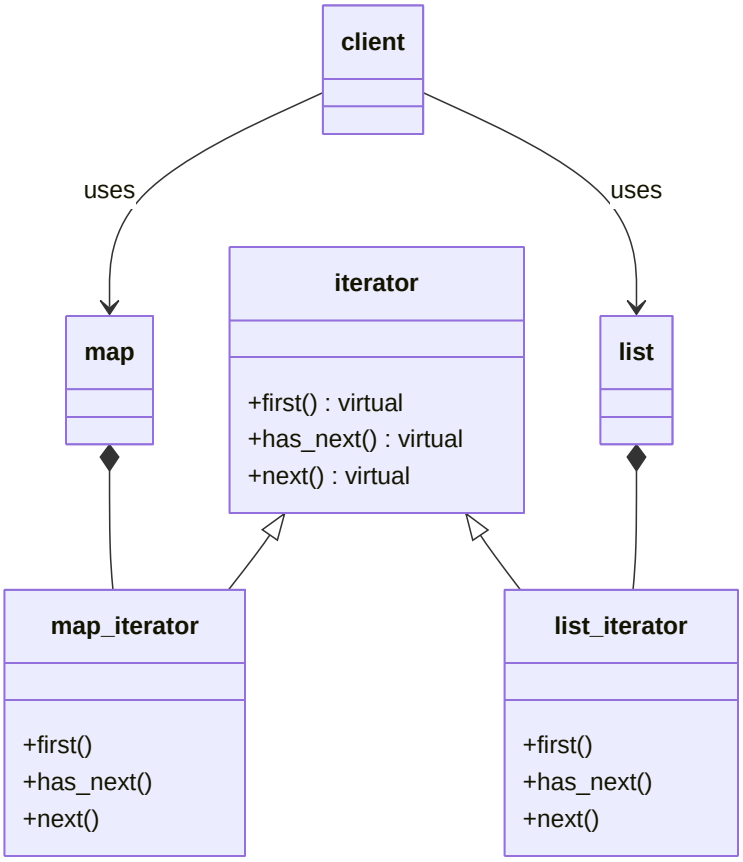
3.14.5 Iterator pattern

The range-for loop works because the function expects a standard interface the loop can use to establish basic facts about the range of elements in the sequence:

- Where does the sequence start?
- Where is the next element?
- When does the sequence end?

  This last question can alternatively be asked as *"Is there a next element?"*

Most OO languages solve this problem using a form of the iterator design pattern.

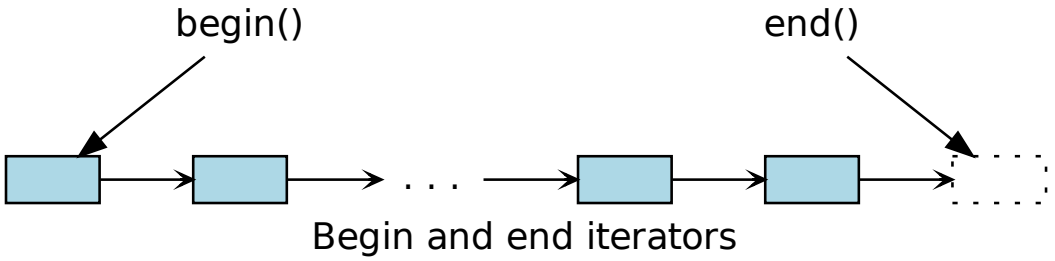


Because design patterns represent general ideas about solving classes of problems, they are language independent. In the case of *iterators*, the idea has solutions in most modern languages, including C++. Each language generally provides iterators using a design appropriate for the language. C++ is no different.

C++ implements iterators using pointer semantics and an *Iterator* base class is generally avoided in C++ iterators. Since classes can overload all of the pointer operations, an iterator can be implemented that exposes a pointer interface.

The key advantage to this solution is that functions can be written more generically. Functions interact with a simple, consistent and well-known interface that works both for user defined types, built-in pointer types, and arrays. However, this solution does require an "end" iterator to test for equality.

Each STL container class provides an *iterator* class that clients can use to retrieve the correct *element* from the *container*.



The element defined by `begin()` is part of the sequence.

The element defined by `end()` is **not part** of the sequence. In C++, the `end` iterator is always one past the end of the sequence. Forgetting this is a common source of error.

More to Explore

- Iterator Library at cppreference.com
- C++ Concepts: Iterator

3.14.6 Basic iterator operations

By design, iterators in C++ feel as if you are using pointers. Most iterators support the same operations as pointers.

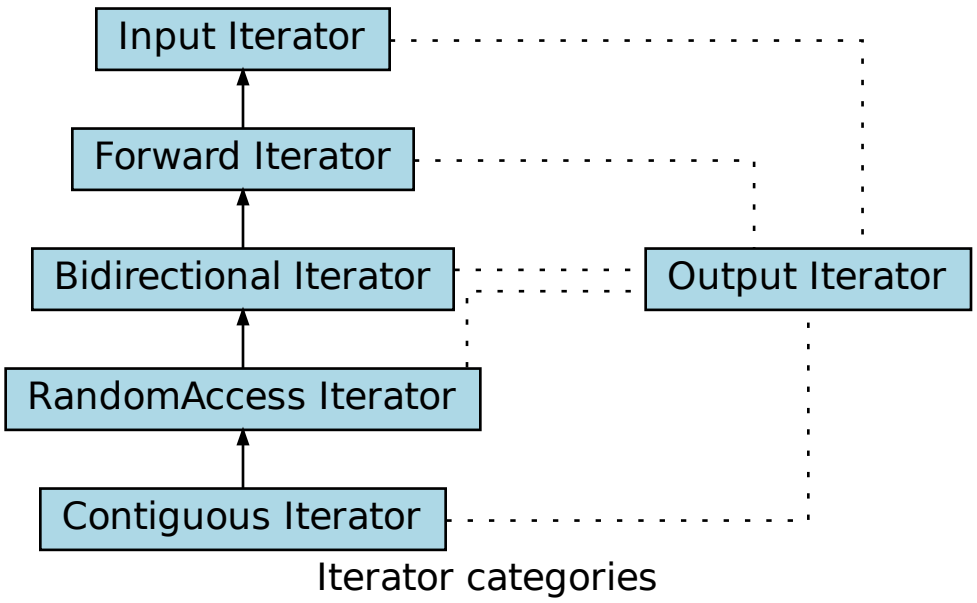
Operation	Result
`p == q`	`true` if and only if `p` and `q` point to the same element or both point to `end`
`p != q`	negation of above
`*p`	refers to the element pointed to by `p`
`*p = val`	writes `val` to the element pointed to by `p`
`val = *p`	reads from the element pointed to by `p` and writes to `val`
`++p`	increment the iterator, making it point to the next element in the container or to `end`

Iterator categories

Different containers need different capabilities from their iterators.

Instead of being defined by specific types, each category of iterator is defined by the operations that can be performed on it. This definition means that any type that supports the necessary operations can be used as an iterator -- for example, a pointer supports all of the operations required by Random Access Iterator, so a pointer can be used anywhere a Random Access Iterator is expected.

All of the iterator categories (except Output Iterator) can be organized into a hierarchy, where more powerful iterator categories Random Access Iterator) support the operations of less powerful categories (e.g. Input Iterator). If an iterator falls into one of these categories and also satisfies the requirements of Output Iterator, then it is called a *mutable iterator* and supports both input and output. Non-mutable iterators are called *constant iterators*.



Input Iterator
: Read elements and increments using `operator++`, without multiple passes. Classes like basic_istream provide this iterator.

Forward Iterator
: InputIterator, plus increment using `operator++`, with multiple passes. The forward_list container provides this iterator.

Bidirectional Iterator
: ForwardIterator, plus decrement using `operator--` Containers like list, map, and set provide this iterator.

Random Access Iterator
: BidirectionalIterator, plus access using `operator[]` Before C++17, containers like vector, array, and string provided this iterator. It is still used for unordered collections like unordered_map and unordered_set

Contiguous Iterator
: RandomAccessIterator, plus the container makes a continuous storage guarantee. This category was added in

C++17. Before C++17, iterators of containers like vector and array were often treated as a separate category. This category simply formalizes what was happening in practice.

Output Iterator

More of a 'sub category' of all of the others. If the iterator allows writing to the element, it is also an OutputIterator An output iterator can **only** be dereferenced on the left-hand side of an expression:

```
vector<int> v = {1,2,3};
auto it = v.begin();
*it = 0;
```

More to Explore

- Iterator Library at cppreference.com
- C++ Named Requirements: Iterator

3.14.7 Using iterators

Assigning an iterator explicitly to a variable works much like any other type:

```
vector<int> nums = {1, 2, 3, 4, 5};
vector<int>::iterator it = nums.begin();
```

The variable `it` now points to the beginning of the container `nums` and can used much like a pointer:

```
vector<int> nums = {1, 2, 3, 4, 5};
vector<int>::iterator it = nums.begin();

cout << *it;  // prints: 1
```

The iterator type always matches the value type of the enclosing container. Just as with pointers, an iterator to a `vector<int>` is a different type from an iterator to a `vector<string>`.

It is possible to declare an iterator and use it in a traditional for loop:

```
vector<int> nums = {1, 2, 3, 4, 5};
cout << "nums contains:";

for (vector<int>::iterator it = nums.begin();
     it != nums.end(); ++it) {
    std::cout << ' ' << *it;
}
```

Which produces:

```
nums contains: 1 2 3 4 5
```

A while loop can produce equivalent results:

```
vector<int> nums = {1, 2, 3, 4, 5};
cout << "nums contains:";
```

(continues on next page)

(continued from previous page)

```
vector<int>::iterator it = nums.begin();
while (it != nums.end())
    std::cout << ' ' << *it;
    ++it;
}
```

We can shorten either example with `auto`, since the compiler can easily determine what type is returned from `begin()`:

```
for (auto it = nums.begin(); it != nums.end(); ++it) {
    std::cout << ' ' << *it;
}
```

Example code like one of the two previous examples is commonly found on the web, even when the point of the example has nothing to do with iterators. When you don't need an iterator, don't use it:

```
for (const auto& n: nums) {
    std::cout << ' ' << n;
}
```

A common source of error for new programmers is confusion about the types used in these two loops:

`begin()`
: **Always** returns an iterator that must be dereferenced in order to access the element value.

The range for declaration
: **Always** is assigned a value from the container. Unless the container is a container of pointers, no dereferencing is needed.

Limits of Range-based for loops

The range-for loop, while convenient, has limitations.

Any situation in which you do not need or want to visit every element requires a traditional loop:

```
for (int i=n; i>0; i/=2) {
  // do something with i . . .
}
```

Similarly, if you need to iterate through multiple containers in a single loop, possibly at different rates, then you need a traditional for loop:

```
for (int i=n, j=0; i>0; i/=2, j++) {
  // do something with i and j . . .
}
```

If you need to *traverse* a container and remove items, then you need an explicit iterator so that you can call the container `erase` method.

See the erase example in the following section.

Container functions that require iterators

Most container functions that use position information do not accept an integral position or an index like `operator[]`. Position information is expressed using iterators.

insert
: Inserts elements at the specified location in the container.

Example

Create a vector<int> and initialize it with 3 values:

```
std::vector<int> nums(3,100);
```

Insert a value at the beginning of the vector:

```
auto it = nums.begin();
it = nums.insert(it, 200);
```

Insert 2 values at the beginning of the vector:

```
it = nums.insert(it, 2, 200);
```

Insert one vector into another vector:

```
auto it = nums.begin();
std::vector<int> fib {1, 1, 2, 3, 5, 8, 13, 21};
nums.insert(it+2, fib.begin(), fib.end());
```

Run it

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

void print(const std::vector<int>& v) {
  for (auto x: v) {
    std::cout << ' ' << x;
  }
  std::cout << '\n';
}

int main () {
  std::vector<int> nums(3,100);
  print(nums);

  auto it = nums.begin();
  it = nums.insert(it, 200);
  print(nums);

  nums.insert(it,2,300);
  print(nums);

  // 'it' no longer valid, get a new one:
  it = nums.begin();

  std::vector<int> fib {1, 1, 2, 3, 5, 8, 13, 21};
  nums.insert(it+2, fib.begin(), fib.end());
  print(nums);

  int arr[] = { 501,502,503 };
  nums.insert(nums.begin(), arr, arr+3);
```

(continues on next page)

(continued from previous page)

```
  print(nums);
}
```

erase
: Removes specified elements from the container. `erase` may remove a single element or a contiguous range of elements.

Example

Given a `vector<int>`:

```
std::vector<int> nums = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

We can erase the first element:

```
nums.erase(nums.begin());
```

or erase a range of adjacent elements:

```
print(nums);
```

or erase other elements:

```
for (auto it = nums.begin(); it != nums.end(); ) {
  if (*it % 2 == 0) {
    it = nums.erase(it);
  } else {
    ++it;
  }
}
print(nums);
```

Things to note about the last erase example:

- `it` is not incremented in the for loop iteration expression
- If an element is erased, the current iterator is *invalidated*. Any further use would be an error in a `vector`.

  The vector::erase function returns the iterator to the next element in the container.
- If an element is **not** erased, *then* increment the iterator.

Run it

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

void print(const std::vector<int>& v) {
  for (auto x: v) {
    std::cout << ' ' << x;
  }
  std::cout << '\n';
}

int main( ) {
```

(continues on next page)

(continued from previous page)

```
  std::vector<int> nums = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  print(nums);

  nums.erase(nums.begin());
  print(nums);

  nums.erase(nums.begin()+2, nums.begin()+5);
  print(nums);

  nums = {2, 7, 1, 8, 2, 8, 1, 8, 2, 8, 4, 5, 9};

  // Erase all even numbers
  for (auto it = nums.begin(); it != nums.end(); ) {
    if (*it % 2 == 0) {
      it = nums.erase(it);
    } else {
      ++it;
    }
  }
  print(nums);

  return 0;
}
```

More to Explore

- From cppreference.com
  - Iterator Library
  - C++ Iterator Named Requirement
  - std::vector::erase
  - std::vector::insert

3.15 Trees and associative data structures

This section introduces what the standard library refers to as "associative data structures" - non-sequential data structures.

3.15.1 Tree ADT concepts

If sequence containers like vector are so great, then why would we need anything else?

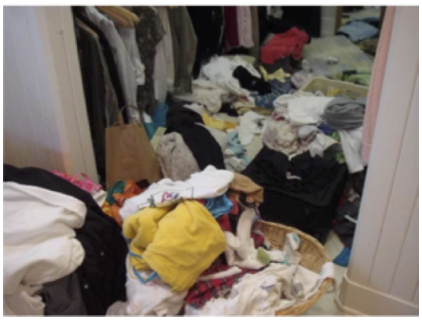
In a word: search.

When we have millions a elements in a data structure and need to find just one element, or a specific range or elements, we *could* use a `vector`.

Inserts are fast. No matter how many elements are already in a `vector` adding one more using push_back takes the same amount of time. That is, the time cost of `push_back` is *constant* for a `vector`.

A `vector` is default ordered only by its index position, not by the values stored within it. It's easy to keep throwing items in without paying any attention to how they are ordered.

But always using `push_back` is analogous to a messy closet. We could consider the closet to be ordered by *depth*: the last things thrown in the closet are on the top of the pile.



This makes getting a specific item from the closet slow. If we only ever want to to access the last item we added, then we know exactly where to go. But if we want to find some arbitrary item, we have to search the vector 1 element at a time until we find it.

```
vector<int> messy_closet (1024 * 1024 * 1024);  // a fairly big vector

// modify closet . . .

cout << "Go get some coffee while I work on this. . . \n";
for (const auto& v: messy_closet) {
  if (v == search_val) {
    do_something(v);
  }
}
```

Sometimes we may get lucky and find the desired element at index position 0. If the data added to the vector is random, then this becomes increasingly less likely as the size grows.

We might sometimes get very unlucky and not find the element until we access the last element. Over many searches, on average, we will have to examine $\frac{N}{2}$ elements.

It's easy to see that the more elements are added, the longer searches will take.

We need a tidy closet.

We could sort the `vector`, which would speed up our search. The basic idea is to sort the vector, then examine the value at position $\frac{N}{2}$. If the value found is greater than the value we are looking for, then examine the value at position $\frac{N}{4}$, else examine the value at position $\frac{3N}{4}$.

At each step, we eliminate the number of remaining elements we need to search in our `vector` by half. For a large `vector`, this saves a lot of time.

This technique requires that we keep the vector sorted. If elements are added or removed frequently, then adding data to our vector, which used to be fast, is now slow. We can either use push_back followed by sort, or use insert. *Every* addition becomes a search and we are back to the original problem. On average, it will take $\frac{N}{2}$ comparisons to add new data.

How can we solve this problem?

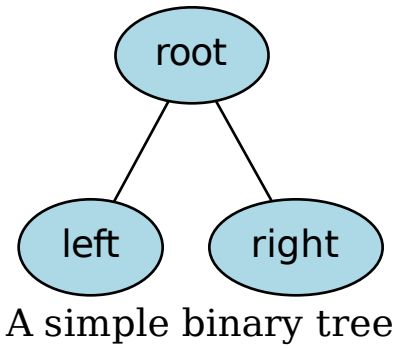
Can we make an *ADT* whose performance does not degrade as the number of elements in the *ADT* grows large?

Yes, but we need a new idea. Instead of a sequential container, we need a *tree*.

The tree ADT

A *tree* is a *hierarchical* abstract data type. Conceptually, it can be thought of as a collection of *nodes* defined by parent-child relationships.

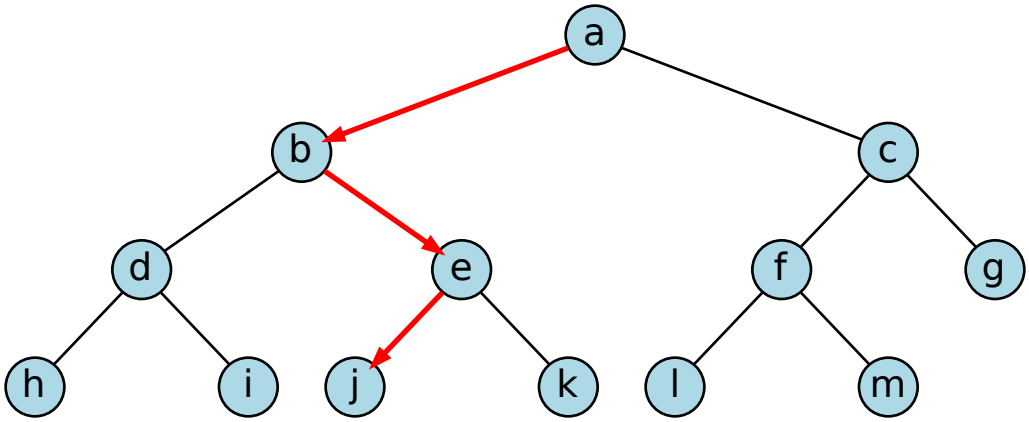
One node is the *root*. It serves as the 'trunk' of the tree and serves the same function as the *head* of a *list* The root node is the *only* node in a tree without a parent. All other nodes in a *tree* refer to exactly 1 parent. In a *binary tree*, the children are commonly referred to as the **left** and **right** nodes.



A simple binary tree

Yes, programmers draw trees upside-down. The *root* is above the branches.

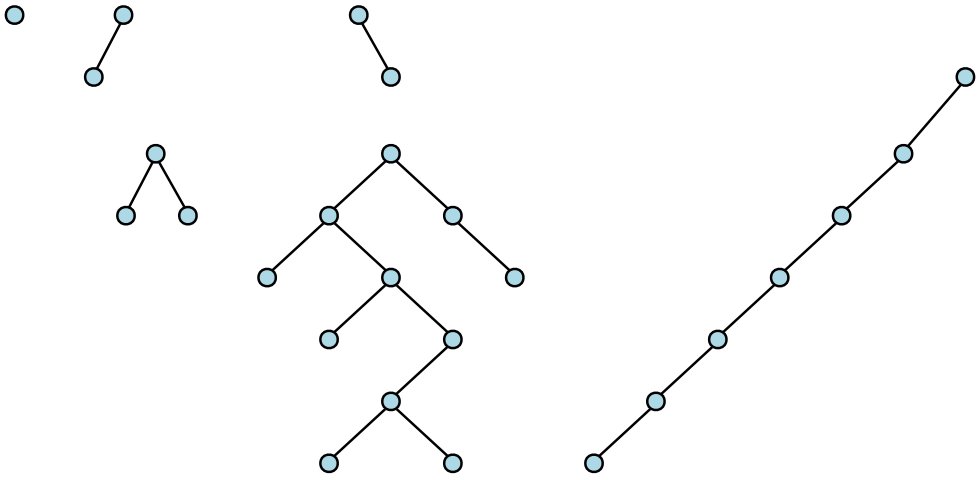
The *height* of a tree is the count of the nodes along the longest path in a tree from the *root* to a *leaf node*.



A tree of height 4

Although there are many different types of trees, we need only worry about *binary trees*. A *binary tree* is a tree in which no node has more than 2 children. Any tree node may have 0, 1, or 2 children. A tree node with no children is a *leaf node*.

All of these are valid *binary trees*:



Example binary trees

A *balanced tree* (one with the roughly equal numbers of nodes in each *subtree*), provides the tidy room we need to

ensure reasonably fast inserts **and** retrievals. A tree must be both balanced and sorted for us to gain benefits from a tree.

When a tree is balanced and sorted, the cost of both inserts and retrievals are on average $log_2 N$. Binary trees provide a way for us to 'formalize' our half-splitting solution.

Unbalanced trees are not much more than fancy *linked lists*. The performance of unbalanced trees degrades back to the messy room, with all of the problems and none of the benefits.

Balanced trees are the data structures that support both sets and maps.

More to Explore

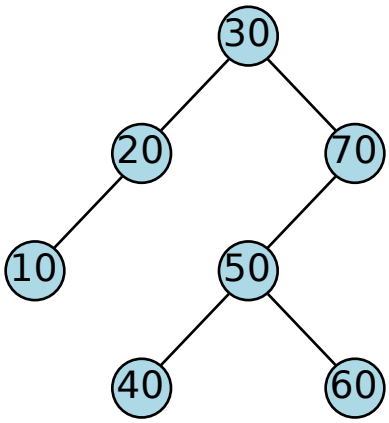
- MyCodeSchool video: Data structures: introduction to trees
- STL containers library
- Visualgo: binary heap
- Wikipedia: binary search algorithm

3.15.2 Binary Search Trees

A binary tree T is a binary search tree if, for each node `n` with sub-trees `left` and `right`,

- The value in `n` is **greater than** the values in every node in `left`.
- The value in `n` is **less than** the values in every node in `right`.
- Both `left` and `right` are binary search trees.

These assertions define the **binary tree property**.



Is this a BST?

Yes.

Each node is greater than or equal to all of its left descendants, and is less than or equal than all of its right descendants.

The Binary Search Tree ADT

Structurally, a BST contains pointers to it's left and right children. As discussed in *Recursion*, a binary tree can be implemented simply as a recursive data structure. A binary search tree can also be implemented recursively.

It is a bit simpler to define the tree nodes as a separate type. Whether you design this class as a completely independent class, like this one, or implement it as a nested (inner) class, is largely a matter of choice.

Since a `tree_node` is a data structure that can exist independently of a tree that enforces the binary search tree property, it makes sense in this case to define it as a completely separate struct with no invariants.

The `tree_node` encapsulates the general characteristics common to all binary trees:

- A variable to store the node value
- Pointers to the left and right child nodes, which might themselves be sub-trees.

```
namespace mesa {
  namespace tree {

   // a binary tree node
   template<class T>
     struct tree_node {
       T value;
       tree_node<T>* left;
       tree_node<T>* right;
       tree_node(const T& value = T{},
           tree_node<T>* left = nullptr,
           tree_node<T>* right = nullptr)
         : value{value}
       , left{left}
       , right{right}
       { }
     };
  } // end namespace tree
} // end namespace mesa
```

In other words, a `tree_node` is a general purpose binary tree data structure and has no knowledge of any binary search tree properties or behavior.

Binary Search Tree Traversal

One benefit of a binary search tree is that when the nodes are visited using infix traversal, the data is sorted ascending.

```
// write a tree to an output stream, infix
template <class T>
   std::ostream& operator<< (std::ostream& os, const mesa::tree::tree_node<T>* node)
   {
     if (node == nullptr) { return os; }
```

```
    os << node->left;
    os << node->value << ' ';
    os << node->right;
    return os;
  }
```

Notice we stream `tree_node` objects here, not `bstree` objects.

Much like our earlier tree objects, all of the functions used to manipulate a `tree_node` will be free functions. To avoid collision with other similarly named functions, all the functions will be defined in the `mesa::tree` namespace.

The binary search tree is built up from individual `tree_node` objects.

The `bstree` class has 1 private member variable: a pointer to a `tree_node`. The basic skeleton of the class should look familiar:

```
namespace mesa {

  // a binary search tree
  template<class T>
    class bstree {
      public:
        typedef T value_type;

        bstree() = default;
        // convert a value into a tree
        explicit
          bstree(const T& value)
          : root{new tree::tree_node<T>{value}}
        { }

        // copy construct and assign
        bstree(const bstree& other);
        bstree& operator=(const bstree& other);

        // move construct and assign
        bstree(bstree&& other);
        bstree& operator=(const bstree&& other);

        constexpr
          bool empty() const noexcept { return root == nullptr; }

      private:
        tree::tree_node<T>* root = nullptr;

    };

} // end namespace mesa
```

Our primary focus for the rest of this section is on the functions that define the key operations associated with a BST:

- contains and find
- insert and erase

Searching binary trees

Efficient search of a binary tree uses the same algorithm you would use when playing the 'number guessing' game. If asked to guess a random number between `1` and `100` in the fewest possible tries, with a hint `higher` or `lower` after each attempt, few people would start at `1`, then guess `2`, `3`, and so on until they guessed correctly. Most people would start with `50` and continue to split the remaining unknown partition in half until they found the correct number.

The strategy most people apply to this problem intuitively is known as the *binary search* algorithm. This algorithm is easily applied to binary search trees.

contains

We always search a binary search tree by comparing the value we're searching for to the 'current' node value. If the target value is smaller, then we search the left subtree. If the target value is larger, then we search the right subtree.

If it is neither of these things, then we found the value.

This function is implemented as a `mesa::tree` free function.

```
template <class T>
bool contains (const T& query_value, tree_node<T>* node)
{
  if(node == nullptr) return false;
  if(query_value < node->value) {
    // search the left sub-tree
    return contains(query_value, node->left);
  }
  if(node->value < query_value) {
    // search the right sub-tree
    return contains(query_value, node->right);
  }
  // we found what we were looking for
  return true;
}
```

Inserting into binary trees

Inserting into a binary tree means adding a new node in the tree such that the binary tree property remains intact.

insert

The insert process begins with a search for a place to insert a new value. But how do we find the place at which to insert that new node? Ask "where would we go if we were searching for this data in the tree?" This process is identical to the search used for the contains function.

This function is implemented as a `bstree` member function.

```
tree::tree_node<T>*
 insert (const T& value, tree::tree_node<T>*& node)
 {
   // add a new leaf
   if(node == nullptr) {
     node = new tree::tree_node<T>(value, nullptr, nullptr);
     return node;
   }
   if(value < node->value) {
```

(continues on next page)

(continued from previous page)

```
    return insert(value, node->left);
  }
  if(node->value < value) {
    return insert(value, node->right);
  }
  // else the value already exists in the tree
  node->value = value;
  return node;
}
```

There are a few important things to notice about this function.

The insert function receives the current node pointer as a reference to a pointer, It can change the value of the pointer provided. It does this specifically when our traversal brings us to a null pointer. A *leaf node* is place where we can insert a new tree node while still adhering to the binary search tree property.

Overwriting an existing value is a design choice. We could have chosen to do nothing and simply return the node.

Try This!

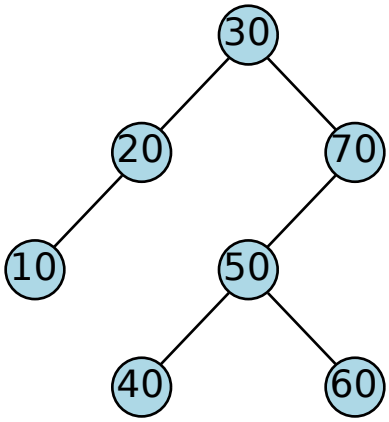
Walk through this algorithm yourself with different sets of values.

Experiment with inserting nodes into binary search trees. Take particular note of what happens if you insert data in ascending or descending order, as opposed to inserting unordered data.

Erasing binary tree nodes

The erase process also begins with a search for the place to erase. This process is identical to the search used for contains and insert.

The tricky part of removing a value from a binary search tree is what to do when we actually find the value we want to delete. We can't just delete the tree node. Consider the following tree.



If we remove values `10`, `40`, or `60` by simply deleting the tree node, that might work. However, deleting any other node would break the links between tree nodes.

We have 3 cases to consider:

- Removing a leaf
- Removing a node that has only one child
  - only a left child
  - only a right child
- Removing a node that has two children

Removing a leaf node

It's easy to see that we can always remove any leaf in a binary search tree without affecting anything else. That is, if we remove any leaf from a binary search tree, we still have a valid binary search tree. There is nothing else to do.

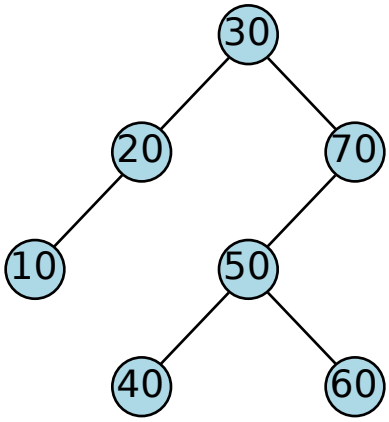
When `node` points to a leaf that contains the data we want to remove we replace the address in `node` by `node->right`. If `node` is pointing to a leaf, then `node->right` is null and we write a nullptr into the parent node, replacing whichever of its two children pointers was used to get to `node`.

In other words, leaf nodes are replaced with the null pointer.

Removing a non-leaf node with a null child

Removing nodes from the interior of the tree is a bit more work as we need to maintain links between nodes.

Given the same tree we have been working with so far:



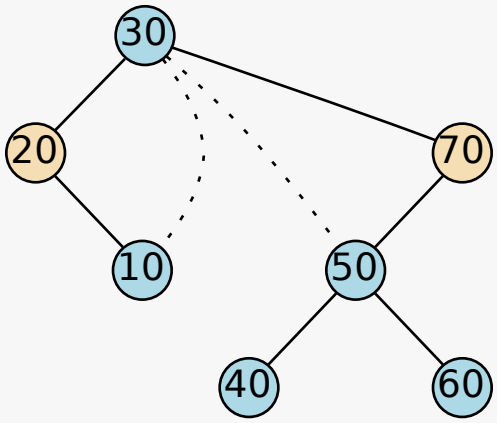
Question Suppose we wanted to remove the `20` or the `70` from this tree. What would we have to do so that the remaining nodes would still be a valid BST?

Show

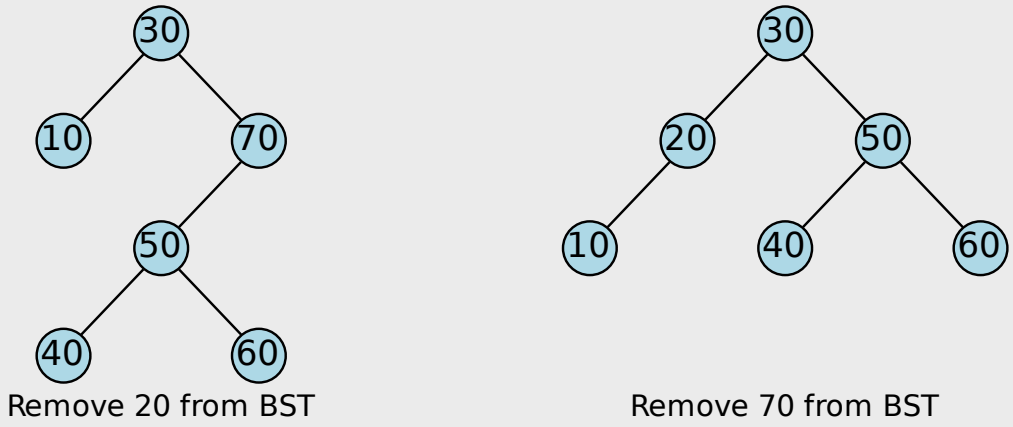
There is one pointer to the node being deleted, and one pointer from that node to its only child.

So this is actually a bit like deleting a node from the middle of a linked list.

All we need to do is to update the pointer from the parent `30` node. That pointer should point to the child of the node we are going to remove.



Verify that if we remove either `20` or `70`, the resulting tree is still a valid binary search tree.



The code we used to remove a leaf also works when there is only one child.

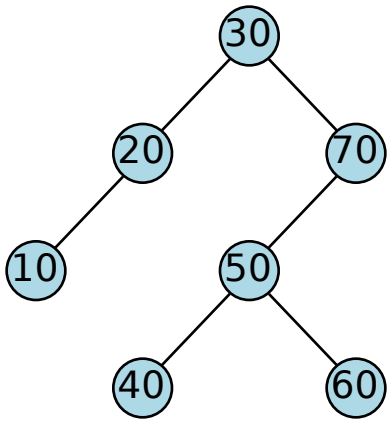
If we reach this code, we know there is at most one non-null child. In the previous case of a leaf node, both children are null, but the same code works for one child also.

If the left child is not null, then reassign the left child to the current node, otherwise assign the right child.

Removing a non-leaf node with a two children

Suppose we wanted to remove the `50` or the `30` from this tree. What must we do so that the remaining nodes would still be a valid BST?

This is a hard case. If we remove either the `50` or `30` nodes, then we break the tree into pieces, with no obvious place to put the now-detached subtrees.



There is an efficient solution to this problem. Instead of deleting the node when we find it, is there some other data value that we could put into that node that would preserve the BST property?

There are, in fact, two values that we could safely put in there:

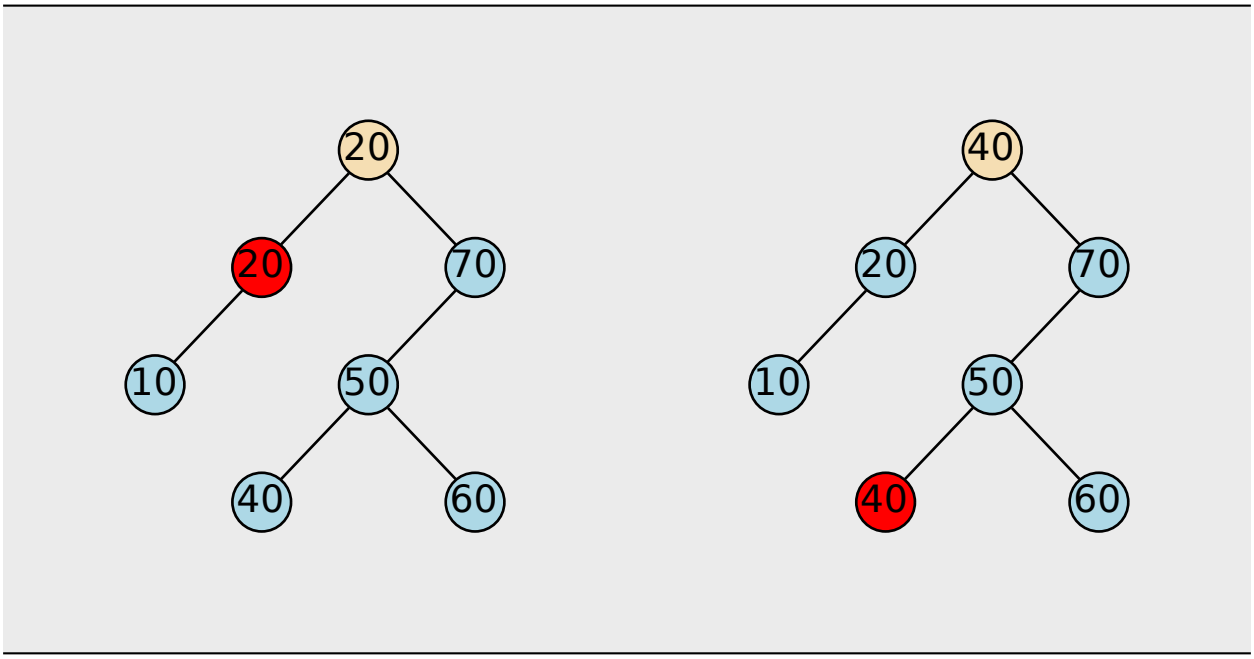
- the smallest value from the right subtree
- the largest value from the left subtree

We can find the **largest** value on the **left** by

- taking one step to the left
- then running as far down to the right as we can go

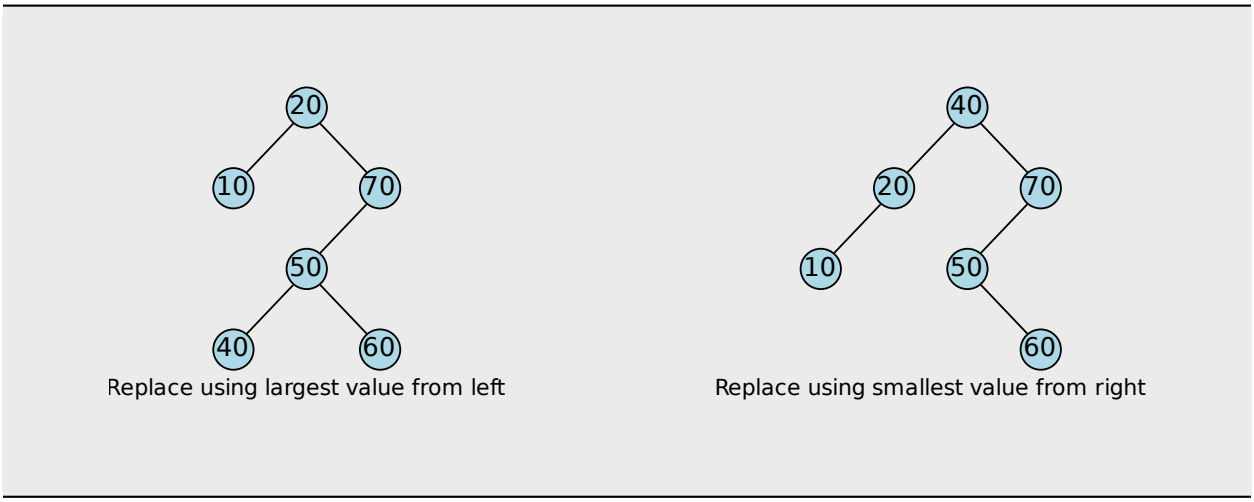
We can find the **smallest** value on the **right** by

- taking one step to the right
- then running as far down to the left as we can go



At this point, we haven't deleted or created any nodes. We simply copy a value from one node to another. Now we have two nodes in our tree with the same value, either `20` or `40`, depending on which approach we used.

We still need to delete the smallest right node or the largest left node. What makes this last step simple is that it falls under our previous case: it is by definition either a leaf, or has at most one child.



erase

Putting it all together.

This function is implemented as a `mesa::tree` free function.

```
template <class T>
  void erase (const T& value, tree_node<T>*& node)
  {
```

(continues on next page)

(continued from previous page)

```
4     if(node == nullptr ) return;
5
6     if(value < node->value) {
7       return erase(value, node->left);
8     }
9     if(node->value < value) {
10      return erase(value, node->right);
11
12    }
13    if(node->left != nullptr && node->right != nullptr ) {
14      // two children
15      // replace node with smallest value from right subtree
16      node->value = min_element(node->right)->value;
17      return erase(node->value, node->right);
18    }
19   // remove a leaf node or node w/ 1 subtree
20   tree_node<T>* trash = node;
21   node = (node->left != nullptr ) ? node->left : node->right;
22   delete trash;
23  }
```

Lines 6-9 handle the search we discussed initially. Here we recursive search for our target value to remove.

The last `if` block handles the case with 2 children. We find the smallest node in the right subtree and assign it's value to the current node. Then we erase this value from the right subtree of the current node.

The final block handles the leaf and the one child cases. This is the only case where a node is actually removed from the tree. This block will also untilately get called when the case handling two child nodes needs to delete the smallest value from the right subtree.

More to Explore

- The content on this page was adapted from Binary Search Trees, by Steven J. Zeil for his data structures course CS361.
- MyCodeSchool video: Data structures: binary search trees
- Wikipedia
  - binary search tree
- Binary tree visualizer

3.15.3 Binary Search Tree iterators

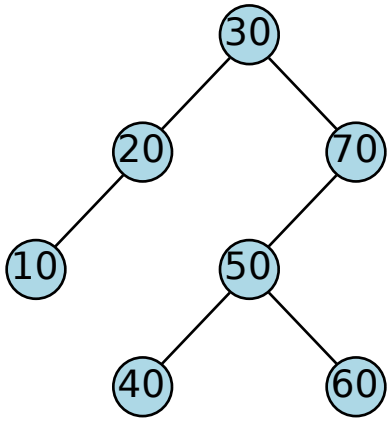
The recursive traversal algorithms work well for implementing tree-based ADT member functions, but if we are trying to hide the trees inside some ADT for example, using binary search trees to implement std::set or using STL algorithms or range-for loops, then we need to provide iterators for walking though the contents of the tree.

Iterators for tree-based data structures can be more complicated than those for linear structures.

For arrays (and vectors and deques and other array-like structures) and linked lists, a single pointer can implement an iterator.

Given the current position, it is easy to move forward to the next element.

For anything but a singly-linked list, we can also easily move backwards.



But look at this binary search tree, and suppose that you were implementing tree iterators as a single pointer. Let's see if we can "think" our way through the process of traversing this tree one step at a time, without needing to keep a whole stack of unfinished recursive calls around.

We're going to try to visit the nodes in the same order we would process them during an "in-order" traversal. For a BST, in-order traversal means that we will visit the data in ascending order.

It's not immediately obvious what our data structure for storing the "current position" (i.e., an iterator) will be. We might suspect that a pointer to a tree node will be part of that data structure, because that worked with iterators over linked lists.

BST iterator `begin()` and `end()`

As in any data structure, `begin` and `end` refer to the first element in the data structure and one past the last element. In the last section, we said that a BST is sorted when a level order traversal is used.

So what algorithm should we use to find the beginning?

Show

Start from the root and working our way down, always taking left children, until we come to a node with no left child.

The left-most child of a BST is always the minimum element.

So what algorithm should we use to find the end?

Show

Just return the `nullptr`.

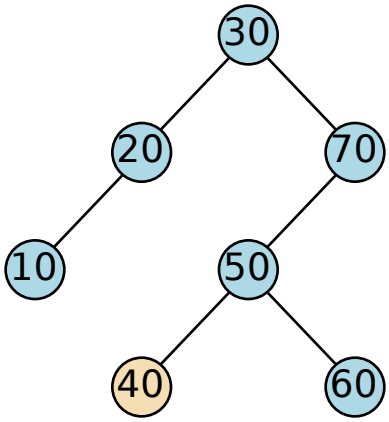
It's tempting to guess that you could do much the same as for `begin()`, this time seeking out the right-most node. But that would leave you pointing to the **last** node in the tree, and `end()`, must always refer to the position after the last element in the container.

BST iterator `operator++()`

A quick review of the definition of iterators and the iterator design pattern. We have a few facts to deal with:

- A tree is a *hierarchical* data structure
- An *iterator* allows users to visit each element in a container *sequentially* - with no awareness of the underlying structure.
- In C++, iterators are implemented using pointer semantics. The function `operator++()` is used to move to the next element.

Given our familiar tree:



If we are iterating through our tree and are currently at the node with value `40`, then how do we get to the next node?

Show

Well, we know that we should wind up at `50`. But how can we get there?

We can't, not with just a pointer to the node and all the nodes pointing only to their children.

The only place you can go within this tree is down, and there is no "down" from our current position.

In a binary tree, to get to the next node, we need to know not only where we are, but also how we got here.

One way is to do that is to implement the iterator as a stack of pointers containing the path to the current node. The stack would be used to simulate the activation stack during a recursive traversal.

But this solution is clumsy and inefficient. Iterators tend to get assigned (copied) a lot, and we'd really like that to be a constant time - an O(1) operation. Having to copy an entire stack of pointers just isn't very attractive.

BST iterator using parent pointers

We can make the task of creating tree iterators much easier if we redesign the tree nodes to add pointers from each node to its parent.

```
// a binary tree node
template<class T>
  struct tree_node {
    T value;
    tree_node<T>* left;
    tree_node<T>* right;
    tree_node<T>* parent; // link to parent simplifies iterators
    tree_node(const T& value = T{},
        tree_node<T>* left = nullptr,
        tree_node<T>* right = nullptr,
        tree_node<T>* parent = nullptr)
      : value{value}
    , left{left}
    , right{right}
    , parent{parent}
    { }
  };
```

These nodes are then used to implement a tree class, which, as usual, keeps track of the root of our tree in a data member.

The outline for a tree iterator is similar to what we have covered before:

```
template <typename T>
  struct tree_iterator {
    typedef T value_type;
    typedef T* pointer;
    typedef T& reference;
    typedef std::ptrdiff_t difference_type;
    typedef std::bidirectional_iterator_tag iterator_category;

    const tree::tree_node<T>* node;
    tree_iterator() = default;
    tree_iterator(const tree::tree_node<T>* n);

    constexpr
       const T& operator*() const noexcept;
    tree_iterator& operator++();
    tree_iterator operator++(int);
    tree_iterator& operator--();
    tree_iterator operator--(int);
 };
```

There is a subtlety when using our tree iterator in a BST.

```
template<class T>
 class bstree {
   public:
     typedef T value_type;
     typedef const tree_iterator<T> const_iterator;
```

(continues on next page)

(continued from previous page)

```
    typedef const_iterator iterator;
    typedef const_iterator reverse_iterator;
    typedef const_iterator const_reverse_iterator;

    // remainder omitted . . .
};
```

Note the type of all the iterators is `const`. We only want `const` behavior for this ADT. If we provided a "true" non-const iterator, it would allow reassigning data in the tree:

```
bstree<int>::iterator it = myTree.find(50);
*it = 10000;
```

which would very likely break the internal ordering of data, violating the binary search tree property, and making it useless for any future searches. A `const` iterator allows us to look at data in the container, but not change that data.

Implementing BST iterators

As discussed earlier, `begin()` is implemented by finding the minimum element in the tree.

A free function that works with the `tree_node` struct is enough:

```
template <class T>
  tree_node<T>* min_element(tree_node<T>* root )
  {
    if(root == nullptr || root->left == nullptr) {
      return root;
    }
    return min_element(root->left);
  }
```

`bstree::begin()` can use this function directly:

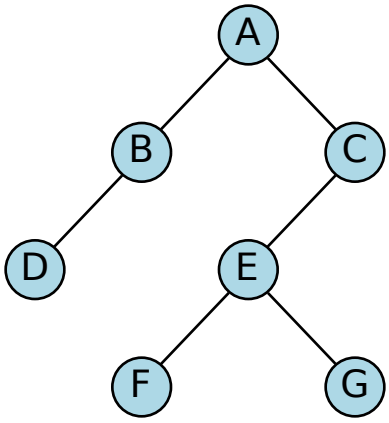
```
constexpr
  const_iterator begin() const noexcept {
    return const_iterator(min_element(root));
  }
```

And `end()` uses the null pointer.

```
constexpr
  const_iterator end() const noexcept {
    return const_iterator(nullptr);
  }
```

Implementing `operator++()`

Before implementing `operator++`, let's think about what is should do. Given the following tree:



(Not a binary search tree, just a tree).

Question: Suppose that we are currently at node E. What is the in-order successor of E? That is, the node that comes next during an in-order traversal of E?

Show

G is the in-order successor of E.

If you answered F, remember that in an in-order traversal, we visit a node only after visiting all of its left descendents and before visiting any of its right descendents. Since we're at E, we must have already visited F.

That example suggests that a node's in-order successor tends to be among its right descendants.

If our previous premise is correct, then what is the in-order successor to A?

Show

F is the in-order successor of A.

If we are at A during an in-order traversal, then we have already visited all of A's left descendents. So the answer has to be C or one of its descendents. It's tempting to pick C because it's only one step away from A.

But, remember, during an in-order traversal, we visit a node only after visiting all of its left descendents and before visiting any of its right descendents.

We have not yet visited C's left descendants. So have to run down from C to the left as far as we can go.

This suggests that, if a node has any right descendants, we should:

- Take a step down to the right, then
- Run as far down to the left as we can.

You can see how this would take us from A to F. The same approach would take us from E to G as well. So both of our prior examples are satisfied.

But that "step to the right, then run left" procedure raises a new question. What happens if we are at a node with no right descendants?

Question: Suppose that we are currently at node C. What is the in-order successor of C?

Show

C does not *have* an in-order successor. C is actually the final node in an in-order traversal. After C is only `end()`.

While node C is an interesting special case, it doesn't make clear what should happen in the more general case where we have no right child.

Question: What is the in-order successor of F?

Show

E is the in-order successor of F.

So, when we have no right child, we may need to move back up in the tree.

Question: What is the in-order successor of G?

Show

C is the in-order successor of G.

Why did we move up two steps in the tree this time, when from F we only moved up one step? The answer lies in whether we moved back up over a left-child edge or a right-child edge.

If we move up over a right-child edge, we're returning to a node that has already had all of its descendants, left and right, visited. So we must have already visited this node as well, otherwise we would never have made it into its right descendants.

If we move up over a left-child edge, then we're returning to a node that has already had all of its left descendants visited but none of its right descendants. That's the definition of when we want to visit a node during an in-order traversal, so it's time to visit this node.

So, if a node has no right child, we move up in the tree (following the parent pointers) until we move back over a left edge. Then we stop.

When applying this procedure to C, we move up to A (right edge), then try to move up again to A's parent. But since A is the tree root, it's parent pointer will be null, which is our signal that C has no in-order successor.

To summarize:

- If the current node has a non-null right child,
  - Take a step down to the right
  - Then run down to the left as far as possible
- If the current node has a null right child,
  - Move up the tree until we have moved over a left child link

operator++

Putting it all together.

```
tree_iterator& operator++() {
  if (node == nullptr) {
    return * this;
  }
  if (node->right != nullptr) {
     // find the smallest node on the right subtree
     node = mesa::tree::min_element(node->right);
  } else {
     // finished with right subtree and there is no right
     // search up for first parent with a non-null right child
     // or nullptr,
     auto parent = node->parent;
     while (parent != nullptr && node == parent->right) {
       node = parent;
       parent = parent->parent;
     }
     node = parent;
  }
  return * this;
}
```

One part of this iterator that needs closer inspection is the `while` loop. This loop continues moving upwards in the tree until it finds:

- A `node` where the current node is in the left subtree of its parent, or
- The root of the tree (`parent == nullptr`), indicating that there is no in-order successor (end of the traversal).

Key Conditions

- `parent != nullptr`: Ensures we do not dereference a nullptr when accessing `parent->right`.
- `node == parent->right`: Asserts the current node is the right child of its parent. When true, we need to keep moving upwards, as the in-order successor is not in this part of the tree.

The loop climbs up the tree until it finds a node for which the current traversal has completed its right subtree. This ensures that the in-order successor is correctly identified.

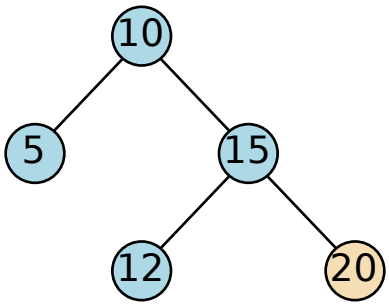
Consider an alternative approach that simply checks the immediate predecessor:

```
if (parent != nullptr && node == parent->right) {
    node = parent;
    parent = parent->parent;
}
```

The issue with this approach is that it would only handle one level of traversal upwards. If the current node is deeply nested in a right subtree, it may skip multiple ancestors that need to be visited to find the in-order successor. This can cause incorrect results when:

- The node is the rightmost descendant of a deep subtree, requiring multiple upward traversals to reach the root or an ancestor in a left subtree.
- Randomly generated trees or trees that have had many insertions and removals from the tree may have imbalanced structures with chains of right children, requiring multiple iterations of the while loop.

Consider this tree:



If the iterator is at `20` (the rightmost node):

- The while loop will correctly climb from 20 --> 15 --> 10, and stop when `parent == nullptr`.
- A single conditional would move only one step (to 15) and fail to reach the root, leaving the iterator in an inconsistent state.

3.15.4 Using parent pointers

Using parent pointers does incur additional overhead. We must store an additional pointer with every tree node. It also means the functions used to manage the tree need to change.

Originally, inserting a node looked like this:

```
tree::tree_node<T>*
 insert (const T& value,
         tree::tree_node<T>*& node)
 {
   // add a new leaf
   if(node == nullptr) {
     node = new tree::tree_node<T>(value, nullptr, nullptr);
     return node;
   }
   if(value < node->value) {
     return insert(value, node->left);
   }
   if(node->value < value) {
     return insert(value, node->right);
   }
   // else the value already exists in the tree
   node->value = value;
   return node;
 }
```

But now when inserting a new node, we also need to maintain correct parent relationships.

```
tree::tree_node<T>*
  insert (const T& value,
          tree::tree_node<T>*& node,
          tree::tree_node<T>* parent)
  {
    // add a new leaf
    if(node == nullptr) {
      node = new tree::tree_node<T>(value, nullptr, nullptr, parent);
      return node;
    }
    if(value < node->value) {
      return insert(value, node->left, node);
    }
    if(node->value < value) {
      return insert(value, node->right, node);
    }
    // else the value already exists in the tree
    node->value = value;
    return node;
  }
```

When we make a new node, we need to pass the parent into the `tree_node` constructor. Even though it won't have children initially, it will have a parent.

When we make our recursive calls, the parent node passed in is the current node.

More to Explore

- The content on this page was adapted from Binary Search Trees, by Steven J. Zeil for his data structures course CS361.
- Binary tree visualizer

3.15.5 Priority queues

If we have a collection of elements that carry a "score", then how can we find and remove the element with the smallest (or largest) score? What if new elements may be added at any time?

This collection is called a *priority queue* because:

- Like std::queue, it is used to simulate objects waiting in line.
- But instead of FIFO, the processing order is determined by the object "priority" or score.

Like std::stack and std::queue, std::priority_queue is an adapter that works on an underlying sequential container, which must provide random-access iterators (vector or deque). For example:

```
priority_queue<event, vector<event>> event_queue;
```

Priority queues are often used to manage scheduling. The classic example of a priority queue is an emergency room. Patients are not served in the order they arrive, but in order of severity.

In a software application, if we wanted to simulate urban traffic, we might create a series of events, some of which spawn other future events:

- Simulated traffic light need an event for the next change. When a light changes to red, we schedule a change event to green some number of seconds later. When a light changes to green, we schedule a change-to-yellow event some time a little later, and so on.
- Parking lots and garages might be simulated as objects that, at random time intervals, toss a new car out on to the street in front of the lot or garage. The car-generation interval might vary with the time of day (e.g., at 5:00PM, when everyone is leaving work, the interval between cars leaving each garage would be reduced).
- Each moving car on the street would have an event associated with the time it needs to reach the next intersection along its path. When it reaches the intersection, we schedule a new event for the intersection after that, and so on.

```
while (simulation has not ended)
   get next event from event_queue
   trigger event
end while
```

In this example, the 'priority' value is time. Regardless of when an event is sent, we want the events to occur in their correct time order.

Although 'queue` is in the name, a queue is not a good starting point for a priority queue implementation. However, there is a data type perfectly suited to this task -- a heap. In fact, heaps are used to implement priority queues so often that the terms are often used interchangeably.

Heaps

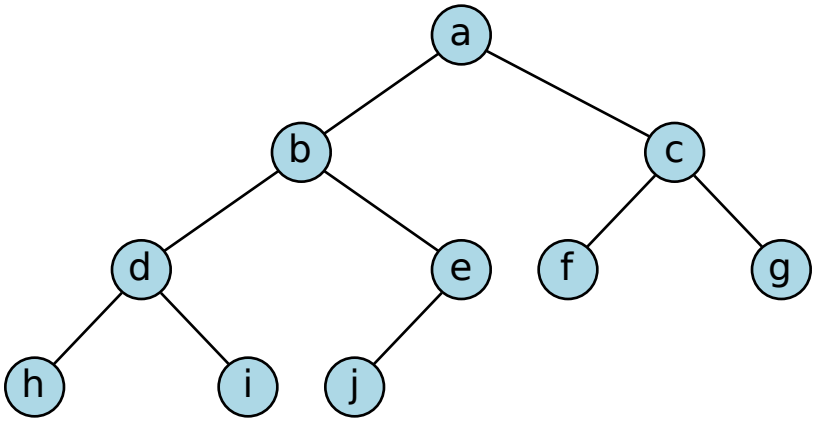
A *heap* is a special tree-like data structure. There are many types of heaps, however, in this chapter, we will discuss only *binary heaps*. In this section, we will refer to binary heaps merely as heaps. A heap may be a binary tree, but it is **not** a binary search tree. Heaps have two key attributes:

- The underlying tree must be **complete**
- The order of elements must obey the *heap property*

  For a *min heap* a parent element must be <= all its children.

  For a *max heap* a parent element must be >= all its children.

One of the side-effects of heap is that the minimum (or maximum) value can always be found at the tree root.



A min heap

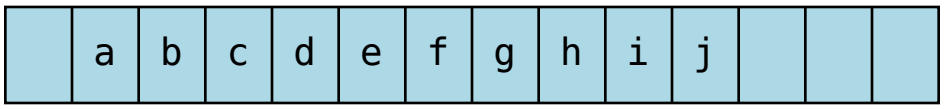
We can show that a complete binary tree of height h has between $2h$ and $2(h+1)-1$ nodes. This implies that the height of a complete binary tree is $\lfloor \log N \rfloor$, which results in $O(\log N)$ performance.

Although logically a heap is a tree-like data structure, because the tree must be complete, it fits easily into an array or vector. It is very common to use an array for the *physical implementation* of a heap, since it is much more efficient than a general purpose tree bound together with pointers. For a tree of height $2^h - 1$:

- The parent of a node *i* is located at index $\lfloor \frac{i}{2} \rfloor$
- The left child of a node *i* is located at index $2i$
- The right child of a node *i* is located at index $2i + 1$

To get our math to work out nicely, we place the root node at index position 1. This storage location won't go to waste -- it is used when we remove nodes from the tree and maintaining the heap property is required.

The array representation of the min heap is:



Array implementation of a min heap

Depending on the implementation, the backing store may or may not have extra storage.

The heap interface can be implemented as follows:

```
template <class T,
       class Container = std::vector<T>
       class Compare = std::greater<typename Container::value_type>>
   // require T is comparable
   class binary_heap
   {
     public:
       using value_type = T;
       using value_compare = Compare;
       static_assert((std::is_same<T, Container::value_type>::value),
         "heap type must match underlying container value type" );

       binary_heap() = default;

       // Construct a heap from an unsorted container
       explicit binary_heap(const Container& items);
       explicit binary_heap(std::initializer_list<T> list)

       constexpr void     clear() noexcept;
       constexpr bool     empty() const noexcept;
       constexpr bool      full() const noexcept;
       constexpr size_t    size() const noexcept;
       constexpr const T& front() const noexcept;

       void pop();
       void push (const T& value) noexcept;

     private:
       size_t size_ = 0;
       Container heap_ = {T{}};

       void percolate_down(size_t hole) noexcept;
       void build_heap() noexcept;
       void percolate_up(const T& value) noexcept;

   };
```

The defining operations of a heap are:

front
: Peek at the heap root element.

pop
: Remove a value while maintaining the heap property.

  Calls `percolate_down` to perform the work.

Constructors
: Creates a new underlying container of the container adaptor from a variety of data sources.

  Calls `build_heap` to ensure the heap property satisfied when construction is complete.

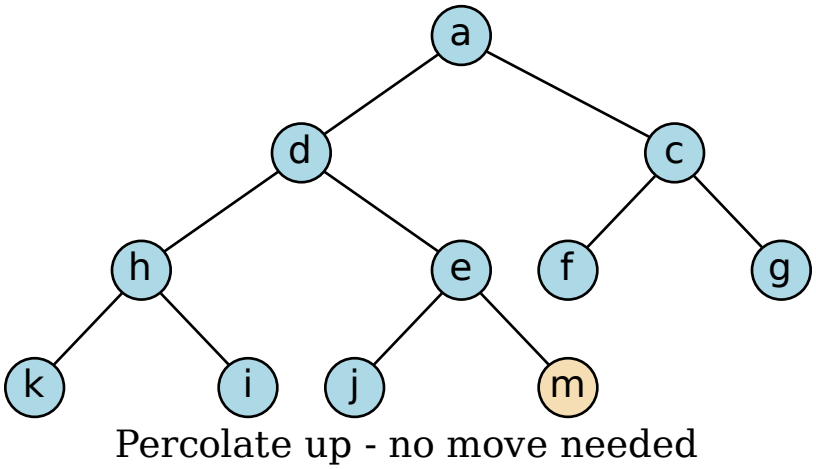
push
: Add a new value to the heap, while maintaining the heap property.

  Calls `percolate_up` to perform the work.

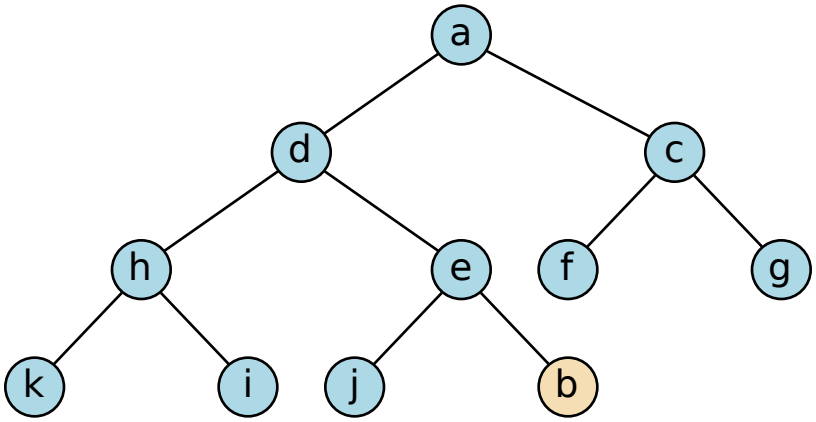
In this implementation, any container that implements `front()`, `push_back()`, and `pop_back()` are candidates for the backing store. This example uses vector by default. The Compare class allows the same class to function as either a min heap (the default), or another comparison function. Using less would transform the heap into a max heap.

Percolate up

When a new node is added to the heap, we initially use `push_back` to append the new value to the 'last' open position in the tree (the hole). At this point the tree is still complete, but the new value is not in the correct position (except through some random stroke of luck). So after the initial `push_back` the heap property is violated and must be restored.

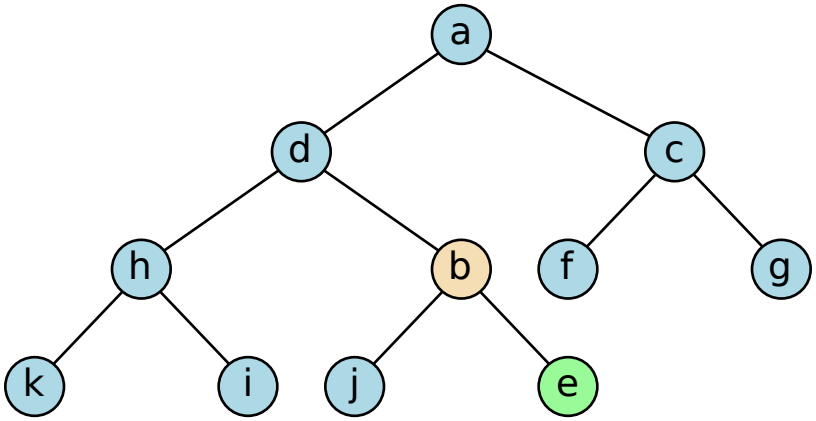


If however, the new value is less then its parent, it must be moved into a valid position.



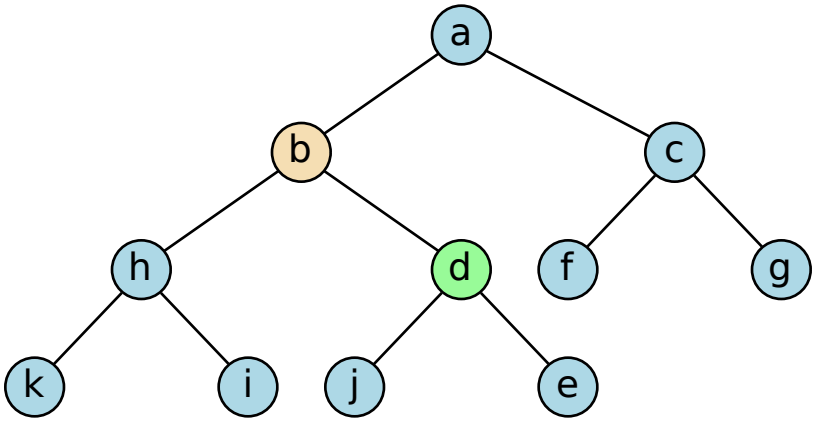
Percolate up - move needed

First we swap the value at position 'b' with the value at position 'e':



Percolate up - first move

The value at position 'd' is still larger then 'b', so we are not done:



Percolate up - last move

Now that 'a' is less than 'b' and all of the children of 'b' are greater than 'b', the heap property has been restored and we are done.

The `percolate_up` function does all the hard work. In truth, it is a fairly short function. The algorithm outline is:

```
percolate_up (value)
   heap[0] <- value
   hole <- last position in heap
   while (value < parent_of_hole)
      move parent value to hole
      set hole to parent position
   done while
   move heap[0] to heap[hole]
done percolate_up
```

The actual implementation is a lab exercise.

To implement `push` using `percolate_up`, we:

- Increase backing store capacity, if needed.
- Push the value onto the end of the tree.
- Increase the tree size by 1.
- Percolate up from the last node in the tree.

Percolate down

When we return a value from the top of the heap, we return the root node value. We now have a hole that needs to be filled. One approach is to move the last node in the heap to the root position and then 'percolate_down' to push the value to its proper location in the tree. This has a few of advantages;

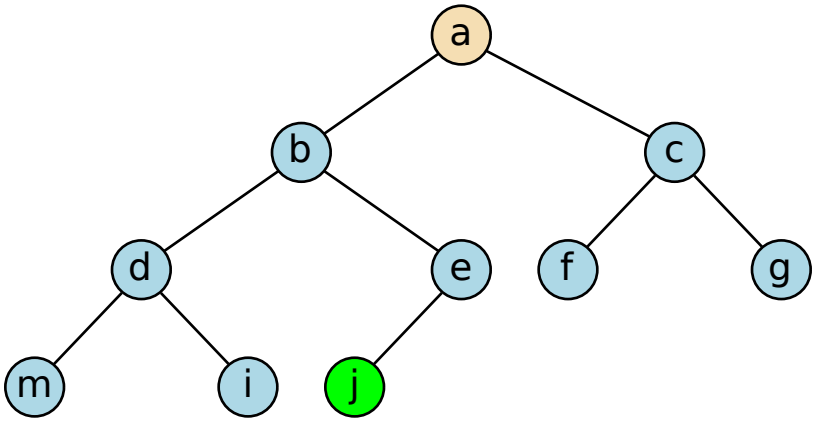
- It maintains the completeness property of our tree

- It is relatively straightforward to implement. The same algorithm can be used independent of the tree structure or any node value.

Since this is a little more complicated than `percolate_up`, the entire function is shown. The main complication is that the current node might have 0 children, 1 child, or 2 children, so we need to be careful that we don't try to access the value of non-existent children.

```
void percolate_down(size_t hole) noexcept
{
  T tmp = std::move(heap_[hole]);

  for (size_t child; hole*2 <= size_; hole = child) {
    child = hole*2;
    if (child != size_ && heap_[child+1] < heap_[child]) {
      ++child;
    }
    if (heap_[child] < tmp) {
      heap_[hole] = std::move(heap_[child]);
    } else {
      break;
    }
  }
  heap_[hole] = std::move(tmp);
}
```

A walk-through follows. When we pop 'a' from the heap, we leave a space that must be filled while maintaining the heap property.

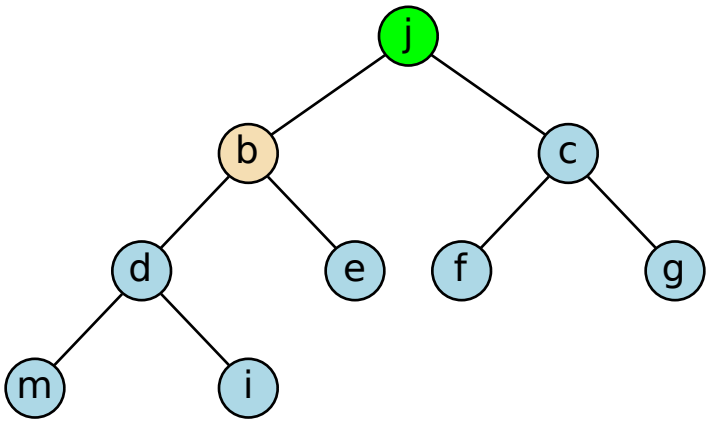


Pop from heap - 'a' is about to become a hole

Grab the last value in the tree, which may or may not be the largest value. Move this value into the root position. Since the largest value is in the root position, the heap property is no longer valid. We have to restore the heap property by pushing this value down until the heap property is restored.

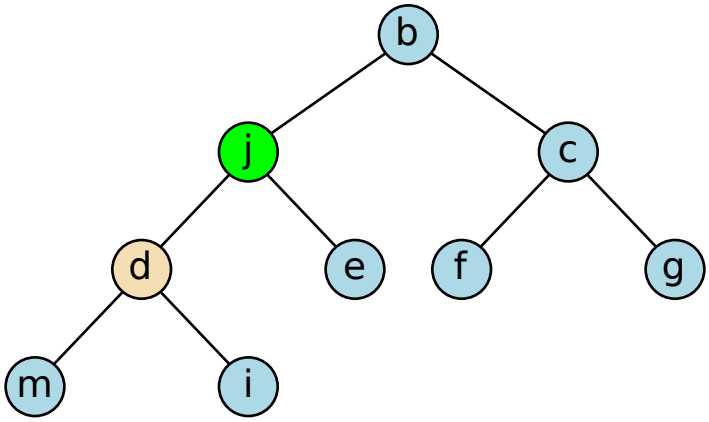
We can achieve this by continually exchanging the smallest child value with the current value until the heap property

is restored.



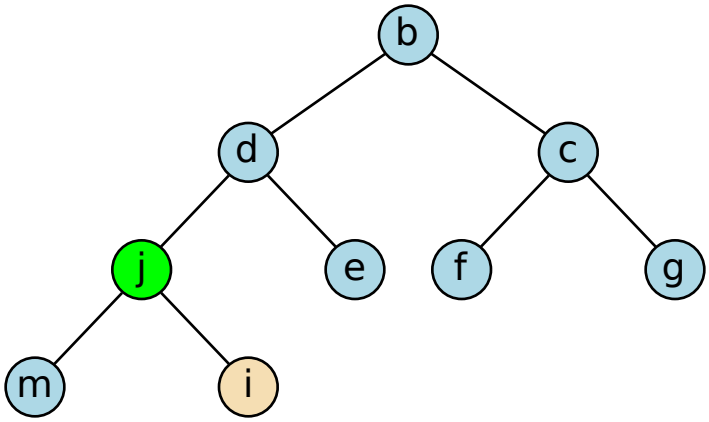
percolate down - first move

The 'j' is still larger than 'd', so again, we exchange the two values.



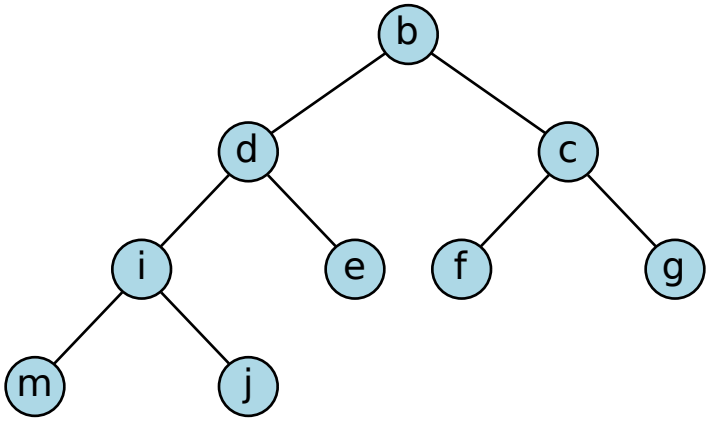
percolate down - second move

When we check the left node, the value 'j' is less then 'm'. We can't stop at this point, because this node has another child. The 'j' is still larger than 'i', so we exchange values, however this time we traverse the right sub tree.



percolate down - third move

At this point, the node 'j' has no children, so we are done.



percolate down complete

To implement `pop` using `percolate_down`, we:

- Move the last tree node to the root.
- Reduce the tree size by 1
- Percolate down from the root node.

Build heap

Frequently we want to create a binary heap from an existing collection which could be the template type `Container` or an `initializer_list<T>` in our example heap. These constructors take N arbitrarily items and transforms them into a heap. We could achieve this with N successive inserts. Each insert will take $O(1)$ average and $O(logN)$ worst-case time, the total running time of this algorithm would be $O(N)$ average but $O(NlogN)$ worst-case. Since this is a special instruction and there are no other operations intervening, and we already know that the instruction can be performed in linear average time, it is reasonable to expect that with reasonable care a linear time bound can be guaranteed.

The general algorithm is to place all the items into the tree in any order, maintaining the structure property. Then, if percolate_down(i) percolates down from node i, the buildHeap routine in Figure 6.14 can be used by the constructor to create a heap-ordered tree.

To implement `build_heap` using `percolate_down`, we:

- Copy all items from the source container or range into the heap backing store in any order. As long no uninitialized values are present in the range $[begin(), end())$, then the heap structure is maintained and only the heap order property needs work.
- Call `percolate_down` starting from the heap midpoint and iterate down to the root node.

The actual implementation of `build_heap` is a lab assignment.

More to Explore

- The content on this page was adapted from Binary Search Trees, by Steven J. Zeil for his data structures course CS361.
- std::priority_queue
- Heap data structures

3.15.6 The set class

A *set* refers to any data structure in which every member of the set is unique. The integers define a set, because every number is unique. The values `{3, 1, 4, 1, 5, 9}` do **not** define a proper set, because the value `1` is repeated.

In C++, a std::set must also be sorted. Like `std::vector`, a `set` is a generic class and declarations must include the object *type* stored in the class:

```
#include <set>

std::set<int> x {2,7,1,8,2,8,1,8,2,8,4,5,9};
```

What will be stored in `x` after initialization?

Show

The two defining characteristics of a `set` are:

- A `set` is sorted.
- A `set` may contain only unique values.

Defining a set with non-unique values is not an error. The new object simply replaces the old.

When initialized, `x` will contain: `1 2 4 5 7 8 9`

Like the sequence containers, each element in a set can be visited one at a time using a range-for loop.

```
#include <iostream>
#include <set>

int main()
{
  std::set<int> x {2,7,1,8,4,5,9};

  for (const auto& i: x) {
    std::cout << i << ' ';
  }

  return 0;
}
```

Because `set` does not provide `operator[]`, traditional for loops using an index are not possible:

```
std::set<int> x {2,7,1,8,4,5,9};

for (int i=0; i < x.size(); ++i) {  // OK
    std::cout << x[i] << ' ';         // compile error
}
```

Sets of any type can be created as long as the type is *comparable*. The comparison operator (*comparator*) used in sets by default is `operator <`. Any type used in a set should overload `operator <`. All of the types are *comparable*.

Use set::insert to add a new element to a `set` or replace an existing element:

```
std::set<int> x {2,7,1,8,4,5,9};
x.insert(6);
```

Because a `set` is not an indexed container, every 'get' is a search:

```
std::set<int> x {2,7,1,8,4,5,9};
auto it = x.find(8);
```

The set::find function returns an *iterator* to the element with a specific key:

```
std::set<int> x {2,7,1,8,4,5,9};
auto it = x.find(8);
std::cout << *it;         // print the value returned from find()
```

The set::erase function is used to remove an element from a `set`. `set::erase` takes an iterator as the position in the `set` to remove:

```
std::set<int> x {2,7,1,8,4,5,9};
auto it = x.find(8);
if (it != x.end()) {
    x.erase(it);
}

it = x.find(8);
assert ( it == x.end() );  // this should be true
```

Variations on std::set

The STL provides 3 alternate forms of `std::set` class:

multiset
: A `set` in which duplicate keys are allowed.

unordered_set
: A `set` of unique objects stored based on the object *hash function*. Added in C++11.

  In order to use a type in an unordered container, the type must override 3 functions:

  - override std::hash
  - override `operator==`
  - override `operator<`

  before the type will compile when added to an unordered container.

unordered_multiset
: An `unordered_set` in which duplicate keys are allowed.

More to Explore

- STL containers library
- Visualgo: binary heap

3.15.7 The map class

A *map* refers to any data structure that 'maps' *keys* to values. The `map` class is arguably the most popular container in the STL after `vector`.

All the containers discussed so far focused on storing 1 thing. That is, each stores values of a single type. Maps add a new wrinkle. A `map` stores pairs of things. Traditionally, the pair stored is referred to as a *key-value pair*.[1]

Nearly every programming language provides some kind of `map` implementation. Some languages use the terms *associative array* or *dictionary List*, but structurally, they are very similar.

Values are retrieved from a `map` using the **key**. Each *key* must be unique. In other words, keys are members of a `set`. Like a std::set, adding a second node with the same key replaces the old value. Unlike a `std::set`, a std::map provides the map::operator[].

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

int main() {
  std::map<std::string, int> name_counts {{"Alice", 27},
                                          {"Bob", 3},
                                          {"Clara", 1}};

  for (const auto& kvp : name_counts) {
    std::cout << kvp.first << ": "
```

(continues on next page)

[1] Sometimes this is abbreviated as 'KVP'. On cppreference.com you'll see it shortened even further to just P

(continued from previous page)

```
              << kvp.second << '\n';
  }
  name_counts["Bob"]   = 42;      // update existing value
  name_counts["Darla"] = 9;       // insert a new value

  // get map values
  std::cout << "Alice is " << name_counts.find("Alice")->second << '\n';
  // or get the key iterator, then print
  auto it = name_counts.find("Alice");
  std::cout << "Alice is " << it->second << '\n';

  std::cout << "Bob is " << name_counts.at("Bob") << '\n';
  std::cout << "Darla is " << name_counts["Darla"] << '\n';
}
```

Selected map functions

Access and assignment
operator=, at and operator[]

Capacity
empty, size, and max_size

Modifiers
clear, emplace, insert, erase, swap

Lookup
count, find, equal_range, upper_bound and lower_bound

For large data sets, the lookup functions in a `map` are faster than their counterparts in a sequential container such as `vector`.

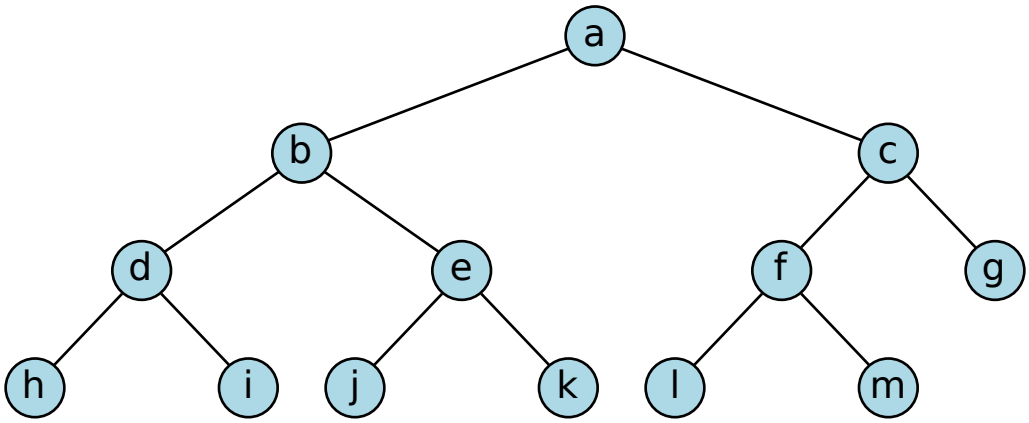
Note

There is no `push_back()` for a map.

The `map` decides where elements go, not you. All access requires either knowing the key or having an iterator.

Map structure

Internally, a `map` is a sorted *complete binary tree*. (Technically it is often implemented as a Red-black tree). Each node in the tree is a std::pair.



A complete binary tree

All *nodes* are sorted by their *keys*. Sorting is managed using `operator<` by default, but this can be configured in the map constructor using a custom compare function or class, just as with a `set`.

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

using std::string;

void print (std::set<string> keys) {
  for (const auto& key: keys) {
    std::cout << key << ' ';
  }
}

int main() {
  std::map<string, int> inventory {
    {"apple", 12},
    {"kiwi", 4},
```

(continues on next page)

(continued from previous page)

```
    {"lemon", 1},
    {"pear", 4},
    {"peach", 4},
    {"grape", 100},
    {"cocoa", 3},
  };

  std::set<string> inventory_keys;

  // extract keys from the inventory map
  for (const auto& i: inventory) {
    auto result = inventory_keys.insert(i.first);
    if (!result.second) std::cout << "no insertion\n";
  }

  std::cout << "All fruit keys:\n";
  print (inventory_keys);

  std::set<string> keys;
  auto it = inventory.upper_bound("kiwi");
  while(it != inventory.end()) {
    auto result = keys.insert(it->first);
    if (!result.second) std::cout << "no insertion\n";
    ++it;
  }
  std::cout << "\n\nAll fruit keys greater than 'kiwi':\n";
  print (keys);

}
```

Using a customer comparator, we can store map items in reverse order:

```
#include <functional>  // provides std::greater
#include <iostream>
#include <map>
#include <string>

using std::string;

// print inventories with different custom comparators
template <class Comparator>
void print (const string title, const std::map<string, int, Comparator>& x) {
   std::cout << title;
   for (const auto& kvp: x) {
     std::cout << kvp.first << ", " << kvp.second << '\n';
   }
}

int main() {
  std::map<string, int> inventory {
     {"apple", 12},
     {"kiwi", 4},
```

(continues on next page)

(continued from previous page)

```
    {"lemon", 1},
    {"pear", 4},
    {"peach", 4},
    {"grape", 100},
    {"cocoa", 3},
  };

  print ("Initial inventory:\n", inventory);


  // define a reverse ordered map
  // a lambda is not the best choice here
  const auto greater_than = [] (string lhs, string rhs) { return lhs > rhs;};
  std::map<string, int, decltype(greater_than)> reverse_inventory1 (greater_than);

  // but it works
  for (auto& i: inventory) {
    reverse_inventory1.insert(i);
  }
  print ("\n\nReverse inventory using lambda:\n", reverse_inventory1);


  // STL provides many basic operations wrapped in a std::function
  std::map<string, int, std::greater<string>> reverse_inventory2;
  for (auto& i: inventory) {
    reverse_inventory2.insert(i);
  }
  print ("\nReverse inventory using std::greater:\n", reverse_inventory2);

  return 0;
}
```

Variations on std::map

The STL provides 3 alternate forms of map class:

multimap
: A `map` in which duplicate keys are allowed.

unordered_map
: A `map` of unique *key-value pairs* stored based on the *key* object *hash function*. Added in C++11.

unordered_multimap
: An `unordered_map` in which duplicate keys are allowed.

More to Explore

- STL containers library
- Red-black tree on Wikipedia

Footnotes

3.16 Hash Tables

This section focuses on alternatives to tree-like associative data structures -- hash tables.

3.16.1 Hashing concepts

Previously, we asserted that a *map* refers to any data structure that 'maps' *keys* to values. They have an advantage over sequential containers in that the cost of searches grows more slowly: $log_2 N$ for maps versus $\frac{N}{2}$ for the sequential containers.

Is there a way to access elements in a tree that does not become more costly as the number of elements grows? We need to:

- Store data in some kind of *indexable data structure*: something we can access using an index, such as a `vector`.
- Compute a value that will result in the index where are data is stored.

This is exactly what the unordered containers do.

The unordered containers all depend on *hashing* to find elements. *Hashing* is a search method that uses a *hash function* to convert a *value* into a *position* within a *hash table*.

Hash tables trade off space for speed, sometimes achieving an average case of $O(1)$ for both search and insert times.

Often the *backing storage* for a hash table is an array. Each indexed location within the array is called a *bucket*.

Generally we want a function that generates values that avoid this kind of clumping and **then** take the modulus to insert the value into whatever hash table size we happen to be working with. So hashing is a two step process:

```
size_t hash  = function(value);
size_t index = hash % array_size;
```

A simple hash function for integers could simply to take the value `% 10`. The results are shown below:

0	1	2	3	4	5	6	7	8	9
60	11	312	23	14	35	26	17	268	799

Simple hash table modulo 10

The data stored in a hash table does not need to be a numeric value. Any function capable of calculating an index position from the data in a data type satisfies the requirements for a hash function.

Suppose, for example, that we were writing an application to work with calendar dates and wanted to quickly be able to translate the names of days of the work week (excluding the weekend) into numbers indicating how far into the week the day is:

Key	Value
Monday	1
Tuesday	2
Wednesday	3
Thursday	4
Friday	5

If we don't care about the unused space, then we could implement our hash function like so:

```
unsigned hash(const std::string& dayName)
{
    return unsigned(dayName[1] - 'a');
}
```

because each of those seven strings has a distinct second character.

So we can set up the table:

```
std::array<string, 5> days = {"Monday", "Tuesday",
                              "Wednesday", "Thursday", "Friday"};
int table[96];
for (int i = 0; i < 5; ++i) {
  table[hash(days[i])] = i+1;
}
```

Something to Consider

Why is the days table size 96?

and then afterwards, we can look up those day names in $O(1)$ time:

```
int dayOfWeek (const string& dayName)
{
  return table[hash(dayName)];
}
```

When we are done we have created a *perfact hash table*. A perfect hash table:

1. Computes values quickly
2. Returns values in the range of the hash table size
3. Returns a unique value for each key.

Perfect hash functions are usually only possible if we know all the keys in advance, which rules out their use in most practical circumstances.

There are some applications where perfect hash functions are possible. For example, most programming languages have a number of reserved words such as "if" or "while", but for any given language the set of reserved words is fixed. Programmers who are writing a compiler for that language may use a perfect hash function over the language keywords to quickly recognize when a word read from the source code file is a reserved word.

Generally we do not expect to have perfect hash functions. This means that some keys will hash to the same table location.

Two keys *collide* if they have the same hash function value.

For example, if we were to expand our days of the week code to include the weekend, then Sunday and Tuesday would collide under our current hash function because both have the same second letter. We could compensate with a more complicated hash function, perhaps one involving a pair of letters, but this could also increase the number of unused/wasted slots in the table.

Collisions are frequently unavoidable simply because we do not know in advance what all of the keys will be.

Consequently, we say that a good hash function will:

1. Computes values quickly
2. Minimizes the number of collisions

Note we also dropped the 'return values in the range of the hash table size', because this requirement is typically enforced inside the hash table code by the simple technique of taking the returned hash value modulo of the hash table size.

More to Explore

- General purpose hash function algorithms

3.16.2 Hash functions

Unless we have special knowledge about the keys, the best we can say about "minimizing the number of collisions" is that we hope that our hashing function will distribute the keys uniformly, that is, if keys are selected at random, then the probability of the next key going into any particular position in the hash table should be the same as for any other position.

This is sometimes harder to achieve in practice than we might expect.

So to recap: a good hash function:

- Is fast and easy to compute
- Distributes keys uniformly across the table.

Note

Don't get hung up on trying to find hash functions that "mean something". Most hash functions don't compute anything useful or "natural". They are simply functions chosen to satisfy our requirements (fast and uniform over the range $0 \rightarrow (table\_size - 1)$.

Integer hashes

If your integers are already in the range $0 \rightarrow (table\_size - 1)$ then there is nothing to do:

```
int hash(int i) {return i;}
```

So integer keys are easy, but you can't always take them for granted. A company once assigned an integer ID number to every employee. When the computerized system for the company payroll was created, the way the IDs were assigned when like this:

- Start with a list of all current employees, in alphabetical order by name.
- Assign the first person in the list the ID `00005`, the next person `00010`, then `00015`, and so on.

This left "gaps" in the ID number sequence that could be used later for new employees.

When a new person was hired, someone would compare the new person's name to the alphabetical list of employee names and would assign the new person a number lying somewhere in the gap between the people already in the list.

Because of this scheme, more than 3/4 of the ID numbers in the company were evenly divisible by 5.

Let's suppose some hash table with `size == 100` simply use the hash function described previously and use `%` to constrain them to the table:

```
int hash(int i) {return i % 100;}
```

There are 20 numbers divisible by 5 in the range from 0 to 99. So 3/4 of the ID numbers would hash into only 1/5 of the table positions. These numbers are not being distributed uniformly.

There is an easy fix: change the hash table size to `101`. Making the table just 1 element larger improves the distribution considerably:

keys	hash to
00005, 00010, ..., 00100	5, 10, ..., 100
00105, 00110, ..., 00200	4, 9, ..., 99
00205, 00210, ..., 00300	3, 8, ..., 98
00305, 00310, ..., 00400	2, 7, ..., 97

The lesson here: the distribution of the original key values is important.

The difference between using 100 vs using 101 is no accident. Choosing prime numbers for hash table sizes tends to increase the uniformity of key distributions.

String hashes

Hash functions for strings generally work by adding up some expression applied to each character in the string (remember that a char is just another integer type in C++).

We need to be a little careful to get an appropriate distribution. Although a char could be any of 255 different values, most strings actually contain only the 96 "printable" characters starting at ASCII value 32 (blank).

Also we often want to make sure that similar strings, likely to occur together, don't hash to the same location. So a simple hash function like this:

```
unsigned hash (const string& words)
{
   unsigned value = 0;
   for (const auto& ch: words) {
     value += ch;
   }
   return value;
}
```

doesn't work very well. Words that differ only by transposition of characters have the same value.

An improved approach is to account for the position of each character in the string.

```
unsigned hash (const string& words)
{
   unsigned value = 1;
   constexpr unsigned factor = 31;  // or any suitable prime
```

(continues on next page)

(continued from previous page)

```
  for (const auto& ch: words) {
    value = value*factor + ch;
  }
  return value;
}
```

If `value` becomes large, eventually this expression may overflow, but for unsigned types this is not a problem. Again, we are looking for a uniform distribution of values we can generate for our hash table.

Manually writing our own hash functions for builtin standard library types is not needed. The STL provides the template std::hash and a set of standard overrides for types in the standard library.

Hashing user defined types

If you define your own `struct` or `class`, you need to write your own hash function. Normally this will be a `std::hash<>` override. Consider a `struct point` and a sample hash function:

```
struct point {
  int x;
  int y;
}

namespace std {
  template <>
  struct hash<point>
  {
    std::size_t operator()(const point& p) const
    {
      return   std::hash<int>()(7919) // or any suitable prime
             + std::hash<int>()(p.x) * 73
             + std::hash<int>()(p.y) * 557;
    }
  };
}
```

The `std::hash` override must be a function template, although in this case, no template parameter is needed. The template declaration `template <>` is perfectly valid.

Note

Notice a recurring theme: prime numbers as multipliers. Prime numbers as multipliers help minimize collisions when the hash values of different parts of an object have the same value or are simple multiples of one another.

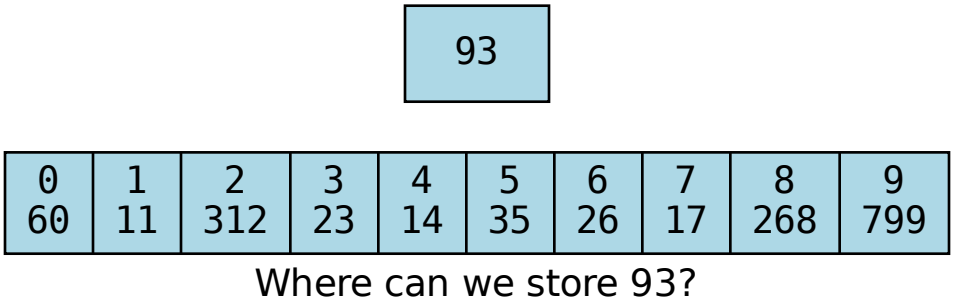
More to Explore

- General purpose hash function algorithms

3.16.3 Resolving collisions

Since we know that perfect hash functions are not generally possible except in limited cases, we must assume that sometimes a hash function will generate the same value for two different objects.

For example, in our simple hash table example, if we need to add another value that *collides* with an existing entry, then how can we store it?



There are two general approaches:

- *Open hashing*: The hash table stores data structures that each holds multiple items.
- *Closed hashing* The keys are stored directly in the table. This requires finding an open bucket in the table for each value.

Historically, one of the most common approaches to dealing with collisions has been to use fixed size *buckets*, for example an array that can hold up to k (some small constant) elements. The problem with this approach is if we get more than k collisions at the same location, then we still need to fall back to some other technique.

More to Explore

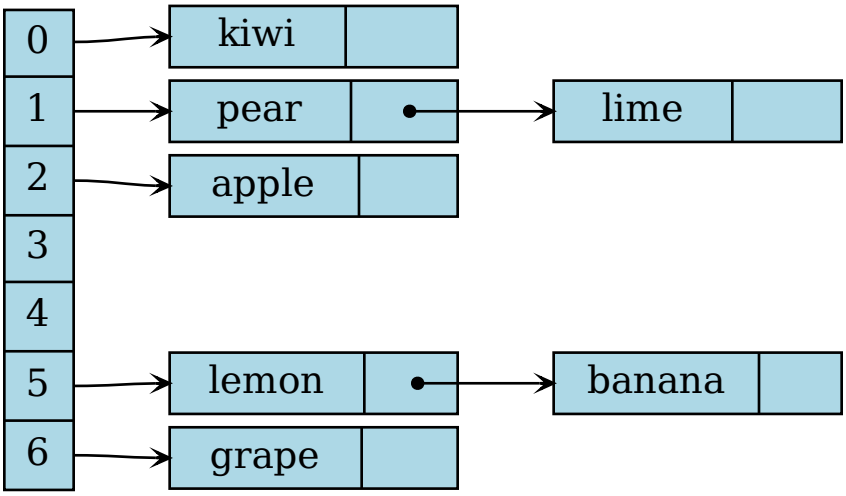
- General purpose hash function algorithms

3.16.4 Open hashing

One collision avoidance strategy is *separate chaining*. In separate chaining the hash table is implemented as an array of variable sized containers that can hold however many elements that have actually collided at each array location.

A linked list is a typical choices for implementing the chain, which is where the term "chaining" actually originates.

Example set

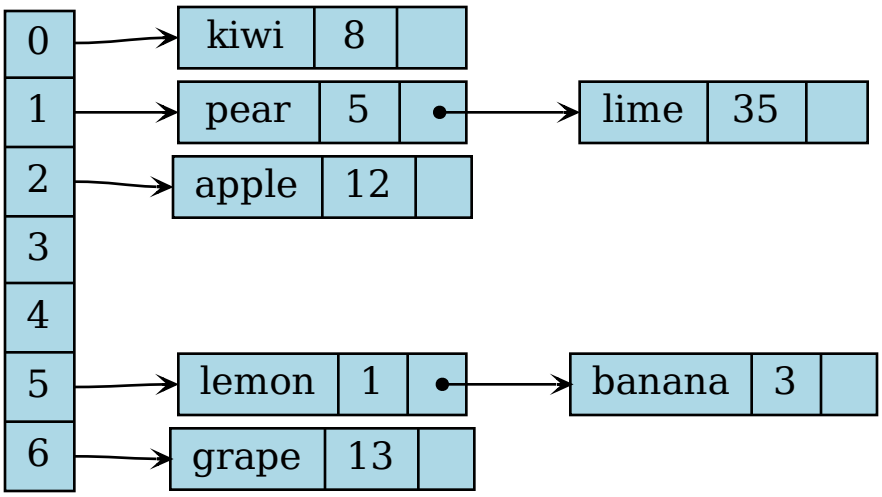


Fruit set hash table

Example map

When the ADT is a map, the process is similar. In a map ADT, the value hashed is the map *key*, since this is what uniquely identifies map items.

Each *bucket* provides access to one or more map entries (*key-value pairs*).



Fruit inventory hash table

The linked lists allows the hash table to be dynamically sized, and each array element is its own *bucket*.

A *set* provides a simple demonstration of the capabilities of a hashed data structure. Recall that set defines a container that stores unique items.

hash_set

The template variables for a hash set defines the type of data stored in the set: the `Key`. This hash table will be fixed size, so we denote that with the non-type template parameter `N`. The Comparator allows the template to accept a function used to find items in the chain. The default is equal, but another *binary predicate* can be substituted.

```
template <class Key,
          size_t N,
          class Comparator=std::equal_to<Key>>
class hash_set
{
  public:
    using Container = std::list<Key>;
    using value_type = Key;
    using key_type   = Key;
    using iterator   = typename Container::iterator;
    using const_iterator  = const iterator;

    hash_set () = default;

  private:
    std::array<Container, N> buckets;
    Comparator compare;
    int sz = 0;
```

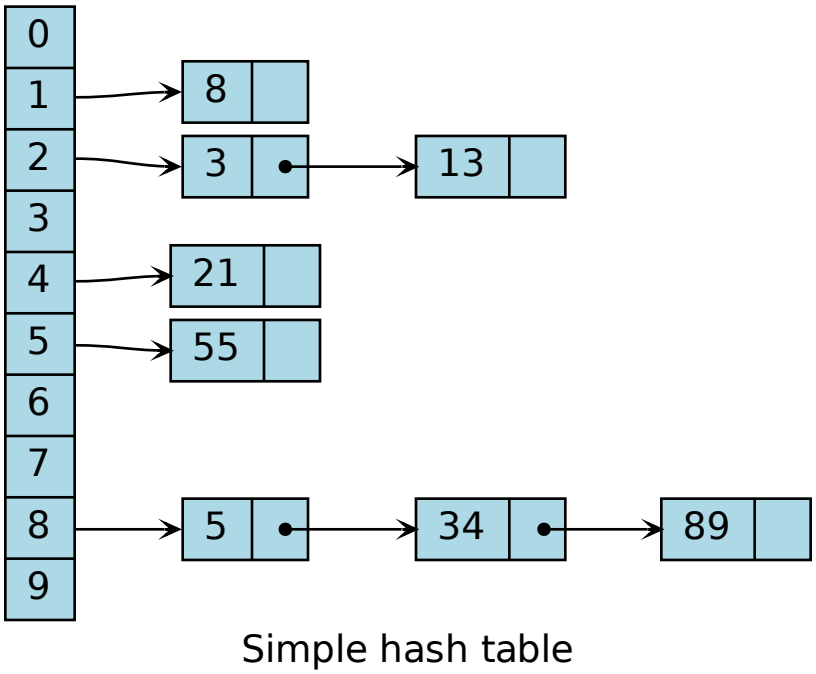
(continues on next page)

(continued from previous page)

```
};
```

find

Finding anything in a hash table using separate chaining is a two step process. Consider the following *hash table*:



Simple hash table

How does the software find the value `34` in this data structure?

First we need to find the right bucket. The `hash` override is used to compute the bucket. In this case the bucket is at index position `8`.

Note we use `std::hash<>` here. Any `Key` type stored in this set must override `std::hash`.

```
private:
   Container& find_bucket (const Key& value)
   {
     return buckets[std::hash<Key>()(value) % N];
   }
```

Next, we search through the list stored in that bucket looking for a specific value. Each element in the list stored in the bucket is evaluated using `operator==` - the default for `std::equal_to`. As soon as `operator==` evaluates to `true` the value is returned.

```
iterator find (const Key& value)
{
```

(continues on next page)

(continued from previous page)

```
  Container& b = find_bucket(value);
  return find_if(b.begin(), b.end(),
           [this, &value](Key x) {
             return compare(x, value);
           });
}
```

It should be clear that the performance of this data structure is highly dependent upon the quality of the *hash function*. Always returning 42 is a *legitimate* value for a hash, but an extremely poor one, because your *hash table* is no better than a *linked list*.

insert

Insert is similar to find. We use `find_bucket` to get the correct array element, if it exists. The we search to see if the value already exists in the linked list. If it does, replace the existing value with the new one. Otherwise, we add it to the list.

```
void insert (const Key& value)
{
  Container& b = find_bucket(value);
  iterator pos =
    find_if(b.begin(), b.end(),
        [this, &value](Key x) { return compare(x, value); });
  if (pos == b.end()) {
    b.push_back(value);
    ++sz;
  }
  else {
    *pos = value;
  }
}
```

erase

Erase is similar to insert.

1. Find the bucket
2. Search for the value
3. If you find it, erase it.

   Otherwise, do nothing.

```
void erase (const Key& value)
{
Container& b = find_bucket(value);
iterator pos =
  find_if(b.begin(), b.end(),
      [this, &value](Key x) { return compare(x, value); });
if (pos != b.end()) {
  b.erase(pos);
  --sz;
  }
}
```

Run it

```
#include <array>
#include <algorithm>
#include <cstddef>
#include <functional>
#include <iomanip>
#include <iostream>
#include <list>
#include <utility>

using std::list;
using std::array;


template <class Key,
          size_t N,
          class Comparator=std::equal_to<Key>>
class hash_set
{
  public:
    using Container = list<Key>;
    using value_type = Key;
    using key_type   = Key;
    using iterator   = typename Container::iterator;
    using const_iterator  = const iterator;

    hash_set() = default;

    iterator find (const Key& value)
    {
      Container& b = find_bucket(value);
      return find_if(b.begin(), b.end(),
            [this, &value](Key x) { return compare(x, value); });
    }

    const_iterator find (const Key& value) const
    {
      const Container& b = find_bucket(value);
      return find_if(b.begin(), b.end(),
            [this, &value](Key x) { return compare(x, value); });
    }

    int count (const Key& value) const
    {
      const Container& b = find_bucket(value);
      return (find_if(b.begin(), b.end(),
            [this, &value](Key x) { return compare(x, value); })
        == b.end()) ? 0 : 1;
    }

    void insert (const Key& value)
    {
```

(continues on next page)

(continued from previous page)

```
    Container& b = find_bucket(value);
    iterator pos =
      find_if(b.begin(), b.end(),
          [this, &value](Key x) { return compare(x, value); });
    if (pos == b.end()) {
      b.push_back(value);
      ++sz;
    }
    else {
      * pos = value;
    }
  }

  void erase (const Key& value)
  {
    Container& b = find_bucket(value);
    iterator pos =
      find_if(b.begin(), b.end(),
          [this, &value](Key x) { return compare(x, value); });
    if (pos != b.end()) {
      b.erase(pos);
      --sz;
    }
  }

  constexpr
    size_t size() const noexcept { return sz; }

  constexpr
    bool empty() const noexcept { return sz == 0; }

private:
  array<Container, N> buckets;
  Comparator compare;
  size_t sz = 0;

  Container& find_bucket (const Key& value)
  {
    return buckets[std::hash<Key>()(value) % N];
  }

  constexpr
    const Container& find_bucket (const Key& value) const
    {
      return buckets[std::hash<Key>()(value) % N];
    }

  friend
  std::ostream& operator<<(std::ostream& os, const hash_set& rhs)
  {
    os << '[';
    int i = 0;
```

(continues on next page)

(continued from previous page)

```
      for (const auto& bucket: rhs.buckets) {
        for (const auto& value: bucket) {
          os << i << ':' << value << ',';
        }
        ++i;
      }
      return os << ']';
    }
};

int main() {
  auto foo = hash_set<int, 11>{};
  foo.insert(72);
  foo.insert(72);
  std::cout << "count: " << foo.count(72) << std::endl;

  foo.erase(72);
  std::cout << "count: " << foo.count(72) << std::endl;

  foo.insert(-1);
  foo.insert(0);
  foo.insert(1);
  foo.insert(2);
  foo.insert(9);
  foo.insert(81);
  foo.insert(121);
  foo.insert(572);
  foo.insert(999);
  std::cout << foo << std::endl;
  auto it = foo.find(572);
  std::cout << "value 572: " << *it << std::endl;
}
```

More to Explore

- The content on this page was adapted from Resolving Collisions <https://www.cs.odu.edu/~zeil/cs361/f25-web/Public/collisions/index.html>, by Steven J. Zeil for his data structures course CS361.

3.16.5 Analysis of separate chaining

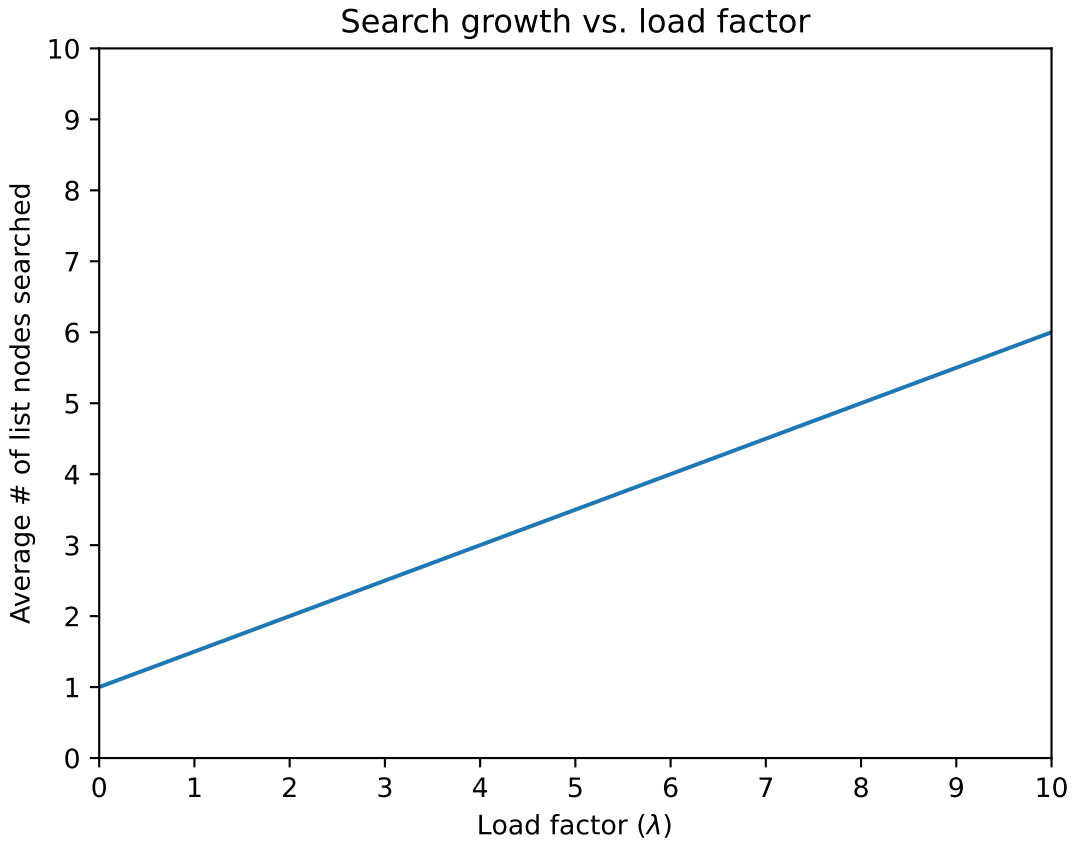
We define λ, the *load factor* of a hash table, as the number of items contained in the table divided by the table size. Given a hash table with a separate bucket for each item to be stored and a well-behaved hash function, then $\lambda = 1.0$ and the length of each list to also 1.

The effort required to perform a search is the constant time required to evaluate the hash function plus the time to traverse the list. In an unsuccessful search, the number of list nodes to examine is λ on average. A successful search requires that about $1 + (\lambda/2)$ links be traversed.

Why? Note that the list that is being searched contains the one node that stores the match plus zero or more other nodes The expected number of "other nodes" in a table of N elements and M lists is $(N-1)/M = \lambda - 1/M$, which is essentially λ, since M is often large. Recall our analysis of sequential data structures: on average, half the list is

searched, so combined with the matching node, the average search cost is $1 + \lambda/2$ nodes. This shows that the table size is not nearly as important as the load factor. The general rule for separate chaining hashing is to make the table size about as large as the number of elements expected.

The graph shows how the cost of finding a node increases as λ increases.



In separate chaining hash tables suffer gradually declining performance as the load factor grows. There is no fixed point beyond which resizing is absolutely needed. Keep in mind that keeping the load factor < 1 in separate chaining may result in too many empty buckets. If memory is at a premium, we can trade some performance for space in memory.

If you are concerned about performance, then you should measure the load factor that maximizes performance on your system. Typically it will be between 1 and 3. One might think that $\lambda = 1$ is the right place to rehash, but the best performance is often seen (for buckets implemented as linked lists) when load factors are in the 1 - 2 range.

If the load factor exceeds our threshold, then we expand the table size by calling `rehash`. This process should also ensure the new hash table storage size maintain a prime number of elements to ensure good distribution among buckets.

Note

For separate chaining, assuming a load factor of 1, this is one version of the classic **balls and bins problem**: Given N balls placed randomly (uniformly) in N bins, what is the expected number of balls in the most occupied bin?

The answer is known to be $\theta(\log N / \log \log N)$, meaning that on average, we expect some queries to take nearly logarithmic time. Similar types of bounds are observed (or provable) for the length of the longest expected probe sequence in a probing hash table. We would like to obtain $O(1)$ worst-case cost.

Suppose we have inserted sz items into a table of size N.

We could use a tree for each bucket, which would reduce the cost of searching buckets to $O(\log_2(sz))$ with an extra cost that `Key` types support a `operator<` comparison in addition to the `operator==` comparison required with list buckets.

In a list-based implementation, if `N` is much larger than `sz`, and if our hash function uniformly distributes our keys, then most lists will have 0 or 1 item, and the average case is approximately $O(1)$. But if `sz` is much larger than `N`, we are looking at an $O(sz)$ linear search sped up by a constant factor (`N`), but still $O(sz)$.

Bottom line: hash tables let us trade space for speed.

More to Explore

- Balls into bins problem on Wikipedia.

3.16.6 Closed hashing

In closed hashing, the hash array contains individual elements rather than a collection of elements. When a key we want to insert collides with a key already in the table, we resolve the collision by searching for another open *slot* within the table where we can place the new key.

Each slot in the hash table contains a `hash_entry`, composed of one data element and a status field indicating whether that slot is *occupied*, *empty*, or *deleted*.

hash_set

```
enum class hash_status { OCCUPIED, EMPTY, DELETED };

template <class T>
struct hash_entry
{
  T data;
  hash_status info;

  hash_entry(): info(hash_status::EMPTY)  {}
  hash_entry(const T& value, hash_status status)
    : data{value}
    , info(status)
    {}
};
```

The `hash_set` backing store is an array of `hash_entry` objects.

```
template <class Key,
          size_t N,
          class Comparator=std::equal_to<Key>>
class hash_set
{
  public:
    using value_type = Key;
    using key_type   = Key;
```

(continues on next page)

(continued from previous page)

```
    hash_set() = default;
  private:
    array<hash_entry<Key>, N> table;
    Comparator compare;
    size_t sz = 0;
};
```

Collisions are resolved by trying a series of locations, $p_0, p_1, p_2, \ldots p_{N-1}$ until we find what we are looking for. Each position is calculated as:

$$pos_i = hash(value) + f(n)$$

where:

- `hash(value)` is the *home slot*
- `f()` is a function taking an integer number of tries and returns an integer offset.

How are new positions used in the hash table?

Searching:
Try positions $p_0, p_1, p_2, \ldots$ until we find the requested value or an `EMPTY` slot.

Inserting:
Try positions $p_0, p_1, p_2, \ldots$ until we find the same value, an `EMPTY` slot, or a `DELETED` slot. Put the new value in the position pound and mark the position as `OCCUPIED`.

Erasing:
Try positions $p_0, p_1, p_2, \ldots$ until we find the requested value or an `EMPTY` slot. If we find the value, then mark the position as `DELETED`.

find

Find takes a value of the `hash_entry` key type as a parameter and returns the position of the value in the table. It returns `N` if the value is not in the table.

```
size_t find (const Key& value) const
{
  size_t hash = std::hash<Key>()(value);
  size_t pos = hash % N;
  size_t count = 0;
  while ((table[pos].info == hash_status::DELETED ||
         (table[pos].info == hash_status::OCCUPIED
         && (!compare(table[pos].data, value))))
      && count < N)
  {
    ++count;
    pos = (hash + next_slot(count)) % N;
  }
  if (count >= N || table[pos].info == hash_status::EMPTY) {
    return N;
  }
  return pos;
}
```

The loop condition is fairly complicated and needs discussion. There are three ways to exit this loop:

- We hit an `EMPTY` space (not `DELETED`, and not `OCCUPIED`)
- We hit an `OCCUPIED` space that has the value we want
- We have tried `N` different positions. (No place left to look!)

contains

With `find` in place, other search operations are easy. Simply call find and evaluate the results.

```
constexpr
  bool contains (const Key& value) const noexcept
{
  return  find(value) != N;
}

int count (const Key& value)
{
  unsigned pos = find(value);
  return  (pos == N) ? 0 : 1;
}
```

Note that since our set is still forcing a uniqueness constraint, `count` will return only `0` or `1`.

erase

The code to remove elements is just as simple. Easier than the `erase` we implemented for open hashing.

We try to find that element.

- If found, we mark that slot `DELETED` and decrement the size.
- Otherwise, do nothing.

```
void erase (const Key& value)
{
  unsigned pos = find(value);
  if (pos != N) {
    table[pos].info = hash_status::DELETED;
    --sz;
  }
}
```

insert

Inserts are a bit more work, because they involve potentially looking for an open slot to store a value.

Because this is a set (and not a multiset) we first call `find` to see if the value is already there.

```
bool insert (const Key& value)
{
  size_t hash = std::hash<Key>()(value);
  unsigned pos = find(value);
  if (pos == N) {
    size_t count = 0;
    pos = hash % N;
    while (table[pos].info == hash_status::OCCUPIED && count < N)
```

(continues on next page)

(continued from previous page)

```
    {
      ++count;
      pos = (hash + next_slot(count)) % N;
    }
    if (count >= N) {
      return false;  // could not add, table is full
    }
    table[pos].info = hash_status::OCCUPIED;
    table[pos].data = value;
    ++sz;
    return true;
  }
  // else replace existing value
  table[pos].data = value;
  return true;
}
```

If not found (`pos == N`), then we need to find a slot. The loop that does this is similar the `find` loop, but unlike `find`, we stop at the first `DELETED` or `EMPTY` slot.

In the other searches, we had kept going past `DELETED` slots, because the element we wanted might have been stored after an element that was later erased. But now we are only looking for an unoccupied slot to put something, so either a slot that has never been occupied (`EMPTY`) or a slot that used to be occupied but is no longer (`DELETED`) works.

Run it

The example contains `#define` statements you can use to change how the next slot is found.

Try it with different hash table sizes to see how clumping changes with the different probing strategies.

```
#include <array>
#include <cstddef>
#include <iomanip>
#include <iostream>
#include <utility>

using std::array;

#define USE_LINEAR_PROBING

#if defined(USE_QUADRATIC_PROBING)
    // find next slot using quadratic probing
    constexpr
      size_t next_slot(size_t count) noexcept { return count*count; }

#elif defined(USE_DOUBLE_HASHING)

    // find next slot using double hashing
    constexpr
      size_t next_slot(size_t count) noexcept { return count * std::hash<size_t>
↪()(count); }
#else   // default to USE_LINEAR_PROBING
    // find next slot using linear probing
```

(continues on next page)

(continued from previous page)

```
    constexpr
      size_t next_slot(size_t count) noexcept { return count; }

#endif


enum class hash_status { OCCUPIED, EMPTY, DELETED };

template <class T>
struct hash_entry
{
  T data;
  hash_status info;

  hash_entry(): info(hash_status::EMPTY)  {}
  hash_entry(const T& value, hash_status status)
    : data{value}
    , info(status)
    {}
};

template <class T>
std::ostream& operator<<(std::ostream& os, const hash_entry<T>& rhs)
{
  if (rhs.info == hash_status::OCCUPIED) {
    os << rhs.data;
  } else if (rhs.info == hash_status::EMPTY) {
    os << 'E';
  } else {
    os << 'D';
  }
  return os;
}

template <class Key,
          size_t N,
          class Comparator=std::equal_to<Key>>
class hash_set
{
  public:
    using value_type = Key;
    using key_type   = Key;

    hash_set() = default;

    size_t find (const Key& value) const
    {
      size_t hash = std::hash<Key>()(value);
      size_t pos = hash % N;
      size_t count = 0;
      while ((table[pos].info == hash_status::DELETED ||
              (table[pos].info == hash_status::OCCUPIED
```

(continues on next page)

(continued from previous page)

```
            && (!compare(table[pos].data, value))))
          && count < N)
    {
      ++count;
      pos = (hash + next_slot(count)) % N;
    }
    if (count >= N || table[pos].info == hash_status::EMPTY) {
      return N;
    }
    return pos;
  }

  constexpr
    bool contains (const Key& value) const noexcept
  {
    return  find(value) != N;
  }

  int count (const Key& value)
  {
    unsigned pos = find(value);
    return  (pos == N) ? 0 : 1;
  }

  void erase (const Key& value)
  {
    unsigned pos = find(value);
    if (pos != N) {
      table[pos].info = hash_status::DELETED;
      --sz;
    }
  }


  bool insert (const Key& value)
  {
    size_t hash = std::hash<Key>()(value);
    unsigned pos = find(value);
    if (pos == N) {
      size_t count = 0;
      pos = hash % N;
      while (table[pos].info == hash_status::OCCUPIED && count < N)
      {
        ++count;
        pos = (hash + next_slot(count)) % N;
      }
      if (count >= N) {
        return false;  // could not add, table is full
      }
      table[pos].info = hash_status::OCCUPIED;
      table[pos].data = value;
      ++sz;
```

(continues on next page)

(continued from previous page)

```
        return true;
      }
      // else replace existing value
      table[pos].data = value;
      return true;
    }


    constexpr
      size_t size() const noexcept { return sz; }

    constexpr
      bool empty() const noexcept { return sz == 0; }

  private:
    array<hash_entry<Key>, N> table;
    Comparator compare;
    size_t sz = 0;

    friend
      std::ostream& operator<<(std::ostream& os, const hash_set& rhs)
      {
        os << '[';
        for (const auto& slot: rhs.table) {
          os << slot << ',';
        }
        return os << ']';
      }

};

int main() {
  using std::cout;
  using std::endl;
  auto foo = hash_set<int, 11>{};
  cout << "sz: " << foo.size() << endl;
  cout << std::boolalpha << "mt?: " << foo.empty() << endl;
  cout << foo << endl;
  foo.insert(72);
  foo.insert(72);
  cout << "insert two 72's count:"<< endl;
  cout << foo.count(72) << endl;
  cout << foo << endl;
  cout << "mt?: " << foo.empty() << endl;

  foo.erase(72);
  cout << "count after erase:"<< endl;
  cout << foo.count(72) << endl;

  foo.insert(-1);
  foo.insert(0);
  foo.insert(1);
```

(continues on next page)

(continued from previous page)

```
  foo.insert(2);
  foo.insert(9);
  foo.insert(81);
  foo.insert(121);
  foo.insert(572);
  foo.insert(999);
  cout << foo << endl;
  foo.erase(-1);
  cout << foo << endl;
}
```

Choosing the next slot

The function `next_slot(n)` in the `find` and `insert` functions controls the sequence of positions that will be checked. It is the implementation of the function $f(n)$ mentioned earlier. Recall the find function:

```
size_t find (const Key& value) const
{
  size_t hash = std::hash<Key>()(value);
  size_t pos = hash % N;
  size_t count = 0;
  while ((table[pos].info == hash_status::DELETED ||
         (table[pos].info == hash_status::OCCUPIED
         && (!compare(table[pos].data, value))))
      && count < N)
  {
    ++count;
    pos = (hash + next_slot(count)) % N;
  }
  if (count >= N || table[pos].info == hash_status::EMPTY) {
    return N;
  }
  return pos;
}
```

On our n_{th} try, we examine the position

$$pos_n = hash(value) + f(n)$$

where $hash(value)$ always returns the *home slot* for any hashed value. This is the location that the value would be stored if currently unoccupied. The `f` function computes an offset from the reference location. The most common schemes for choosing the next slot are *linear probing*, *quadratic probing*, and *double hashing*.

Linear probing

$$f(n) = n$$

If a collision occurs at location `pos`, we next check locations $pos + 1 \pmod N, pos + 2 \pmod N, pos + 3 \pmod N, \ldots$ and so on.

Because collisions get stored in a location originally intended for another hash code, values have a tendency to clump together in the hash table.

Quadratic probing

$$f(n) = n^2$$

If a collision occurs at location `pos`, we next check locations $pos + 1 \pmod N, pos + 4 \pmod N, pos + 9 \pmod N, \ldots$ and so on.

Because the jumps between slots increases as the number of tries increases, this function tends to reduce clumping (and results in shorter searches). ``*But*`` it is not guaranteed to find an available empty slot if the table is more than half full or if `N` is not a prime number.

Note

Again, prime numbers!

Remember the earlier discussion about how `% N` tends to improve the key distribution when `N` is prime? You can see why it's part of programming "folklore" that hash tables should be prime-number sized, even if most programmers can't say *why* that's supposed to be good.

Double hashing

$$f(n) = n * h_2(value)$$

where h_2 is an alternate hash code function.

If a collision occurs at location `pos`, we next check locations $(pos+1*h_2(value)) \pmod N, pos+2*h_2(value)) \pmod N, pos + 3 * h_2(value)) \pmod N, \ldots$ and so on.

This also tends to reduce clumping, but, as with quadratic hashing, it is possible to get unlucky and miss open slots when trying to find a place to insert a new key.

Analysis of closed hashing

We define λ, the *load factor* of a hash table, as the number of items contained in the table divided by the table size. In other words, the load factor measures what fraction of the table is full. By definition, $0 \leq \lambda \leq 1$.

- Given an ideal collision strategy, the probability of an arbitrary cell being full is λ.
- Therefore, the probability of an arbitrary cell being empty is $1 - \lambda$.
- The average number of table elements we expect to examine before finding an open position is therefore $\frac{1}{1-\lambda}$.

Since we never look at more than `N` positions, given an ideal collision strategy, finds and inserts are on average

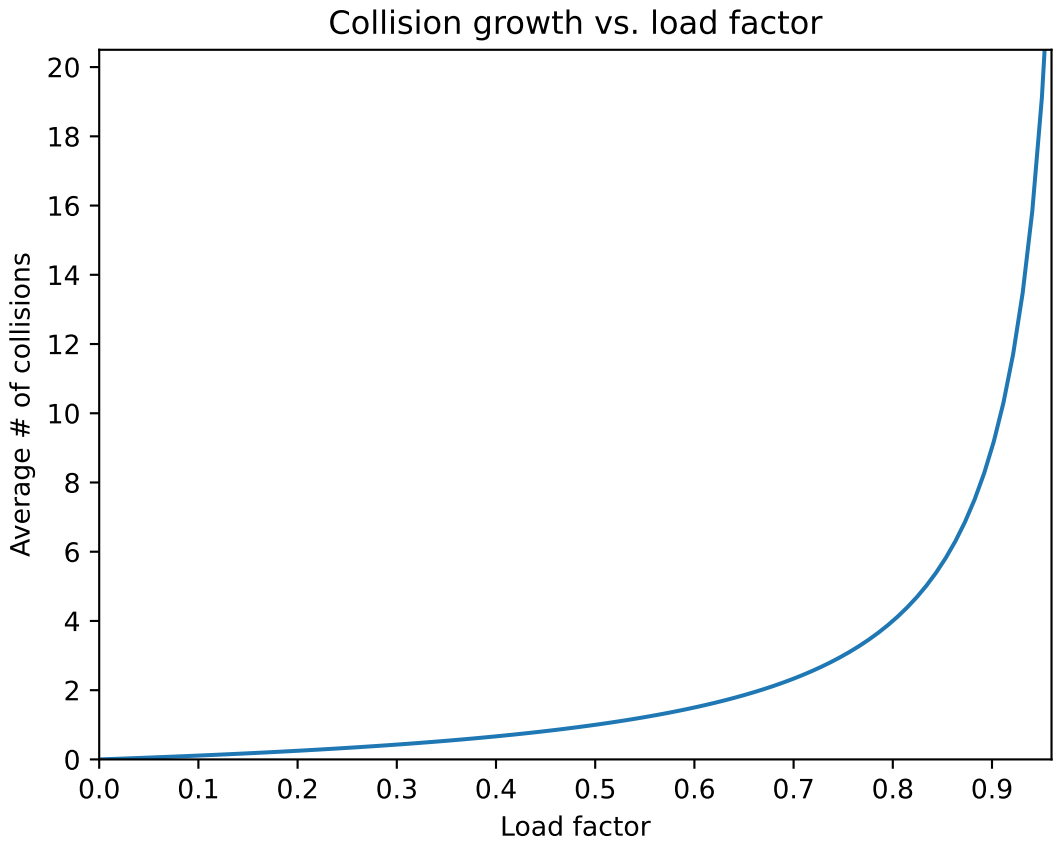
$$O\left(min\left(\frac{1}{1-\lambda}, N\right)\right)$$

The graph shows how $\frac{1}{1-\lambda}$ changes as λ increases.

If the table is less than half full ($\lambda < 0.5$) then we expect to try **on average** no more than 2 slots during a search or insert. Not too bad. But as λ gets larger, the average number of slots examined grows toward `N`. As the table fills and `sz` approaches `N`, the performance degenerates toward $O(N)$ behavior.

Because of this, a general rule of thumb for hash tables is to keep them no more than half full. At that load factor, we can treat searches and inserts as $O(1)$ operations. If we let the load factor get much higher, we start seeing $O(N)$ performance.

No collision resolution scheme is truly ideal, so keeping the load factor low enough is even more important in practice than this idealized analysis indicates.


Collision growth vs. load factor
Average # of collisions
Load factor
0
2
4
6
8
10
12
14
16
18
20
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

More to Explore

- The content on this page was adapted from Resolving Collisions <https://www.cs.odu.edu/~zeil/cs361/f25-web/Public/collisions/index.html>, by Steven J. Zeil for his data structures course CS361.

3.16.7 Analysis of hash tables

The table below shows the **average case** complexity of some basic unordered_map operations. For each operation that has constant time average case efficiency, the *worst case* complexity is $O(n)$.

Table 5: **Average complexity of C++ unordered map operations**

Operation	Complexity
assignment =	O(1)
insert()	O(1)
find()	O(1)
contains()	O(1)
erase()	O(1)
clear()	O(n)

The reason these operations may have $O(n)$ complexity is because the performance of the container is ultimately controlled by the quality of the hash function for the key type in the container. If the hash function performs poorly (many collisions), then the benefits of hash tables are lost and we decay into list performance. When the hash function quality is high, then the performance is good.

When we discussed the messy and neat closets in *Tree ADT concepts*, we mentioned the primary motivation for non-sequential containers was search. Unlike even a sorted vector or a tree, hash tables provide constant time access tot he correct bucket containing our data and linear search is required only when collisions exist.

The following code shows what happens when searching in an unordered map vs a vector.

```
#include <algorithm>
#include <chrono>
#include <iomanip>
#include <iostream>
#include <numeric>
#include <unordered_map>
#include <vector>

int main() {
    using clock = std::chrono::high_resolution_clock;
    std::cout << std::setw(6) << "size"
              << std::setw(10) << "vector"
              << std::setw(20) << "hash table\n";
    // for(int size = 10'000; size < 100'001; size += 20'000) {
    int size = 35000;
        // fill vector
        std::vector<int> sequence (size);
        std::iota(sequence.begin(), sequence.end(), 0);
        // search vector
        auto begin = clock::now();
        for(const auto& it: sequence){
```

(continues on next page)

(continued from previous page)

```
            if(std::find(sequence.begin(), sequence.end(), it) == sequence.end()) {
                std::cerr << "Failed to find an expected value in vector! Halting.\n";
                return -2;
            }
        }
        auto end = clock::now();
        std::chrono::duration<double> elapsed_secs = end - begin;
        // fill hash table
        std::unordered_map<int, int> table;
        for(int item = 0; item < size; ++item){
            table[item] = item;
        }
        begin = clock::now();
        // search hash table
        for(const auto& it: table){
            if(table.find(it.first) == table.end()) {
                std::cerr << "Failed to find an expected value in map! Halting.\n";
                return -2;
            }
        }
        end = clock::now();
        std::chrono::duration<double> elapsed_secs_ht = end - begin;

        // Printing final output
        std::cout << std::fixed   << std::setprecision(4)
                  << std::setw(6) << size << '\t'
                  << std::setw(8) << elapsed_secs.count() << '\t'
                  << std::setw(8) << elapsed_secs_ht.count() << '\n';
    // }
    return 0;
}
```

Try This!

The online compiler is limited in both memory and time allowed.

Run this example on your own computer with the loop enabled and with larger values and compare.

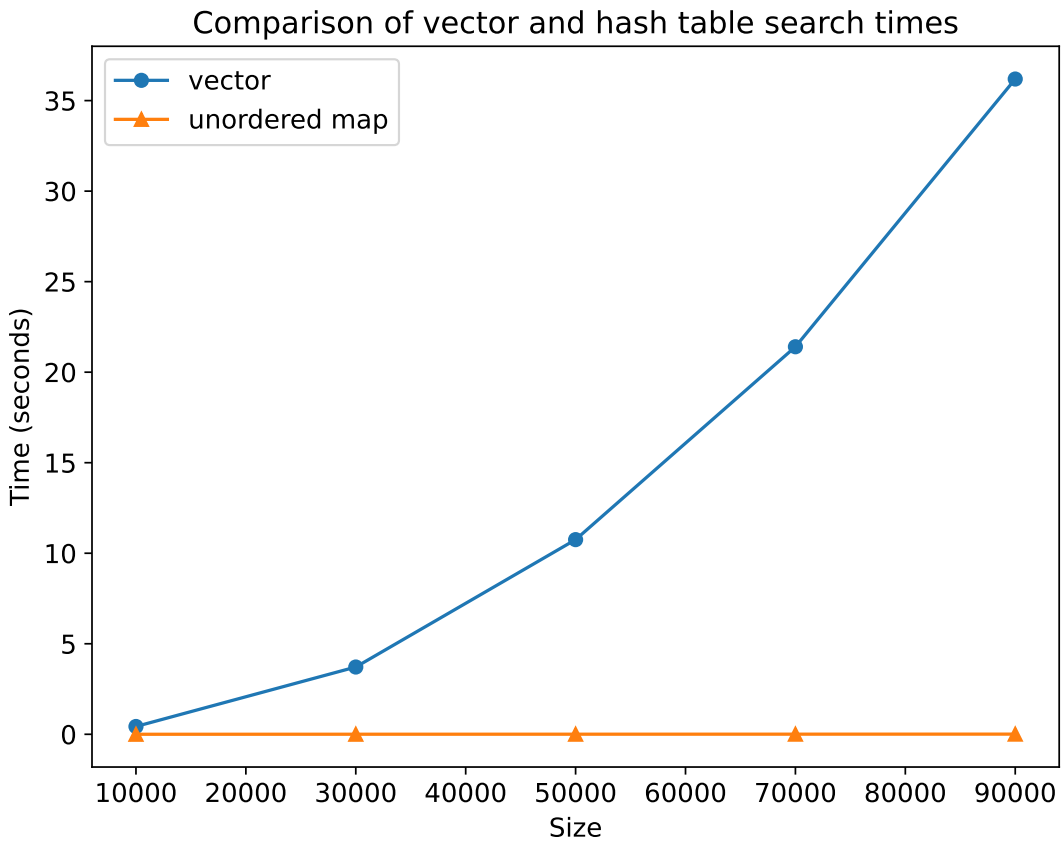
The vector is linear in `std::distance(begin, end)` and as expected, the hash table is constant time. Running the previous code should produce results similar to this:

So what about the tree ADT? The *std::set* is generally implemented as a tree. The C++ standard guarantees logarithmic complexity in the size of the container.

How does std::set find compare to std::unordered_map find?

```
#include <algorithm>
#include <chrono>
#include <iomanip>
#include <iostream>
#include <numeric>
#include <set>
```

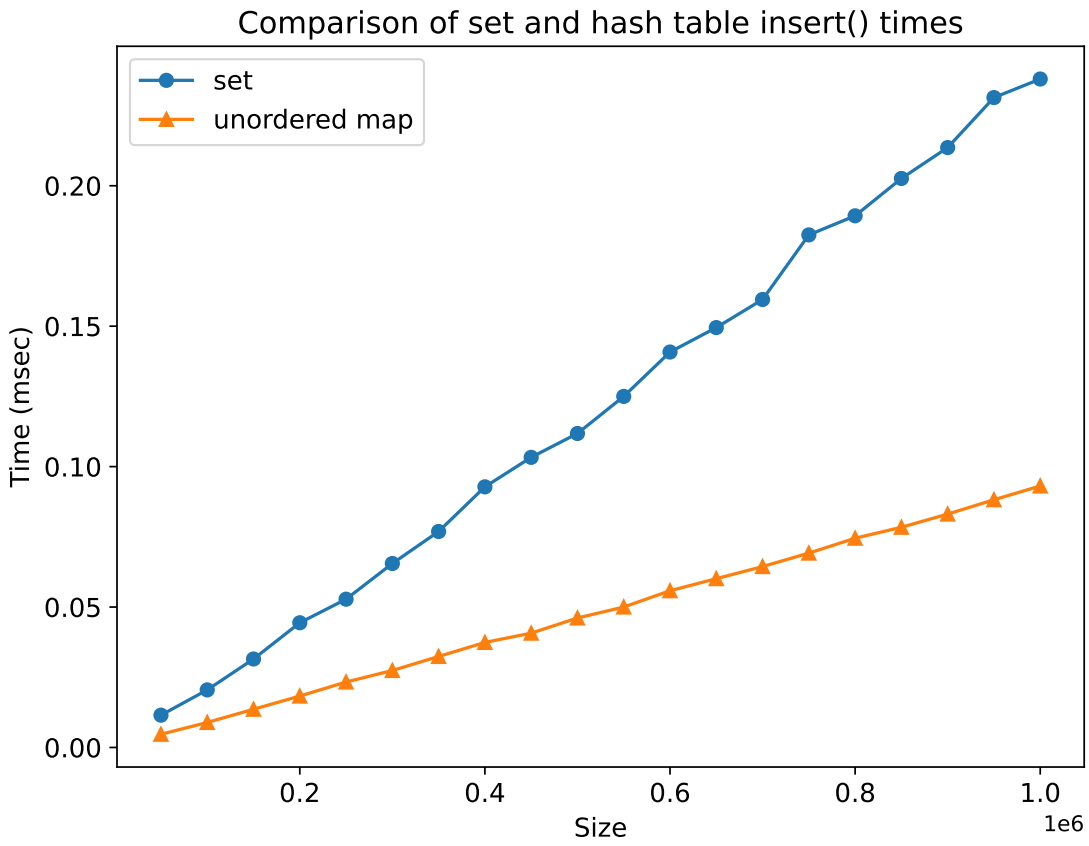
(continues on next page)


Comparison of vector and hash table search times
vector
unordered map
Time (seconds)
0
5
10
15
20
25
30
35
10000
20000
30000
40000
50000
60000
70000
80000
90000
Size

(continued from previous page)

```cpp
#include <unordered_map>

int main() {
    using clock = std::chrono::high_resolution_clock;
    std::cout << std::setw(6) << "size"
              << std::setw(10) << "set"
              << std::setw(20) << "hash table\n";
    for(int size = 5'000; size < 100'001; size += 5'000) {
        // fill set
        std::set<int> tree;
        for(int item = 0; item < size; ++item){
            tree.insert(item);
        }
        // search set
        auto begin = clock::now();
        for(const auto& it: tree){
            if(tree.find(it) == tree.end()) {
                std::cerr << "Failed to find an expected value in set! Halting.\n";
                return -2;
            }
        }
        auto end = clock::now();
        std::chrono::duration<double> elapsed_secs = end - begin;
        // fill hash table
        std::unordered_map<int, int> table;
        for(int item = 0; item < size; ++item){
            table[item] = item;
        }
        begin = clock::now();
        // search hash table
        for(const auto& it: table){
            if(table.find(it.first) == table.end()) {
                std::cerr << "Failed to find an expected value in map! Halting.\n";
                return -2;
            }
        }
        end = clock::now();
        std::chrono::duration<double> elapsed_secs_ht = end - begin;

        // Printing final output
        std::cout << std::fixed   << std::setprecision(4)
                  << std::setw(6) << size << '\t'
                  << std::setw(8) << elapsed_secs.count() << '\t'
                  << std::setw(8) << elapsed_secs_ht.count() << '\n';
    }
    return 0;
}
```

Although the std::set find is logarithmic complexity, from a practical sense, it compares favorably with the hash table. The graph below shows example output for values up to 1,000,000.


Comparison of set and hash table insert() times
set
unordered map
Time (msec)
0.00
0.05
0.10
0.15
0.20
0.2
0.4
0.6
0.8
1.0
1e6
Size

Try This!

The online compiler is limited in both memory and time allowed.

Run this example on your own computer with larger values and compare.

More to Explore

- TBD

3.17 Algorithms

3.17.1 Background

Recall from *Container classes* that a container is a generic collection. Containers allow us to store data using *well-known* data structures. The STL containers provide the patterns we can use to make custom containers, if needed.

Recall from *Iterable ADT's* that an iterator is a *type* that performs operations that *feel* like a pointer. Although an iterator allows syntax very similar to a pointer, it is not a pointer. Each container is responsible for its own iterators. When a container is created, it has the ability to create an iterator that knows how to visit elements of the type stored in the container.

Now that we have these two great tools in the STL, we want to use them to solve problems. It turns out that many programming tasks fall into basic groups:

- find
- copy
- sum
- count
- sort

These are all *actions* that we perform on *sequences*. The goal of the STL algorithms is to define these actions in a generic way. The STL algorithms satisfy this goal using small, reusable functions that avoid writing repetitive code and define a consistent, portable interface.

The *abstractions* in the STL are primarily concerned with performing actions on data stored in STL containers. Consider that counting elements in a *list* is not very different from counting elements in a *vector*.

STL Algorithms at a glance

The STL algorithms are part of the ISO C++ Standard. Currently, it contains more than 150 algorithms for searching, counting, and manipulating ranges. C++17 alone added 69 more algorithms to the library. While most of these (but not all) were new overloads to existing algorithms, it does demonstrate the dynamic nature of the STL and its growth.

The algorithms are organized into broad categories:

Algorithm operations	Example algorithms
Non-modifying sequence operations	for_each, count_if, find_if, search
Modifying sequence operations	copy_if, move, swap, transform
Partitioning operations	is_partitioned, partition_copy, stable_partition
Sorting operations	is_sorted, sort, stable_partition
Binary search operations	lower_bound, binary_search, equal_range
Set operations	merge, includes, set_difference, set_union
Heap operations	is_heap, make_heap, sort_heap
Min/max operations	max, min, max_element, clamp
Comparison operations	equal, lexicographical_compare
Permutation operations	is_permutation, next_permutation
Numeric operations	iota, accumulate, inner_product, reduce
Uninitialized memory operations	uninitialized_copy, uninitialized_fill, destroy

Notice that only a single category of algorithms is considered 'numeric'. In C++11, only 5 algorithms specifically do numeric computations. C++17 adds 6 more.

STL algorithms and loops

Most STL algorithms are essentially wrappers around loops. They often take a range of elements and an operation that is performed on each element. Structurally, this makes them similar to loops.

Most tasks you've written so far could be rewritten using algorithms.

One way to think about STL algorithms is to consider them *named loops*. That is, a loop that is important and general enough to justify getting named and encapsulated in its own function.

iota is a STL algorithm that fills a range `[first, last)` with sequentially increasing values. This is the sort of algorithm that occurs often enough that it was decided to include it in the standard library (but not until C++11).

Example: iota

The parameter `value` defines the start value. This value is assigned to `first`, and both `first` and `value` are incremented.

```
template<typename ForwardIterator, typename T>
void iota(ForwardIterator first,
          ForwardIterator last, T value) {
  while(first != last) {
    *first++ = value;
    ++value;
  }
}
```

Note the order of operations on 5.

- First `first` is dereferenced and `value` is assigned.
- **Then** the iterator is incremented.

Run It

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

(continues on next page)

(continued from previous page)

```

// possible implementation for iota
template<typename ForwardIterator, typename T>
void iota(ForwardIterator first,
          ForwardIterator last, T value) {
  while(first != last) {
    *first++ = value;
    ++value;
  }
}

void print(const std::vector<int>& v) {
  for (auto x: v) {
    std::cout << std::setw(3) << x;
  }
  std::cout << '\n';
}

int main () {
  std::vector<int> nums(13);
  std::cout << "Before iota:";
  print(nums);

  iota(nums.begin(), nums.end(), -6);
  std::cout << "After iota: ";
  print(nums);
}
```

Why prefer algorithms to hand-written loops?

- Efficiency, for one.

  Algorithms are often more efficient than the loops programmers produce. The developers of the STL have had overt 20 years to consider how to make these algorithms efficient. Many algorithms have code to handle specific edge cases your initial implementations might overlook.

- Correctness

  Writing loops is more subject to errors than algorithm calls. As a programmer you have to worry about initializing the loop, incrementing the loop, terminating the loop as well as the loop body.

  When calling an algorithm, you only need to get the start and end iterators correct.

  Often you don't even need to care about the body - the algorithm takes care of all the details for you. Sometimes a lambda or function pointer is expected.

  The STL implementations have been carefully reviewed and tested in many thousands of programs. It is safe to say that any STL algorithm has been tested more thoroughly than any comparable code you write yourself.

- Maintainability

  Algorithm calls result in clearer code. The STL is designed around a simple consistent set of interfaces. The more you use these interfaces, the more consistently your own code will be structured.

  When combined together, algorithms can eliminate lots of code you other wise would have needed to write and results in more straightforward than the corresponding explicit loops.

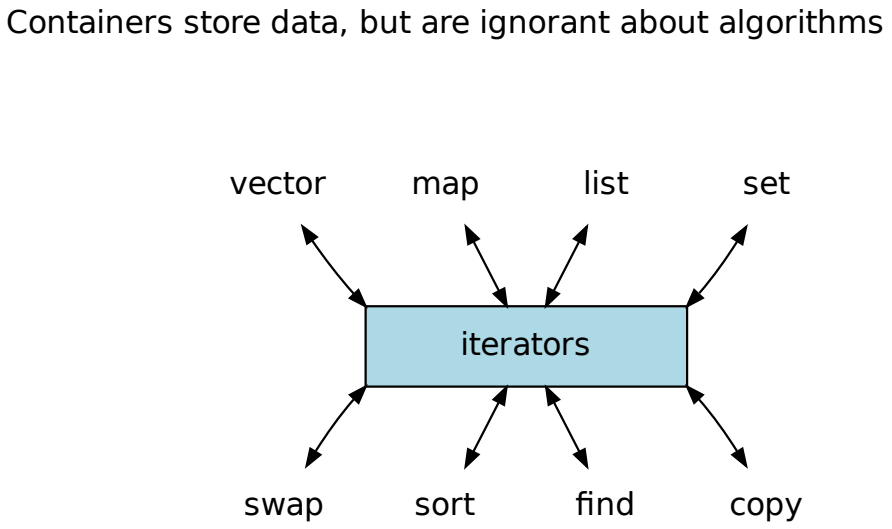
Code you use from the STL is code you don't need to maintain. The less code you have to maintain, the cheaper and easier it is to maintain.

More to Explore

- From cppreference.com
  - Overview of the algorithms library

3.17.2 Basic Model

One of the primary goals of the STL is avoiding repetition & using regular, compact syntax. The STL achieves these goals using *separation of concerns*.



Algorithms and containers interact through **iterators**.

Basic Model in Action: find()

Let's suppose we need to find the first element in a container that equals a value. Specifically, we want to find a specific int in a vector.

This seems like a function we will need to use frequently, so we decide right away that it should be written as a free function:

Example

We *could* choose to pass an entire container of a specific type to our find function.

```
std::vector<int>::iterator
find(std::vector<int>& v, int x) {
```

and loop over the entire container:

```
for(auto p = v.begin(); p != v.end(); ++p) {
  if (x == *p) return p;
}
```

Run It

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

// function to find 'x' in v
std::vector<int>::iterator
find(std::vector<int>& v, int x) {
  for(auto p = v.begin(); p != v.end(); ++p) {
    if (x == *p) return p;
  }
  // did not find x
  return v.end();
}

int main () {
  int value = 144;
  std::vector<int> nums(999);
  std::iota(nums.begin(), nums.end(), -72);
  auto it = find(nums, value);

  if (it == nums.end()) {
    std::cout << "Did not find " << value <<  '\n';
  } else {
    std::cout << "Found " << value << " at position "
              << (std::distance(nums.begin(), it)) << '\n';
  }
  return 0;
}
```

While this seems easier at first, this version is not **nearly** as *generic*, or *general purpose* as a version that defines a generic type and uses iterators.

- No way to run this function over part of a container.
- Need a different function for every container type.

How do we refactor our find function to satisfy our goals?

- Replace `vector<int> v` with a pair of iterators
- Make the iterators generic types

- Make the value type generic

Example

```
// function to find 'x' between first and last
template<class InputIt, class T>
// requires: InputIt is Convertible to T when dereferenced
//        && InputIt is EqualityComparable
//        && T is Regular
InputIt my_find(InputIt first, InputIt last, const T& value)
{
  for (; first != last; ++first) {
    if (*first == value) {
      return first;
    }
```

Run It

And we can prove to ourselves that we get the same results as find.

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

// function to find 'x' between first and last
template<class InputIt, class T>
// requires: InputIt is Convertible to T when dereferenced
//        && InputIt is EqualityComparable
//        && T is Regular
InputIt my_find(InputIt first, InputIt last, const T& value)
{
  for (; first != last; ++first) {
    if (*first == value) {
      return first;
    }
  }
  return last;
}

int main () {
  int value = 144;
  std::vector<int> nums(999);
  std::iota(nums.begin(), nums.end(), -72);

  auto it  = find(nums.begin(), nums.end(), value);

  if (it != nums.end()) {
    std::cout << "my_find() found " << value << " at position "
              << (std::distance(nums.begin(), it)) << '\n';
  }
  return 0;
}
```

And since it is arguably the same function as `std::find`, we now know we no longer need it.

Try this!

Change the name of the function `my_find` to `find` and change the matching name on line 24.

Does this program still compile? Explain.

Rewrite the previous example to use find.

More to Explore

- From cppreference.com
  - Overview of the algorithms library.
  - std::find (and find_if).

3.17.3 Refactoring to Algorithms

The primary objective of refactoring is to improve code. Those improvements might take many forms. In this section we are going to focus on refactoring a pair of functions that at first glance do not appear to be doing the same thing. However, we will see the similarities and how refactoring is accomplished, step-by-step.

Given two functions, each sums the values provided.

The first function adds all of the integers in a raw array:

```
int sum(int array[], int n) {
  int sum = 0;
  for (int i = 0; i < n; ++i ) {
    sum += array[i];
  }
  return sum;
}
```

The second adds all of the elements in a simple, home-grown linked list.

```
// create a simple node in a linked list
struct node {
  int value = 0;
  node* next = nullptr;
};

int sum(node* first) {
  int s = 0;
  while (first) {        // first not false or zero
    s += first->value;
    first = first->next;
  }
  return s;
}
```

How can we generalize and combine these two functions into one? We can rewrite both functions in a form of pseudo-code.

```
// we need a generic type 'T'
T sum(/* data */ )                    // somehow parameterize this
{
  T s = 0;
  while (/* not at end */ ) {         // loop through all elements
    s = s + /* get value */;          // compute sum
    /* get next data element */;
  }
  return s;
}
```

We need several generic operations on **data**:

- Determine if we are not at end of data
- Get value
- Get next element

Example

The STL style supports both data structures.

Like find, we define a pair of iterators. `first` and `last`. The iterator type should support the requirements of InputIterator.

A separate template parameter for the initial sum finishes the signature.

The value must be a *regular type* and the dereferenced iterator must be convertible to the value type.

The function signature becomes:

```
template <typename InputIt, typename T>
// requires: InputIt is convertible to T when dereferenced
//        && InputIt is EqualityComparable
//        && T is Regular
T sum (InputIt first, InputIt last, T value) {
```

The main loop checks whether we should continue and accumulates the sum:

```
while (first != last) {
  value = value + *first;
  ++first;
}
```

Run It

And we can use this algorithm with either a raw array or a linked list.

```
#include <iostream>
#include <list>

// accumulate the sum of values in range [first, last)
template <typename InputIt, typename T>
// requires: InputIt is convertible to T when dereferenced
```

(continues on next page)

(continued from previous page)

```
//        && InputIt is EqualityComparable
//        && T is Regular
T sum (InputIt first, InputIt last, T value) {
  while (first != last) {
    value = value + *first;
    ++first;
  }
  return value;
}

int main () {
  float a[]  = {1,1,2,3,5,8,13,21,34};
  float* end = a+sizeof(a)/sizeof(*a);
  double d   = 0;

  d = sum (a, end, d);
  std::cout << "array sum  = " << d << '\n';

  // now do list
  d = 0;
  std::list<float> b = {1,1,2,3,5,8,13,21,34};

  d = sum (b.begin(), b.end(), d);
  std::cout << "list sum  = " << d << '\n';
  return 0;
}
```

Removing a final assumption

Can we make `sum` even more generic?

Sum still has a hard-coded assumption that addition (the `operator+` function) is the operation that we always want to perform.

Might we want to perform **any** binary operation on a sequence? If yes, then we can add one more template parameter allowing callers to pass in a function pointer (or equivalent).

Example

The function signature becomes:

```
template <typename InputIt, typename T, typename BinaryOp>
// requires: InputIt is convertible to T when dereferenced
//        && InputIt is EqualityComparable
//        && T is Regular
T accumulate (InputIt first,
              InputIt last,
              T value,
              BinaryOp op) {
```

The main loop replaces the explicit + with a call to a provided binary operator:

```
  value = op(value, *first);
```

This *could* be addition: `operator+`, but can now support any binary operation that the type `T` supports.

A default operation can be provided with a supporting template that calls accumulate with plus.

```
template <typename InputIt, typename T>
T accumulate (InputIt first, InputIt last, T value) {
  return accumulate(first, last, value, std::plus<T>());
}
```

Run It

```
#include <cstddef>
#include <iostream>
#include <functional>
#include <vector>

// accumulate the sum of values in range [first, last)
// using the binary operation op
template <typename InputIt, typename T, typename BinaryOp>
// requires: InputIt is convertible to T when dereferenced
//        && InputIt is EqualityComparable
//        && T is Regular
T accumulate (InputIt first,
              InputIt last,
              T value,
              BinaryOp op) {
  while (first != last) {
    value = op(value, *first);
    ++first;
  }
  return value;
}

// version that provides a default operation
template <typename InputIt, typename T>
T accumulate (InputIt first, InputIt last, T value) {
  return accumulate(first, last, value, std::plus<T>());
}

int main () {
  std::size_t sum = 0;
  std::vector<std::size_t> x = {1,1,2,3,5,8,13,21,34};
  sum = accumulate (x.begin(), x.end(), sum);
  std::cout << "vector sum  = " << sum << '\n';

  std::size_t product = 1;
  product = accumulate (x.begin(), x.end(), product, std::multiplies<std::size_t>());
  std::cout << "vector product  = " << product << '\n';
  return 0;
}
```

Note that we did not pass + or * to a function. The symbol + is not a type.

The parameter passed through `BinaryOp op` **must** be a valid *type*.

A function *can* take a pointer or a type as a parameter. Function objects passed as parameters must satisfy the requirements of function. Lambda expressions, function objects, and functions pointers are all acceptable. The STL has a large collection of operator types that can be passed to functions.

More to Explore

- From CPP Core Guidelines
  - T.2 Use templates to express algorithms that apply to many argument types

3.17.4 std::copy and Iterator Adapters

TBD.

Need an example showing how some algorithms break with default iterators if there is no 'next' element.

More to Explore

- From cppreference.com
  - std::copy
  - std::inserter

CHAPTER

FOUR

BACK MATTER

4.1 Frequently asked questions (FAQ)

This section is a collection of my answers to some frequently asked questions you may find useful.

4.1.1 Why not use an IDE?

You can use an Integrated Development Environment (IDE) if you feel strongly about it. An IDE can make you more productive in some environments. But I don't recommend an IDE *ever* when learning a language.

Why?

Lots of reasons.

1. When learning, you are still trying to commit the syntax and grammar of the new language to memory. If every other keystroke is <CTRL>-space and you let the tool autocomplete everything, you will never really master the language.
2. An IDE is not a 'free lunch'. They are typically complex applications with loads of options that require as much learning as the modes and keyboard commands in vim. Unless you put in the time and effort to learn the IDE, you won't realize most of the productivity gains anyways.
3. IDEs typically only support one, or a few language really well. Microsoft Visual Studio is an exception. I would rate it as very good for Visual Basic, C#, and Web Development (including JavaScript).

   There are no 'great' C or C++ IDEs.

   But even if you find a great IDE, if you work in a different language, then you need to potentially learn a completely different tool to work with the other language.
4. IDEs are great if you are working on a single, medium sized project for an extended period of time. Think more than 1,000 lines of code.

   On a project, the time it takes to setup a new project is worth the investment, because you'll be working on it for potentially a long time.

   This is not what we do in the classroom.

   We create lots of very small programs, because we are trying to distill new concepts down to the smallest program that will demonstrate the idea.

   In this environment, IDEs get in the way, because it takes more time to make a new project in the IDE, than it does to just type `make my_program` on the command line or in vim.

More to Explore

- Textbook: *Command-line compiling*

4.1.2 What language should I learn?

A closely related question is 'Why are there so many languages'?. The short answer is because languages themselves are tools. Perhaps a more correct question is 'what kind of software projects am I interested in working on'?

- If you are interested in web applications or cross-platform applications based on Node.js, you should learn HTML, CSS, and JavaScript.
- If you are interested in iPhone development, you should learn Swift and Objective-C.
- If you are interested in Android development, you should learn Java.
- If you are interested in embedded development, you should learn C and C++.
- If you are interested in systems development, game development, or creating virtual machines, you should learn C++.
- If you are interested in Windows application development, you should learn C#.
- If you are interested in Gnu/Linux development, you should learn C, C++, and Python.

So many popular languages are derived from C, it is a good language to know eventually, but it does not need to be the first language you learn. C, one of the oldest languages still in use today, was voted 'TIOBE language of the year' in 2017 because it was the fastest growing language according to the TIOBE index.

In the past, a relatively small number of languages tended to dominate. Most programs were written using relatively few languages. This is less the case today.

All programming languages have something to offer. The most important thing to learn is how to think like a programmer:

- Lifehacker has an article
- and so does Yevgeniy (Jim) Brikman

There is even a book that can help.

The best way to learn any language is to have a reason that is important to you. That is, you have a problem you really want to solve. Pick an appropriate language and start coding!

As the programming is terrible blog says:

> The first programming language you learn will likely be the hardest to learn. Picking something small and fun makes this less of a challenge and more of an adventure. It doesn't really matter where you start as long as you keep going - keep writing code, keep reading code. Don't forget to test it either. Once you have one language you're happy with, picking up a new language is less of a feat, and you'll pick up new skills on the way.
>
> —tef, programming is terrible, 2013-01-13

Don't worry if you are not completely happy with the results of your early programs. You'll improve with practice.

Every program I ever wrote is horrible about a year after I finish it. I have never written a program and said:

> Wow, this is just perfect. I don't want to change a thing.
>
> —Dave Parillo, A bold-faced lie, 2018.

What languages did I learn first?

In the late 1970's, I began programming using Applesoft-BASIC and Sinclair BASIC. In the 1980's, I progressed from Rocky Mountain BASIC, to Pascal, to C and FORTRAN. Since then, I have been all over the map, coding in a variety of languages, depending on the project.

More to Explore

- Predicted top languages of 2018
- Github's State of the Octoverse
- TIOBE Index

4.1.3 Should I learn C first?

Maybe.

The answer depends partly on what kind of software projects might interest you (see *What language should I learn?*).

You certainly do not need to learn C as a prerequisite for learning C++. If you already know some C, it doesn't hurt, but there is no real advantage to learning C first. C does many things 'the hard way' and many solutions to the same problems in the two languages end up being quite different.

4.1.4 What other good C++ resources are out there?

Congratulations! You made to the end of the semester and this book, now what?

First, take a well-earned rest. When you are ready to think about programming again, consider the following for further exploration.

First, just for fun there is the Map of the C++17 Lands. Now that you have finished this material, you now know enough to see the humor in the map.

Coding challenges and puzzles

If you haven't figured it out by now, I think the best way to learn to program in any language is by writing code.

- exercism Code practice and mentorship for everyone. Level up your programming skills with 2,545 exercises across 67 languages.
- edabit - coding challenges for beginners in 8 different languages
- Advent of Code - Help elves, learn to code. 25 Puzzles per year since 2015. Technically 50 per year since each puzzle has 2 parts, each of which involves writing code.

  Puzzles can be solved using any programming language.
- Codewars - coding challenges on medium level
- LeetCode - medium to difficult algorithm challenges
- Project Euler a series of challenging mathematical/computer programming problems that require more than just mathematical insights to solve.

  Puzzles can be solved using any programming language.

  Although mathematics will help you arrive at elegant and efficient methods, the use of a computer and programming skills will be required to solve most problems.

- C++ Quiz a simple online quiz you can use to test your knowledge of C++.
- Code Chef Code Chef is a not-for-profit educational initiative. They provide this platform as a way to practice and compete in friendly competition.
- UVa Online Judge online problems sets - some with published solutions.

Suggested reading

Some of these books and articles have been around a long time, but they are all classics that have retained considerable value.

First, a good, short blog post about how to teach yourself programming. Good advice.

- A Philosphy of Software Design, John Ousterhout. A great and short book that does not require any special programming skill - targeted at students, but generally useful.
- Code Complete A true compendium. Which may be a 'pro' or a 'con' for you.

  Originally written in 1993 and redone in 2004, it is a collection of software best practices with essentially no competition. Years later, there remains little to disagree with
- The Pragmatic Programmer. I really like the style of this book.

  It's thin & you can read it in any order. You can skip around or read it straight through.

  It's primary focus is on honing the behaviors that turn OK programmers into great ones. If you read this, you see may similarities between sections in Code Complete. This is just less comprehensive - it's 'agile'!
- Chapter 16 of the Mythical Man-Month: *No silver bullets*.

  This book gets a lot of endorsements, but honestly, I think most of it is not too relevant to modern computing problems, with a few exceptions. The "No silver bullets" chapter complements the 'religion' section at the end of Code Complete.
- *The Design of Everyday Things*, Don Norman and *The Inmates are running the Asylum*, Alan Cooper.

  These two books are focused on how people interact with things in general (Norman), and software in particular (Cooper). Cooper's main point is that knowing how to design a user interface is a distinct skill and you can't just expect any old programmer to be a great designer - which is a point made by Fred Brooks in No Silver bullets, just in less space.
- Effective Java - yes Java.
- Effective C++ (but it's a bit dated - along with parts of Effective C++, More Effective C++ and Effective Modern C++)

  You'll find a lot of the material in Code Complete also mentioned these books as well. I prefer Effective Java over Effective C++, even though Josh Bloch was inspired to write Effective Java *by* Effective C++. His version is less specific - that has allowed it to stay relevant even though Java has changed a lot. I also like that Josh is not shy about calling out specifics where the design of the Java libraries was just wrong. He uses the design of `Object.clone()`, `Object.notify()` & `wait()`, `java.util.Date`, `java.util.Stack` and others as ways to point out how not to make the same mistakes the creators of Java did. And he should know - he is one of them.

  One of the interesting things about the Effective C++ series is that you can see how the design of the language has evolved over the last 15 or 20 years. For example:

  - Effective C++ says to prefer new & delete over malloc & free
  - More Effective C++ says to prefer smart pointers over new & delete
- https://www.open-std.org/ is a bare bones site with a huge amount of detailed information related to open standards: C, C++, Python, Vulnerability information, and more.

- Working Well with Legacy Code

  A clear, approachable article providing insights and strategies for dealing with legacy codebases. Thanks to Tracy Cavanaugh and the young ladies of Mrs. C's High School Club for this recommendation.

  Their suggestion is timely because legacy code has been on my mind lately.

  Working with legacy code matters because it represents the accumulated knowledge, decisions, and solutions of the past. Mastering it allows programmers to preserve functionality, reduce technical debt, and build a foundation for future innovation. By understanding and improving existing systems, developers gain insight into software design, architecture, and maintainability which are valuable skills on the journey from apprentice to mastery.

4.1.5 What if I need help in this class right now?

You think my book needs help. I'm sorry to hear that, but I understand this text may not work for everyone.

You need help, what other resources are out there?

First consider your first semester C++ textbook. It may not cover everything in this course, but it will certainly provide review of fundamentals if you need that.

Every page in the textbook has a "More to Explore" section with links related to the material on that page. I admit that many of these links are "cppreference.com heavy" and *reference* and not exactly a tutorial. However, the links are still worth checking out, especially if you are currently reading a page that is giving you trouble.

There is a first semester textbook I am working on for C++. I trust it as far as it goes, but it is incomplete.

> https://daveparillo.github.io/cisc192-reader/

Videos

I have a love hate relationship. I can't learn anything from them, but I get everyone learns differently. Also, it bugs me that people always are trying to sell you something on YouTube. That said, here are some video sources that I trust. Some are linked from my textbook:

- Bucky's C++ Tutorials: https://www.youtube.com/playlist?list=PLAE85DE8440AA6B83
- My Code School C++ Pointers: https://www.youtube.com/playlist?list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_ There is a C playlist that has a lot to offer: https://www.youtube.com/playlist?list=PL2_aWCzGMAwLSqGsERZGXGkA5AfMhcknE
- The Cherno is a game developer from Australia with a C++ channel: https://www.youtube.com/watch?v=18c3MTX0PK0
- Jason Turner C++ weekly: https://www.youtube.com/user/lefticus1/videos
  - These are a bit more random as opposed to a linear tutorial, but this guy is incredibly smart. But this is an area where you really do not want to try to see them all. It may be too overwhelming and some advanced concepts are mixed in with the basics. Just try one on for size and if it isn't helping, then forget it.

All these videos are usually short, single topic tutorials.

Other resources

I highly recommend trying out excerism.io. You can solve simple problems in C++ and many other languages too. The problems are designed like a ladder with gradually increasing difficulty.

If you learn of anything particularly useful to you, please share!

4.1.6 What major should I pick?

People feel a lot pressure to pick the 'perfect major': one that is guaranteed to get them a great job, doing something they love, with an excellent salary.

I can't speak to all options, but if your interest is *programming*, then relax.

Many students agonize over the distinctions between computer engineering, computer science, and software engineering,

There are no wrong answers. Careers related to computing are so diverse and widespread, there is a home for everyone regardless of your interests and talents.

Computer science is an extremely broad field. In school, the emphasis tends to be on underlying science, abstract concepts, and theory.

Closely related, software engineering tends to focus on large scale systems. More emphasis is placed on practices and techniques from other engineering disciplines.

Computer engineering tends to involve more hardware as well as software, and the development of software that involves hardware/software interactions.

If your interest is writing programs, you can major in any of these fields and get an excellent and fun job.

In my experience, programming is a *meritocracy*: talented individuals are rewarded for their effort. One of the most successful programmers I know was a dual major in English and Philosophy. Not everyone can carve out a great career in computer science as a liberal arts major, or by dropping out completely, but it happens.

The primary common characteristic shared by all successful people in this field is that they worked their tails off for years, often decades, to get where they are today.

The point is, worry less about the exact major and more about the effort you put into it.

It turns out that it takes about 10 years to truly master anything. A short blog post about it and resisting the urge to how to teach yourself programming in 21 days.

More to Explore

- Consider the Goshen College quiz if you really have **no idea** what you want to do and aren't even sure computers are your thing.

4.1.7 Do I need to know math?

Math is a broad topic.

Nearly every programming statement in any language is either a *logic expression* or an *algebraic statement*. A firm grasp of boolean expressions, basic (high school) algebra, and an ability to think through problems is critical. Programmers avoid this material at their peril.

Don't panic. Do most programmers spend their days working through advanced calculus problems? No.

The final answer ultimately depends on what kind of programs you write. If you are writing game graphics or physics engines, numerical modeling, complex financial analysis, or something similar, then yes, math is critical.

If you spend a lot of time in a specialized domain like these, then you may find a physics degree as useful as a math or computer science degree. Otherwise, a solid understanding of what you learned in high school will suffice most of the time.

One last word:

In my experience, programmers who know more math, or who have specialized domain expertise are more highly compensated than those who don't

---

More to Explore

- ISO C++ FAQ - this FAQ also includes everything from the old Marshall Cline FAQ at Parasoft
- Bjarne Stroustrup's C++11 FAQ
- cppreference.com's C++ FAQ
- stackoverflow.com's C++ FAQ
- Microsoft's C++ standard library FAQ

4.2 ASCII Character Set

ASCII is the American Standard Code for Information Interchange. It is a 7-bit code. Many 8-bit codes for example, ISO 8859-1, contain ASCII as their lower half. The international counterpart of ASCII is known as ISO 646-IRV.

The following table contains the 128 ASCII characters.

C program 'X' escapes are noted.

Oct	Dec	Hex	Char	Description	Oct	Dec	Hex	Char
000	0	00	NUL	'0' (null character)	100	64	40	@
001	1	01	SOH	(start of heading)	101	65	41	A
002	2	02	STX	(start of text)	102	66	42	B
003	3	03	ETX	(end of text)	103	67	43	C
004	4	04	EOT	(end of transmission)	104	68	44	D
005	5	05	ENQ	(enquiry)	105	69	45	E
006	6	06	ACK	(acknowledge)	106	70	46	F
007	7	07	BEL	'a' (bell)	107	71	47	G
010	8	08	BS	'b' (backspace)	110	72	48	H
011	9	09	HT	't' (horizontal tab)	111	73	49	I
012	10	0A	LF	'n' (new line)	112	74	4A	J
013	11	0B	VT	'v' (vertical tab)	113	75	4B	K

continues on next page

Table 1 – continued from previous page

Oct	Dec	Hex	Char	Description	Oct	Dec	Hex	Char
014	12	0C	FF	'f' (form feed)	114	76	4C	L
015	13	0D	CR	'r' (carriage ret)	115	77	4D	M
016	14	0E	SO	(shift out)	116	78	4E	N
017	15	0F	SI	(shift in)	117	79	4F	O
020	16	10	DLE	(data link escape)	120	80	50	P
021	17	11	DC1	(device control 1)	121	81	51	Q
022	18	12	DC2	(device control 2)	122	82	52	R
023	19	13	DC3	(device control 3)	123	83	53	S
024	20	14	DC4	(device control 4)	124	84	54	T
025	21	15	NAK	(negative ack.)	125	85	55	U
026	22	16	SYN	(synchronous idle)	126	86	56	V
027	23	17	ETB	(end of trans. Blake)	127	87	57	W
030	24	18	CAN	(cancel)	130	88	58	X
031	25	19	EM	(end of medium)	131	89	59	Y
032	26	1A	SUB	(substitute)	132	90	5A	Z
033	27	1B	ESC	(escape)	133	91	5B	[
034	28	1C	FS	(file separator)	134	92	5C	\
035	29	1D	GS	(group separator)	135	93	5D	]
036	30	1E	RS	(record separator)	136	94	5E	^
037	31	1F	US	(unit separator)	137	95	5F	_
040	32	20	SPACE		140	96	60	`
041	33	21	!		141	97	61	a
042	34	22	"		142	98	62	b
043	35	23	#		143	99	63	c
044	36	24	$		144	100	64	d
045	37	25	%		145	101	65	e
046	38	26	&		146	102	66	f
047	39	27	'		147	103	67	g
050	40	28	(		150	104	68	h
051	41	29	)		151	105	69	i

continues on next page

Table 1 – continued from previous page

Oct	Dec	Hex	Char	Description	Oct	Dec	Hex	Char
052	42	2A	•		152	106	6A	j
053	43	2B	•		153	107	6B	k
054	44	2C	,		154	108	6C	l
055	45	2D	•		155	109	6D	m
056	46	2E	.		156	110	6E	n
057	47	2F	/		157	111	6F	o
060	48	30	0		160	112	70	p
061	49	31	1		161	113	71	q
062	50	32	2		162	114	72	r
063	51	33	3		163	115	73	s
064	52	34	4		164	116	74	t
065	53	35	5		165	117	75	u
066	54	36	6		166	118	76	v
067	55	37	7		167	119	77	w
070	56	38	8		170	120	78	x
071	57	39	9		171	121	79	y
072	58	3A	:		172	122	7A	z
073	59	3B	;		173	123	7B	{
074	60	3C	<		174	124	7C	|
075	61	3D	=		175	125	7D	}
076	62	3E	>		176	126	7E	~
077	63	3F	?		177	127	7F	DEL

For convenience, below are more compact tables in hex and decimal.

```
   2 3 4 5 6 7        30 40 50 60 70 80 90 100 110 120
 -------------      ---------------------------------
0:   0 @ P ` p     0:    (  2  <  F  P  Z  d   n   x
1: ! 1 A Q a q     1:    )  3  =  G  Q  [  e   o   y
2: " 2 B R b r     2:    *  4  >  H  R  \  f   p   z
3: # 3 C S c s     3: !  +  5  ?  I  S  ]  g   q   {
4: $ 4 D T d t     4: "  ,  6  @  J  T  ^  h   r   |
5: % 5 E U e u     5: #  -  7  A  K  U  _  i   s   }
6: & 6 F V f v     6: $  .  8  B  L  V  `  j   t   ~
7: ' 7 G W g w     7: %  /  9  C  M  W  a  k   u  DEL
8: ( 8 H X h x     8: &  0  :  D  N  X  b  l   v
9: ) 9 I Y i y     9: '  1  ;  E  O  Y  c  m   w
A: * : J Z j z
B: + ; K [ k {
C: , < L \ l |
D: - = M ] m }
E: . > N ^ n ~
   F: / ? O _ o DEL
```

More to Explore

- ASCII Chart from ccpreference.com

4.3 Glossary

abstract base class
A class in which some functions are not implemented. Abstract bases classes cannot be instantiated --- a derived class must override the abstract virtual function with an implementation.

abstract data type
Abbreviated *ADT*. The realization of a *data type* as a software component.

abstraction
A technique used to reduce the complexity of systems and data. An abstraction often involves a simplified model of the 'real world' for the purposes of achieving goals within a specific problem domain.

Data abstraction enforces a clear separation between the abstract properties of a *data type* and the concrete details of its implementation.

activation record
The entity that is stored on the *runtime stack* during program execution. It stores any active *local variable* and the return address from which a new subroutine is being called, so that this information can be recovered when the subroutine terminates.

activecode
A unique interpreter environment that allows code to be executed within a web browser.

address
A location in memory.

adjacency list
An implementation for a *graph* that uses an (array-based) *list* to represent the *vertices* of the graph, and each vertex is in turn represented by a (linked) list of the vertices that are *neighbors*.

adjacency matrix
An implementation for a *graph* that uses a 2-dimensional array where each row and each column corresponds to a *vertex* in the *graph*. A given row and column in the matrix corresponds to an edge from the *vertex* corresponding to the row to the vertex corresponding to the column.

adjacent
Two *nodes* of a *tree* or two *vertices* of a *graph* are said to be adjacent if they have an *edge* connecting them. If the edge is directed from a to b, then we say that a is adjacent to b, and b is adjacent from a.

ADT
Abbreviation for *abstract data type*.

adversary argument
A type of *lower bounds proof* for a problem where a (fictional) "adversary" is assumed to control access to an algorithm's input, and which yields information about that input in such a way that will drive the cost for any proposed algorithm to solve the problem as high as possible. So long as the adversary never gives an answer that conflicts with any previous answer, it is permitted to do whatever necessary to make the algorithm require as much cost as possible.

aggregate type
A *data type* whose *members* have subparts. For example, a typical database record. Another term for this is *composite type*.

algorithm
A general step by step process for solving a *problem*.

algorithm analysis

to-term
growth rate :label: key concept

to-term
upper bound :label: key concept

to-term
lower bound :label: key concept

to-term
asymptotic analysis :label: synonym

to-term
asymptotic algorithm analysis :label: formal synonym

A less formal version of the term *asymptotic algorithm analysis*, generally used as a synonym for *asymptotic analysis*.

alphabet trie
A *trie* data structure for storing variable-length strings. Level i of the tree corresponds to the letter in position i of the string. The root will have potential branches on each intial letter of string. Thus, all strings starting with "a" will be stored in the "a" branch of the tree. At the second level, such strings will be separated by branching on the second letter.

ancestor
In a tree, for a given node A, any node on a path from A up to the root is an ancestor of A.

anti-pattern
A common response to a recurring problem that is generally ineffective. Anti-patterns represent examples that you **should not** emulate! As bad as they are, they can still be instructive. Compare to *design pattern*.

antisymmetric
In set notation, relation R is antisymmetric if whenever aRb and bRa, then $a = b$, for all $a, b \in \mathbf{S}$.

api
API
An Application Programming Interface (API) is a set of functions, or classes used by a program. Often an API provides a family of functions or classes that work together to provide a complete set of capabilities.

approximation algorithm
An algorthm for an *optimization problem* that finds a good, but not necessarily cheapest, solution.

array-based list
An implementation for the *list* ADT that uses an array to store the list elements.

array-based queue
Analogous to an *array-based list*, this uses an array to store the elements when implementing the *queue* ADT.

array-based stack
Analogous to an *array-based list*, this uses an array to store the elements when implementing the *stack* ADT.

ASCII character coding
American Standard Code for Information Interchange. A commonly used method for encoding characters using a binary code. Standard ASCII uses an 8-bit code to represent upper and lower case letters, digits, some punctuation, and some number of non-printing characters (such as carrage return). Now largely replaced by UTF-8 encoding.

assembly code

to-term
intermediate code :label: form of

A form of *intermediate code* created by a *compiler* that is easy to convert into the final form that the computer can execute. An assembly language is typically a direct mapping of one or a few instructions that the CPU can execute into a mnemonic form that is relatively easy for a human to read.

assignable
A *type* is *assignable* if the type can be copy-assigned a new value as the left-hand side of the operation.

References are not assignable because once initialized, they cannot be assigned a new value.

associative container
A set of sorted data structures that can be quickly searched. `map` and `set` are examples.

asymptotic algorithm analysis
A more formal term for *asymptotic analysis*.

asymptotic analysis

to-term
algorithm analysis :label: synonym

to-term
asymptotic algorithm analysis :label: formal synonym

A method for estimating the efficiency of an algorithm or computer program by identifying its *growth rate*. Asymptotic analysis also gives a way to define the inherent difficulty of a *problem*. We frequently use the term *algorithm analysis* to mean the same thing.

attribute
In *object-oriented programming*, a synonym for *data member*.

average case
In *algorithm analysis*, the average of the costs for all *problem instances* of a given input size n. If not all problem instances have equal probability of occurring, then average case must be calculated using a weighted average.

backing storage
backing store
The underlying storage for an *ADT*.

bag
In set notation, a bag is a collection of elements with no order (like a set), but which allows for duplicate-valued elements (unlike a set). A synonym for *multilist*.

balanced tree
A *tree* where the *subtrees* meet some criteria for being balanced. Two possibilities are that the tree is *height balanced*, or that the tree has a roughly equal number of *nodes* in each subtree.

base
Synonym for *radix*.

base case
In *recursion*, the base case is the termination condition. This is a simple input or value that can be solved without resorting to a recursive call.

base class
In *object-oriented programming*, a class from which another class *inherits*. The class that inherits is called a *subclass* or *derived class*.

base type
The *data type* for the elements in a set. For example, the set might consist of the integer values 3, 5, and 7. In this example, the base type is integers.

best case
In algorithm analysis, the *problem instance* from among all problem instances for a given input size n that has least cost. Note that the best case is **not** when n is small, since we are referring to the best from a class of inputs (i.e, we want the best of those inputs of size n).

big-O notation
big-Oh notation
In algorithm analysis, a shorthand notation for describing the upper bound for an algorithm or problem.

binary search
A standard *recursive* algorithm for finding the *record* with a given *key* within a sorted list. It runs in $O(\log n)$ time. At each step, look at the middle of the current sublist, and throw away the half of the records whose keys are either too small or too large.

binary tree
A non-linear data structure with a set of nodes which is either empty, or else has a root node together two binary trees, called the left and right *subtrees*, which are disjoint from each other and from the *root*.

binary trie
A *binary tree* whose structure is that of a *trie*. Generally this is an implementation for a *search tree*. This means that the *search key* values are thought of a binary digits, with the digit in the position corresponding to this a node's *level* in the tree indicating a left branch if it is "0", or a right branch if it is "1". Examples include the *Huffman coding tree* and the *bintree*.

binning
In *hashing*, binning is a type of *hash function*. Say we are given keys in the range 0 to 999, and have a hash table of size 10. In this case, a possible hash function might simply divide the key value by 100. Thus, all keys in the range 0 to 99 would hash to slot 0, keys 100 to 199 would hash to slot 1, and so on. In other words, this hash function "bins" the first 100 keys to the first slot, the next 100 keys to the second slot, and so on. This approach tends to make the hash function dependent on the distribution of the high-order bits of the keys.

bintree

to-term
flyweight :label: uses

A *spatial data structure* in the form of binary *trie*, typically used to store point data in two or more dimensions. Similar to a *PR quadtree* except that at each level, it splits one dimension in half. Since many leaf nodes of the PR quadtree will contain no data points, implementation often makes use of the *flyweight design pattern*.

block
Defines a *scope* within a program. A synonym for *code block*.

boolean variable
A variable that takes on one of the two values `true` and `false`.

bucket
In *bucket hashing*, a bucket is a sequence of *slots* in the *hash table* that are grouped together.

bucket hashing
A method of *hashing* where multiple *slots* of the *hash table* are grouped together to form a *bucket*. The *hash function* then either hashes to some bucket, or else it hashes to a *home slot* in the normal way, but this home slot is part of some bucket. *Collision resolution* is handled first by attempting to find a free position within the same bucket as the home slot. If the bucket if full, then the record is placed in an *overflow bucket*.

bug
An error in a program.

call stack
Known also as execution stack. A stack that stores the function call sequence and the return address for each function.

cartesian product
For sets, this is another name for the *set product*.

ceiling
Written $\lceil x \rceil$, for real value x the ceiling is the least integer $\geq x$.

child
In a tree, the set of *nodes* directly pointed to by a node R are the *children* of R.

class
In the *object-oriented programming paradigm* an ADT and its implementation together make up a class. An instantiation of a class within a program is termed an *object*.

class hierarchy
In *object-oriented programming*, a set of classes and their interrelationships. One of the classes is the *base class*, and the others are *derived classes* that *inherit* either directly or indirectly from the base class.

class invariant
type invariant
invariants
A class invariant is an assertion about conditions which must be true in order for a class to remain valid.

client
The user of a service.

closed
A set is closed over a (binary) operation if, whenever the operation is applied to two members of the set, the result is a member of the set.

closed hashing
closed hash system
A *hash system* where all records are stored in slots of the *hash table*. This is in contrast to an *open hash system*.

closed-form solution
An algebraic equation with the same value as a *summation* or *recurrence relation*. The process of replacing the summation or recurrence with its closed-form solution is known as solving the summation or recurrence.

code block
Defines a *scope* within a program. A synonym for *block*.

code generation
A phase in a *compiler* that transforms *intermediate code* into the final executable form of the code. More generally, this can refer to the process of turning a parse tree (that determines the correctness of the structure of the program) into actual instructions that the computer can execute.

code optimization

> **to-term**
> assembly code :label: changes

A phase in a *compiler* that makes changes in the code (typically *assembly code*) with the goal of replacing it with a version of the code that will run faster while performing the same computation.

codelens
An interactive environment that allows the user to control the step by step execution of a program

cohesion
In *object-oriented programming*, a term that refers to the degree to which a class has a single well-defined role or responsibility.

collision
In a *hash system*, this refers to the case where two search *keys* are mapped by the *hash function* to the same slot in the *hash table*. This can happen on insertion or search when another record has already been hashed to that slot. In this case, a *closed hash system* will require a process known as *collision resolution* to find the location of the desired record.

collision resolution
The outcome of a *collision resolution policy*.

collision resolution policy
In *hashing*, the process of resolving a *collision*. Specifically in a *closed hash system*, this is the process of finding the proper position in a *hash table* that contains the desired record if the *hash function* did not return the correct position for that record due to a *collision* with another record.

comment
Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

comparable
The concept that two objects can be compared to determine if they are equal or not, or to determine which one is greater than the other. In set notation, elements x and y of a set are comparable under a given relation R if either xRy or yRx. To be reliably compared for a greater/lesser relationship, the values being compared must belong to a *total order*. In programming, the property of a data type such that two elements of the type can be compared to determine if they the same (a weaker version), or which of the two is larger (a stronger version).

comparator
A function given as a parameter to a method of a library (or alternatively, a parameter for a C++ template or a Java generic). The comparator function concept provides a generic way encapulates the process of performing a comparison between two objects of a specific type. For example, if we want to write a generic sorting routine, that can handle any record type, we can require that the user of the sorting routine pass in a comparator function to define how records in the collection are to be compared.

comparison
The act of comparing two *keys* or *records*. For many *data types*, a comparison has constant time cost. For others, such as *linked list* the cost often increases as the number of elements increases.

comparison sort
A class of algorithms that sort data by comparing pairs of elements to determine their relative order, using operations like "less than" or "greater than" to decide which element comes first, with examples including *Merge sort*, *Quick sort*, and *Insertion sort*, all fundamentally limited to $\Omega(n \log n)$ time complexity in the worst case because they rely on these pairwise comparisons.

The act of comparing two *keys* or *records*. For many *data types*, a comparison has constant time cost. For others, such as *linked list* the cost often increases as the number of elements increases.

compile
To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.

compile-time error
Errors detected by the compiler. Compare to *runtime error*, *link error*, and *semantic error*.

compile-time polymorphism
Compile-time polymorphism
A form of *polymorphism* known as Overloading. Overloaded methods have the same names, but different signatures as a method available elsewhere in the class. Compare to *runtime polymorphism*.

compiler
A computer program that reads computer programs and converts them into a form that can be directly excecuted by some form of computer. The major phases in a compiler include *lexical analysis*, *syntax analysis*, *intermediate code generation*, *code optimization*, and *code generation*. More broadly, a compiler can be viewed as *parsing* the program to verify that it is syntactically correct, and then doing *code generation* to convert the hig-level program into something that the computer can execute.

complete binary tree
A binary tree where the nodes are filled in row by row, with the bottom row filled in left to right. Due to this

requirement, there is only one tree of n nodes for any value of n. Since storing the records in an array in row order leads to a simple mapping from a node's position in the array to its *parent*, *siblings*, and *children*, the array representation is most commonly used to implement the complete binary tree. The *heap* data structure is a complete binary tree with partial ordering constraints on the node values.

Composite design pattern
Given a class hierarchy representing a set of objects, and a container for a collection of objects, the composite *design pattern* addresses the relationship between the object hierarchy and a bunch of behaviors on the objects. In the composite design, each object is required to implement the collection of behaviors. This is in contrast to the procedural approach where a behavior (such as a tree *traversal*) is implemented as a method on the object collection (such as a *tree*). Procedural tree traversal requires that the tree have a method that understands what to do when it encounters any of the object types (*internal* or *leaf nodes*) that the tree might contain. The composite approach would have the tree call the "traversal" method on its root node, which then knows how to perform the "traversal" behavior. This might in turn require invoking the traversal method of other objects (in this case, the children of the root).

composite type
A type whose *members* have subparts. For example, a typical database record. Another term for this is *aggregate type*.

composition
Relationships between classes based on usage rather than *inheritance*, i.e. a **HAS-A** relationship. For example, some code in class 'A' has a reference to some other class 'B'.

compound type
A *data type* built up from simpler parts. Compare to *simple type* and *composite type*.

constant running time
constant time
The cost of a function whose running time is not related to its input size. In Theta notation, this is traditionally written as $\Theta(1)$.

container
container class
A *data structure* that stores a collection of *records*. Typical examples are arrays and *hash tables*.

cost
The amount of resources that the solution consumes.

cost model
In *algorithm analysis*, a definition for the cost of each basic operation performed by the algorithm, along with a definition for the size of the input. Having these definitions allows us to calculate the *cost* to run the algorithm on a given input, and from there determine the *growth rate* of the algorithm. A cost model would be considered "good" if it yields predictions that conform to our understanding of reality.

CPU
Acronym for Central Processing Unit, the primary processing device for a computer.

current position
A property of some list ADTs, where there is maintained a "current position" state that can be referred to later.

data field
In *object-oriented programming*, a synonym for *data member*.

data item
A piece of information or a record whose value is drawn from a type.

data member
The variables that together define the space required by a data item are referred to as data members. Some of the commonly used synonyms include *data field*, *attribute*, and *instance variable*.

data structure
The implementation for an *ADT*.

data type
A type together with a collection of operations to manipulate the type.

debugging
The process of finding and removing any of the three kinds of programming errors.

decision tree
A theoretical construct for modeling the behavior of algorithms. Each point at which the algorithm makes a decision (such as an if statement) is modeled by a branch in the tree that represents the algorithms behavior. Decision trees can be used in *lower bounds proofs*, such as the proof that sorting requires $\Omega(n \log n)$ comparisons in the *worst case*.

declaration
A declaration introduces a new *name* and *type* into a *scope*.

depth
The depth of a node M in a tree is the length of the path from the root of the tree to M.

depth-first search

to-term
DFS :label: abbreviation

to-term
depth-first search tree :label: generates

A *graph traversal* algorithm. Whenever a v is *visited* during the traversal, DFS will *recursively* visit all of v 's *unvisited neighbors*.

depth-first search tree
A *tree* that can be defined by the operation of a *depth-first search* (DFS) on a *graph*. This tree would consist of the *nodes* of the graph and a subset of the *edges* of the graph that was followed during the DFS.

dequeue
A specialized term used to indicate removing an element from a queue.

derivation
In formal languages, the process of executing a series of *production rules* from a *grammar*. A typical example of a derivation would be the series of productions executed to go from the *start symbol* to a given string.

derived class
In *object-oriented programming*, any class within a *class hierarchy* that *inherits* from some other class. A synonym for *derived class*.

descendant
In a tree, the set of all nodes that have a node A as an *ancestor* are the descendants of A. In other words, all of the nodes that can be reached from A by progressing downwards in tree. Another way to say it is: The *children* of A, their children, and so on.

deserialization
The process of returning a *serialized* representation for a data structure back to its original in-memory form.

design pattern
An abstraction for describing the design of programs, that is, the interactions of objects and classes. Experienced software designers learn and reuse patterns for combining software components, and design patterns allow this design knowledge to be passed on to new programmers more quickly. Examples are *Composite design pattern*, *flyweight*, *iterator*, *strategy*, and *visitor*.

dictionary
An abstract data type or interface for a data structure or software subsystem that supports insertion, search, and deletion of records.

discriminator
A part of a *multi-dimensional search key*. Certain tree data structures such as the *bintree* and the *kd tree* operate by making branching decisions at nodes of the tree based on a single attribute of the multi-dimensional key, with the attribute determined by the level of the node in the tree.

For example, in 2 dimensions, nodes at the odd levels in the tree might branch based on the x value of a coordinate, while at the even levels the tree would branch based on the y value of the coordinate. Thus, the x coordinate is the discriminator for the odd levels, while the y coordinate is the discriminator for the even levels.

disjoint
Two parts of a *data structure* or two collections with no objects in common are disjoint. This term is often used in conjunction with a data structure that has *nodes* (such as a *tree*). Also used in the context of *sets*, where two *subsets* are disjoint if they share no elements.

disjoint sets
A collection of *sets*, any pair of which share no elements in common. A collection of disjoint sets partitions some objects such that every object is in exactly one of the disjoint sets.

divide and conquer
A technique for designing algorithms where a solution is found by breaking the problem into smaller (similar) subproblems, solving the subproblems, then combining the subproblem solutions to form the solution to the original problem. This process is often implemented using *recursion*.

divide-and-conquer recurrences
A common form of *recurrence relation* that have the form

$$\mathbf{T}(n) = a\mathbf{T}(n/b) + cn^k; \quad \mathbf{T}(1) = c$$

where a, b, c, and k are constants. In general, this recurrence describes a problem of size n divided into a subproblems of size n/b, while cn^k is the amount of work necessary to combine the partial solutions.

domain
The set of possible inputs to a function.

double hashing
A *collision resolution* method. A second hash function is used to generate a value c on the key. That value is then used by this key as the step size in *linear probing by steps*. Since different keys use different step sizes (as generated by the second hash function), this process avoids the clustering caused by standard linear probing by steps.

doubly linked list
A *linked list* implementation variant where each list node contains access pointers to both the previous element and the next element on the list.

dynamic array
Arrays, once allocated, are of fixed size. A dynamic array puts an interface around the array so as to appear to allow the array to grow and shrink in size as necessary. Typically this is done by allocating a new copy, copying the contents of the old array, and then returning the old array to *free store*. In some programming languages, the term *vector* is used as a synonym for dynamic array.

dynamic memory allocation
A programming technique where linked objects in a data structure are created from *free store* as needed. When no longer needed, the object is either returned to *free store* or left as *garbage*, depending on the programming language.

edge
The connection that links two *nodes* in a *tree*, *linked list*, or *graph*.

element
One value or member in a set.

empty
For a *container* class, the state of containing no *elements*.

encapsulation
In programming, the concept of hiding implementation details from the user of an ADT, and protecting *data members* of an object from outside access.

enqueue
A specialized term used to indicate inserting an element onto a queue.

enumeration
The process by which a *traversal* lists every object in the *container* exactly once. Thus, a traversal that prints the *nodes* is said to enumerate the nodes. An enumeration can also refer to the actual listing that is produced by the traversal (as well as the process that created that listing).

equivalence class
An *equivalence relation* can be used to partition a set into equivalence classes.

equivalence relation
Relation R is an equivalence relation on set $\mathbf{S}$ if it is *reflexive*, *symmetric*, and *transitive*.

exception
Another name for a runtime error.

exchange
A swap of adjacent records in an array.

executable
Another name for object code that is ready to be executed.

exponential growth rate
A *growth rate* function where n (the input size) appears in the exponent. For example, 2^n.

external fragmentation
A condition that arises when a series of memory requests result in lots of small *free blocks*, no one of which is useful for servicing typical requests.

external sort
A sorting algorithm that is applied to data stored outside the program, such as a disk file. This is in contrast to an *internal sort* that is meant to work on data stored in memory.

FIFO
Abbreviation for "first-in, first-out". This is the access paradigm for a *queue*, and an old terminolgy for the queue is "FIFO list".

fixed-length coding
Given a collection of objects, a fixed-length coding scheme assigns a code to each object in the collection using codes that are all of the same length. Standard ASCII and Unicode representations for characters are both examples of fixed-length coding schemes. This is in contrast to *variable-length coding*.

floor
Written $\lfloor x \rfloor$, for real value x the floor is the greatest integer $\leq x$.

flyweight
A *design pattern* that is meant to solve the following problem: You have an application with many objects. Some of these objects are identical in the information that they contain, and the role that they play. But they must be reached from various places, and conceptually they really are distinct objects. Because there is so much duplication of the same information, we want to reduce memory cost by sharing that space. For example, in document layout, the letter "C" might be represented by an object that describes that character's strokes and

bounding box. However, we do not want to create a separate "C" object everywhere in the document that a "C" appears. The solution is to allocate a single copy of the shared representation for "C" objects. Then, every place in the document that needs a "C" in a given font, size, and typeface will reference this single copy. The various instances of references to a specific form of "C" are called flyweights.

folding method
: In *hashing*, an approach to implementing a *hash function*. Most typically used when the key is a string, the folding method breaks the string into pieces (perhaps each letter is a piece, or a small series of letters is a piece), converts the letter(s) to an integer value (typically by using its underlying encoding value), and summing up the pieces.

free block
: A block of unused space in a memory pool.

free block list
: In a memory manager, the list that stores the necessary information about the current *free blocks*. Generally, this is done with some sort of *linked list*, where each node of the linked list indicates the start position and length of the free block in the memory pool.

free store
: Space available to a program during runtime to be used for *dynamic memory allocation* of objects. The free store is distinct from the *runtime stack*. The free store is sometimes referred to as the *heap*, which can be confusing because *heap* more often refers to a specific data structure. Most programming languages provide functions to allocate (and maybe to deallocate) objects from the free store, such as `new` in C++ and Java.

full tree
: A *binary tree* is full if every *node* is either a *leaf node* or else it is an *internal node* with two non-empty *children*.

function
: In programming, a subroutine that takes input parameters and uses them to compute and return a value. In this case, it is usually considered bad practice for a function to change any global variables (doing so is called a side effect).

fundamental type
: One of the *simple types* provided by the language. Examples are `bool`, `char`, `int`, and `double`. Types provided by the STL, such as `std::string` and `std::vector` are not considered 'fundamental' types.

garbage
: In memory management, any memory that was previously (dynamically) allocated by the program during runtime, but which is no longer accessible since all pointers to the memory have been deleted or overwritten. In some languages, garbage can be recovered by *garbage collection*. In languages such as C and C++ that do not provide built-in garbage collection, so creating garbage is considered a *memory leak*.

garbage collection
: Languages with garbage collection such Java, JavaScript, Lisp, and Scheme will periodically reclaim *garbage* and return it to *free store*.

general tree
: A tree in which any given node can have any number of *children*. This is in contrast to, for example, a *binary tree* where each node has a fixed number of children (some of which might be `null`). General tree nodes tend to be harder to implement for this reason.

generic programming
: A computer programming style in which functions are written using *placeholders for* types. In C++ this is accomplished using :term:templates<template>`. Templates are used to create actual functions for specific types as needed.

grammar
: A formal definition for what strings make up a *language*, in terms of a set of *production rules*.

graph
A *graph* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ consists of a set of *vertices* $\mathbf{V}$ and a set of *edges* $\mathbf{E}$, such that each edge in $\mathbf{E}$ is a connection between a pair of vertices in $\mathbf{V}$.

greedy algorithm
An algorithm that makes locally optimal choices at each step.

growth rate

to-term
lower bound :label: type

to-term
upper bound :label: type

In *algorithm analysis*, the rate at which the cost of the *algorithm* grows as the size of its input grows.

hash function
In a *hash system*, the function that converts a *key* value to a position in the *hash table*. The hope is that this position in the hash table contains the record that matches the key value.

hash system
The implementation for search based on hash lookup in a *hash table*. The *key* is processed by a *hash function*, which returns a position in a *hash table*, which hopefully is the correct position in which to find the record corresponding to the search key.

hash table
The data structure (usually an array) that stores data records for lookup using *hashing*.

hashing
A search method that uses a *hash function* to convert a *key* into a position within a *hash table*. In a properly implemented *hash system*, that position in the table will have high probability of containing the record that matches the key value. Sometimes, the hash function will return an position that does not store the desired key, due to a process called *collision*. In that case, the desired record is found through a process known as *collision resolution*.

head
The beginning of a *list*.

header guard
In C and C++, used to prevent definitions copied into a file using the `#include` directive from being defined more than once.

```
#ifndef FOO_H_INCLUDED // any name uniquely mapped to file name
#define FOO_H_INCLUDED
// contents of the file are here
#endif
```

header node
Commonly used in implementations for a *linked list* or related structure, this *node* precedes the first element of the list. Its purpose is to simplify the code implementation by reducing the number of special cases that must be programmed for.

heap
This term has two different meanings. Sometimes, it is a synonym for *free store*.

A heap may also refer to a particular data structure. This data structure is a *complete binary tree* with the requirement that every *node* has a value greater than its *children* (called a *max heap*), or else the requirement that every node has a value less than its children (called a *min heap*). Since it is a complete binary tree, a heap is nearly always implemented using an array rather than an explicit tree structure. To add a new value to a heap, or to remove the extreme value (the max value in a max-heap or min value in a min-heap) and update the heap,

takes $\Theta(\log n)$ time in the worst case. However, if given all of the values in an unordered array, the values can be re-arranged to form a heap in only $\Theta(n)$ time. Due to its space and time efficiency, the heap is a popular choice for implementing a *priority queue*.

height
The height of a tree is one more than the *depth* of the deepest *node* in the tree.

height balanced
The condition the *depths* of each *subtree* in a tree are roughly the same.

heuristic
A way to solve a problem that is not guarenteed to be optimal. While it might not be guarenteed to be optimal, it is generally expected (by the agent employing the heuristic) to provide a reasonably efficient solution.

heuristic algorithm
A type of *approximation algorithm*, that uses a *heuristic* to find a good, but not necessarily cheapest, solution to an *optimization problem*.

high-level language
A programming language that is designed to be easy for humans to read and write.

home position
In *hashing*, a synonym for *home slot*.

home slot
In *hashing*, this is the *slot* in the *hash table* determined for a given key by the *hash function*.

Huffman codes
The codes given to a collection of letters (or other symbols) through the process of Huffman coding. Huffman coding uses a *Huffman coding tree* to generate the codes. The codes can be of variable length, such that the letters which are expected to appear most frequently are shorter. Huffman coding is optimal whenever the true frequencies are known, and the frequency of a letter is independent of the context of that letter in the message.

Huffman coding tree
A Huffman coding tree is a *full binary tree* that is used to represent letters (or other symbols) efficiently. Each letter is associated with a node in the tree, and is then given a *Huffman code* based on the position of the associated node. A Huffman coding tree is an example of a binary *trie*.

Huffman tree
Shorter form of the term *Huffman coding tree*.

identifier
An identifier is used to name a type introduced into a program by a *declaration*.

An identifier is an arbitrarily long sequence of digits, underscores, lowercase and uppercase Latin letters. A valid identifier must begin with a non-digit character (Latin letter, underscore, Identifiers are case-sensitive (lowercase and uppercase letters are distinct), and every character is significant.

image-space decomposition
A from of *key-space decomposition* where the *key space* splitting points is predetermined (typically by splitting in half). For example, a *Huffman coding tree* splits the letters being coded into those with codes that start with 0 on the left side, and those with codes that start with 1 on the right side. This regular decomposition of the key space is the basis for a *trie* data structure. An image-space decomposition is in opposition to an *object-space decomposition*.

indexing
The process of associating a *search key* with the location of a corresponding data record. The two defining points to the concept of an index is the association of a key with a record, and the fact that the index does not actually store the record itself but rather it stores a *reference* to the record.

In this way, a collection of records can be supported by multiple indices, typically a separate index for each key field in the record.

induction step
Part of a *proof by induction*. In its simplest form, this is a proof of the implication that if the theorem holds for $n-1$, then it holds for n. As an alternative, see *strong induction*.

inherit
In *object-oriented programming*, the process by which a *subclass* gains *data members* and *methods* from a *base class*.

inorder traversal
In a *binary tree*, a *traversal* that first *recursively visits* the left *child*, then visits the *root*, an then recursively visits the right child.

instance variable
In *object-oriented programming*, a synonym for *data member*.

integrated development environment
IDE
A software suite that consolidates many tools developers need to write and test software. An IDE normally consists of a source code editor, version control, build automation tools, and a debugger. Most also automatically complete partially typed keywords and create commonly used code from templates.

interface
An interface is a class-like structure that only contains method signatures and fields. An interface does not contain an implementation of the methods or any *data members*.

intermediate code
A step in a typical *compiler* is to transform the original high-level language into a form on which it is easier to do other stages of the process. For example, some compilers will transform the original high-level source code into *assembly code* on which it can do *code optimization*, before translating it into its final executable form.

intermediate code generation

to-term
Parse tree :label: walks through

to-term
intermediate code :label: produces

A phase in a *compiler*, that walks through a *parse tree* to produce simple *assembly code*.

internal fragmentation
A condition that occurs when more than N bytes are allocated to service a memory request for N bytes, wasting free storage. This is often done to simplify memory management.

internal node
In a tree, any node that has at least one non-empty *child* is an internal node.

internal sort
A sorting algorithm that is applied to data stored in memory. This is in contrast to an *external sort* that is meant to work on data stored on disk.

interpreter
In contrast to a *compiler* that translates a high-level program into something that can be repeatedly executed to perform a computation, an interpreter directly performs computation on the high-level langauge.

This tends to make the computation much slower than if it were performed on the directly executable version produced by a compiler.

irreflexive
In set notation, binary relation R on set S is irreflexive if aRa is never in the relation for any $a \in \mathbf{S}$.

iterable
A *container* in which each element can be visited using an *iterator*.

iterator
In a *container* such as a vector or set, a separate class that indicates position within the container, with support for *traversing* through all *elements* in the container.

kd tree

to-term
discriminator :label: uses

A *spatial data structure* that uses a binary tree to store a collection of data records based on their (point) location in space. It uses the concept of a *discriminator* at each level to decide which single component of the *multi-dimensional search key* to branch on at that level. It uses a *key-space decomposition*, meaning that all data records in the left subtree of a node have a value on the corresponding discriminator that is less than that of the node, while all data records in the right subtree have a greater value.

key

to-term
key space :label: has

A field or part of a larger record used to represent that record for the purpose of searching or comparing.

key sort

to-term
key :label: uses

Any sorting operation applied to a collection of *key-value pairs* where the value in this case is a *reference* to a complete record (that is, a pointer to the record in memory or a position for a record on disk). This is in contrast to a sorting operation that works directly on a collection of records. The intention is that the collection of key-value pairs is far smaller than the collection of records themselves. As such, this might allow for an *internal sort* when sorting the records directly would require an *external sort*. The collection of key-value pairs can also act as an *index*.

key space
The range of values that a *key* value may take on.

key-space decomposition

to-term
object-space decomposition :label: type

to-term
image-space decomposition :label: type

The idea that the range for a *search key* will be split into pieces. There are two general approaches to this: *object-space decomposition* and *image-space decomposition*.

key-value pair
A standard solution for solving the problem of how to relate a *key* value to a record (or how to find the key for a given record) within the context of a particular index. The idea is to simply store as records in the index pairs of keys and records. Specifically, the index will typically store a copy of the key along with a reference to the record. The other standard solution to this problem is to pass a *comparator* function to the index.

language
A set of strings with specific meanings.

leaf node
In a *binary tree*, leaf node is any node that has two empty *children*. (Note that a binary tree is defined so that every node has two children, and that is why the leaf node has to have two empty children, rather than no children.) In a general tree, any node is a leaf node if it has no children.

level
In a tree, all nodes of *depth* d are at level d in the tree. The root is the only node at level 0, and its depth is 0.

lexical analysis

to-term
interpreter :label: is

A phase of a *compiler* or *interpreter* responsible for reading in characters of the program or language and grouping them into *tokens*.

lexical scoping
Within programming languages, the convention of allowing access to a variable only within the block of code in which the variable is defined. A synonym for static scoping.

lifetime
For a variable, lifetime is the amount of time it will exist before it is destroyed.

LIFO
Abbreviation for "Last-In, First-Out". This is the access paradigm for a *stack*, and an old terminolgy for the stack is "LIFO list".

linear growth rate
For input size n, a growth rate of cn (for c any positive constant). In other words, the cost of the associated function is linear on the input size.

linear order
Another term for *total order*.

linear probing
In *hashing*, this is the simplest *collision resolution* method. Term i of the *probe sequence* is simply i, meaning that collision resolution works by moving sequentially through the hash table from the *home slot*. While simple, it is also inefficient, since it quickly leads to certain free *slots* in the hash table having higher probability of being selected during insertion or search.

linear probing by steps
In *hashing*, this *collision resolution* method is a variation on simple *linear probing*. Some constant c is defined such that term i of the *probe sequence* is ci. This means that collision resolution works by moving sequentially through the hash table from the *home slot* in steps of size c. While not much improvement on linear probing, it forms the basis of another collision resolution method called *double hashing*, where each key uses a value for c defined by a second *hash function*.

link error
After compiling, a link error occurs when each compilation unit compiles correctly, but in the next stage, the linker is unable to combine all the object code into a single valid executable file. Compare to *compile-time error*, *runtime error*, and *semantic error*.

linked list
An implementation for the list ADT that uses *dynamic memory allocation* of link nodes to store the list elements. Common variants are the *singly linked list* and *doubly linked list*.

list
A finite, ordered sequence of data items known as *elements*. This is close to the mathematical concept of a *sequence*. Note that "ordered" in this definition means that the list elements have position. It does not refer to the relationship between *key* values for the list elements (that is, "ordered" does not mean "sorted").

literal
In a boolean expression, a `literal` is a boolean variable or its negation.

In the context of compilers, it is any constant value. Similar to a *terminal*.

load factor
In a *hash table*, the number of items contained in the table divided by the table size.

local variable
A variable declared within a function or method. It exists only from the time when the function is called to when the function exits. When a function is suspended (due to calling another function), the function's local variables are stored in an *activation record* on the *runtime stack*.

logical form
The definition for a data type in terms of an ADT. Contrast to the *physical form* for the data type.

low-level language
A programming language that is designed to be easy for a computer to execute; also called machine language or assembly language.

lower bound
In *algorithm analysis*, a *growth rate* that is always less than or equal to the growth rate of the *algorithm* in question. In practice, this is the fastest-growing function that we know grows no faster than all but a constant number of inputs. It could be a gross under-estimate of the truth. Since the lower bound for the algorithm can be very different for different situations (such as the *best case* or *worst case*), we typically have to specify which situation we are referring to.

lower bounds proof

to-term
adversary argument :label: example

to-term
sorting lower bound :label: example

to-term
search lower bound :label: example

A proof regarding the lower bound, with this term most typically referring to the lower bound for any possible algorithm to solve a given *problem*. Many problems have a simple lower bound based on the concept that the minimum amount of processing is related to looking at all of the problem's input. However, some problems have a higher lower bound than that. For example, the lower bound for the problem of sorting ($\Omega(n \log n)$) is greater than the input size to sorting (n). Proving such "non-trivial" lower bounds for problems is notoriously difficult.

lvalue
An expression that identifies a non-temporary object.

lvalue reference
An alias or synonymn for an existing object. Often just referred to as a reference.

map
A *data structure* that relates a *key* to a *record*.

mapping
A *function* that maps every element of a given *set* to a unique element of another set; a correspondence.

max heap
A *heap* where every *node* has a *key* value greater than its *children*. As a consequence, the node with maximum key value is at the *root*.

member
In set notation, this is a synonym for *element*. In abstract design, a *data item* is a member of a *type*. In an object-oriented language, *data members* are data fields in an object.

member function
Each operation associated with the ADT is implemented by a member function or *method*.

memory leak
In programming, the act of creating *garbage*. In languages such as C and C++ that do not support *garbage collection*, repeated memory leaks will evenually cause the program to terminate.

method
In the *object-oriented programming paradigm*, a method is an operation on a *class*. A synonym for *member function*.

min heap
A *heap* where every *node* has a *key* value less than its *children*. As a consequence, the node with minimum key value is at the *root*.

mod
Abbreviation for the *modulus* function.

modulus
The modulus function returns the remainder of an integer division. Sometimes written $n \bmod m$ in mathematical expressions, the syntax in many programming languages is `n % m`.

multi-dimensional search key
A search key containing multiple parts, that works in conjunction with a *multi-dimensional search structure*. Most typically, a *spatial* search key representing a position in multi-dimensional (2 or 3 dimensions) space. But a multi-dimensional key could be used to organize data within non-spatial dimensions, such as temperature and time.

multi-dimensional search structure

to-term
multi-dimensional search key :label: uses

A data structure used to support efficient search on a *multi-dimensional search key*. The main concept here is that a multi-dimensional search structure works more efficiently by considering the multiple parts of the search key as a whole, rather than making independent searches on each one-dimensional component of the key. A primary example is a *spatial data structure* that can efficiently represent and search for records in multi-dimensional space.

multilist
A list that may contain sublists. This term is sometimes used as a synonym to the term *bag*.

name
See *identifier*.

natural language
Any one of the languages that people speak that evolved naturally.

natural order
An ordering of a sequence of objects that seems 'natural' to most people. The 'natural order' of whole numbers is the sequence used to count things: 1,2,3,4,5,6... The natural order for words is sorted alphabetically.

neighbor

to-term
adjacent :label: is

to-term
graph :label: context

In a *graph*, a *node* w is said to be a neighbor of *node* v if there is an *edge* from v to w.

node
The objects that make up a linked structure such as a linked list or binary tree. Typically, nodes are allocated using *dynamic memory allocation*.

non-strict partial order
In set notation, a relation that is *reflexive*, *antisymmetric*, and *transitive*.

non-terminal
In contrast to a *terminal*, a non-terminal is an abstract state in a *production rule*. Begining with the *start symbol*, all non-terminals must be converted into terminals in order to complete a *derivation*.

object
An instance of a class, that is, something that is created and takes up storage during the execution of a computer program. In the *object-oriented programming paradigm*, objects are the basic units of operation. Objects have state in the form of *data members*, and they know how to perform certain actions (*functions*).

object code
The output of the compiler after it translates the program.

object-oriented programming
object-oriented programming paradigm
An approach to problem-solving where all computations are carried out using *objects*.

object-space decomposition
A from of *key-space decomposition* where the *key space* is determined by the actual values of keys that are found. For example, a binary search tree stores a key value in its root, and all other values in the tree with lesser value are in the left *subtree*. Thus, the root value has split (or decomposed) the key space for that key based on its value into left and right parts. An object-space decomposition is in opposition to an *image-space decomposition*.

octree
The three-dimensional equivalent of the *quadtree* would be a tree with 2^3 or eight branches.

Omega notation
In *algorithm analysis*, Ω notation is used to describe a *lower bound*. Roughly (but not completely) analogous to *big-Oh notation* used to define an *upper bound*.

open addressing
A synonym for *closed hashing*.

open hashing
open hash system
A *hash system* where multiple records might be associated with the same slot of a *hash table*. Typically this is done using a linked list to store the records. This is in contrast to a *closed hash system*.

operating system
The control program for a computer. Its purpose is to control hardware, manage resources, and present a standard interface to these to other software components.

optimization problem
Any problem where there are a (typically large) collection of potential solutions, and the goal is to find the best solution. An example is the Traveling Salesman Problem, where visiting n cities in some order has a cost, and the goal is to visit in the cheapest order.

overflow
The condition where the amount of data stored in an entity has exceeded its capacity. For example, a node in a

array can store a certain number of records. If a record is attempted to be inserted into a node that is full, then something has to be done to handle this case.

overflow bucket
In *bucket hashing*, this is the *bucket* into which a record is placed if the bucket containing the record's *home slot* is full. The overflow bucket is logically considered to have infinite capacity, though in practice search and insert will become relatively expensive if many records are stored in the overflow bucket.

parameter
parameters
The values making up an input to a *function*.

parent
In a tree, the *node* P that directly links to a node A is the parent of A. A is the *child* of P.

parent pointer representation
For *trees*, a *node* implementation where each node stores only a pointer to its *parent*, rather than to its *children*. This makes it easy to go up the tree toward the *root*, but not down the tree toward the *leaves*.

parity
The concept of matching even-ness or odd-ness, the basic idea behind using a *parity bit* for error detection.

parity bit
A common method for checking if transmission of a sequence of bits has been performed correctly. The idea is to count the number of 1 bits in the sequence, and set the parity bit to 1 if this number is odd, and 0 if it is even. Then, the transmitted sequence of bits can be checked to see if its parity matches the value of the parity bit. This will catch certain types of errors, in particular if the value for a single bit has been reversed. This was used, for example, in early versions of *ASCII character coding*.

parse
To examine a program and analyze the syntactic structure.

parse tree
A tree that represents the syntactic structure of an input string, making it easy to compare against a *grammar* to see if it is syntactically correct.

parser

to-term
compiler :label: part of

to-term
parse tree :label: build

A part of a *compiler* that takes as input the program text (or more typically, the tokens from the *scanner*), and verifies that the program is syntactically correct. Typically it will build a *parse tree* as part of the process.

partial order
In set notation, a binary relation is called a partial order if it is *antisymmetric* and *transitive*. If the relation is also *reflexive*, then it is a *non-strict partial order*. Alternatively, if the relation is also *irreflexive*, then it is a *strict partial order*.

partially ordered set
The set on which a *partial order* is defined is called a partially ordered set.

partition
The process of splitting a set into two parts, typically centering around a predicate expression or a value.

In quick sort, the central value is called the `pivot` value. One partition will contain values less then the pivot, while the other partition will contain values greater than the pivot value.

pass by reference
A *reference* to the variable is passed to the called function. So, any modifications will affect the original variable.

pass by value
A copy of a variable is passed to the called function. So, any modifications will not affect the original variable.

path

to-term
tree :label: In

to-term
vertex :label: sequence of

In *tree* or *graph* terminology, a sequence of *vertices* $v_1, v_2, ..., v_n$ forms a path of length $n-1$ if there exist edges from v_i to v_{i+1} for $1 \leq i < n$.

permutation
A permutation of a sequence $\mathbf{S}$ is the *elements* of $\mathbf{S}$ arranged in some order.

physical form
The implementation of a data type as a data structure. Contrast to the *logical form* for the data type.

Pigeonhole Principle
A commonly used lemma in Mathematics. A typical variant states: When $n+1$ objects are stored in n locations, at least one of the locations must store two or more of the objects.

POD
An abbreviation for 'plain old data'. Used to indicate a data structure containing no member functions and only publicly accessible data.

pointer
A variable whose value is the *address* of another variable; a link.

pointer-based implementation for binary tree nodes
A common way to implement *binary tree nodes*. Each node stores a data value (or a reference to a data value), and pointers to the left and right children. If either or both of the children does not exist, then a null pointer is stored.

polymorphism
Polymorphism
An *object-oriented programming* term meaning *one name, many forms*. It describes the ability of software to change its behavior dynamically. Two basic forms exist: *runtime polymorphism* and *compile-time polymorphism*.

pop
A specialized term used to indicate removing an *element* from a *stack*.

portability
A property of a program that can run on more than one kind of computer.

poset
An abbreviation for a *partially ordered set*.

position
The defining property of the list ADT, this is the concept that list elements are in a position. Many list ADTs support access by position.

postorder traversal
In a *binary tree*, a *traversal* that first *recursively visits* the left *child*, then recursively visits the right child, and then visits the *root*.

powerset
For a *set* $\mathbf{S}$, the power set is the set of all possible *subsets* for $\mathbf{S}$.

PR quadtree
A type of *quadtree* that stores point data in two dimensions. The root of the PR quadtree represents some square region of 2d space. If that space stores more than one data point, then the region is decomposed into four equal subquadrants, each represented *recursively* by a subtree of the PR quadtree. Since many leaf nodes of the PR quadtree will contain no data points, implementation often makes use of the *flyweight design pattern*. Related to the *bintree*.

predicate
predicate function
A function that returns a boolean value.

preorder traversal
In a *binary tree*, a *traversal* that first *visits* the *root*, then *recursively* visits the left *child*, then recursively visits the right child.

primary clustering
In *hashing*, the tendency in certain *collision resolution* methods to create clustering in sections of the hash table. The classic example is *linear probing*. This tends to happen when a group of keys follow the same :term`probe sequence` during collision resolution.

primitive element
In set notation, this is a single element that is a member of the base type for the set. This is as opposed to an element of the set being another set.

primitive type
A *data type* whose values contain no subparts. An example is the integers. A synonym for *simple type* and *code block*.

priority
A quantity assigned to each of a collection of tasks that indicate importance for order of processing. For example, in an operating system, there could be a collection of processes (jobs) ready to run. The operating system must select the next task to execute, based on their priorities.

priority queue
An ADT whose primary operations of insert of records, and deletion of the greatest (or, in an alternative implementation, the least) valued record. Most often implemented using the *heap* data structure. The name comes from a common application where the records being stored represent tasks, with the ordering values based on the *priorities* of the tasks.

probe function
In *hashing*, the function used by a *collision resolution* method to calculate where to look next in the *hash table*.

probe sequence
In *hashing*, the series of *slots* visited by the *probe function* during *collision resolution*.

problem
A task to be performed. It is best thought of as a *function* or a mapping of inputs to outputs.

problem instance
A specific selection of values for the parameters to a problem. In other words, a specific set of inputs to a problem. A given problem instance has a size under some *cost model*.

problem solving
The process of formulating a problem, finding a solution, and expressing the solution.

procedural
Typically referring to the *procedural programming paradigm*, in contrast to the *object-oriented programming paradigm*.

procedural programming paradigm
Procedural programming uses a list of instructions (and procedure calls) that define a series of computational

steps to be carried out. This is in contrast to the *object-oriented programming paradigm*.

production
production rule
A *grammar* is comprised of production rules. The production rules consist of *terminals* and *non-terminals*, with one of the non-terminals being the *start symbol*. Each production rule replaces one or more non-terminals (perhaps with associated terminals) with one or more terminals and non-terminals. Depending on the restrictions placed on the form of the rules, there are classes of languages that can be represented by specific types of grammars. A *derivation* is a series of productions that results in a string (that is, all non-terminals), and this derivation can be represented as a *parse tree*.

program
An instance, or concrete representation, of an algorithm in some programming language. A sequence of instructions that specifies to a computer actions and computations to be performed. A program can refer to the *compiled* system *object code*, or to the original *source code*.

programming language
A formal notation for representing solutions.

proof

to-term
lower bounds proof :label: example

to-term
NP-Completeness proof :label: example

to-term
proof by contradiction :label: type

to-term
proof by induction :label: type

The establishment of the truth of anything, a demonstration.

proof by contradiction
A mathematical proof technique that proves a theorem by first assuming that the theorem is false, and then uses a chain of reasoning to reach a logical contradiction. Since when the theorem is false a logical contradiction arises, the conclusion is that the theorem must be true.

proof by induction
A mathematical proof technique similar to *recursion*. It is used to prove a parameterized theorem $S(n)$, that is, a theorem where there is a induction variable involved (such as the sum of the numbers from 1 to n). One first proves that the theorem holds true for a *base case*, then one proves the implication that whenever $S(n)$ is true then $S(n+1)$ is also true. Another variation is *strong induction*.

pseudo-random probing
In *hashing*, this is a *collision resolution* method that stores a random permutation of the values 1 through the size of the *hash table*. Term i of the *probe sequence* is simply the value of position i in the permuation.

push
A specialized term used to indicate inserting an *element* onto a *stack*.

push_back
A specialized term used to indicate appending an *element* onto a *vector*.

quadratic growth rate
A growth rate function of the form cn^2 where n is the input size and c is a constant.

quadratic probing
In *hashing*, this is a *collision resolution* method that computes term i of the *probe sequence* using some quadratic

equation $ai_b^2 i + c$ for suitable constants a, b, c. The simplest form is simply to use i^2 as term i of the probe sequence.

quadtree
A *full tree* where each internal node has four children. Most typically used to store two dimensional *spatial data*. Related to the *bintree*. The difference is that the quadtree splits all dimensions simultaneously, while the bintree splits one dimension at each level. Thus, to extend the quadtree concept to more dimensions requires a rapid increase in the number of splits (for example, 8 in three dimensions).

queue
A list-like structure in which elements are inserted only at one end, and removed only from the other one end.

radix
Synonym for *base*. The number of digits in a number representation. For example, we typically represent numbers in base (or radix) 10. Hexidecimal is base (or radix) 16.

RAII
Resource Acquisition Is Initialization is the C++ term for a programming style in which critical resources are tied to the object which owns them. Because they are typically allocated in class constructors and destroyed in class destructors, in other languages, this is sometimes called *Constructor Acquires, Destructor Releases*

range
The set of possible outputs for a function.

record
A collection of information, typically implemented as an *object* in an *object-oriented programming language*. Many data structures are organized containers for a collection of records.

recurrence relation
A recurrence relation (or less formally, recurrence) defines a function by means of an expression that includes one or more (smaller) instances of itself. A classic example is the *recursive* definition for the factorial function, $F(n) = n * F(n-1)$.

recursion
The process of using recursive calls. An algorithm is recursive if it calls itself to do part of its work. See *recurrence relation*.

recursive call
Within a *recursive function*, it is a call that the function makes to itself.

recursive data structure
A data structure that is partially composed of smaller or simpler instances of the same data structure. For example, *linked lists* and *binary trees* can be viewed as recursive data structures.

recursive function
A function that includes a *recursive call*.

reference
A value that enables a program to directly access some particular data item. An example might be a byte position within a file where the record is stored, or a pointer to a record in memory. (Note that Java makes a distinction between a reference and the concept of a pointer, since it does not define a reference to necessarily be a byte position in memory.)

reference count algorithm
An algorithm for *garbage collection*. Whenever a reference is made from a variable to some memory location, a counter associated with that memory location is incremented. Whenever the reference is changed or deleted, the reference count is decremented. If this count goes to zero, then the memory is considered free for reuse. This approach can fail if there is a cycle in the chain of references.

reflexive
In set notation, binary relation R on set S is reflexive if aRa for all $a \in \mathbf{S}$.

regular type
A user defined *type* that behaves like a 'regular' built-in (fundamental) type. Regular types support the following operations:

Operation	Definition
Default constructor	T a;
Copy constructor	T a = b;
Destructor	~T (a);
Assignment	a = b;
Equality	a == b;
Inequality	a != b;
Ordering	a < b;

relation
In set notation, a relation R over set $\mathbf{S}$ is a set of ordered pairs from $\mathbf{S}$.

root
In a *tree*, the topmost *node* of the tree. All other nodes in the tree are *descendants* of the root.

runtime environment
The environment in which a program (of a particular programming language) executes. The runtime environment handles such activities as managing the *runtime stack*, the *free store*, and the *garbage collector*, and it conducts the execution of the program.

runtime error
An error that does not occur until the program has started to execute but that prevents the program from continuing. Compare to *compile-time error*, *link error*, and *semantic error*.

runtime polymorphism
Runtime polymorphism
A form of *polymorphism* known as Overriding. Overridden methods are those which implement a new method with the same signature as a method inherited from its base class. Compare to *compile-time polymorphism*.

runtime stack
The place where an *activation record* is stored when a subroutine is called during a program's runtime.

rvalue
An expression that identifies a temporary object or a value not associated with any object, such as a literal.

rvalue reference
Sometimes called a forwarding reference. A reference that is allowed to refer to an rvalue. That is, a temporary object or an rvalue not associated with any object.

scanner

to-term
compiler :label: part of

to-term
lexical analysis :label: responsible for

The part of a *compiler* that is responsible for doing *lexical analysis*.

scope
A region of the program where a defined variable, definition, or function exists. Beyond that point the variable can not be accessed.

search key
A field or part of a record that is used to represent the record when searching. For example, in a database of customer records, we might want to search by name. In this case the name field is used as the search key.

search lower bound
The problem of searching in an array has provable lower bounds for specific variations of the problem. For an unsorted array, it is $\Omega(n)$ *comparisons* in the *worst case*, typically proved using an *adversary argument*. For a sorted array, it is $\Omega(\log n)$ in the worst case, typically proved using an argument similar to the *sorting lower bound* proof. However, it is possible to search a sorted array in the average case in $O(\log \log n)$ time.

search problem
Given a particular key value K, the search problem is to locate a *record* (k_j, I_j) in some collection of records $\mathbf{L}$ such that $k_j = K$ (if one exists). *Searching* is a systematic method for locating the record (or records) with key value $k_j = K$.

search tree

to-term
Binary Search Tree :label: example

to-term
search trie :label: example

A *tree* data structure that makes search by *key* value more efficient. A type of *container*, it is common to implement an *index* using a search tree. A good search tree implementation will guarantee that insertion, deletion, and search operations are all $\Theta(\log n)$.

search trie

to-term
alphabet trie :label: example

to-term
binary trie :label: example

Any *search tree* that is a *trie*.

searching
Given a *search key* K and some collection of records $\mathbf{L}$, searching is a systematic method for locating the record (or records) in $\mathbf{L}$ with key value $k_j = K$.

secondary clustering
In *hashing*, the tendency in certain *collision resolution* methods to create clustering in sections of the hash table. In *primary clustering*, this is caused by a cluster of keys that don't necessarily hash to the same slot but which following significant portions of the same *probe sequence* during collision resolution. Secondary clustering results from the keys hashing to the same slot of the table (and so a collision resolution method that is not affected by the key value must use the same probe sequence for all such keys). This problem can be resolved by *double hashing* since its probe sequence is determined in part by a second hash function.

semantic error
An error in a program or expression that makes it do something other than what the programmer intended. Compare to *compile-time error*, *link error*, and *runtime error*.

semantics
The meaning of a program or piece of text.

separate chaining
In *hashing*, a synonym for *open hashing*

sequence
In set notation, a collection of elements with an order, and which may contain duplicate-valued elements. A sequence is also sometimes called a *tuple* or a *vector*.

sequence container
A container in which elements can be accessed sequentially. The underlying data may be a contiguous block of memory, as with `vector` and `array`, or may be non-contiguous memory, as with `list`.

sequential tree representation
A representation that stores a series of node values with the minimum information needed to reconstruct the tree structure. This is a technique for *serializing* a tree.

serialization
The process of taking a data structure in memory and representing it as a sequence of bytes. This is sometimes done in order to transmit the data structure across a network or store the data structure in a *stream*, such as on disk. *Deserialization* reconstructs the original data structure from the serialized representation.

set
A collection of distinguishable *members* or *elements*.

set former
A way to define the membership of a set, by using a text description. Example: $\{x \mid x \text{ is a positive integer}\}$.

set product
Written $\mathbf{Q} \times \mathbf{P}$, the set product is a set of ordered pairs such that ordered pair (a, b) is in the product whenever $a \in \mathbf{P}$ and $b \in \mathbf{Q}$. For example, when $\mathbf{P} = \{2, 3, 5\}$ and $\mathbf{Q} = \{5, 10\}$, $\mathbf{Q} \times \mathbf{P} = \{(2, 5),\ (2, 10),\ (3, 5),\ (3, 10),\ (5, 5),\ (5, 10)\}$.

shallow copy
Copying a *reference* or *pointer* value without copying the actual content.

sibling
In a *tree*, a sibling of *node* A is any other node with the same *parent* as A.

signature
In a programming language, the signature for a function is its return type and its list of parameters and their types.

simple type
A *data type* whose values contain no subparts. An example is the integers. A synonym for *primitive type* and :term"*fundamental type*.

simulating recursion
If a programming language does not support *recursion*, or if you want to implement the effects of recursion more efficiently, you can use a *stack* to maintain the collection of subproblems that would be waiting for completion during the recursive process. Using a loop, whenever a recursive call would have been made, simply add the necessary program state to the stack. When a return would have been made from the recursive call, pop the previous program state off of the stack.

singly linked list
A *linked list* implementation variant where each list node contains access an pointer only to the next element in the list.

slot
In *hashing*, a position in a *hash table*.

software engineering
The study and application of engineering to the design, development, and maintenance of software.

software reuse
In *software engineering*, the concept of reusing a piece of software. In particular, using an existing piece of software (such as a function or library) when creating new software.

sorting lower bound
The lower bound for the *problem* of *sorting* is $\Omega(n \log n)$. This is traditionally proved using a *decision tree* model

for sorting algorithms, and recognizing that the minimum depth of the decision tree for any sorting algorithm is $\Omega(n \log n)$ since there are $n!$ permutations of the n input records to distinguish between during the sorting process.

sorting problem
Given a set of records r_1, r_2, ..., r_n with *key* values k_1, k_2, ..., k_n, the sorting problem is to arrange the records into any order s such that records r_{s_1}, r_{s_2}, ..., r_{s_n} have keys obeying the property $k_{s_1} \leq k_{s_2} \leq ... \leq k_{s_n}$. In other words, the sorting problem is to arrange a set of records so that the values of their key fields are in non-decreasing order.

source code
A program, stored in a file, in a high-level language before being compiled or interpreted.

spatial
Referring to a position in space.

spatial application
An application what has spatial aspects. In particular, an application that stores records that need to be searched by location.

spatial attribute
An attribute of a record that has a position in space, such as the coordinate. This is typically in two or more dimensions.

spatial data
Any object or record that has a position (in space).

spatial data structure

to-term
bintree :label: example

to-term
kd tree :label: example

to-term
PR quadtree :label: example

A *data structure* designed to support efficient processing when a *spatial attribute* is used as the key. In particular, a data structure that supports efficient search by location, or finds all records within a given region in two or more dimensions. Examples of spatial data structures to store point data include the *bintree*, the *PR quadtree* and the *kd tree*.

stack
A list-like structure in which elements may be inserted or removed from only one end.

start symbol
In a *grammar*, the designated *non-terminal* that is the initial point for *deriving* a string in the language.

static scoping
A synonym for *lexical scoping*.

strategy
An approach to accomplish a task, often encapsulated as an algorithm. Also the name for a *design pattern* that separates the algorithm for performing a task from the control for applying that task to each member of a collection. A good example is a generic sorting function that takes a collection of records (such as an array) and a "strategy" in the form of an algorithm that knows how to extract the key from a record in the array. Only subtly different from the *visitor* design pattern, where the difference is primarily one of intent rather than syntax. The strategy design pattern is focused on encapsulating an activity that is part of a larger process, so that different ways of performing that activity can be substituted. The visitor design pattern is focused on encapsulating an activity

that will be performed on all members of a collection so that completely different activities can be substituted within a generic method that accesses all of the collection members.

stream
The process of delivering content in a *serialized* form.

strict partial order
In set notation, a relation that is *irreflexive*, *antisymmetric*, and *transitive*.

strong induction
An alternative formulation for the *induction step* in a *proof by induction*. The induction step for strong induction is: If **Thrm** holds for all $k, c \leq k < n$, then **Thrm** holds for n.

subclass
In *object-oriented programming*, any class within a *class hierarchy* that *inherits* from some other class. A synonym for *derived class*.

subset
In set theory, a set A is a subset of a set B, or equivalently B is a *superset* of A, if all elements of A are also elements of B.

subtree
A subtree is a subset of the nodes of a binary tree that includes some node R of the tree as the subtree *root* along with all the *descendants* of R.

summation
The sum of costs for some function applied to a range of parameter values. Often written using Sigma notation. For example, the sum of the integers from 1 to n can be written as $\sum_{i=1}^{n} i$.

superset
In set theory, a set A is a *subset* of a *set* B, or equivalently B is a *superset* of A, if all *elements* of A are also elements of B.

symbol table
As part of a *compiler*, the symbol table stores all of the identifiers in the program, along with any necessary information needed about the identifier to allow the compiler to do its job.

symmetric
In set notation, relation R is symmetric if whenever aRb, then bRa, for all $a, b \in \mathbf{S}$.

symmetric matrix
A square matrix that is equal to its *transpose*. Equivalently, for a $n \times n$ matrix A, for all $i, j < n$, $A[i, j] = A[j, i]$.

syntax
The set of rules that defines the valid symbol combinations that define valid statements or expressions in a specific language.

syntax analysis

to-term
parse tree :label: generates

to-term
tokens :label: accepts

A phase of *compilation* that accepts *tokens*, checks if program is syntactically correct, and then generates a *parse tree*.

syntax error
An error in a program that makes it impossible to parse --- and therefore impossible to interpret.

tail
The end of a *list*.

template
A template is a specific way in C++ to write *generic* functions and classes.

A template by itself is not a class, type, function, or any other entity. It defines a **recipe** for generating a class or function.

terminal
A specific character or string that appears in a *production rule*. In contrast to a *non-terminal*, which represents an abstract state in the production. Similar to a *literal*, but this is the term more typically used in the context of a *compiler*.

test-driven development
Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only. This is opposed to software development that allows software to be added that is not proven to meet requirements.

Kent Beck, who is credited with having developed the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.

token
One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

total order
A binary relation on a set where every pair of distinct elements in the set are *comparable* (that is, one can determine which of the pair is greater than the other).

trailing return type
A C++11 language feature that allows a function or lambda expression to defer evaluating the function return type. Example:

```
template<class T>
auto mul(T a, T b) -> decltype(a*b){
  return a*b;
}
```

or

```
[](double x, double y) -> int {return x*y;}
```

transitive
In set notation, relation R is transitive if whenever aRb and bRc, then aRc, for all $a, b, c \in \mathbf{S}$.

transpose
In the context of linear algebra, the transpose of a matrix A is another matrix A^T created by writing the rows of A as the columns of A^T.

traversal
traverse
Any process for visiting all of the objects in a collection (such as a *tree* or *list*) in some order.

tree
A tree $\mathbf{T}$ is a finite set of one or more *nodes* such that there is one designated node R, called the *root* of $\mathbf{T}$. If the set $(\mathbf{T} - \{R\})$ is not empty, these nodes are partitioned into $n > 0$ disjoint sets $\mathbf{T}_0$, $\mathbf{T}_1$, ..., $\mathbf{T}_{n-1}$, each of which is a tree, and whose *roots* $R_1, R_2, ..., R_n$, respectively, are *children* of R.

tree traversal
A *traversal* performed on a tree. Traditional tree traversals include *preorder* and *postorder* traversals for both *binary* and *general* trees, and :term`inorder traversal` for binary search trees.

trie

to-term
alphabet trie :label: example

to-term
binary trie :label: example

to-term
search trie :label: example

A form of *search tree* where an internal node represents a split in the *key space* at a predetermined location, rather than split based on the actual *key* values seen. For example, a simple binary search trie for key values in the range 0 to 1023 would store all records with key values less than 512 on the left side of the tree, and all records with key values equal to or greater than 512 on the right side of the tree. A trie is always a *full tree*. Folklore has it that the term comes from "retrieval", and should be pronounced as "try" (in contrast to "tree", to distinguish the differences in the space decomposition method of a search tree versus a search trie). The term "trie" is also sometimes used as a synonym for the *alphabet trie*.

truth table
In symbolic logic, a table that contains as rows all possible combinations of the boolean variables, with a column that shows the outcome (true or false) for the expression when given that row's truth assignment for the boolean variables.

tuple
In set notation, another term for a *sequence*.

In C++, the class tuple.

two's complement
A mathematical operation on binary numbers, as well as a binary signed number representation based on this operation.

type
A collection of values.

unary function
A function that accepts one parameter.

unit test
A software test method in which a single *unit* of source code, for example, a single function is tested in isolation. Unit tests are short code fragments, typically created by programmers as part of the development process. In a process like *test-driven development* the unit tests are written before any other code.

unsorted list
A *list* where the records stored in the list can appear in any order (as opposed to a sorted list). An unsorted list can support efficient ($\Theta(1)$) insertion time (since you can put the record anywhere convenient), but requires $\Theta(n)$ time for both search and deletion.

unvisited
In *graph* algorithms, this refers to a node that has not been processed at the current point in the algorithm.

upper bound
In *algorithm analysis*, a *growth rate* that is always greater than or equal to the growth rate of the *algorithm* in question. In practice, this is the slowest-growing function that we know grows at least as fast as all but a constant number of inputs. It could be a gross over-estimate of the truth. Since the upper bound for the algorithm can be very different for different situations (such as the *best case* or *worst case*), we typically have to specify which situation we are referring to.

variable-length coding
Given a collection of objects, a variable-length coding scheme assigns a code to each object in the collection using

codes that can be of different lengths. Typically this is done in a way such that the objects that are most likely to be used have the shortest codes, with the goal of minimizing the total space needed to represent a sequence of objects, such as when representing the characters in a document. This is in contrast to *fixed-length coding*.

vector
In set notation, another term for a *sequence*. As a data structure, the term vector usually used as a synonym for a *dynamic array*.

vertex
Another name for a *node* in a *graph*.

visit
During the process of a *traversal* on a *list* or *tree* the action that takes place on each *node*.

visitor
A *design pattern* where a *traversal* process is given a function (known as the visitor) that is applied to every object in the collection being traversed. For example, a generic tree or graph traversal might be designed such that it takes a function parameter, where that function is applied to each node.

volatile
In the context of computer memory, this refers to a memory that loses all stored information when the power is turned off.

worst case
In algorithm analysis, the *problem instance* from among all problem instances for a given input size n that has the greatest cost. Note that the worst case is **not** when n is big, since we are referring to the worst from a class of inputs (i.e, we want the worst of those inputs of size n).

- search
- *Glossary*
- genindex

Acknowledgements

Portions of this book would not be possible without the generosity of others who have created excellent textbooks and materials and also chose to release them in a shareable and extensible manner. Some content in this book is adapted from On Complexity, by Janice L. Pearce released under the Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0), Problem Solving with Algorithms and Data Structures using C++, by Brad Miller and David Ranum, Luther College, and Jan Pearce, Berea College also released under the CC BY-NC-SA 4.0, and Open Data Structures (OpenDSA) by Ville Karavirta and Cliff Shaffer which is distributed under the MIT License

Download the PDF version.

BIBLIOGRAPHY

[Aspnes2014] Adapted from The IEEE-754 floating-point standard (PineWiki) and Class 223 notes . Retrieved 2026-07-05.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

W

X