

**CS361**

**Advanced Data Structures and Algorithms**

**May 27, 2020**

# CS361 Outline

Summer 2020

## 1 Preamble

- [Syllabus](#)
- [Policies](#)
- [Library](#)

### Welcome to CS361

- The course is divided into three parts. Each part contains multiple modules, including some assignments, and an exam.
- Each module consists of a series of activities. These include reading, taking self-assessment quizzes, and doing assignments.
  - Not every assigned activity requires you to submit something for grading. Nonetheless, you are expected to complete all of them.
  - If an activity is not marked with its own due date, then it is considered to be due at the end of the date range given for that module.

### KEYS TO SUCCESS IN THIS COURSE:

- READ THE SYLLABUS

The syllabus lays out the basic course policies. It tells you what you need to do to earn a passing grade. It tells you when you need to have done that by. It tells you how to get in touch with me if you run into problems.

- HAVE A SCHEDULE

You have some freedom to schedule your own time in this course, but you *DO* need to set up a schedule. Don't forget that this course exists and that you are registered for it. Don't think you can repeatedly set it aside for weeks at a time and make up the time later.

- IF YOU DON'T UNDERSTAND SOMETHING, ASK QUESTIONS

In a web course, my role as Instructor changes from "lecturer" to "tutor". You can ask questions in the course Forums. You can send me email. You can also contact me during office hours. You'll find more information on these options in the syllabus and other documents on the Course Policies page.

Some people are too shy to ask questions. Some are too proud to ask questions. My advice to both groups is to get over it! Part of being educated is knowing how to exploit your available information resources. In this course, I am one of those resources.

## 1 Part I 05/18/2020 - 06/16/2020

### 1.1 Overview and ADTs Review 05/18/2020 - 05/24/2020

#### Overview

Coming into this course, you should already be familiar with Abstract Data Types and with the way that C++ classes are used to implement them.

To be certain that everyone is on the same page, however, this module examines some critical information about ADTs, classes, templates in C++. Most of this material should be review for you. (If not, you can find more detailed treatments on the websites for the pre-requisite courses CS250 or CS333.) But it may be some time since you have seen it, or, if you are a recent transfer to ODU, your prior courses might not have covered all of it.

Because the C++ language is evolving, there may be some code that looks unfamiliar simply because it takes advantage of the C++11 standard.

- 
1. Add your personal intro in the Forums on Blackboard.
  2. [Orientation](#) 05/18/2020, 7:30PM EDT
  3. Weiss, Ch 1
  4. [Setting Up Your Personal Programming Environment](#) 05/20/2020
  5. [Abstraction and Abstract Data Types](#)
  6. [ADT Interfaces in C++](#)
  7. [Implementing ADTs in C++](#)
  8. Visibility & Type Annotations (in Self-Assessments in [Blackboard](#))
  9. [Copying and Moving Data](#)
  10. [Review session](#) 05/20/2020, 7:30PM EDT

11. See website for assignment

### 1.2 Templates and Iterators 05/25/2020 - 05/30/2020

#### Overview

- *Templates* are a mechanism for writing algorithmic patterns that can be applied to a wide variety of different data types.
- *Iterators* are a data abstraction for the notion of a position within a container of data. Iterators allow us to express many simple algorithms in a simple form, regardless of whether the underlying container is an array, a linked list, a tree, or some other data structure.

Templates and iterators are often used together to provide patterns for code that can be applied to a wide range of underlying data structures.

---

1. [Algorithms as Patterns for Code](#)
  2. [Function Templates](#)
  3. [Class Templates](#)
  4. [The Standard Library: Overview](#)
  5. [Iterators: an ADT for Positions](#)
  6. [Review session](#) 05/27/2020, 7:30PM EDT
7. See website for assignment

## 1.3 Algorithm Analysis 05/31/2020 - 06/05/2020

### Overview

An important theme throughout this semester will be viewing the process of developing software as an *engineering* process. Now, engineers in traditional engineering disciplines, civil engineers, electrical engineers, and the like, face trade offs in developing a new product, trade offs in cost, performance, and quality.

Software developers face the same kinds of choices. Early on, you may have several alternative designs and need to make a decision about which of those designs to actually pursue. It's no good waiting until the program has already been implemented, written down in code. By then you've already committed to one design and invested significant resources into it.

In this module, we'll look at mathematical techniques for analyzing algorithms to determine what their speed will be, or, more precisely, how badly their speed will degrade as we apply them to larger and larger amounts of data. The key to doing this will be to *analyze* the code for its *worst case complexity*.

---

1. [Analysis of Algorithms: Motivation](#)
  2. Weiss, Ch 2
  3. [Analysis of Algorithms: Worst Case Complexity](#)
  4. [The Algebra of Big-O](#)
  5. Big-O Algebra
  6. [Worst-Case Complexity Analysis](#)
  7. [Case Studies: Analyzing Standalone Functions](#)
  8. Complexity (in Blackboard)
  9. [Review session](#) 06/03/2020, 7:30PM EDT
10. Quiz: Worst Case Complexity (in Blackboard) Due: 06/05/2020

## 1.4 Sequences 06/06/2020 - 06/14/2020

### Overview

A substantial amount of the data that we work with is arranged into a simple linear ordering, one thing after another. Of course, you are already quite familiar with one way of doing this, by putting the data into arrays.

In this module we explore the common variations on ADTs for maintaining data in a sequence: vectors, linked lists, stacks, and queues.

The combination of templates and iterators, combined with a wider range of container types, brings us to *generic programming*, an important and pervasive style of coding in C++.

---

### 1.4.1 Vectors

1. Weiss, Ch 3
2. [Vectors](#)
3. [Implementing the Vector Class](#)

### 1.4.2 Lists

1. [Linked Lists](#)
2. [Linked List Applications](#)
3. [Standard Lists](#)
4. Linear Sequences
5. [Review session](#) 06/10/2020, 7:30PM EDT

6. See website for assignment

### 1.4.3 Generic Programming

1. [Generic Programming](#)
2. [Generic Programming](#)

## 1.5 Stacks and Queues 06/10/2020 - 06/14/2020

1. [Recursion](#)
2. [Stacks](#)
3. [Queues](#)
4. See website for assignment
5. [Deques](#)

## 1.6 End of Part I

1. Exam 1 (on Blackboard) 06/14/2020 - 06/15/2020

## 2 Part II 06/15/2020 - 07/13/2020

### 2.1 Average Case Complexity

#### 2.1.1 Average Complexity 06/15/2020 - 06/21/2020

##### Overview

In Part I, we analyzed the speed of algorithms exclusively from the point of view of the worst case. One might argue that this is unnecessarily pessimistic on our part. There are some algorithms for the worst case input is rare enough that we might not be worried about it, particularly if we believe that typical inputs can be handled much more quickly.

We therefore next turn to the idea of *average case complexity* a measure of how the average behavior of a program degrades as the input sets get larger and larger.

- 
1. [Average Case Analysis](#)
  2. [Case Study: Schemes to Improve the Average Case of Sequential Search](#)
  3. Average Case Analysis
  4. [Review session](#) 06/17/2020, 7:30PM EDT
  5. See website for assignment

#### 2.1.2 Sorting 06/22/2020 - 06/25/2020

##### Overview

Sorting algorithms arrange data stored in a sequence into a new desired order.

Because the data structures involved are elementary (arrays, vectors, and, occasionally, linked lists) and because the need for sorted data arises in so many practical applications, you probably learned one or more sorting algorithms in your earliest programming classes.

But sorting is actually a fairly subtle problem, and the sorting algorithms taught to beginning programmers are chosen for simplicity, not performance. They are often slow and rather clumsy.

In this section we'll look at more sophisticated sorting algorithms. We'll also consider the fundamental limits on just how fast a sorting algorithm can get, and we'll see that some practical algorithms actually approach that upper speed limit.

- 
1. Weiss, Ch 7
  2. [Sorting --- Insertion Sort](#)
  3. [Sorting Speed Limits](#)
  4. [Sorting --- Merge Sort](#)
  5. [Sorting --- Quick Sort](#)
  6. [Review session](#) 06/24/2020, 7:30PM EDT
  7. Self-Assessment: Sorting

## 2.2 Trees 06/26/2020 - 07/05/2020

### Overview

Most of the data structures we have looked at so far have been devoted to keeping a collection of elements in some linear order.

Trees are the most common non-linear data structure in computer science. Trees are useful in representing things that naturally occur in hierarchies (e.g., many company organization charts are trees) and for things that are related in a “is-composed-of” or “contains” manner (e.g., this country is composed of states, each state is composed of counties, each county contains cities, each city contains streets, etc.)

Trees also turn out to be exceedingly useful in implementing associative containers like `std::set`. Properly implemented, a tree can lead to an implementation that can be both searched and inserted into in  $O(\log N)$  time. Compare this to the data structures we’ve seen so far, which may allow us to search in  $O(\log N)$  time but insert in  $O(N)$ , or insert in  $O(1)$  but search in  $O(N)$ .

---

### 2.2.1 General Trees

1. Weiss, Ch 4
2. [Trees](#)
3. [Review session](#) 07/01/2020, 7:30PM EDT
4. See website for assignment

### 2.2.2 Search Trees

1. [Binary Search Trees](#)
2. [Traversing Trees with Iterators](#)
3. Weiss, Ch 12.2
4. [Balanced Search Trees](#)

### 2.2.3 Sets and Maps

1. [Sets and MultiSets](#)
2. [Maps and MultiMaps](#)
3. See website for assignment

## 2.3 Hashing 07/06/2020 - 07/12/2020

### Overview

Hashing is an alternative to trees for providing fast associative containers (sets and maps).

Hashing stores data in arrays (primarily), but does not store them in any predictable order, or even contiguously. Instead, hashing uses a special “hash function” to compute a desired location for any key we want to insert. If you don’t actually know the internal details of the hash function, its choices of locations would seem arbitrary, almost random.

Nonetheless, it works, and in many cases works well. Hash tables can often store and search for data in  $O(1)$  average time.

---

1. [Hashing](#)
2. Weiss, Ch 5
3. [Resolving Collisions](#)
4. [Review session](#) 07/08/2020, 7:30PM EDT
5. [Rehashing \(Variable Hashing\)](#)
6. [Hash-Based Sets and Maps](#)
7. See website for assignment

## 2.4 End of Part II

1. [Exam 2](#) 07/12/2020 - 07/13/2020

## 3 Part III 07/13/2020 - 08/07/2020

### 3.1 Algorithm Design Techniques 07/13/2020 - 07/24/2020

## Overview

By this point in the semester, you've learned a lot of algorithms. Many practical problems can be solved by direct application of these. But what do you do when faced with an unfamiliar problem, one for which none of the "canned" algorithms in your personal toolbox are suitable?

When you have to design your own algorithms, you should consider some of the common patterns or styles that are available to you. This lesson looks at these styles, many of which we have seen before, and a few new ones as well.

- 
1. Weiss, Ch 10
  2. [Converting Recursion to Iteration](#)
  3. [A Gallery of Algorithmic Styles](#)
  4. [Review session](#) 07/15/2020, 7:30PM EDT
  5. See website for assignment
  6. [Review session](#) 07/22/2020, 7:30PM EDT
  7. See website for assignment
  8. [Exponential and NP Algorithms](#)

## 3.2 Heaps and Priority Queues 07/25/2020 - 07/30/2020

### Overview

A priority queue is an ADT that allows us to repeatedly find and remove the largest (or smallest) item from a collection of data. They take their name from the idea that they implement a "queue" of items awaiting processing, but one in which some items have higher priority than others and so get to jump to the head of the line if nothing ahead has even higher priority.

Priority queues are generally implemented using *heaps*, a tree with very special ordering properties.

- 
1. Weiss, Ch 6
  2. [Priority Queues](#)
  3. [Heaps](#)
  4. [Heapsort](#)
  5. [Review session](#) 07/29/2020, 7:30PM EDT

## 3.3 Graphs 07/31/2020 - 08/06/2020

### Overview

A graph is a collection of vertices (nodes) connected by edges in arbitrary fashion. Graphs are used to represent data relationships that are far more complicated than could be represented using trees or lists.

- 
1. Weiss, Ch 9
  2. [Graphs --- the Basics](#)
  3. [Graphs --- ADT and Traversing](#)
  4. [Graphs --- Sample Algorithms](#)
  5. Graph Basics
  6. [Sharing Pointers and Garbage Collection](#)
  7. [Review session](#) 08/05/2020, 7:30PM EDT
  8. See website for assignment

## 3.4 End of Part III

1. [Final exam](#) (cumulative) 08/06/2020 - 08/07/2020

## 4 Frequently Asked Questions

- [FAQ](#)

# CS 361: Syllabus -- Summer 2020

Steven Zeil

Last modified: Feb 12, 2020

## Contents:

- [1 Course Description](#)
  - [1.1 When and Where](#)
  - [1.2 Objectives:](#)
- [2 Basic Information](#)
  - [2.1 Instructor](#)
  - [2.2 Text](#)
  - [2.3 Course Prerequisites](#)
  - [2.4 Software Requirements](#)
  - [2.5 Computer Accounts](#)
- [3 Course Policies](#)
  - [3.1 Due Dates:](#)
  - [3.2 Academic Honesty:](#)
  - [3.3 Grading:](#)
- [4 Educational Accessibility:](#)

## 1 Course Description

### 1.1 When and Where

Website: [Blackboard](#) and <https://www.cs.odu.edu/~zeil/cs361/sum18/>

This is an internet-based course. There are no regularly scheduled class lecture times. For the most part, students will work at their own chosen times, subject to deadlines for assignments and exams (described later under [Course Policies](#)).

There will be an optional [Orientation session](#) conducted via network conferencing at the start of the semester. Additional similar sessions may be announced later in the semester.

### 1.2 Objectives:

This course explores data structures, algorithms for manipulating them, and the practical problems of implementing those structures in real programming languages and environments. Heavy emphasis is placed upon the analysis of algorithms to characterize their worst and average case requirements for running time and memory.

Perhaps more than any other course, CS361 should expand the students “toolbox” of basic techniques for manipulating data at both the conceptual and the concrete level. At the conceptual level, the student will see a broad selection of standard practices and approaches used in program design. At the concrete level, the student will begin what should be a career-long practice of accumulating useful, reusable code units.

## 2 Basic Information

### 2.1 Instructor

<a href="#">Steven J. Zeil</a>	E&CS 3208
(757) 683-4928	Fax: (757) 683-4900
<a href="mailto:zeil@cs.odu.edu">zeil@cs.odu.edu</a>	

**Important:** All email related to this course should have the phrase “cs361” somewhere in the subject line. This flags your message in my mailbox for faster attention. Omit this, and your message may get lost amid the ton of daily spam and ODU administrative messages I get each day.

I try to respond to all (properly marked) messages within 24 hours M-F, within 48 on weekends & holidays.

#### 2.1.1 Office Hours

Students may meet with me in person, by telephone, or via internet-conferencing. A week-by-week schedule of available meeting times can be found by going to the my home page <http://www.cs.odu.edu/~zeil/> and clicking on “[Office Hours and Appointments](#)”.

### 2.2 Text

In addition to the readings at the course web site, listed at the top of this document, the (required) textbook for this course is:

- Weiss, *Data Structures & Algorithm Analysis in C++*, 4e, 2013, Pearson PH, 013284737X 978-0132847377

## 2.3 Course Prerequisites

The prerequisites for this course are:

- CS 250, Problem Solving and Programming, or [CS 333](#), Problem Solving and Programming in C++
- [CS 252](#) Introduction to Unix for Programmers
- MATH 163, Pre-Calculus II,

or equivalents.

## 2.4 Software Requirements

### 2.4.1 Required

- Web browser: Most up-to-date web browsers should suffice for this course. Chrome and Firefox are recommended. Internet Explorer and Safari are discouraged.

Your browser will need to run Javascript, particularly when taking self-assessments, quizzes and exams, which are hosted on the ODU BlackBoard system.

- ssh, sftp: Any program should do. The [CS252 website](#) has some recommendations.
- [Java 8 \(or 9\) JRE](#), used for algorithm demos

### 2.4.2 Optional:

- C++ compiler: The “official” compiler for this course is the Free Software Foundation’s g++ (also known as gcc or GNU CC), version 5.5 or higher. This is the compiler that the instructor and/or grader will use in evaluating and grading projects. If you have access to other compilers, you may use them, but *you* are responsible for making sure that their projects can be compiled by the instructor and/or the course’s grader using the official compiler.

You may want to develop your programs on the most convenient compiler and then port it over to the Dept’s Linux machines for final testing. Please don’t underestimate the amount of time that may be involved in coping with subtle differences among compilers.

You can do all work in this course using g++ on the CS Dept Linux servers via ssh/X. If you like, however, you can obtain the g++ compiler for free from a variety sources.

- X, and its compressed cousin, X2Go, allows Linux programs running on our servers to open up windows on your PC. This opens up a whole world of window/graphics/mouse-based software that you may find easier to work with than the command-line based alternatives. The [CS 252](#) website has a substantial discussion of free X server software.

## 2.5 Computer Accounts

Students will need two network accounts to participate in this class:

- An ODU ITS (Midas) account. This is the account associated with your @odu.edu email. It will allow you to log into the course’s Blackboard site when taking quizzes and exams.

All ODU students automatically receive this account, though you may need to activate yours, particularly if you are new to ODU.

- An account on the CS Dept. network. This will be used for access to the CS dept computing resources, and for accessing and submitting assignments.

You may have a CS account already if you were registered for a CS class last semester. If not, the account setup and password can be initiated at <http://www.cs.odu.edu/> by clicking on “Account Creation” under “Online Services”.

A few notes about this:

- Typically, new accounts can be created no earlier than 1-2 weeks before the start of classes.
- There are time lags in the way that information flows around the University and within the CS network.
  - Typically you will need to have been enrolled in a CS course for 24-48 hours before you can create an account.
  - Once your account is activated, you may need to wait another 24 hours before your account information becomes available to the course website and you are able to access the course’s assignment pages.

All students in this course are responsible for making sure they have working accounts prior to the first assignment.

Students will have access to the Dept’s local network of Linux workstations and Windows PCs in Dragas Hall and the E&CS building. All students can access the Unix network and the [Virtual PC Lab](#) from off campus or from other computer labs on campus.

## 3 Course Policies

## 3.1 Due Dates:

The course is divided into three parts. Each part has associated assignments and a closing exam. (The final exam following Part III, is cumulative). Most assignments are marked with an explicit due date, and are due at the end of that day (11:59:59PM, ET). You will find these dates on the [outline page](#) and on the course Announcements page on Blackboard.

Late submissions for programming assignments (one in which the principal thing being submitted is source code) will be accepted, at a 5% per day penalty, up until the start of the final exam. Late submissions will **not** be accepted once the scheduled starting time of the final exam has begun. Late submissions of quizzes, non-programming assignments, and exams are not accepted.

Exceptions to these dates will be made only in situations of unusual and unforeseeable circumstances beyond the student's control.

"I've fallen behind and can't catch up", "I'm having a busier semester than I expected", or "I registered for too many classes this semester" are **not** grounds for an extension.

## 3.2 Academic Honesty:

Everything turned in for grading in this course must be your own work.

The instructor reserves the right to question a student orally or in writing and to use his evaluation of the student's understanding of the assignment and of the submitted solution as evidence of cheating. Violations will be reported to the Office of Student Conduct & Academic Integrity for consideration for possible punitive action.

Students who contribute to violations by sharing their code/designs with others may be subject to the same penalties.

Students are expected to use standard Unix protection mechanisms (`chmod`) to keep their assignments from being read by their classmates. Failure to do so will result in grade penalties, at the very least.

This policy is **not** intended to prevent students from providing legitimate assistance to one another. Students are encouraged to seek/provide one another aid in learning to use the operating system, in issues pertaining to the programming language, or to general issues relating to the course subject matter.

Students should avoid, however, explicit discussion of approaches to solving a particular programming assignment, and under no circumstances should students show one another their code for an ongoing assignment, nor discuss such code in detail.

### Use of Online Resources

You may **not** post details of course assignments, projects, or tests at online Forums, Bulletin Boards, Homework sites, etc., soliciting help.

You may use information that you have not solicited but have located, subject to the following restrictions:

- Just as when writing a paper, if you use someone else's ideas, you must cite your sources appropriately. Within code, such citations appear in comments.

Example:

```
:  
double x = 23.0;  
double xsqrt = sqrt(x);  
// Search algorithm based upon code by S Zeil at  
// https://www.cs.odu.edu/~zeil/cs361/latest/Public/functionAnalysis/index.html#orderedsequentialsearch  
int loc = 0;  
while (loc < arraySize && numbers[loc] < xsqrt)  
:  
:
```

- Just as when writing a paper, if you use someone else's words (code), you must cite your sources appropriately **and** mark the quoted text. Within code, such citations appear in comments.

Example:

```
:  
double x = 23.0;  
double xsqrt = sqrt(x);  
// Begin quoted code from S Zeil at  
// https://www.cs.odu.edu/~zeil/cs361/latest/Public/functionAnalysis/index.html#orderedsequentialsearch  
int loc = 0;  
while (loc < listLength && list[loc] < searchItem)  
{  
    ++loc;  
}  
// End quoted code  
:  
:
```

- Failure to appropriately cite any such "found code" will be taken as evidence of plagiarism.
- The overall principle stated in the first sentence of this section remains in effect. "Everything turned in for grading in this course must be your own work." If the bulk of your assignment, project, test answer, etc., are copied, even with appropriate citation, to the degree that, in the judgment of the instructor, you have

not demonstrated your own knowledge of the course material, you will receive a zero for that submission.

### **3.3 Grading:**

Assignments & Quizzes:	45%
Part 1 Exam:	15%
Part 2 Exam:	15%
Final Exam:	25%

[Further notes](#) on grading.

## **4 Educational Accessibility:**

Old Dominion University is committed to ensuring equal access to all qualified students with disabilities in accordance with the Americans with Disabilities Act (ADA). The Office of Educational Accessibility (OEA) is the campus office that works with students who have disabilities to provide and/or arrange reasonable accommodations.

- If you experience a disability which will impact your ability to access any aspect of the course, present me with an accommodation letter from OEA so that we can work together to ensure that appropriate accommodations are available to you.
- If you feel that you will experience barriers to your ability to learn and/or complete examinations in the course but do not have an accommodation letter, consider scheduling an appointment with OEA to determine if academic accommodations are necessary.

The Office of Educational Accessibility is located at 1021 Student Success Center, and their phone number is (757)683-4655. Additional information is available at the [OEA website](#).

# Course Policies

- [Syllabus](#)

Please read this at the start of the semester, if not sooner.

- [Communications](#)

Please read this at the start of the semester.

- [Preparing and Submitting Assignments](#)

Skim this at the start of the semester, and read in detail before starting on an assignment.

- Notes on [grading](#).

# Library

## 1 Reference Material

### 1.1 PDF Copies of Lecture Notes

For those who might want to read the lecture notes offline, each of the PDF documents below contains all lecture notes and assorted other documents for the course. Assignments are *not* included.

The content of each file is the same. Only the page sizes have been varied to facilitate reading on different types of devices.

- A conventional [letter size](#) page.
- A [4x3](#) landscape page for viewing on tablets with that ratio of width to height (usually larger tablets, 10-12in diagonals).
- An [8x5](#) landscape page for viewing on tablets with that ratio of width to height (usually somewhat smaller tablets, 7-9in diagonals, designed for watching “wide-screen” videos).

This is something of an experimental feature. Let me know if you try it and find it useful.

### 1.2 Review material from Earlier Courses

- [Unix & Linux \(CS252\)](#)
- [C++ Programming \(CS150/250\)](#) (CS333 was an online course that combined material from CS150 and CS250 into a single, accelerated course. Although we no longer offer CS333, I have preserved the website from its last offering as a convenient place to review content from CS150 and CS250.)

### 1.3 The C++ Language

- [cplusplus.com](#) has an extensive set of tutorials and reference material on C++
- [cppreference.com](#) is also an excellent reference site.
- The [C++ FAQ](#) (Frequently Asked Questions) has a wealth of information about object-oriented programming in general and about C++ specifically.

### 1.4 The C++ Standard Library (std/STL)

- The already-mentioned [cplusplus.com](#) describes the STL and also the portions of the std library that SiliconGraphics does not (e.g., iostream, string).
- Also very useful is this [C++ reference](#).

In particular, take note of

- An excellent [summary sheet for std::generic\\_algorithm](#)
  - This [listing of std::containers](#)
- My own reference sheets for [containers](#) are not as complete and up-to-date, but they do include complexity info as part of the summary.

## 2 Complexity

- [The Idiot's Guide to Big-O](#)
- [Big-O notation explained by a self-taught programmer](#)

## 3 Algorithm Animations

A number of the more important or interesting algorithms covered in this course are available as animations using my [AlgAE](#) (Algorithm Animation Engine) system.

These allow you to view a picture of the data being manipulated while you “play” an algorithm like a DVD, forward or reverse, pausing, moving continuously or just a short hop at a time.

Each animation is packaged as a separate Jar file. To play it, you will need to have Java Runtime Environment (JRE) or Development Kit (JDK), version 1.8 or better. The JRE allows you to run compiled Java programs. The JDK is the JRE plus the Java compiler. Get either of them [here](#).

## 4 Software Downloads

### 4.1 Compilers

A *compiler* translates your programming language source code into an executable. A compiler is **not** a program that allows you to create, edit, run, test, & debug your code. Code::Blocks, Eclipse, XCode, etc. are *not* compilers – they are IDEs (see below).

- [Installing a \(free\) C++ compiler on your own PC](#)

## 4.2 IDEs

An IDE is a program that “surrounds” a compiler and provides support for a variety of programming activities, including writing code, compiling it, correcting errors, running and testing the resulting program, and debugging the program.

The most popular IDEs for C++ programming at ODU are Code::Blocks and Eclipse. Both IDEs are free, and both can be installed on Windows, OS/X, and Linux PCs.

- [Installing a \(free\) C++ IDE on your own PC](#)

# Course Orientation and Intro

Steven Zeil

## Contents:

- [1 Course Website](#)
- [2 Course Policies](#)
  - [2.1 Questions](#)
  - [2.2 Communications](#)
  - [2.3 Grading](#)
- [3 Weekly Reviews](#)
- [4 Keys to Success](#)
  - [4.1 Where to Go from Here?](#)

## Welcome to CS 361

- Course website: [Blackboard](#)
- Instructors:
  - [Steven Zeil](#)
    - email: [zeil@cs.odu.edu](mailto:zeil@cs.odu.edu)
    - [Office hours](#)
  - [Susan Zehra](#)
    - email: [szehra@cs.odu.edu](mailto:szehra@cs.odu.edu)
    - [Office hours](#)

---

## Course Themes

- Common data structures and algorithms
  - expanding your programming toolkit
  - when/how to choose among alternatives
- How do they do that in C++?
  - Interfaces to common data structures provided in the C++ std library
- Predicting program performance
  - Analysis of algorithms (“big-O”)
    - (Yes, there *will* be math.)

## 1 Course Website

### Quick Tour of the Course Website

- Start at the BlackBoard site
  - Hosts announcements, calendars, tests, forums
- Most of the content is on a CS Dept server
  - Lecture notes, labs, assignments
  - This material remains accessible after the end of the semester

---

### The Outline Page

The Outline page is the “heart” of the course

- Lecture notes
- Assigned readings from the text book
- Labs and self-assessments (ungraded activities)
- Assignments, tests

## 2 Course Policies

- Details in [course syllabus](#)

- All students are responsible for reading the syllabus and adhering to the policies described there!

## 2.1 Questions

I expect questions.

- This is one of the big differences between a web course and an F2F (face-to-face) course.
  - In an F2F course, I am primarily a *lecturer*.
  - In a web course, I am primarily a tutor.
- You have options for how to ask questions: office hours, email, & forums.

## 2.2 Communications

- Office hours will be conducted by network conferencing.
- email to instructor
  - Make sure that “CS361” appears somewhere in the subject line
    - But that should not be the **only** thing in the subject line
  - I try to answer all emailed questions within 24 hours on weekdays, 48 on the weekend.
    - But email does get lost/blocked, so if you don’t hear from me in that time frame, send it again.
- Course Forums (Blackboard)
  - For open discussions unrelated to graded activities

## 2.3 Grading

Assignments & Quizzes:	45%
Part 1 Exam:	15%
Part 2 Exam:	15%
Final Exam:	25%

- All assignment, quiz, and exam scores are normalized
- ### 2.3.1 Due Dates
- Course is divided into three parts
    - Each part ends with an exam
      - final (part III) exam is cumulative
    - Exam dates are fixed
  - Assignments & quizzes in each part are due on the date shown in the outline, at 11:59:59PM EDT.
    - Late submissions on programming assignments will be accepted, at a penalty of 5% per day.
      - No submissions are accepted after the scheduled start of the final exam.
    - Late submissions will not be accepted on quizzes and non-programming assignments.
  - Readings, self-assessments, & other activities, if not given an explicit due date, should be completed by the ending date given for that entire module.

## 3 Weekly Reviews

We will hold an open weekly Review Meeting each Weds at 7:30PM.

- Link to attend is in the [Outline](#).
- Meetings will be recorded for those unable to attend.
- Primary purpose of these meetings will be to answer student questions
- First meeting will walk through the process of submitting assignments and the grading procedures associated with them.

## 4 Keys to Success

### Pace Yourself

- There’s a lot of material in this course
- Plan to finish the readings and ask any questions you have about them in the first half of the time allotted to a module.

Work on the assignments in the 2nd half. Not just the last day!

- There's not time to waste by staying stuck on one assignment.
  - If you get stuck ask questions
    - Take the time to frame your question carefully so that I have enough info to actually give you a helpful answer.
  - If you are stuck for more than a few days, it's time to move on.
    - Read my solution and tests.
    - Run the tests through your own code
    - If you don't understand how my solution works, **ask**.

## 4.1 Where to Go from Here?

- Go to the [Policies page](#).
  - Read the [Syllabus](#).
  - Read the [Communications](#) policy.
- Then head off to the [Outline Page](#)
  - Before the first Review Session,

# Setting Up Your Personal Programming Environment

Steven Zeil

Last modified: May 19, 2020

## Contents:

- [1 What You Need](#)
- [2 How to get what you need](#)
- [3 Test Your Environment Out](#)

You are going to be doing a lot of C++ programming in this course.

Now, before you actually get involved with the details of an assignment, is the time to decide how you are going to do that and to get everything set up that you need.

## 1 What You Need

You will need an environment that includes

- a C++ compiler in the g++ family
- a debugger that works with your compiler
- an IDE that can tie these things together, that provides
- support for makefiles for projects with multiple executables and easily controlled compiler settings.

Most students entering this course will have used Code::Blocks in earlier C++ courses. Code::Blocks is a fine IDE for beginners, but **Code::Blocks will not be sufficient in this course.**

## 2 How to get what you need

1. You can work remotely. All of the software that you need is available on the CS network machines.

Use [X2Go](#) to connect to our Linux servers and run [Eclipse](#) as your IDE. CS252 is a pre-requisite to this class, so everyone in this course should already know how to do these things.

One advantage of doing this is that you will be developing your code using the exact same machines and software that I will be using to grade it, so there will be minimal chances of unpleasant surprises.

You may be able to use the CS Dept's [Virtual Computer Portal](#) to run Eclipse and C++ for some assignments. But it won't work for all of them, so I don't recommend it.

2. You can set things up to work locally on your own PC. Most people will probably prefer to do this most of the time, for convenience, improved performance, and independence from possible network problems.

You can find my guide on setting up your personal programming environment [here](#).

If you go this route (and I fully expect that most people will), it may be worth your while to transfer your completed code to one of our Linux servers and check it there just before submitting. Again, the idea is to avoid unpleasant surprises when you get your grade back from me.

## 3 Test Your Environment Out

...before it's time to actually start working on an assignment.

1. Download [this sample C++ application](#), and unpack it into a convenient directory.

2. Launch Eclipse.

3. Import this code as a project in Eclipse:
  - From the `File` menu, select "Import...", then "C/C++", then "Existing Code as Makefile Project".
  - Browse the "Existing Code Location" to find the directory where you unpacked the source code files.
  - Give the project a name, select "C++", and make sure that the ToolChain is the one you want.
  - Click Finish.

4. Try compiling the code. You will encounter some syntax errors. Fix them and recompile.

5. When compilation is successful, this project produces two executables, one the "real" application, and one a set of unit tests on the functions. Both will be listed under "Binaries".

6. Try running the main application, `spellCheck`. Right-click on the `spellCheck` binary, and select "Run as...Local C/C++ application".

You should see the program run (after re-compiling, if necessary). In the Console area, you will be prompted for input. Type some words, hit Enter, and see what happens.

7. As a general rule, if a program expects input from standard in, you don't want to type the text every time you run the program. It makes much more sense to save the input text in a file and redirect that to the program input.

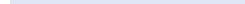
You should know how to do [redirection](#) from the command line. You can also do it from within Eclipse.

- Right-click on the spell-check binary, and select Run as...Run Configurations....
- Find the C/C++ Application configuration for spellCheck in the column on the left and select it.
- On the right, go to the Common tab. Place a check by Input File, click the Workspace... button, and navigate to the test0.txt file in your project directory.
- Click Run and watch the results.

8. Look in the file spellcheck.cpp. Set a breakpoint in the function loadDictionary at the first line that says "getLine(...".

Right-click on the spell-check binary, and select Debug as...Local C/C++ Application. (This will reuse the configuration you set up a moment ago to redirect the input.)



When the debugger pauses, you will see a line of controls that look like this: . Hover your mouse over each of these to see what they do.

The program will have stopped at the first executable line inside main. Use the Resume control and verify that the debugger stops at your selected breakpoint. Examine the variables such as word and dict.

Use the Step over control a few times, watching how the values of the variables change.

When you are done, use the Stop control or use Resume and let the program run to completion.

9. If necessary, return to the normal C/C++ perspective. You can do this from the Window menu, or look in the upper-right-hand corner for this small control:



Right-click on the unittest perspective and select Run as...Local C/C++ Application. Watch the unit tests run. (Don't be concerned about the fact that one test case fails. That's deliberate.)

10. That's one way to run unit tests. But there's a better way.

Right-click on the unittest executable and select Run as...Run Configurations. On the left, select "C/C++ Unit Test".

There is no unit test configuration for this executable yet, so find the New Configuration button and click it to create one.

On the right, go to the C/C++ Testing tab. The unit test framework we are using follows the "Test Anywhere Protocol", so from the Tests Runner list, select "Tap Tests Runner".

Click "Run"

In a moment you will be able to see, at a glance, which test cases passed and which one failed.

11. To get more information on the failed test case, click on the test with the blue 'X' icon. You will see a message that describes the test that failed, including both what data the test expected to see and what it actually observed.

Click on location line with the red 'X' icon. Eclipse will take you right to the point of the failure.

12. Now, suppose that you wanted to debug the failed test. To save your own time, you probably don't want to rerun all of the tests. As it happens, the unit test framework we are using will allow you to specify what tests you want to run as a command-line parameter. That is, if you were executing this unittest program from the command line, you could type

```
unittest InitListSearch
```

to run just that one test case. (In fact, you can abbreviate test case names, so "I" would be sufficient.)

Eclipse allows you to run executables with command-line parameters. Right-click the unittest executable and select Run as...Run Configurations. Select your unittest configuration under "C/C++ Unit Test".

On the right, go to the Arguments tab. Type "Init" into the Program Arguments box and then click Run. you will see that, this time, only that one test is run.



13. Finally, look in the main tool bar for this control: . Click on the drop-down arrow and look at the menu. You have here quick access to relaunch any of the Run Configurations you have used recently.

# Abstraction and Abstract Data Types

Steven J. Zeil

Last modified: May 13, 2020

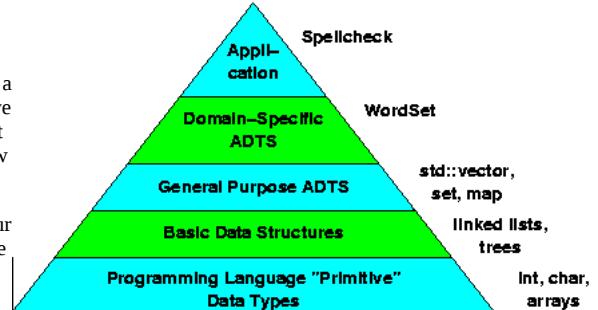
## Contents:

- [1 Abstraction](#)
  - [1.1 Procedural Abstraction](#)
  - [1.2 Data Abstraction](#)
- [2 Abstract Data Types](#)
  - [2.1 ADTs as contracts](#)
  - [2.2 ADT Implementations](#)
  - [2.3 Where Do ADTs Come From?](#)

If we were to look at a program that is actually large enough to require a full team of programmers to implement it, you would probably not be surprised to find that it would not be organized as a single, large, monolithic unit, but instead as a large number of cooperating functions. You already know how to design and write functions in C++. What may, however, come as a bit of a surprise if you have not worked much with programs that size is that even these functions will be further organized into higher level structure.

I've tried to illustrate that structure in the diagram that you see here. At the top we have the main application program, for example, a spell checker. The code that occurs at this level is very specific to this application and is the main thing that differentiates a spell checker from, let's say, a spreadsheet. On the other hand, at the very bottom of the hierarchy, we have all the basic primitive data types and operations, such as the int type, the char type, addition, subtraction, and so on, that are provided by our programming language, C++. These primitive operations may very well show up in almost any kind of program.

In between those, we have all the things that we can build on top of the language primitives on our way working up towards our application program. Just above the language primitives we have the basic data structures, structures like linked lists or trees. We're going to spend a lot of time this semester looking at these kinds of structures - you may already be familiar with some of them. They are certainly very important. And yet, if we stopped at that level, we would wind up "building to order" for every application. As we move from one application to another we would often find ourselves doing the same kinds of coding, over and over again.



What's wrong with that? Most companies, and therefore most programmers, do not move from one application to a wildly different application on the next project. Programmers who been working on "accounts receivable" are unlikely to start writing compilers the next week, and programmers who have been writing compilers are not going to be writing control software for jet aircraft the week after. Instead, programmers are likely to remain within a general application domain. The people who are currently working on our spell checker may very well be assigned to work on a grammar checker next month, or on some other text processing tool. That means that any special support we can design for dealing with text, words, sentences, or other concepts natural to this application to make may prove valuable in the long run because we can share that work over the course of several projects.

And so, on top of the basic data structures, we expect to find layers of reusable libraries. Just above the basic data structures, the libraries are likely to provide fairly general purpose structures, such as support for look-up tables, user interfaces, and the like. As we move up in the hierarchy, the libraries become more specialized to the application domain in which we're working. Just below the application level, we will find support for concepts that are very close to the spell checker, such as "words", "misspellings", and "corrections".

The libraries that make up all but the topmost layer of this diagram may contain individual functions or groups of functions organized as Abstract Data Types. In this lesson, we'll review the idea of Abstract Data Types and how to use C++ classes to implement them. None of the material in this lesson should be entirely new to you - all of it is covered in CS 250.

## 1 Abstraction

In general, abstraction is a creative process of focusing attention on the main problems by ignoring lower-level details.

In programming, we encounter two particular kinds of abstraction: procedural abstraction and data abstraction.

### 1.1 Procedural Abstraction

A procedural abstraction is a mental model of what we want a subprogram to do (but not how to do it).

**Example:** if you wanted to compute the length of the a hypotenuse of a right triangle, you might write something like

```
double hypotenuse = sqrt(side1*side1 + side2*side2);
```

We can write this, understanding that the sqrt function is supposed to compute a square root, even if we have no idea how that square root actually gets computed.

- That's because we understand what a square root is.

When we start actually writing the code, we [implement](#) a procedural abstraction by

- assigning an appropriately named “function” to represent that procedural abstraction,
- in the “main” code, calling that function, trusting that it will actually do *what* we want but not worrying about *how* it will do it, and
- finally, writing a function body using an appropriate algorithm to do what we want.

In practice, there may be many algorithms to achieve the same abstraction, and we use engineering considerations such as speed, memory requirements, and ease of implementation to choose among the possibilities.

For example, the `sqrt` function is probably implemented using a technique completely unrelated to any technique you may have learned in grade school for computing square roots. On many systems, `sqrt` doesn’t compute a square root at all, but computes a polynomial function that was chosen as a good approximation to the actual square root and that can be evaluated much more quickly than an actual square root. (This initial approximation may then be refined by several iterations of a numerical technique known as the Newton-Raphson method.)

Does it *bother* you to hear that the implementation of `sqrt` might not, in fact, work by computing a square root? It shouldn’t. You are still being guaranteed that if you call `sqrt(x)`, then multiply the return value by itself, you will get something that is approximately `x`. And isn’t that what you were presumably looking for?

## 1.2 Data Abstraction

Data abstraction works much the same way. A [data abstraction](#) is a mental model of *what* can be done to a collection of data. It deliberately excludes details of *how* to do it.

### 1.2.1 Example: elapsed time

Elapsed time refers to a period or extent of time, as opposed to an instant in time that you might read in a single glance at a clock.

Elapsed time is generally measured in a mixture of hours, minutes, and seconds.

That’s it. That’s probably all you need to know for you and I to agree that we are talking about a common idea.

### 1.2.2 Example: a book

How to describe a *book*?

- If we are implementing a card catalog and library checkout, it is probably enough to list the [metadata](#)
  - (e.g., title, authors, publisher, date).
- If, however, we are going to be working on a project involving the full text of the document (e.g., [automatic metadata extraction and indexing](#)), then we might need all the pages and all the text.
- Of course, if we were building bookshelves, we might need more physical attributes such as size and weight!

### 1.2.3 Example: Publishing

In many cases, we have a collection of related data abstractions that work together to define a simulated “world” in which our later programming design will take place.

Let’s explore that by expanding on the abstraction of a book, putting into a context.

#### Books

We start with Books:

- Books have titles
- Books have ISBN numbers

These numbers serve as unique identifiers. There may be many books with the same title, but the ISBN number distinguishes between them.

- Books have a Publisher – we’ll come back to this.
- Books have a list of Authors.

There may be other properties of books, but we’re going to stick with just these few simple ones.

The diagram does not attempt to show all of the data associated with books, just enough to illustrate our idea.

#### Authors

- Authors have names.
- Authors have addresses.

#### Books and Authors

However, the most interesting thing about Authors is how they work in relationship with books:

- For any Book, we should be able to list the authors of that book.
- For any Author, we should be able to list the books they have written.

## Addresses

Addresses are also an abstraction. For our purposes, we will follow U.S. conventions:

- An Address has a street address.
- An Address has a City.
- An Address has a State.
- An Address has a Zip Code.

## Publishers

Our final abstraction in this world is that of Publishers.

- Publishers have names.

Again, however, the most interesting thing about Publishers is their relationship to other abstractions:

- A Publisher publishes Books.
  - And, as already noted, every book has one Publisher.
- A Publisher has a list of authors under contract.

However, not every Author has a contract with a Publisher. In our example diagram, AddisonWesley publishes a book by Oscar Wilde, but Wilde is long dead and his book is actually in the public domain, meaning that anyone can publish it.

---

We will return to this Publishing world example and its component abstractions many times over the course of the semester.

### 1.2.4 Summary

A data abstraction is a mental model. It's not enough to start programming with. The evolution from mental model to implemented code proceeds in two steps:

1. We devise an ADT *interface* for the data abstraction that describes what we want to do with it.
2. We *implement* the data abstraction by
  - choosing an appropriate *data structure* - a specific construction of data, and
  - providing appropriate *operations* (algorithms) to manipulate that data.

This course is primarily about data structures and algorithms - the implementation level of the abstractions.

## 2 Abstract Data Types

The mental model offered by a data abstraction gives us an informal understanding of how and when to use it. But because it is simply a mental model, it does not tell us enough information to program with it.

An *abstract data type* (ADT) captures this model in a programming language interface.

Definition: An *abstract data type* (ADT) is a type name and a set of operations on that type where

- Users of the ADT are expected to alter/examine values of this type only via the operations provided.
- The creator of the ADT promises to leave the operation specifications unchanged.
- The creator of the ADT is allowed to change the code of the operations at any time, as long as it continues to satisfy the specifications.
- The creator of the ADT is also allowed to change the data structure actually used to implement the type.

### 2.1 ADTs as contracts

An ADT represents a contract between the ADT developer and the users (application programmers).

What do we gain by holding ourselves to this contract?

- Application programmers can be designing and even implementing the application before the details of the ADT implementation have been worked out. This helps in team development.
- The ADT implementors know exactly what they must provide and what they are allowed to change.
- ADTs designed in this manner are often re-usable. By reusing code, we save time in
  - implementation, and

- debugging
- We gain the flexibility to try/substitute different data structures to actually implement the ADT, *without needing to alter the application code*.
- By encouraging modularity, application code becomes more readable.

### Example 1: Book ADT

Suppose we are working on a program to track the offerings of a book publishing house. Among the abstractions that we might want to capture are:

- A book has a title, one or more authors, a publisher, and a unique identification code.
- An author has a name, an address, and a numeric identifier that never changes. One author may have written or co-written multiple books.
- A publisher has a name and address and a catalog of books that they publish.
- An address consists of a street address, city, state, and zip code.

These are, of course, highly simplified for the purpose of this example. There's probably a lot more information that would really need to be captured about books and authors, and our address structure assumes U.S. addresses and is fairly limited, even given that assumption.

So, among the ADTs for this system, we will have

#### Address

```
get/set Street
get/set City
get/set State
get/set Zipcode
```

#### Author

```
get/set Name
get/set Address
get/set ID
```

#### Book

```
get/set Title
get number of Authors
get/set Author[i]
get/set ID
```

#### Publisher

```
get/set Name
get/set Address
get the catalog
```

#### Catalog

```
get # of books
get a specific book from the catalog
```

One possible C++ realization for the Book ADT for that abstraction is shown here:

```
typedef ... Book;

void initialize (Book&);

void setTitle(Book&, std::string);
std::string getTitle (const Book&);

void setPublisher(Book&, Publisher);
Publisher getPublisher (const Book&);

int getNumAuthors(const Book&);
void addAuthor (Book&, Author*);
void removeAuthor (Book&, Author*);
Author* getAuthor (const Book&, int);

void setIdentifier(Book&, std::string);
std::string getIdentifier (const Book&);
```

This isn't a particularly good ADT, but it illustrates the main point — with the information provided here, we could write code that manipulates Books, even though we haven't yet established how the Book will actually be implemented (e.g., how to store and retrieve the multiple authors).

## 2.2 ADT Implementations

An ADT is *implemented* by supplying

- a *data structure* for the type name.
- coded *algorithms* for the operations.

We sometimes refer to the ADT itself as the *ADT specification* or the *ADT interface*, to distinguish it from the code of the *ADT implementation*.

In C++, implementation is generally done using a C++ class, e.g.,

```
class Book {  
public:  
    std::string getTitle() const;  
    void setTitle(std::string theTitle);  
  
    Publisher* getPublisher() const;  
    void setPublisher(const Publisher*);  
  
    int getNumberOfAuthors() const;  
  
    Author getAuthor (int authorNumber) const;  
    void addAuthor (Author);  
    void removeAuthor (Author);  
  
    std::string getISBN() const;  
    void setISBN(std::string id);  
  
private:  
    std::string title;  
    Publisher* publisher;  
    int numAuthors;  
    std::string isbn;  
    :  
};
```

Like most classes, this uses public/private to enforce the ADT contract.

Would it really be so awful if we did not hide the data members in the private area? Why not just do:

```
struct Book {  
    std::string title;  
    Publisher* publisher;  
    int numAuthors;  
    std::string identifier;  
  
    void addAuthor (Author*);  
    void removeAuthor (Author*);  
    :  
};
```

Well, for one thing, we have not yet figured out how we actually intend to store all the authors. As it is, our “proper” ADT version would allow you or a member of your team to start writing application code that *uses* Books even before we've made up our mind on that issue.

Second, the ADT version gives us the opportunity to change our minds later if we decide to store the authors in a different way. For example, right now you might be inclined to write something like this:

```
struct Book {  
    std::string title;  
    Publisher* publisher;  
    int numAuthors;  
    std::string identifier;  
  
    void addAuthor (Author*);  
    void removeAuthor (Author*);  
    Author* authors[1000];  
};
```

but, by the time this course is finished, we will have explored many better options. However, once that array has been made a public part of the interface, changing our mind later would be possible only at the risk of breaking application code that already references that array. By contrast, the ADT interface effectively raises a firewall around our choice of data structure for the authors - no matter what destruction we might later wreak on that *authors* data structure, none of the rest of the code in our program will get burned.

## 2.3 Where Do ADTs Come From?

ADTs may be

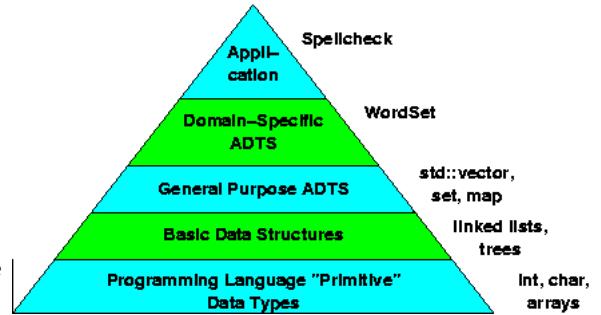
- General-Purpose: often containers of “smaller” ADTs
  - There are the focus of this course.
- Domain-specific, used in multiple related applications
- Application-specific

Domain and application-specific ADTs generally reflect the real-world objects found in the application domain.

- For example, if you were working on software for managing course enrollment at a college or university, even without knowing what the specific program was supposed to do, you might guess that you would need ADTs for:

- Students
- Courses
- Faculty
- Semesters
- Classrooms

The process of discovering good domain-specific and application-specific ADTs is a major theme in CS 330.



# ADT Interfaces in C++

Steven J. Zeil

Last modified: May 13, 2020

## Contents:

- [1 Declaring New Data Types](#)
- [2 Manipulating an ADT: Attributes and Operations](#)
  - [2.1 Attributes](#)
  - [2.2 Operations](#)
  - [2.3 Example: the Publishing World](#)
- [3 Realizing the Interface in C++](#)
  - [3.1 Basic Attributes & Operations](#)
- [4 Constructors](#)
  - [4.1 Constructing Books with Multiple Authors](#)
  - [4.2 The Default Constructor](#)
  - [4.3 The Copy Constructor](#)
- [5 Destructors](#)
- [6 Operators](#)
  - [6.1 Assignment](#)
  - [6.2 I/O](#)
  - [6.3 Comparisons](#)
- [7 Other Interface Details](#)
  - [7.1 Const Correctness](#)
  - [7.2 Inline Functions](#)

An ADT is an interface to a collection of data.

In C++, the usual mechanism for realizing an ADT is the public part of a class.

In this lesson, we look at how to write interfaces that express the abstract idea that we have about different kinds of data.

## 1 Declaring New Data Types

In the course of designing a program, we discover that we need a new data type. What are our options?

Well, to a certain degree, it depends upon whether or not we already have a data type that provides the interface (data and functions) that we want for this new one.

- On rare occasions, we may be lucky enough to have an existing type that provides *exactly* the capabilities we want for our new type. In that case, we can get by just using a `typedef` to create an “alias” for the existing type. For example, suppose that we were writing a program to look up the zip code for any given city. Obviously, one of the kinds of data we will be manipulating will be “city names”. Now, a city name is pretty much just an ordinary character string, and anything we can do to a character string we could probably also do to a city name. So we might very well decide to take advantage of the existing C++ `std::string` data type and declare our new one this way:

```
typedef std::string CityName;
```

or, a newer syntax,

```
using CityName = std::string;
```

These basically gives us a way to attach a more convenient name to an existing type.

- We may need to build our ADT “from scratch”, specifying exactly what data and functions we want to associate with the new ADT. This is done by declaring a new class and listing the data and function members we need to achieve our desired abstraction. For example, in building a system to track the output of a publishing company, we might write the class shown here (we’ll talk about where the data and function members come from just a bit later).

```
class Book {  
public:  
    std::string getTitle() const;  
    void setTitle(std::string theTitle);  
  
    std::string getISBN() const;  
    void setISBN(std::string id);  
    ...  
private:  
    ...  
};
```

- In between those two extremes, we sometimes have an existing type that provides almost everything we want for our new type, but needs just a little more data or a few more functions to make it what we want. In that case we may be able to create an extended version of the existing type by using inheritance. For example, suppose that our publishing company has a number of books that are grouped into series (e.g., “Cookbooks of the World”, volumes 1-28). This leads to the idea (abstraction) of a “book in series”, that would be identical to a regular Book except for providing two addition data items, a series title and a volume

number. We might then write the declaration shown here, which creates a new type named “BookInSeries” that is identical to the existing type Book except that it carries two additional data fields and the functions that provide access to them.

```
class BookInSeries: public Book {  
public:  
    std::string getSeriesTitle() const;  
    void setSeriesTitle(std::string theSeries);  
  
    int getVolume() const;  
    void setVolume(int);  
private:  
    std::string seriesTitle;  
    int volume;  
};
```

Inheritance is a powerful technique that lies at the heart of object-oriented programming. We won’t use much of it in this course, but it is covered extensively in CS 330.

## 2 Manipulating an ADT: Attributes and Operations

An ADT is more than just a type name. It also provides a description of what other code can do to manipulate that data.

### 2.1 Attributes

As you work with ADTs, it becomes very clear that many things we do to data are little more than fetching and storing smaller pieces of data that we think of as components of the ADT.

For example,

- A Book has a title. So we can imagine fetching the title of a book or storing a title in a book.
- A Book has a publisher. So we can imagine fetching the publisher of a book or storing the publisher of a book.
- A Book has an [ISBN](#). So we can ...

We call these kinds of properties the [attributes](#) of an ADT.

We sometimes show the attributes of an ADT using a UML class diagram, like this:

For the most part, attributes will eventually be implemented as private data members in C++, with public get/set functions to provide access to the data. e.g.,

```
class Book {  
    string title;  
    Publisher publisher;  
    string isbn;  
public:  
    :  
    string getTitle() const;  
    void setTitle(const string& newTitle);  
  
    const Publisher& getPublisher() const;  
    void setPublisher (const Publisher& publ);  
  
    string getISBN() const;  
    void setISBN(const string& newISBN);  
};
```

But that’s not always the case.

For one thing, the data members might need to be a little different. We will see in a later lesson that we almost certainly want to keep a pointer to the publisher in our data:

```
class Book {  
    string title;  
    Publisher* publisher;  
    string isbn;  
public:  
    :  
    const Publisher& getPublisher() const;  
    void setPublisher (const Publisher& publ);  
    :  
};
```

but that decision would not alter our intended interface or our *idea* that we are fetching and storing publishers of a book.

Or, we may decide that some of our attributes should be *immutable* – not subject to change once the Book has been initialized. We might decide, for example, that a book can never change its ISBN, in which case we would provide a “get” but not a “set” function:

```
class Book {  
    string title;  
    Publisher publisher;  
    string isbn;
```

```

public:
:
string getTitle() const;
void setTitle(const string& newTitle);

const Publisher& getPublisher() const;
void setPublisher (const Publisher& publ);

string getISBN() const; // no setISBN() -- ISBN is immutable
};

```

Both of these variants would still be described as

i.e., a Book has title, publisher, and ISBN attributes.

In addition, there may be clever ways to store data that don't follow the simple "attribute == data member" rule. One of my favorite examples is the idea of a calendar date. We think of dates as having attributes day of the month, month, and year. But it's rare to see this implemented as

```

class Date {
    int day;
    int month;
    int year;
public:
:
int getYear() const;
void setYear(int);

int getMonth() const;
void setMonth(int);

int getDay() const;
void setDay(int);
};

```

That's because some of the most common operations we would like to perform on calendar dates are things like "get the day one week before this one", or "how many days apart are these two dates?" And those sorts of calculations are much easier with a different underlying data structure:

```

class Date {
    int totalDaysSinceEpoch;
    // Epoch is Jan 1, 1970
public:
:
int getYear() const;
void setYear(int);

int getMonth() const;
void setMonth(int);

int getDay() const;
void setDay(int);
};

```

With this, we can answer our "date calculation" queries easily: "get the day one week before this one" is simply `totalDaysSinceEpoch - 7`, and "how many days apart are these two dates?" is `date1.totalDaysSinceEpoch - date2.totalDaysSinceEpoch`. Now, the calculation to get/set the year/month/day are kind of horrible, but the three-int version of `Date` would need to do roughly the same calculations in order to answer the date calculation queries.

The moral of this story:

An attribute is a property that we think of as being fetched from and stored in an ADT, regardless of whether we eventually implement it that way.

## 2.2 Operations

Not everything we do to an ADT is an attribute. There are also [operations](#), manipulations of the data that do not appear to us as simple data storage.

The date calculations are examples of "operations". In UML, we list operations separately in the bottom third area of a class diagram.

In our Book example, we might consider another part of our idea of what makes something book-like:

- A Book has a list of authors.

Now, we *could* treat that as just another attribute:

```

class Book {
    string title;
    Publisher publisher;
    string isbn;
    authors: Author[25];
public:
:
string getTitle() const;
void setTitle(const string& newTitle);

```

```

    const Publisher& getPublisher() const;
    void setPublisher (const Publisher& publ);

    string getISBN() const;
    void setISBN(const string& newISBN);

    const Author* getAuthors() const;
    void setAuthors(Author[] authorList);
};

}

```

But this feels clunky. Just imagining writing code to, say, remove an author or change one author's address if you needed to retrieve the entire array first, find the author you wanted to change, make the change, and then put the whole modified array into place.

It makes much more sense to instead envision *operations* that let us work with authors one at a time:

Now, you might object that this no longer captures the abstract idea that

- A Book has a list of authors.

But that just shows that "has a" does not always mean "attribute". In fact, UML has a way of saying that one thing is part of another without representing them as attributes:

The diamond-headed arrow is called the "aggregation" arrow, and can be read as "is part of", i.e., "an Author is part of a Book". The "\*" markers stand for "many" or "an arbitrary number", because a book can have multiple authors, but many authors have written more than one book.

## 2.3 Example: the Publishing World

In the prior lesson, we used the following to illustrate our data abstraction for things related to books.

As we move forward into considering an ADT, we would shift our focus away from the individual objects and more towards the classes:

# 3 Realizing the Interface in C++

Eventually, a C++ programmer must sit down to capture the abstract ideas of attributes and operations in a C++ class. At this stage, we are really only worrying about the public interface of a class. Generally this involves adding public member functions and, rarely, public data members.

Of course, C++ being what it is, there's lots to think about even when only working on the interface.

## 3.1 Basic Attributes & Operations

As a first pass, we can translate the attributes and operations into C++.

For example,

```

class Address {
public:
    std::string getStreet() const;
    void setStreet (std::string theStreet);

    std::string getCity() const;
    void setCity (std::string theCity);

    std::string getState() const;
    void setState (std::string theState);

    std::string getZip() const;
    void setZip (std::string theZip);

private:
    ...
};

```

Remember, this lesson is about ADT *interfaces*. We will worry about the function bodies and data members when we look at [ADT implementation](#):

```

class Author
{
public:

    std::string getName() const;
    void setName (std::string theName);

    const Address& getAddress() const;
    void setAddress (const Address& addr);

    int numberOfBooks() const;
    Book& getBook(int i);

```

```

void addBook (Book& b);
void removeBook (Book& b);

private:
    ;
};
```

## 4 Constructors

Most ADTs need a way to initialize their attributes. Of course, we could do it one data field at a time:

```

Address addr;
addr.setStreet ("21 Pennsylvania Ave.");
addr.setCity ("Washington");
addr.setState ("D.C.");
addr.setZip ("10001");

Author doe;
doe.setName ("Doe, John");
doe.setAddress (addr);
```

Now this isn't really all that great.

- It would be quite easy to forget to initialize one or more of the data fields.
- As the code gets modified over the course of the project, some misguided programmer might place some additional lines of code in between the declaration and the last line of initialization. (This is particularly likely if we need a nontrivial computation to come up with the initial values for these data fields.)

That, in turn, leads to a real possibility of our using `addr` or `doe` before all the data fields have been initialized, with potentially disastrous results.

- There's no way to initialize data members that are not attributes, or attributes that are immutable, because these will not have their own "set" functions.
- Last, but not least, the process just takes too much effort — too many lines of code written just to initialize one data object.

C++ provides a special kind of member function to streamline the initialization process. It's called a [constructor](#). Constructors are unusual in that their name must be the same as their "return type", the name of the class being initialized. A constructor is called when we define a new variable, and any parameters supplied in the definition are passed as parameters to the constructor.

Suppose we add a constructor to each of our `Address` and `Author` classes.

```

class Address {
public:
    Address (std::string theStreet,
              std::string theCity,
              std::string theState,
              std::string theZip);

    std::string getStreet() const;
    void setStreet (std::string theStreet);

    ;
};

class Author
{
public:
    Author (std::string theName, const Address& theAddress);

    std::string getName() const;
    void setName (std::string theName);

    ;
};
```

Then we can create a new author object much more easily:

```

Address addr ("1600 Pennsylvania Ave.",
              "Washington",
              "D.C.", "10001");

Author doe ("Doe, John", addr);
```

or, since the `addr` variable is probably only there only for the purpose of initializing this author and is probably not used elsewhere, we can do:

```

Author doe ("Doe, John",
           Address (
               "1600 Pennsylvania Ave.",
               "Washington",
               "D.C.", "10001"),
           1230157);
```

## 4.1 Constructing Books with Multiple Authors

Our Book class will need a constructor as well, but this one is slightly complicated by the fact that books can have multiple authors. How can we pass an arbitrary number of authors to a constructor (or to any function, for that matter)?

```
class Book {
public:
    Book(const std::string& title,
          const std::string& isbn,
          const Publisher& publisher,
          Author* authors = nullptr, ①
          int numAuthors = 0); ②
    ...
};
```

For now, we'll do it by passing an array of Authors (①), together with an integer indicating how many items are in the array. (In a later lesson, we will see much cleaner ways to handle this problem.)

- Remember, the data type “Author\*” could denote either a pointer to a single Author value, or an array of Author objects.
- The = `nullptr` (①) and = 0 (②) define default values for their function parameters. If we write a call to this (constructor) function and supply only the first three parameters, the compiler will fill in these defaults for the missing final two parameters.

With that constructor in place, we could initialize books by first building an appropriate author array, then declaring our new Book object. For example, given these authors:

```
Author weiss (
    "Weiss, Mark",
    Address("21 Nowhere Dr.", "Podunk", "NY", "01010")
);
Author doe (
    "Doe, John",
    Address("212 Baker St.", "Peoria", "IL", "12345")
);
Author smith (
    "Smith, Jane",
    Address("47 Scenic Ct.", "Oahu", "HA", "54321")
);
```

we can create some books like this:

```
Publisher prenticeHall = ...;
Author textAuthors[] = {weiss};
Book text361 ("Data Structures and Algorithms in C++", "013284737X",
              macmillan,
              textAuthors, 1);

Author recipeAuthors[] = {doe, smith};
Book recipes ("Cooking with Gas", "0-124-46821", prenticeHall,
              recipeAuthors, 2);
```

Taking advantage of the default parameters, we could write

```
Book unknownText ("Much Ado About Nothing", "123456789", prenticeHall);
```

In cases where we really do have multiple authors (e.g., `recipes`), that's probably as simple as it's going to get. It's a bit awkward for books that only have a single author, however. But, just as with any other functions in C++, nothing limits us to having a single constructor function as long as the formal parameter list for each constructor is different. Since single authorship is likely to be a common case, it might be convenient to declare another Book constructor to serve that special case.

```
class Book {
public:
    Book(const std::string& title,
          const std::string& isbn,
          const Publisher& publisher,
          Author* authors = nullptr,
          int numAuthors = 0);

    Book(const std::string& title,
          const std::string& isbn,
          const Publisher& publisher,
          Author& author1);
    ...
};
```

We could then use either constructor, as appropriate, when creating Books.

```
Book text361 ("Data Structures and Algorithms in C++", "013284737X",
              macmillan,
              weiss);

Author recipeAuthors[] = {doe, smith};
```

```
Book recipes ("Cooking with Gas", "0-124-46821", prenticeHall,  
            recipeAuthors, 2);
```

## 4.2 The Default Constructor

The [default constructor](#) is a constructor that takes no arguments. This is the constructor you are calling when you declare an object with no explicit initialization parameters. E.g.,

```
std::string s;
```

A default constructor might be declared to take no parameters at all, like this:

```
class Author  
{  
public:  
    Author();  
  
    Author (std::string theName, const Address& theAddress);  
  
};
```

Or by supplying a default value for *every* parameter in an existing constructor:

```
class Address {  
public:  
  
    Address (std::string theStreet = std::string("unknown"),  
             std::string theCity = std::string(),  
             std::string theState = std::string(),  
             std::string theZip = std::string());  
  
};
```

Either way, we can *call* it with no parameters.

Why is this called a “default” constructor? There’s actually nothing particularly special about it. It’s just an ordinary constructor, but it is *used* in a special way.

Whenever we create an array of elements, the compiler implicitly calls the default constructor to initialize each element of the array.

For example, if we declared:

```
std::string words[5000];
```

then each of the 5000 elements of this array will be initialized using the default constructor for string.

In fact, if we don’t *have* a default constructor for our class, then we *can’t* create arrays containing that type of data. Attempting to do so will result in a compilation error.

Because arrays are so common, it is rare that we would actually want a class with no default constructor.

### 4.2.1 Compiler-Generated Constructors

It’s hard to imagine a situation in which we create a class and would deliberately prefer that, when we declare a variable of that class type, its data members should remain uninitialized.

That’s just poor programming practice.

The designers of the C++ language felt the same way. So the C++ compiler tries to be helpful:

If we create no constructors at all for a class, the compiler generates a default constructor for us.

Why does the compiler generate, specifically, a *default* constructor? Well, any other choice would mean that the compiler would have to figure out what data we needed to supply to initialize each data member, and what we wanted to do with that data to carry out the initialization. That’s well beyond the intelligence of any compiler, so it opts for the simple case of generating a constructor that takes no parameters at all.

That automatically generated default constructor works by initializing every data member in our class with its own default constructor. In many cases (e.g., strings), this does something entirely reasonable for our purposes.

Be careful, though: the “default constructors” for the primitive types such as int, double and pointers work by doing nothing at all — leaving us with an *uninitialized* value containing whatever bits happened to have been left at that particular memory address by earlier calculations/programs.

## 4.3 The Copy Constructor

Another specialized constructor, the [copy constructor](#) is a constructor that takes a (const reference to a) value of the same type its only parameter. For example:

```
Book (const Book& b);
```

- like the default constructor, there is nothing unusual about the copy constructor itself.
  - It's just an ordinary constructor.
- It's special because of the number of common situations in which it gets [used](#).
  - The compiler often generates implicit calls to this constructor in places where we might not expect it.

### 4.3.1 Where are Copy Constructors Used?

The copy constructor gets used in 5 situations:

1. When you declare a new object as a copy of an old one:

```
Book book2 (book1);
```

or

```
Book book2 = book1;
```

2. When a function call passes a parameter “by copy” (i.e., the formal parameter does not have a &):

```
void foo (Book b, int k);
:
Book text361 (...);
foo (text361, 0); // foo actually gets a copy of text361
```

3. When a function returns an object:

```
Book foo (int k);
{
    Book b;
    :
    return b; // a copy of b is placed in the caller's memory area
}
```

4. When data members are initialized in a constructor’s initialization list from a single parameter of the same type:

```
Author::Author (std::string theName,
                Address theAddress, long id)
: name(theName),
  address(theAddress),
  identifier(id)
{
```

We’ll cover initialization lists in the next lesson.

5. When an object is a data member of another class for which the compiler has generated its own copy constructor.

### 4.3.2 Compiler-Generated Copy Constructors

As you can see from that list, the copy constructor gets used a [lot](#). It would be very awkward to work with a class that did not provide a copy constructor.

So, again, the compiler tries to be helpful.

If we do not create a copy constructor for a class, the compiler generates one for us.

- This automatically generated version works by copying each data member via [their](#) individual copy constructors.
- For data members that are primitives, such as `int` or pointers, the copying is done by copying all the bits of that primitive object.
  - For things like `int` or `double`, that’s just fine.
  - But, we will see later that this may or may not be what we want for pointers.

## 5 Destructors

The flip-side of initializing new objects is cleaning up when old objects are going away. Just as C++ provides special functions, constructors, for handling initialization, it also provides special functions, [destructors](#), for handling clean-up.

A destructor for the class `Foo` is a function of the form

```
-Foo();
```

You should never call a destructors explicitly. Instead the compiler generates a call to an object's destructor when

- Execution passes outside of the `{ ... }` within which the object was declared. For example, if we wrote

```
if (someTest)
{
    Book b = text361;
    cout << b.getTitle() << endl;
}
```

what the compiler would actually generate would be something along the lines of

```
if (someTest)
{
    Book b = text361;
    cout << b.getTitle() << endl;
    b.-Book(); // implicitly generated by the compiler
}
```

- When we delete a pointer to an object, the object's destructor is (implicitly) called prior to actually recovering the memory occupied by the object. For example, if we were to write:

```
Book *bPointer = new Book(text361); // initialized using copy constructor
:
cout << bPointer->getTitle() << endl;
delete bPointer;
```

what the compiler would actually generate would be something along the lines of

```
Book *bPointer = new Book(text361); // initialized using copy constructor
:
cout << bPointer->getTitle() << endl;
bPointer->-Book();
free(bPointer); // recover memory at the address in bPointer
```

Now, we have not declared or implemented a destructor for any of our classes, yet. That might be OK.

If you don't provide a destructor for a class, the compiler generates one for you automatically.

The automatically generated destructor simply invokes the destructors for any data member objects. We will [later](#) discuss the circumstances under which we need to provide our own destructors.

## 6 Operators

One of the truly delightful, or demonic, depending on your point of view, aspects of C++ is that almost every operator in the language can be declared in our own classes. In most programming languages, we can write things like "`x+y`" or "`x<y`" only if `x` and `y` are integers, floating point numbers, or some other pre-defined type in the language. In C++, we can add these operators to any class we design, if we feel that they are appropriate.

The basic idea is really very simple. Almost of the things we use as operators, including

```
+ - * / | & < > <= >= == != ++ -- -> += -= *= /=
```

can be overloaded. This is also true of some things you probably don't consider as operators (the `[ ]` used in array indexing and the `( )` used in function calls). There are only a few things that look like operators but *cannot* be overloaded: `'.'` and `::`.

All of those operator symbols are actually "syntactic sugar" for a function whose name is formed by appending the operator itself to the word "operator".

For example, if you write `a + b*(-c)`, that's actually just a shorthand for

```
operator+(a, operator*(b, operator-(c)))
```

and if you write

```
testValue = (x <= y);
```

that's actually a shorthand for

```
operator=(testValue, operator<=(x, y));
```

Now, this shorthand is so much easier to deal with that there's seldom any good reason to actually write out the long form of these calls. But knowing the shorthand means that we can now declare new operators in the same way we declare new functions. For example, we could declare a + operator for Books this way:

```
Book operator+ (const Book& left, const Book& right);
```

and then call it like this:

```
Book b = book1 + book2;
```

but that's probably not a good idea, because it's not clear just what it means to add two books together. Consequently any implementation we wrote for the operator+ function for Books would probably be nonintuitive and confusing to other programmers.

There are, however, a few operators that *would* make sense for Book and for almost all ADTs.

## 6.1 Assignment

Chief among these is the assignment operator. When we write book1 = book2, that's shorthand for either book1.operator=(book2) or operator=(book1, book2), depending on whether operator= has been declared as a member function of the Book class or as an ordinary, stand-alone function.

Assignment is used so pervasively that a class without assignment would be severely limited (although sometimes designers may *want* that limitation). Consequently,

If a class does not provide an explicit assignment operator, the compiler will attempt to generate one. The compiler-generated version will simply assign each data member in turn.

We'll look more closely at when and how to write our own assignment operators in a [later lesson](#).

## 6.2 I/O

Another, **very** common set of operators that programmers often write for their own code are the I/O operators << and >>, particularly the output operator <<.

Indeed, I would argue that you should *always* provide an output operator for every class you write, even if you don't expect to use it in your final application.

My reason for this is quite practical. Sooner or later, you're going to discover that your program isn't working right. So how are you going to debug it? The most common thing to do is to add debugging output at all the spots where you think things might be going wrong. So you come to some statement:

```
doSomethingTo(book1);
```

and you realize that it might be really useful to know just what the value of that book was before and after the call:

```
cerr << "Before doSomethingTo: " << book1 << endl;
doSomethingTo(book1);
cerr << "After doSomethingTo: " << book1 << endl;
```

Now, if you have already written that operator<< function for your Book class, you can proceed immediately. If you *haven't* written it already, do you really want to be writing and debugging that new function *now*, when you are already dealing with a different bug?

We can add output to some of our existing classes:

```
class Address {
public:
    Address (std::string theStreet = std::string("unknown"),
              std::string theCity = std::string(),
              std::string theState = std::string(),
              std::string theZip = std::string());
    std::string getStreet() const;
    void setStreet (std::string theStreet);
    std::string getCity() const;
    void setCity (std::string theCity);
    std::string getState() const;
    void setState (std::string theState);
    std::string getZip() const;
    void setZip (std::string theZip);
private:
    ...
};

std::ostream& operator<< (std::ostream& out, const Address& addr);
```

The output operator must be declared **outside** of the class. That's because, function members of a class are always called with an object of that class type on the left. For example

```
Address addr;  
addr.setState("VA");
```

For the operator shorthand, that translates to the object being to the left of the operator. E.g., "address1 = address2" is equivalent to "address1.operator=(address2)".

But look at how we use an output operator:

```
cout << addr;
```

The thing on the left is not the ADT we are trying to write – it's a C++ `std::ostream`. So `operator<<` could only be a member function of class `std::ostream`. And that's no good for our purposes. So if we want to support output of `Address` or `Author` or any of our other classes, we need to do it as an ordinary, non-member function.

Another unusual point: notice the return value on the `operator<<` declaration. Each implementation of `operator<<` is supposed to return the output stream to which we are writing. It's the fact that `<<` is an expression that returns the stream we are writing to that let's us write "chains" of output expressions like:

```
cout << book1 << " is better than " << book2 << endl;
```

which is treated by the compiler as if we had written:

```
((cout << book1 << " is better than ") << book2) << endl;
```

Look at the second `<<`. What is it being given as its left-side parameter? It's being given `(cout << book1)`, the value returned by the first `<<` operator. What do we expect a `<<` to be given as its left-side parameter? We expect it to be given the output stream to which we want to write.

So, by convention, every implementation of `operator<<` returns the stream that it has just written into, so that a chain of `<<` calls will, each in turn, pass the stream on to the next one in line.

## 6.3 Comparisons

After assignments and I/O, the most commonly programmed operators would be the relational operators, especially `==` and `<`. The compiler never generates these implicitly, so if we want them, we have to supply them.

```
class Address  
{  
    ...  
    bool operator==(const Address&) const;  
    ...  
};
```

The trickiest thing about providing these operators is making sure we understand just what they should *mean* for each individual ADT. For example, if I were to write

```
if (address1 == address2)
```

what would I expect to be true of two Addresses that passed this test? Probably that the two addresses would have the same street, city, state, and zip - in other words, that all the data fields should themselves be equal. In that case, this would be a reasonable implementation:

```
bool Address::operator==(const Address& right) const  
{  
    return (street == right.street)  
        && (city == right.city)  
        && (state == right.state)  
        && (zip == right.zip);  
}
```

We could do something similar for `Author`:

```
bool Author::operator==(const Author& right) const  
{  
    return (name == right.name)  
        && (address == right.address);  
}
```

which, interestingly, makes use of the `Address::operator==` that we have just defined.

In C++, we traditionally provide `operator<` whenever possible (and many code libraries assume that this operator is available). Again, just what this should *mean* depends upon the uses we intend to make of our ADT, but there are a few hard and fast rules. We should always design `operator<` so that

- If  $x < y$  is true, then  $y < x$  must be false and  $x == y$  must be false.
- If  $x == y$  is true, then  $x < y$  must be false and  $y < x$  must be false.
- If  $x == y$  is false, then either  $x < y$  must be true or  $y < x$  must be true, but both of those should not be true.

# 7 Other Interface Details

## 7.1 Const Correctness

In C++, we use the keyword `const` to declare constants. E.g.,

```
const double PI = 3.14159;
```

But it also has two other important uses:

- indicating what formal parameters a function will look at, but promises not to change
- indicating which member functions don't change the object they are applied to

These last two uses are important for a number of reasons

- This information often helps make it easier for programmers to understand the expected behavior of a function.
- The compiler may be able to use this information to generate more efficient code.
- This information allows the compiler to detect many potential programming mistakes.

A class is *const-correct* if

- Any formal function parameter that will not be changed by the function is passed by copy<sup>1</sup> or as a const reference (`const &`).
- Every member function that does not alter the object it's applied to is declared as a const member.

Our current version of Author gives some good examples of this process:

```
class Author
{
public:
    Author();

    Author (std::string theName, const Address& theAddress);

    std::string getName() const;
    void setName (std::string theName);

    const Address& getAddress() const;
    void setAddress (const Address& addr);

    int numberOfBooks() const;
    std::string& getBook(int i);
    const std::string& getBook(int i) const;

    void addBook (Book& b);
    void removeBook (Book& b);

    bool operator== (const Author& right) const;
    bool operator< (const Author& right) const;

private:
    :
};
```

- For example, the `setName` function does not alter the string that it receives as a parameter, so that parameter is passed by copy, as are all the others highlighted like this.
- Similarly, the `setAddress` function does not alter the `Address` that it receives as a parameter, so that parameter is passed by `const reference`.

How do you know whether to declare your functions to pass by copy or by const reference? Either works for input parameters, but copy tends to be faster for non-class data types or very small classes and const reference tends to be faster for classes with more than a few bytes worth of data members.

- The `getName` and `getID` functions do not alter the `Author` they are applied to, so they are marked as `const member functions`. It's a pretty sure bet that most functions named "get..." are supposed to fetch data and not change the object that they are applied to.

Similarly, when we use `==` or `<` to compare one author to another, we would expect that such a comparison would not affect the value of the author to the left of the operator, so we mark these functions as `const` also.

### 7.1.1 Const Member Function Pairs

It's worth highlighting one peculiarity that const correctness introduces to ADT interfaces. Look at these member functions:

```
class Author
{
```

```

public:
    ...
    std::string& getBook(int i);
    const std::string& getBook(int i) const;
    ...
};

```

The first of these will be applied to Authors that are not `const`. The second will be applied to `Author` objects that are `const`. For example,

```

void transferFirstBook (const Author& a, Author& b)
{
    string title = a.getBook[0]; // uses 2nd, const versions
    b.getBook[0] = title; // uses 1st, non-const version
    a.getBook[0] = "out of print"; // Compilation error!
}

```

The first version returns a reference (address) of the author's book and so can actually be used to change the book inside the `Author` object.

The second version returns a `const` reference, which cannot be changed. And that is entirely consistent with the fact that the book `a` is itself marked as `const`, which is a promise by `transferFirstBook` that it will not do anything that could change the value of `a`.

This use of paired functions differing only in the `const`-labels on the function, its parameters, and its return type, is a **very** common pattern in C++.

## 7.2 Inline Functions

Many of the member functions in this example are simple enough that we might consider an alternate approach, declaring them as inline functions.

When we define a function, it is usually compiled into a self-contained unit of code. For example, the function

```

int foo(int a, int b)
{
    return a+b-1;
}

```

would compile into a block of code equivalent to

```

stack[1] = stack[3] + stack[2] - 1;
jump to address in stack[0]

```

where the "stack" is the [runtime stack](#) a.k.a. the [activation stack](#) used to track function calls at the system level, `stack[0]` is the top value on the stack, `stack[1]` the value just under that one, and so on. A function call like

```
x = foo(y,z+1);
```

would be compiled into a code sequence along the lines of

```

push y onto the runtime stack;
evaluate z+1;
push the result onto the runtime stack
push (space for the return value) onto the runtime stack
save all CPU registers
push address RET onto the runtime stack
jump to start of foo's body
RET: x = stack[1]
pop runtime stack 4 times
restore all CPU registers

```

As you can see, there's a fair amount of overhead involved in passing parameters and return address information to a function when making a call. The amount of time spent on this overhead is not really all that large. If the function body contains several statements in any kind of loop, then the overhead is probably a negligible fraction of the total time spent on the call. But if the function body is only a line or two and does not involve a significant amount of computation, then the time spent on that overhead can be, by comparison, quite large.

Many ADTs have member functions that are only one or two lines long, and often trivial lines at that. In our `Book` class, for example, we might feel that all of our `get...` and `set...` functions meet that description.

For these functions, the overhead associated with each call may exceed the time required to do the function body itself. Furthermore, because these functions are often the primary means of accessing the ADT's contents, sometimes these functions get called thousands of times or more inside the application's loops, causing the total amount of time spent on the overhead of making the function call to become significant.

For these kinds of trivial functions, C++ offers the option of declaring them as [`inline`](#). An inline function can be written one of two ways. **First**, it can be written inside the class declaration. **Second**, we can place the reserved word `inline` in front of the function definition written in its usual place outside the class declaration.

For example, I might rewrite our `Address` class like this:

```

class Address {
public:
    Address (std::string theStreet = std::string("unknown"),

```

```

        std::string theCity = std::string(),
        std::string theState = std::string(),
        std::string theZip = std::string());

std::string getStreet() const {return street;}
void setStreet (std::string theStreet) {street = theStreet;}

std::string getCity() const {return city;}
void setCity (std::string theCity) {city = theCity;}

std::string getState() const {return state;}
void setState (std::string theState) {state = theState;}

std::string getZip() const {return zipcode;}
void setZip (std::string theZip) {zipcode = theZip; }

bool operator==(const Address& right) const;
bool operator< (const Address& right) const;

private:
    std::string street;
    std::string city;
    std::string state;
    std::string zipcode;
};

inline Address::Address (std::string theStreet, std::string theCity,
                       std::string theState, std::string theZip)
{
    street = theStreet;
    city = theCity;
    state = theState;
    zipcode = theZip;
}

std::ostream& operator<< (std::ostream& out, const Address& addr);

```

Either way, we then remove the function body from the .cpp file.

When we make a call to an inline function, the compiler simply replaces the call by a compiled copy of the function body (with some appropriate renaming of variables to avoid conflicts). So, if we have

```

inline int foo(int a, int b)
{
    return a+b-1;
}

```

and we later make a call

```
x = foo(y,z+1);
```

This would be compiled into a code sequence along the lines of

```

evaluate z+1, storing result in tempB
evaluate y + tempB - 1, storing result in x

```

Most of the overhead of making a function call has been eliminated.

Inline functions can reduce the run time of a program by removing unnecessary function calls, but, used unwisely, may also cause the size of the program to explode. Consequently, they should be used only by frequently-called functions with bodies that take only 1 or 2 lines of code. For larger functions, the times savings would be negligible (as a fraction of the total time) while the memory penalty is more severe, and for infrequently used functions, who cares?

Inlining is only a *recommendation* from the programmer to the compiler. The compiler may ignore an inline declaration and continue treating it as a conventional function if it prefers. In particular, inlining of functions with recursive calls is impossible, as is inlining of most virtual function calls (don't worry if you don't know what those are). Many compilers will refuse to inline any function whose body contains a loop. Others may have their own peculiar limitations.

---

1: This is the default method of parameter passing in C++. It's what you get when your formal parameter types do not have an &.

# Implementing ADTs in C++

Steven J. Zeil

Last modified: May 13, 2020

## Contents:

- [1 Data Members](#)
- [2 Function Members](#)
  - [2.1 Many Function Members are Simple](#)
  - [2.2 Separately Compiled Member Functions](#)
  - [2.3 Initialization Lists](#)
  - [2.4 What is “this”?](#)
- [3 Filling in the Related ADTs](#)
- [4 The Perils of Pointers](#)
  - [4.1 Pointers and References](#)
  - [4.2 Pointers are Perilous](#)
- [5 Pointers in the Publishing World](#)

Once we have defined our desired ADT interface, it's time to look at implementation – choosing a data structure and writing algorithms for the member functions.

In C++, this is generally done using a C++ class. The class enforces the ADT contract by dividing the information about the implementation into public and private portions. The ADT designer declares all items that the application programmer can use as public, and makes the rest private. An application programmer who then tries to use the private information will be issued error messages by the compiler.

## 1 Data Members

To be useful, an ADT must usually contain some internal data. These are declared as *data members* of the class. To continue our prior example, what data should we associate with a book? Earlier, we said that “A Book has a title, one or more authors, and a unique identification code.” and we can declare these as shown here.

```
class Book {  
    ;  
private:  
    std::string title;  
    std::string isbn;  
    Publisher publisher;  
    int numberOfAuthors;  
    static const int maxAuthors = 12;  
    Author authors[maxAuthors]; // array of authors  
};
```

Note that it's not unusual for data members to involve still other ADTs (e.g., the Author and Publisher data types in this example are presumably declared elsewhere as classes with a number of different data fields, including name, address, etc.).

## 2 Function Members

### 2.1 Many Function Members are Simple

Many ADTs are rich in attributes and lean in operations. That means that many of the function members will be “gets” and “sets” that do little more than fetch and store in private data members.

```
class Book {  
public:  
    Book();  
  
    Book(const std::string& title, const std::string& isbn,  
          const Publisher& publisher,  
          Author* authors = nullptr, int numAuthors = 0);  
  
    std::string getTitle() const {return title;}  
    void setTitle(std::string theTitle) {title = theTitle;}  
  
    const Publisher& getPublisher() const {return publisher;}  
    void setPublisher(const Publisher& publ) {publisher = publ;}  
  
    int getNumberOfAuthors() const {return numberOfAuthors;}  
  
    Author getAuthor (int authorNumber) const;  
    void addAuthor (Author);  
    void removeAuthor (Author);  
  
    std::string getISBN() const {return isbn;}  
    void setISBN(std::string id) {isbn = id;}  
  
    bool operator== (const Book& right) const;  
    bool operator< (const Book& right) const;
```

```

private:
    std::string title;
    std::string isbn;
    Publisher publisher;
    int numAuthors;
    static const int maxAuthors = 12;
    Author authors[maxAuthors]; // array of authors
};

```

Because these are so simple, they are good candidates for inlining.

## 2.2 Separately Compiled Member Functions

The remaining member functions would then have their bodies placed in a separate .cpp file. The usual convention is to package each ADT into its own pair of appropriately named .h and .cpp files. The class declaration above would typically appear in a file named book.h, as shown here.

```

#ifndef BOOK_H
#define BOOK_H

#include <iostream>
#include <string>

#include "author.h"

class Publisher;           ①

class Book {
public:
    ...
private:
    std::string title;
    Publisher* publ;
    int numAuthors;
    static const int maxAuthors = 12;
    Author* authors; // array of authors
    std::string isbn;
};

std::ostream& operator<< (std::ostream& out, const Book& book)

#endif

```

- ① This declaration of Publisher is called an “incomplete” or “forward” declaration. It’s really more of a promise to provide a complete declaration elsewhere. In the meantime, we can, given this simple statement that we will eventually have a class named Publisher, write declarations that use pointers or references to Publisher. Forward declarations are used when we need to break a circular dependency: every book has a publisher, but every publisher has a catalog of books. If we tried to store a whole publisher object inside each book, and (several) whole books inside each publisher, ... Well, we’d never see an end to it. So we break that cycle by making one of those data fields a pointer (in this case, Publisher\*) which, given that forward declaration, we are allowed to do without getting all the details of the Publisher class.)

Then, in a separate file named book.cpp we would place the remaining function definitions (bodies).

```

#include "book.h"
#include "publisher.h"

#include <cassert>
#include "arrayManipulation.h"

using namespace std;

Book::Book()           ①
: title(), isbn(),      ②
  publisher(),
  numAuthors(0)
{
}

Book::Book(const std::string& theTitle, const std::string& theISBN,
           const Publisher& thePublisher,
           Author* theAuthors, int theNumAuthors)
: title(theTitle), isbn(theISBN),          ②
  publisher(thePublisher), numAuthors(0)
{
    for (int i = 0; i < theNumAuthors; ++i)
    {
        addAuthor(theAuthors[i]);
    }
}

void Book::addAuthor (const Author& au)      ①
{
    assert (numAuthors < MaxAuthors);
    addIfNotPresent(authors, numAuthors, au.getName()); ③
}

```

```

}

void Book::removeAuthor (const Author& au)
{
    removeIfPresent (authors, numAuthors, au.getName()); ③
}

Author Book::getAuthor(int i) const
{
    return Author(authors[i], Address());
}

bool Book::operator==(const Book& right) const
{
    return getISBN() == right.getISBN(); ④
}

bool Book::operator<(const Book& right) const
{
    return getISBN() < right.getISBN(); ④
}

std::ostream& operator<< (std::ostream& out, const Book& book)
{
    out << book.getTitle() << ", by ";
    for (int i = 0; i < book.numberofAuthors(); ++i)
    {
        if (i > 0)
            out << ", ";
        out << book.getAuthor(i);
    }
    out << ";" << book.getPublisher() << ", " << book.getISBN();
    return out;
}

```

- ① Notice that all the member functions follow the pattern of describing the function that is being defined as “Book::”. Why is this necessary?

Book:: means “inside the class Book”.

If we write

```

void addAuthor (const Author& au)
{
    :
}

```

then we are writing a perfectly normal C++ function named “addAuthor” that is not a member of any class. There’s nothing wrong with that, but it’s not what we want. Adding the “Book::”

```

void Book::addAuthor (const Author& au)
{
    :
}

```

makes it clear that we are supplying the body to the “addAuthor” function *that is a member of the Book class*.

Notice that there is no Book:: attached to operator<<. That’s because, as explained [in the last lesson](#), the output operator is **not** a member function of the class it is applied to.

- ② This is an example of an initialization list, which we will explain shortly.
- ③ These functions come from the included header `arrayManipulation.h` which we will not discuss at this time, but you can find it [here](#).
- ④ The comparison operators only compare books by ISBN, because we stated as part of our [Book abstraction](#) that the ISBN functioned as a unique identifier for books.

## 2.3 Initialization Lists

There is a subtle but important distinction in C++ between *initializing* data and *assigning* data.

Initializing data is something that happens only once, when that data value is created. For example, we assign variables like this:

```

int x;
string s1, s2;
:
x = 23;
s1 = s2;

```

We initialize variables like this:

```

int x = 0;
double pi (3.14159);
string s1 = "abcdef";
string s2 ("abcdef");
string s3 {"abcdef"};
string s4;
Address holmesHome ("221b Baker St", "London", "", "England");

```

You can easily tell the difference between assignment and initialization because the initialization is combined with declaring the variables, so the type name appears in front of the variable being initialized.

Initialization of a value takes place by invoking the constructor for that value's data type. The first three strings are each initialized by calling the constructor for class `std::string` that expects a single parameter of type `const char*` (the data type of string literals like "abcdef"). If a constructor takes exactly one parameter, then the initialization can be written in any of the three forms shown above for `s1`, `s2`, and `s3`. They all mean exactly the same thing. The `{ }` form for `s3` is a relatively recent addition to C++, and is intended to make initialization look less like ordinary assignment or like an ordinary function call.

When writing constructors for classes, it's tempting to write them much like ordinary functions, using a series of assignments:

```

Book::Book()
{
    title = "";
    isbn = "";
    publisher = Publisher();
    numAuthors = 0;
}

Book::Book(const std::string& theTitle, const std::string& theISBN,
           const Publisher& thePublisher,
           Author* theAuthors, int theNumAuthors)
{
    title = theTitle;
    isbn = theISBN;
    publisher = thePublisher;
    numAuthors = 0;
    for (int i = 0; i < theNumAuthors; ++i)
    {
        addAuthor(theAuthors[i]);
    }
}

```

But these are all assignments. Before they even begin, the data members of the class have already been initialized (using their class's default constructor). Then our assignments come along and write all over those initial values. This is rather inefficient.

We could simplify the `Book` default constructor a bit. Since we are overwriting each of the data members with the same value that their default constructors will have already initialized them to, we could write

```

Book::Book()
{
    numAuthors = 0;
}

```

(Integers, by default, simply keep whatever random bits happened to be in that block of memory, so we still need to put some better value into `numAuthors`.) Although faster, this has the disadvantage of being less helpful to someone reading the code trying to perceive what data members are initialized and how.

Another way to write the same constructor is to make use of an [initialization list](#), a special C++ syntax for initializing data members. It's shown in the highlighted portion of the constructor code below.

```

Book::Book()
: title(), isbn(),
  publisher(),
  numAuthors(0)
{ }

Book::Book(const std::string& theTitle, const std::string& theISBN,
           const Publisher& thePublisher,
           Author* theAuthors, int theNumAuthors)
: title(theTitle), isbn(theISBN),
  publisher(thePublisher), numAuthors(0)
{
    for (int i = 0; i < theNumAuthors; ++i)
    {
        addAuthor(theAuthors[i]);
    }
}

```

These describe how to *initialize* the data members. Each item in the list represents a call to a constructor. For example, in the first constructor

```
title()
```

means to initialize `title` (a `string`) by invoking the `string` constructor that takes no parameters (the default constructor for `string`), while, in the second constructor,

```
title(theTitle)
```

means to initialize `title` by calling the `string` constructor that takes a single expression of type `string` as its only parameter (the `string` copy constructor).

#### Initialization lists

- appear only in constructors and
- *must* used for constants, references (which also cannot be assigned to) and to initialize data members that need elaborate constructor calls of their own.
- *can* be used to initialize any data member.

Initialization lists are often faster and more efficient than doing the same steps via normal assignment.

#### How data members are initialized in constructors

Each data member of a constructor is initialized, before the `{ }` body of the constructor is started, according to the following rules:

1. If a data member `x` is of a class/struct type `T` and `x` is listed in the initialization list as `x(parameters)`, then `x` is initialized using the constructor `T(parameters)`.
2. If a data member `x` is of a primitive (not a class or struct) type `T` and `x` is listed in the initialization list as `x(expression)`, then `x` is initialized as if assignment `x = expression`.
3. If a data member `x` is of a class/struct type `T` and `x` is not listed in the initialization list, then `x` is initialized using the constructor `T()`.

So you can see that all non-primitive data members will be initialized before the constructor's function body begins. It only makes sense to try and make that first initialization be the one that we actually want.

## 2.4 What is “this”?

With the ADT we have now prepared, we could write code like:

```
Book b1;
Book b2;
:
if (b1.getAuthor(0) == b2.getAuthor(0))
{
    :
```

Now, let's look at `getAuthor` again:

```
const Author& Book::getAuthor (int i) const
{
    return authors[i];
}
```

`authors` is, of course, a data member of the `Book` class. But in the \hi{two calls} above, we expect that the `getAuthor` function will retrieve `b1.authors` on the first call and `b2.authors` on the second. But how does the code produced by compiling the function body above “know” which book object it should be working with?

The answer lies in a bit of legerdemain carried out by the C++ language designers. If you were to look at the code actually generated for member functions, you would discover that all (non-static) member functions have a hidden parameter, a pointer to the object that will appear to the left of the ‘.’ when we write calls like `b1.getAuthor(0)`. So when we write:

```
class Book {
public:
    :
    std::string getTitle() const;
    void setTitle(std::string theTitle);
    :
    Author getAuthor (int i) const;
    :
};
```

what the compiler actually generates is

```
class Book {
public:
    :
    std::string getTitle(const Book* this) const;
    void setTitle(Book* this, std::string theTitle);
    :
    Author getAuthor (const Book* this, int i) const;
    :
};
```

Every non-static member function has a hidden first parameter. Its name is “this” and its data type is a pointer to the class being declared.

If the member function is `const`, this will be a `const` pointer. `const` pointers can be used to look at the data they point to, but cannot be used to change that data.

So when we write

```
if (b1.getAuthor(0) == b2.getAuthor(0))
```

how does `Book::getAuthor` know whether to access `b1`'s data members or `b2`'s data members? The answer is that

1. Function calls with the member `'.'` on the left, like

```
b1.getAuthor(0)
```

are actually translated as if we had written

```
Book::getAuthor(&b1, 0);
```

The address of the variable on the left of the `'.'` is passed as the first, hidden parameter.

2. The C++ compiler will insert a dereference of the hidden pointer parameter whenever it sees a data or function member name that can't be properly compiled otherwise. So when we write:

```
Author* Book::getAuthor (int authorNumber) const
{
    return authors[authorNumber];
}
```

the compiler pretends that we wrote

```
Author* Book::getAuthor (const Book* this, int authorNumber)
{
    return this->authors[authorNumber];
}
```

We never *need* to write `this->` in our code. But there are times when we do need to do things to the object on the left of a `'.'` call other than use it with a `->` dereference. In those cases, we need to know that `this` is very real even though it is hidden, that it is a pointer, and that is it available for us to use in our code just like any other function parameter.

## 3 Filling in the Related ADTs

The `Book` is not our only class in this world. Let's start to fill in the other ADTs from our example:

Address is pretty simple:

```
class Address {
public:
    Address (std::string theStreet = std::string("unknown"),
              std::string theCity = std::string(),
              std::string theState = std::string(),
              std::string theZip = std::string());

    std::string getStreet() const {return street;}
    void setStreet (std::string theStreet) {street = theStreet;}

    std::string getCity() const {return city;}
    void setCity (std::string theCity) {city = theCity;}

    std::string getState() const {return state;}
    void setState (std::string theState) {state = theState;}

    std::string getZip() const {return zipcode;}
    void setZip (std::string theZip) {zipcode = theZip;}

    bool operator== (const Address& right) const;
    bool operator< (const Address& right) const;

private:
    std::string street;
    std::string city;
    std::string state;
    std::string zipcode;
};

inline Address::Address (std::string theStreet, std::string theCity,
                       std::string theState, std::string theZip)
: street(theStreet), city(theCity), state(theState), zipcode(theZip)
{ }

std::ostream& operator<< (std::ostream& out, const Address& addr);
```

Note that, again, you can see the use of an initialization list in the Address constructor.

Author seems no more complicated than Book

```
class Author
{
public:
    Author();

    Author (std::string theName, const Address& theAddress);

    std::string getName() const {return name;}
    void setName (std::string theName) {name = theName;}

    const Address& getAddress() const {return address;}
    void setAddress (const Address& addr) {address = addr;}

    int numberOfBooks() const;
    Book& getBook(int i);
    const Book& getBook(int i) const;

    void addBook (Book& b);
    void removeBook (Book& b);

    bool operator==(const Author& right) const;
    bool operator<(const Author& right) const;

private:
    std::string name;
    Address address;

    static const int BookMax = 10;
    int numBooks;
    Book books[BookMax];
};

std::ostream& operator<< (std::ostream& out, const Author& author);
```

But when we consider this together with Book, we can see a problem. If every Author object contains the Books that person has written, and every Book object contains the list of Authors of that book, we have a conflict. So the block of memory for one Book will include bytes reserved to contain several Authors, each of which will set aside some bytes to contain several Books, each of which will... This never ends.

We have a similar problem when we introduce the Publishers:

```
class Publisher
{
public:
    Publisher (std::string theName = std::string());

    std::string getName() const {return name;}
    void setName (std::string theName) {name = theName;}

    int numberOfBooks() const;
    Book& getBook(int i);
    const Book& getBook(int i) const;
    void addBook (Book& b);

    int numberOfAuthors() const;
    Author& getAuthor(int i);
    const Author& getAuthor(int i) const;
    void addAuthor (const Author& au);

    bool operator==(const Publisher& right) const;
    bool operator<(const Publisher& right) const;

private:
    std::string name;

    static const int BookMax = 10;
    int numBooks;
    Book books[BookMax];

    static const int AuthorMax = 20;
    int numAuthors;
    Author authors[AuthorMax];

};

std::ostream& operator<< (std::ostream& out, const Publisher& publ);
```

If every Book contains its Publisher, which contains all of the Books it publishes, each of which contains its Publisher, ... There we go again.

There are a couple of ways out of this dilemma:

### 3.0.1 Cheating – Break the Abstraction

We can “cheat” by not storing the authors and publishers directly in the books, but only storing their names. We would need to alter the interface accordingly, e.g.,

```
class Book {
public:
    Book();

    Book(const std::string& title, const std::string& isbn,
        const Publisher& publisher,
        Author* authors = nullptr, int numAuthors = 0);

    std::string getTitle() const {return title;}
    void setTitle(std::string theTitle) {title = theTitle;}

    const std::string& getPublisher() const {return publisher;}
    void setPublisher(const std::string& publ) {publisher = publ;}

    int getNumberOfAuthors() const {return numAuthors;}

    std::string getAuthor (int authorNumber) const;
    void addAuthor (std::string);
    void removeAuthor (std::string);

    std::string getISBN() const {return isbn;}
    void setISBN(std::string id) {isbn = id;}

    bool operator==(const Book& right) const;
    bool operator<(const Book& right) const;

private:
    std::string title;
    std::string isbn;
    std::string publisher;
    int numAuthors;
    static const int maxAuthors = 12;
    std::string authors[maxAuthors]; // array of author names
};
```

I consider this cheating because it's not the way our original abstract idea worked. If we ever want, for example, to get the address of the author of a book, or to find out if a book's author has other books under the same publisher, we're going to need some significant additional coding and data structures to obtain that kind of information.

Nonetheless, it's a workable approach, and relatively simple, and is an example that we will use in several later lessons.

### 3.0.2 Use Pointers

If we want to stay faithful to our original abstraction, we can do so by breaking the unending “contains” recursion using pointers.

Indeed, this was hinted at in our earlier diagram.

## 4 The Perils of Pointers

...and pointer-like types.

### 4.1 Pointers and References

First, a quick review.

Both pointers and references hold the address of a value.

#### 4.1.1 Pointers

A pointer type is indicated by following a type name with \*, e.g. Author\*.

Usually, we generate a pointer value by allocating an object on the heap:

```
Author* auPtr = new Author("William Shakespeare", avonAddress);
```

Once that pointer value has been obtained, we can copy it to other places, pass it to functions, store it in other data structures, etc.

If we want to get at the entire object denoted by a pointer, we use the \* operator. If we want to get at a function or data member of an object denoted by a pointer, we use the -> operator.

```
Author shakespeare = *auPtr;
string name = auPtr->getName();
```

A special pointer value, the *null pointer*, written in C++ as nullptr, indicates a pointer that does not actually contain a valid address.

Objects allocated on the heap stay there until our code explicitly removes them, which it does via delete.

```
delete auPtr; // hands the allocated storage  
// back to the system for later reuse.
```

There's a special case for arrays. If we use a pointer to point to (the start of) an array:

```
Book* readingList = new Book[30];
```

then to later recover the storage we say

```
delete [] readingList;
```

## 4.1.2 References

References also hold addresses to objects, but they have several key differences from pointers.

A reference type is indicated by following a type name with &, e.g. Author&.

We generate a reference value by initializing it with the object we want it to point to:

```
Author shakespeare ("William Shakespeare", avonAddress);  
Author& auRef = shakespeare; // auRef holds the memory address of shakespeare
```

Once that reference value has been obtained, we can use it to initialize other references, pass it to functions, but we *cannot change where it points to*. This is unlike pointer variables, which can be reassigned the addresses contained in other pointers/

If we want to get at the entire object denoted by a reference, we just use the name of the reference. If we want to get at a function or data member of an object denoted by a reference, we use the . operator.

```
Author shakespeare2 = auRef;  
string name = auRef.getName();
```

In fact, once a reference has been initialized, it looks pretty much like an ordinary variable.

Because references are always initialized by giving them the address of a real object, they cannot be null.

You are probably most familiar with reference types being used in the parameter lists for functions, e.g.,

```
class Book {  
    :  
    void addAuthor(Author& au);  
    :  
}
```

We understand that the & means that we don't want our code to make an actual copy of the Author object that we pass to this function (which might take considerable time) but simply to pass the address of that object – just a few bytes.

However, references can also be used to simplify and speed up code. Consider some code like this:

```
Point points[30000];  
:  
for (int i = 0; i < 10000, ++i)  
{  
    double z = points[2*i+1].y;  
    points[2*i+1].y = points[2*i+1].x;  
    points[2*i+1].x = z;  
}
```

It takes time to do the array calculation points[2\*i+1] and, given how many times we will go around that loop, we might not want to do the same pointless (sorry!) calculation three times in a row. And frankly, this is just easier to read:

```
Point points[30000];  
:  
for (int i = 0; i < 10000, ++i)  
{  
    Point& p = points[2*i+1];  
    double z = p.y;  
    p.y = p.x;  
    p.x = z;  
}
```

Without the &, this code would not actually change any of the values in the array. With the &, however, p holds the address of an element inside the array and lets us look at and alter that array element efficiently.

## 4.1.3 Converting Between Pointers and References

If we have a pointer p, then \*p is actually a reference to the same object.

If we have a reference r, then &r is a pointer to the same object.

#### 4.1.4 const Pointers and const References

Both pointers and references can be declared as `const`. In both cases, it means the same thing: we can use that address to look at the value at that address but not to change that value in any way.

## 4.2 Pointers are Perilous

Pointers introduce a whole host of potential run-time errors if used improperly.

- If we allocate data on the heap and never delete it, we have a [memory leak](#). Leak enough memory and the program may crash or slow the entire machine down.
- If we delete the same address twice, we can corrupt the heap (i.e., the data structure used by the system software to track available memory). Corrupting the heap can cause our program to crash or yield incorrect output.
- If we delete an array without the `[]`, or use `delete []` on a non-array, we could corrupt the heap.
- If we have two or more pointers that hold the same address, and we delete one of them, the others are said to be [dangling](#) – they point to a location whose contents are now unpredictable. Using a dangling pointer can lead to incorrect results or a crashed program.
- If we forget to initialize a pointer, it contains random bits. Using that pointer to access memory will have unpredictable results.

All of these various pointer errors are pernicious and hard to debug. Unlike most “simple” bugs in our code, pointer errors can have different effects on different execution of the program, even if we consistently give the program the same input each time. Worse, pointer errors can have effects that are only seen long after the mistaken pointer action took place and can affect data seemingly unrelated to the pointer itself. This makes it very hard to reason backwards from, say, seeing an incorrect value in the program output to the actual cause of the problem.

## 5 Pointers in the Publishing World

Let's face it. Many programmer would gladly forgo the risk of working pointers at all if the things weren't so blasted useful.

Remember our issue with books inside of authors inside of books inside of...? A few pointers can save the day.

```
class Author
{
public:
    Author();
    Author (std::string theName, const Address& theAddress);

    std::string getName() const {return name;}
    void setName (std::string theName) {name = theName;}

    const Address& getAddress() const {return address;}
    void setAddress (const Address& addr) {address = addr;}

    int numberOfBooks() const;
    Book* getBook(int i);
    const Book* getBook(int i) const;

    void addBook (const Book* b);
    void removeBook (const Book* b);

    bool operator== (const Author& right) const;
    bool operator< (const Author& right) const;

private:
    std::string name;
    Address address;

    static const int BookMax = 10;
    int numBooks;
    Book* books[BookMax];
};
```

Do the same with the Authors inside of Books, and with the Publisher class, and we wind up being able to support our abstraction as originally envisioned.

We're going to leave that version for a while, though, because actually figuring out how and when to delete these objects is something of a nightmare. But we will get to it eventually.

And, in the meantime, we will continue to use pointers, in a more limited fashion, to give ourselves some flexibility in our data structures.

# Copying and Moving Data

Steven J Zeil

Last modified: Feb 12, 2020

## Contents:

- [1 Copying Data – Shallow and Deep Copies](#)
  - [1.1 Copying Blocks of Bits](#)
  - [1.2 Shallow vs Deep Copy](#)
  - [1.3 Choosing Between Shallow and Deep Copy](#)
- [2 The Big 3](#)
  - [2.1 Copy Constructors](#)
  - [2.2 Assignment Operators](#)
  - [2.3 Destructors](#)
- [3 The Rule of the Big 3](#)
- [4 Moving Data – l-values and r-values](#)
  - [4.1 L-values and R-values](#)
  - [4.2 R-values and Returns](#)
  - [4.3 The Rule of the Big 5?](#)
- [5 Summary](#)

Next we turn our attention to a set of issues that are often given short shrift in both introductory courses and textbooks, but that are extremely important in practical C++ programming.

As we begin to build up our own ADTS, implemented as C++ classes, we quickly come to the point where we need more than one of each kind of ADT object. Sometimes we will simply have multiple variables of our ADT types. Once we do, we will often want to copy or assign one variable to another, and *we need to understand what will happen when we do so*. Even more important, we need to be sure that what *does* happen is what we *want* to happen, depending upon our intended behavior for our ADTs.

As we move past the simple case of multiple variables of the same ADT type, we may want to build *collections* of that ADT. The simplest case of this would be an array or linked list of our ADT type, though we will study other collections as the semester goes on. We will need to understand *what happens* when we initialize such a collection and when we copy values into and out of it, and we need to make sure that behavior is what we *want* for our ADTs.

In this lesson, we set the stage for this kind of understanding by looking at how we control initialization and copying of class values.

## 1 Copying Data – Shallow and Deep Copies

One of the most common things we do with data is to copy it from one place to another. After all, the basic assignment statement:

```
x = y;
```

may be the first statement of C++ statement you learned, and assignments constitute the bulk of many programmers' code.

### 1.1 Copying Blocks of Bits

We often envision copying data as a simple process of copying the bits from one location to another. This is a simple, intuitive view, and works well with some data.

#### 1.1.1 Book - simple arrays

```
book1.h +  
  
#ifndef BOOK_H  
#include "author.h"  
#include "publisher.h"  
  
class Book {  
public:  
    Book();  
  
    Book (std::string theTitle, const Publisher* thePubl,  
          int numberofAuthors, Author* theAuthors,  
          std::string theISBN);  
  
    Book (std::string theTitle, const Publisher* thePubl,  
          const Author& theAuthor,  
          std::string theISBN);  
  
    std::string getTitle() const {return title;}  
    void setTitle(std::string theTitle) {title = theTitle;}  
  
    Publisher* getPublisher() const {return publisher;}  
    void setPublisher(const Publisher* publ) {publisher = publ;}  
  
    int getNumberOfAuthors() const {return numAuthors;}
```

```
Author getAuthor (int authorNumber) const;
void addAuthor (const Author&);
void removeAuthor (cosnt Author&);

std::string getISBN() const {return isbn;}
void setISBN(std::string id) {isbn = id;}

private:
    std::string title;
    Publisher* publisher;
    int numAuthors;
    static const int maxAuthors = 12;
    Author authors[maxAuthors];
    std::string isbn;
};

#endif
```

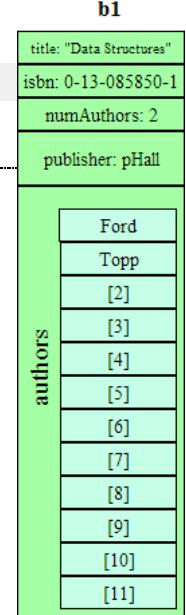
Consider the problem of copying a book, where this list of authors has been implemented as a basic (fixed-size) array.

## Book - simple arrays (cont.)

If we start with a single book object, b1, as shown here, and then we execute

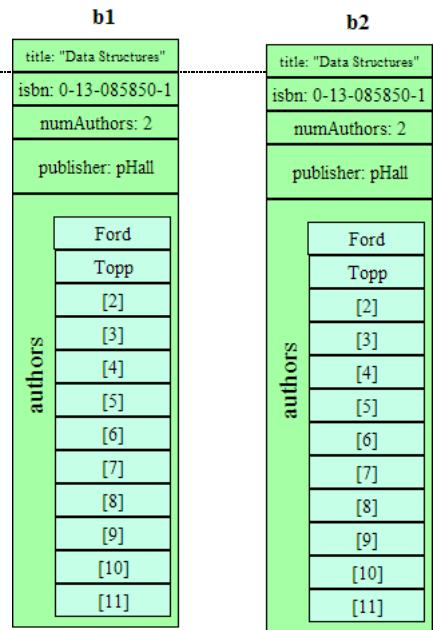
```
Book b2 = b1;
```

and assume that the copy of b1 is created by simply copying the block of bits that make up b1 to the location of the new variable b2,



The new result would be something like this, which looks just fine.

Of course, this fixed-length array design has a lot of drawbacks, so let's look at another possible implementation.



### 1.1.2 Book - dynamic arrays

Now let's consider the problem of copying a book implemented using dynamically allocated arrays.

```
class Book {  
public:  
    ...  
private:  
    std::string title;  
    Publisher* publisher;  
    int numAuthors;  
    static const int maxAuthors = 12;  
    Author* authors; // array of authors  
    std::string isbn;  
};
```

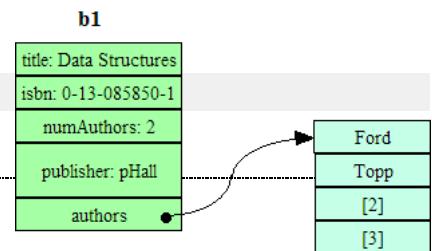
Here we have replaced the simple array with a [pointer](#). When initializing a `Book` object, we would allocate an appropriately-sized array of `Authors` on the heap, storing the address of that array in the pointer.

## Copying dynamic arrays

If we start with a single book object, `b1`, as shown here, and then we execute

```
Book b2 = b1;
```

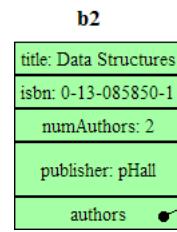
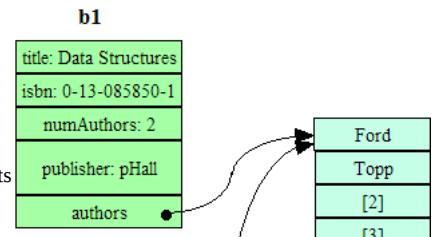
again carrying out the copy by copying the block of bits making up `b1`



The new result would be something like this.

- The *authors* pointer is copied bit-by-bit
  - In effect, copying the address of the array into the new book object.
- two book objects now share the same author array

The key factor here is that the *authors* data member, of data type *AuthorNode\** is copied along with all the other bits of the book. That has the effect of placing the same array address in both book objects. That's a really, really, bad idea!



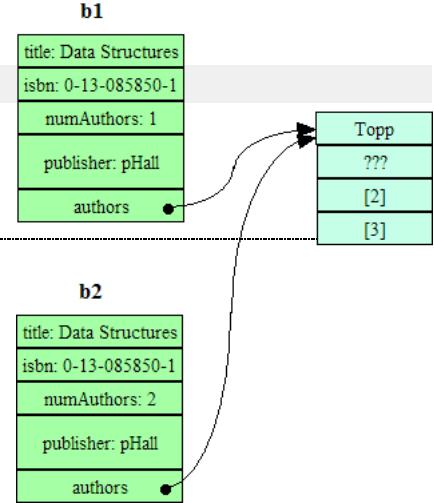
## Unplanned Sharing Leads to Corruption

To understand why this is so bad, suppose that we later did

```
b1.removeAuthor(b1.getAuthor(0));
```

to remove the first author from book b1.

Afterwards, we would have something like this. Notice that the change to b1 has, in effect, corrupted b2. The book object b2 still believes it has two authors (`numAuthors == 2`) but there is only one in the array.



## Unplanned Sharing Leads to Corruption

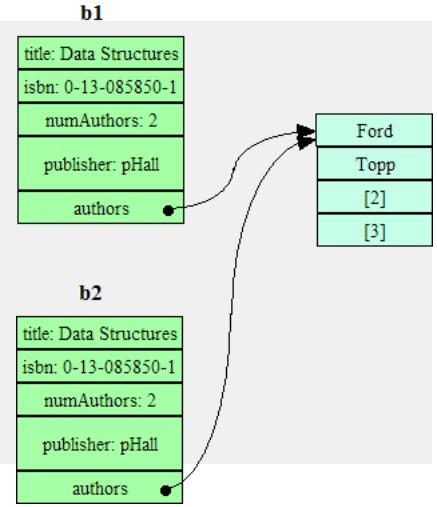
That's not even the worst possible scenario. The code below is a more elaborate and, arguably, safer approach to adding authors that does not crash when run out of room in the array for more authors.

```
void Book::addAuthor (int at, const Author* author)
{
    if (numAuthors >= MAXAUTHORS) ①
    {
        Author* newAuthors = new Author[2*MAXAUTHORS]; ②
        for (int i = 0; i < MAXAUTHORS; ++i) ③
            newAuthors[i] = authors[i];
        MAXAUTHORS *= 2;
        delete [] authors; ④
        authors = newAuthors;
    }
    int i = numAuthors;
    int atk = at - authors;
    while (i > atk)
    {
        authors[i] = authors[i-1];
        i--;
    }
    authors[atk] = author;
    ++numAuthors;
}
```

It works by

- Checking to see if there was room in the array. ①
- if not,
  1. allocating a larger array, ②
  2. copying the data ③ , and
  3. deleting the old array: ④

So if we start from this data state, and then add 3 authors to book 1 (“Doe”, “Smith”, and “Jones”), ...

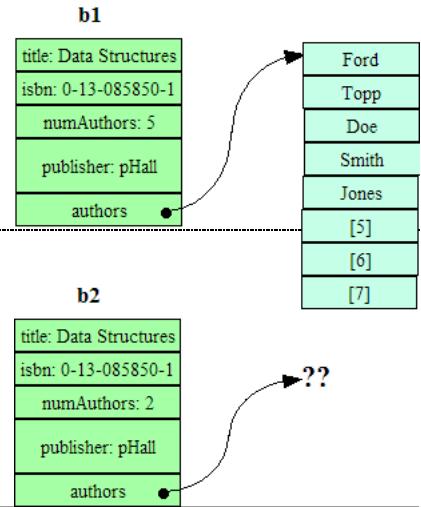


## Corrupted Data

... we would end up with the state shown here.

- b1 deleted the old array,
- but b2 still has the old array address in its data member.

So any attempt to access b2's authors is now essentially a throw of the dice — nobody can really predict what would happen.



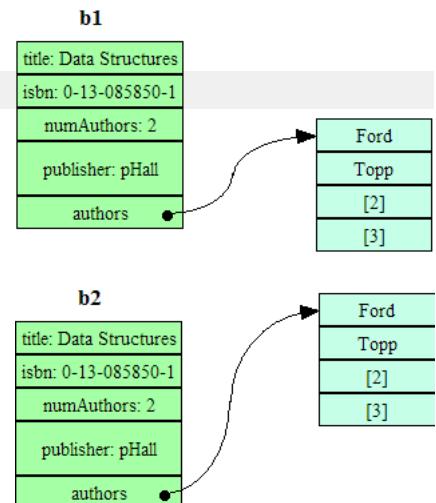
## Sometime it's Better to Have 2 Copies

What we really wanted, after the copy:

```
Book b2 = b1;
```

is something more like this:

But to get that, we will not be able to rely on copying books as simple blocks of bits.

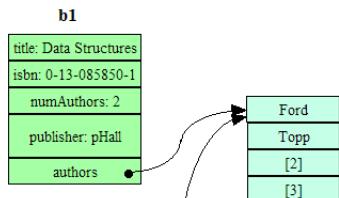


## 1.2 Shallow vs Deep Copy

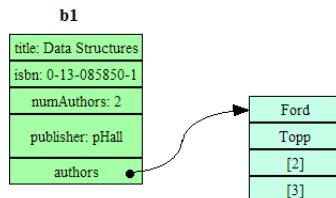
Copy operations are distinguished by how they treat pointers:

- In a shallow copy, all pointers are copied.
  - Leads to shared data on the heap.
- In a deep copy, objects pointed to are copied, then the new pointer set to the address of the copied object.
  - Copied objects keep exclusive access to the things they point to.

### This was a Shallow Copy



### This was a Deep Copy



### Shallow versus Deep

- In this example, deep is preferred.
- That's not always the case.
  - When we design an ADT we have to *think* about what is right for the abstraction we want to provide
- “Shallow” and “deep” are two extremes of a range of possible copies

For any ADT, we must decide whether the things it points to are things we want to *share* with other objects or whether we want to *own* them exclusively. That's a matter of just how we want our ADTs to behave, which depends in turn on what we expect to do with them.

In this context, it should be noted that “shallow” and “deep” are actually two extremes of a range of possible copy depths - sometimes our ADTs call for a behavior that has us treat some pointers shallowly and others deeply.

#### 1.2.1 Copying Books Two Ways

Here is a shallow copy:

```
Book shallowCopyOf (const Book& b)
: title(b.title), isbn(b.isbn), publisher(b.publisher),
  numAuthors(b.numAuthors), MAXAUTHORS(b.MAXAUTHORS)
{
    Book copy;
    copy.title = b.title;
    copy.isbn = b.isbn;
    copy.publisher = b.publisher;
    copy.numAuthors = b.numAuthors;
    copy.MAXAUTHORS = b.MAXAUTHORS;
    copy.authors = b.authors;
    return copy;
}
```

If we replace the highlighted line (the only one involving a pointer):

```
Book deepCopyOf (const Book& b)
: title(b.title), isbn(b.isbn), publisher(b.publisher),
  numAuthors(b.numAuthors), MAXAUTHORS(b.MAXAUTHORS)
{
    Book copy;
    copy.title = b.title;
    copy.isbn = b.isbn;
    copy.publisher = b.publisher;
    copy.numAuthors = b.numAuthors;
    copy.MAXAUTHORS = b.MAXAUTHORS;
    copy.authors = new Author[MAXAUTHORS];
    for (int i = 0; i < numAuthors; ++i)
        copy.authors[i] = b.authors[i];
    return copy;
}
```

then we get a deep copy.

In both copies, the non-pointer data members can be copied easily. But for the deep copy, the authors pointer is “copied” by allocating a new array big enough to hold the existing data; then all the existing data has to be copied into the new array.

(In truth, the second copy is not actually a fully deep copy. `publisher` is also a pointer, and we are still copying `publisher` shallowly. But that's a deliberate choice. `publisher` was made a pointer so that multiple books published by the same company could point back to their common publisher – we *want* to share the `publisher`.)

## 1.3 Choosing Between Shallow and Deep Copy

How do we decide which form of copy we want for any particular ADT?

First, if none of your data members are pointers (or references), then there's no problem. Shallow and deep copy are entirely equivalent in the absence of pointers.

If you *do* have pointers among your data members, then you have to ask whether your ADT should share the data it points to. There's no magic answer to this question. An ADT exists to support some mental model of a collection of data.

Sometimes sharing is a part of that mental model. For example, a publisher might want to keep all of its information about an author in one place, even though that author has written many books. In that case, it would make sense to use pointers to a shared `Author` object in our `Books`.

On the other hand, the dynamically allocated arrays we use to collect a group of co-authors of any particular book are properties specific to that book. So sharing of those arrays makes less sense and, as we have seen, can lead to easily corrupted data.

We can offer up this observation:

Shallow copy is wrong for any ADT that has pointers among its data members to things that it does not want to share.

# 2 The Big 3

The Big 3 in C++ are the copy constructor, assignment operator, and destructor. These three functions are closely related to one another in regards to their treatment of shallow and deep copying.

## 2.1 Copy Constructors

The copy constructor for a class `Foo` is the constructor of the form:

```
Foo (const Foo& oldCopy);
```

- Taken on its own, there is nothing special about the copy constructor itself.
  - It's just an ordinary constructor.
- It's special because of the number of common situations in which it gets *used*.
  - The compiler often generates implicit calls to this constructor in places where we might not expect it.

### Where are Copy Constructors Used?

The copy constructor gets used in 5 situations:

1. When you declare a new object as a copy of an old one:

```
Book book2 (book1);
```

or

```
Book book2 = book1;
```

2. When a function call passes a parameter “by copy” (i.e., the formal parameter does not have a &):

```
void foo (Book b, int k);
:
Book text361 (0201308787, budd,
    "Data Structures in C++ Using the Standard Template Library",
    1998, 1);
foo (text361, 0); // foo actually gets a copy of text361
```

3. When a function returns an object:

```
Book foo (int k);
{
    Book b;
    :
    return b; // a copy of b is placed in the caller's memory area
}
```

4. When data members are initialized in a constructor's initialization list from a single parameter of the same type:

```
Author::Author (std::string theName,
               Address theAddress, long id)
: name(theName),
  address(theAddress),
  identifier(id)
{}
```

5. When an object is a data member of another class for which the compiler has generated its own copy constructor.

## 2.1.1 Compiler-Generated Copy Constructors

As you can see from that list, the copy constructor gets used a *lot*. It would be very awkward to work with a class that did not provide a copy constructor.

So, again, the compiler tries to be helpful.

If we do not create a copy constructor for a class, the compiler generates one for us.

- This automatically generated version works by copying each data member via *their* individual copy constructors.
- For data members that are primitives, such as `int` or pointers, the copying is done by copying all the bits of that primitive object.
  - For things like `int` or `double`, that's just fine.
  - But, as you can guess, We'll see shortly, however, that this may or may not be what we want for pointers.

### Example 1: copying Address

```
addressDecl.h +
```

```
class Address {
public:
    Address (std::string theStreet, std::string theCity,
              std::string theState, std::string theZip);

    std::string getStreet() const;
    void putStreet (std::string theStreet);

    std::string getCity() const;
    void putCity (std::string theCity);

    std::string getState() const;
    void putState (std::string theState);

    std::string getZip() const;
    void putZip (std::string theZip);

private:
    std::string street;
    std::string city;
    std::string state;
    std::string zip;
};
```

In the case of our `Address` class, we have not provided a copy constructor, so the compiler would generate one for us.

The implicitly generated copy constructor would behave as if it had been written this way:

```
Address::Address (const Address& a)
: street(a.street), city(a.city),
  state(a.state), zip(a.zip)
{}
```

- This is, in fact, a perfectly good copy function for this class, so we might as well use the compiler-generated version.

If our data members do not have explicit copy constructors (and *their* data members do not have explicit copy constructors, and ...) then the compiler-provided copy constructor amounts to a shallow copy.

The compiler is all too happy to generate a copy constructor for us, but can we trust what it generates? To understand when we can and cannot trust it, we need to understand the different ways in which copying can occur.

### Example 2: copying Books

```
book1.h +
```

```

#ifndef BOOK_H
#include "author.h"
#include "publisher.h"

class Book {
public:
    Book();

    Book (std::string theTitle, const Publisher* thePubl,
          int numAuthors, Author* theAuthors,
          std::string theISBN);

    Book (std::string theTitle, const Publisher* thePubl,
          const Author& theAuthor,
          std::string theISBN);

    std::string getTitle() const {return title;}
    void setTitle(std::string theTitle) {title = theTitle;}

    Publisher* getPublisher() const {return publisher;}
    void setPublisher(const Publisher* publ) {publisher = publ;}

    int getNumberOfAuthors() const {return numAuthors;}

    Author getAuthor (int authorNumber) const;
    void addAuthor (const Author&);
    void removeAuthor (const Author&);

    std::string getISBN() const {return isbn;}
    void setISBN(std::string id) {isbn = id;}

private:
    std::string title;
    Publisher* publisher;
    int numAuthors;
    static const int maxAuthors = 12;
    Author authors[maxAuthors];
    std::string isbn;
};

#endif

```

If we provide no copy constructor for Book, the compiler generates one for us. It would be equivalent to

```

Book::Book (const Book& b)
: title(b.title), isbn(b.isbn), publisher(b.publisher),
  numAuthors(b.numAuthors), MAXAUTHORS(b.MAXAUTHORS),
  authors(b.authors)
{
}

```

We've already seen that this would be fine for a Book implemented using a simple array, but a disaster for a Book implemented using a dynamically allocated array.

So we conclude

**The compiler-generated copy constructor is wrong for classes that have pointers among their data members to data that they don't want to share.**

## 2.1.2 Implementing a deep Copy Constructor

So for our dynamic array version of Book, we need to implement our own copy constructor.

We start by adding the constructor declaration:

```

class Book {
public:
    Book();

    Book (std::string theTitle, const Publisher* thePubl,
          int numAuthors, Author* theAuthors,
          std::string theISBN);

```

```

Book (std::string theTitle, const Publisher* thePubl,
      const Author& theAuthor,
      std::string theISBN);

Book(const Book&);

std::string getTitle() const {return title;}
void setTitle(std::string theTitle) {title = theTitle;}
;

private:
    std::string title;
    Publisher* publisher;
    int numAuthors;
    static const int maxAuthors = 12;
    Author* authors; // array of authors
    std::string isbn;
};

```

Then we supply a function body for this constructor.

```

Book::Book (const Book& b)
: title(b.title), isbn(b.isbn), publisher(b.publisher),
  numAuthors(b.numAuthors), authors(new Author[maxAuthors])
{
    for (int i = 0; i < numAuthors; ++i)
        authors[i] = b.authors[i];
}

```

Most of the data members can be copied easily. But the `authors` pointer is copied by allocating a new array big enough to hold the existing data; then all the existing data has to be copied into the new array.

## 2.2 Assignment Operators

In most cases, when we think of copying, we think of assignment, not the copy constructor.

When we write `book1 = book2`, that's shorthand for `book1.operator=(book2)`.

The difference between the assignment operator and the copy constructor seems subtle to some people, but it typically comes down to whether the statement is a declaration or not. Look for the type name at the start of the statement:

- |   |   |
|---|---|
| <pre>MyClass x = y;</pre> <ul style="list-style-type: none"> <li>• invokes the copy constructor</li> <li>• creates a new variable, <code>x</code></li> <li>• initializes that new variable as a copy of <code>y</code></li> </ul> | <pre>x = y;</pre> <ul style="list-style-type: none"> <li>• invokes the assignment operator</li> <li>• changes the value of an existing variable, <code>x</code></li> <li>• replaces the value of that variable by a copy of <code>y</code></li> </ul> |
|---|---|

It's arguable which actually gets used more in typically C++ programming, the copy constructor or the assignment operator. Most of us probably write a lot more assignments, but the compiler generates a lot of copy constructor calls for us.

Assignment is so common in most people's programming that, once again, the compiler tries to be helpful:

If you don't provide your own assignment operator for a class, the compiler generates one automatically.

- The automatically generated assignment operator works by assigning each data member in turn.
- If none of the members have programmer-supplied assignment ops, then this is a *shallow copy*.

### Example 3: assigning Address

```

addressDecl.h +
```

```

class Address {
public:
    Address (std::string theStreet, std::string theCity,
              std::string theState, std::string theZip);

    std::string getStreet() const;
    void putStreet (std::string theStreet);

    std::string getCity() const;
    void putCity (std::string theCity);

    std::string getState() const;
}

```

```

    void putState (std::string theState);

    std::string getZip() const;
    void putZip (std::string theZip);

private:
    std::string street;
    std::string city;
    std::string state;
    std::string zip;
};

```

For example, we have not provided an assignment operator for Address class. Therefore the compiler will attempt to generate one, just as if we had written

```

class Address {
public:
    Address (std::string theStreet, std::string theCity,
              std::string theState, std::string theZip);

    Address& operator= (const Address&);

    ...

```

The automatically generated body for this assignment operator will be the equivalent of

```

Address& Address::operator= (const Address& a)
{
    street = a.street;
    city = a.city;
    state = a.state;
    zip = a.zip;
    return *this;
}

```

And that automatically generated assignment is just fine for Address.

#### Return values in Asst Ops

The return statement in the prior example returns the value just assigned, allowing programmers to chain assignments together:

```
addr3 = addr2 = addt1;
```

- This can simplify code where a computed value needs to be tested and then maybe used again if it passes the test, e.g.,

```

while ((x = foo(y)) > 0) {
    do_something_useful_with(x);
}

```

The compiler is all too happy to generate an assignment operator for us, but can we trust what it generates? To understand when we can and cannot trust it, we need to understand the different ways in which copying can occur.

#### Example 4: Assigning Books

[book1.h](#) +

```

#ifndef BOOK_H
#include "author.h"
#include "publisher.h"

class Book {
public:
    Book();

    Book (std::string theTitle, const Publisher* thePubl,
          int numberOfAuthors, Author* theAuthors,
          std::string theISBN);

    Book (std::string theTitle, const Publisher* thePubl,
          const Author& theAuthor,
          std::string theISBN);

    std::string getTitle() const {return title;}
    void setTitle(std::string theTitle) {title = theTitle;}

    Publisher* getPublisher() const {return publisher;}
}

```

```

    void setPublisher(const Publisher* publ) {publisher = publ; }

    int getNumberOfAuthors() const {return numAuthors;}

    Author getAuthor (int authorNumber) const;
    void addAuthor (const Author&);
    void removeAuthor (const Author&);

    std::string getISBN() const {return isbn;}
    void setISBN(std::string id) {isbn = id;}

private:
    std::string title;
    Publisher* publisher;
    int numAuthors;
    static const int maxAuthors = 12;
    Author authors[maxAuthors];
    std::string isbn;
};

#endif

```

If we provide no assignment operator for Book, the compiler generates one for us. It would be equivalent to

```

Book& Book::operator= (const Book& b)
{
    title = b.title;
    isbn = b.isbn;
    publisher = b.publisher;
    numAuthors = b.numAuthors;
    MAXAUTHORS = b.MAXAUTHORS;
    authors = b.authors;
    return *this;
}

```

Again, we've seen that this would be fine for a Book implemented using a simple array, but a disaster for a Book implemented using a dynamically allocated array.

So we conclude

**The compiler-generated assignment operator is wrong for classes that have pointers among their data members to data that they don't want to share.**

### 2.2.1 Implementing a deep assignment operator

So for our dynamic array version of Book, we need to implement our own assignment operator. We start by adding the operator declaration:

```

class Book {
public:
    Book();

    Book (std::string theTitle, const Publisher* thePubl,
          int numberofAuthors, Author* theAuthors,
          std::string theISBN);

    Book (std::string theTitle, const Publisher* thePubl,
          const Author& theAuthor,
          std::string theISBN);

    Book(const Book&);

    const Book& operator= (const Book&);

    std::string getTitle() const {return title;}
    void setTitle(std::string theTitle) {title = theTitle;}

    :

private:
    std::string title;
    Publisher* publisher;
    int numAuthors;
    static const int maxAuthors = 12;
    Author* authors; // array of authors
    std::string isbn;
};

```

Then we supply a function body for this operator.

```
const Book& Book::operator= (const Book& b)
{
    title = b.title;
    isbn = b.isbn;
    publisher = b.publisher;
    numAuthors = b.numAuthors;
    delete [] authors;           ①
    authors = new Author[MAXAUTHORS];
    for (int i = 0; i < numAuthors; ++i) ②
        authors[i] = b.authors[i];
    return *this;
}
```

Most of the data members can be copied easily. But the `authors` pointer is copied by allocating a new array big enough to hold the existing data; then all the existing data has to be copied into the new array (②).

Note also that one of the big differences between a copy constructor and an assignment operator is that copy constructors build new values, but assignment operators replace existing values. That means that one of the tasks of an assignment operator has to be to clean up the old value, which is what you see in step ①, above.

And that leads us to an interesting issue...

## 2.2.2 Self-Assignment

If we assign something to itself:

```
x = x;
```

we normally expect that nothing really happens.

But when we are writing our own assignment operators, that's not always the case. Sometimes assignment of an object to itself is a nasty special case that breaks things badly.

In the `Book` assignment operator we have just developed, what happens if we do `b1 = b1`?

In step ①, we deleted the existing `authors` array. In step ②, we copy the old `authors` into the new array. But if we are assigning a book to itself, there won't be any old `authors` left to copy, because we will have just deleted them.

So, instead of `b1 = b1`; leaving `b1` unchanged, it would actually destroy `b1`.

---

### Checking for Self-Assignment

```
const Book& Book::operator= (const Book& b)
{
    if (this != &b)
    {
        title = b.title;
        isbn = b.isbn;
        publisher = b.publisher;
        numAuthors = b.numAuthors;
        delete [] authors;
        authors = new Author[MAXAUTHORS];
        for (int i = 0; i < numAuthors; ++i)
            authors[i] = b.authors[i];
    }
    return *this;
}
```

[bookSelfAsst.cpp](#) +

```
const Book& Book::operator= (const Book& b)
{
    if (this != &b)
    {
        title = b.title;
        isbn = b.isbn;
        publisher = b.publisher;
        numAuthors = b.numAuthors;
        MAXAUTHORS = b.MAXAUTHORS;
        delete [] authors;
        authors = new Author[MAXAUTHORS];
        for (int i = 0; i < numAuthors; ++i)
            authors[i] = b.authors[i];
    }
    return *this;
}
```

This is safer.

- We check to see if the object we are assigning *to* (`this`) is at the same address as the one we are assigning *from*

- the & in the expression `&b` is the C++ [address-of](#) operator).
- If the two are the same, we leave them alone.
- Only if the two addresses are different do we carry on with the assignment.

You might think that self-assignment is so rare that we wouldn't need to worry about it. But, in practice, you might have lots of ways to reach the same object.

For example, we might have passed the same object as two different parameters of a function call `foo(b1, b1)`. If the function body of `foo` were to assign one parameter to another, we would then have a self-assignment that would likely not have been anticipated by the author of `foo` and that would be very hard to detect in the code that called `foo`.

As another example, algorithms for sorting arrays often contain statements like

```
array[i] = array[j];
```

with a very real possibility that, on occasion, `i` and `j` might be equal.

So self-assignment does occur in practice, and it's a good idea to check for this whenever you write your own assignment operators.

## 2.3 Destructors

We've already talked about the purpose of destructors. They are used to clean up objects that are no longer in use.

Once again, we find that we can't do without them:

If you don't provide a destructor for a class, the compiler generates one for you automatically.

- The automatically generated destructor simply invokes the destructors for any data member objects.
  - If none of the members have programmer-supplied destructors, the net effect is that the compiler-generated destructor does nothing.

### Example 5: Destroying addresses

`addrNoDestructor.h` [+]

```
class Address {
public:
    Address (std::string theStreet, std::string theCity,
              std::string theState, std::string theZip);

    std::string getStreet() const;
    void putStreet (std::string theStreet);

    std::string getCity() const;
    void putCity (std::string theCity);

    std::string getState() const;
    void putState (std::string theState);

    std::string getZip() const;
    void putZip (std::string theZip);

private:
    std::string street;
    std::string city;
    std::string state;
    std::string zip;
};

class Author
{
public:
    Author (std::string theName, Address theAddress, long id);

    std::string getName() const {return name;}
    void putName (std::string theName) {name = theName;}

    const Address& getAddress() const {return address;}
    void putAddress (const Address& addr) {address = addr;}

    long getIdentifier() const {return identifier;}

private:
    std::string name;
```

```

    Address address;
    const long identifier;
};


```

We have not declared or implemented a destructor for any of our classes. For Address and Author, that's OK.

- Note that the strings probably do contain pointers internally, but we trust the `std::string` to handle its own cleanup.

### Example 6: Destroying Books

Start with our simple array version.

[book1.h](#) +

```

#ifndef BOOK_H
#include "author.h"
#include "publisher.h"

class Book {
public:
    Book();
    Book (std::string theTitle, const Publisher* thePubl,
          int numAuthors, Author* theAuthors,
          std::string theISBN);

    Book (std::string theTitle, const Publisher* thePubl,
          const Author& theAuthor,
          std::string theISBN);

    std::string getTitle() const {return title;}
    void setTitle(std::string theTitle) {title = theTitle;}

    Publisher* getPublisher() const {return publisher;}
    void setPublisher(const Publisher* publ) {publisher = publ;}

    int getNumberOfAuthors() const {return numAuthors;}

    Author getAuthor (int authorNumber) const;
    void addAuthor (const Author&);
    void removeAuthor (const Author&);

    std::string getISBN() const {return isbn;}
    void setISBN(std::string id) {isbn = id;}

private:
    std::string title;
    Publisher* publisher;
    int numAuthors;
    static const int maxAuthors = 12;
    Author authors[maxAuthors];
    std::string isbn;
};

#endif

```

This version of the book has all of its data in a single block of memory. Assuming that each data member knows how to clean up its own internal storage, there's really nothing we would have to do when this book gets destroyed.

We can rely on the compiler-provided destructor.

Now, let's think about the dynamically allocated array.

[book2.h](#) +

```

#ifndef BOOK_H
#include "author.h"
#include "publisher.h"

class Book {
public:
    Book();
    Book (std::string theTitle, const Publisher* thePubl,
          int numAuthors, Author* theAuthors,
          std::string theISBN);

private:
    std::string title;
    Publisher* publisher;
    int numAuthors;
    static const int maxAuthors = 12;
    Author* authors;
    std::string isbn;
};

#endif

```

```

        std::string theISBN);

Book (std::string theTitle, const Publisher* thePubl,
      const Author& theAuthor,
      std::string theISBN);

Book(cponst Book&);
const Book& operator= (const Book&);

std::string getTitle() const {return title;}
void setTitle(std::string theTitle) {title = theTitle;}

Publisher* getPublisher() const {return publisher;}
void setPublisher(const Publisher* publ) {publisher = publ;}

int getNumberOfAuthors() const {return numAuthors;}

Author getAuthor (int authorNumber) const;
void addAuthor (const Author&);
void removeAuthor (cosnt Author&);

std::string getISBN() const {return isbn;}
void setISBN(std::string id) {isbn = id}

private:
    std::string title;
    Publisher* publisher;
    int numAuthors;
    static const int maxAuthors = 12;
    Author* authors; // array of authors
    std::string isbn;
};

#endif

```

In this version of the Book class, a portion of the data is kept on the heap.

- If this object were destroyed, we would need to be sure that the storage allocated for the array is recovered.
- That won't happen in the compiler-generated destructor, because the default action on pointers is to do nothing.

To implement our own destructor, we start by adding the **destructor declaration**:

```

class Book {
public:
    Book();

    Book (std::string theTitle, const Publisher* thePubl,
          int numberofAuthors, Author* theAuthors,
          std::string theISBN);

    Book (std::string theTitle, const Publisher* thePubl,
          const Author& theAuthor,
          std::string theISBN);

    Book(cponst Book&);
    const Book& operator= (const Book&);
    ~Book();
    :

```

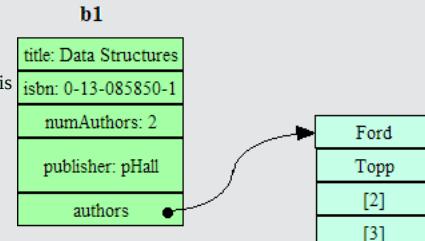
Then we supply a function body for this destructor.

```

Book::~Book()
{
    delete [] authors;
}

```

Not much needs to be done - just delete the pointer to the array of authors.



## 2.3.1 Trusting the Compiler-Generated Destructor

By now, you may have perceived a pattern.

#### Compiler-generated destructors are wrong for an ADT when...

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.

Under those circumstances, the compiler-generated destructor would result in a memory leak by failing to recover storage of an allocated object that is not accessible from anywhere else.

## 3 The Rule of the Big 3

The “[Big 3](#)” are the

- copy constructor
- assignment operator, and
- destructor

We've seen that, for each of these, the compiler will provide them if we don't but that the compiler-generated versions will be wrong for our ADT under identical circumstances.

This leads to the [Rule of the Big 3](#):

*If you provide your own version of any one of the Big 3, you should provide your own version of all 3.*

This is an important rule of thumb for C++ programmers. Like all “rules of thumb”, there are exceptions, but they are rare.

Watch for situations where you have pointers to data you don't share. When you see that, plan on implementing your own version of these three functions.

## 4 Moving Data – l-values and r-values

A common, and valid, criticism of C++ is that it forces so much copying to take place that programs are noticeably slowed down by all the copying.

The 2011 C++ standard responded to that criticism by introducing some new features for moving data rather than copying it. To understand this, though, we need to go back to some ideas from some of the very earliest programming languages.

### 4.1 L-values and R-values

Assignment is actually a rather interesting operation (even setting aside the fact that, in C++, you can override `operator=` to make assignment mean almost anything you want). Ask someone for an example of assignment, and they might respond with something like this:

```
x = y;
```

But that really over-simplifies things, because we know that we can put almost any kind of expression on the right:

```
x = y;
x = y + 1;
x = a[i].data + f(x);
```

So we can see a difference in how the left and right hand sides of the assignment are treated. The left side names a *location* where we want to store something. The right denotes a *value* to store there. That value can be given as another location from which to fetch a value, or as an expression to compute the value to be stored.

But a little more thought shows that this picture is still over-simplified. There are many expressions that we can use on the left hand side as well:

```
x = y;
a[i] = y + 1;
b.title = a[i].data + f(x);
c.foo() = 42;
```

So is it just expressions on either side? No, not quite. There are some expressions that make no sense at all on the left hand side of an assignment:

```
y + 1 = 23; // No!
sqrt(x) = 2.0; // No!
```

So how is it that some expressions can appear on the left of an assignment but others cannot? Well, the important distinction is that any expression that appears on the left of the assignment must somehow compute a location. On the right hand side, we can have expressions that yield locations or “pure” calculated values.

An [l-value](#) is an expression that denotes a location where data can be stored.

An *r-value* is an expression that denotes a value that can be stored in a location.

```
int x;
int a[100];
x = 1; // OK: x is a location
x+1 = 1; // No: x+1 is not a location
a[2] = 1; // OK: a[2] is a location
a[x+1] = 1; // OK: a[x+1] is a location
```

I've seen authors explain the names by indicating that the "l" stands for "location" and the "r" for, well, that one varies a bit, sometimes "reference", sometimes something else. But that's pure revisionism. It's clear from the earliest uses of the terms that the "l" and the "r" stand for "left" and "right", because they describe the idea that an assignment is legal if it has the form

*l-value* = *r-value*;

So how it is that, on the right, we can sometimes have a location and sometimes a "real" data value?

```
x = 2*y; // OK: 2*y is an int
x = y; // OK: y is a location (an int&)
```

In older programming languages, this was explained by claiming that the l-values are a special case of r-values. After all, the very existence of pointers shows that locations can be stored as data. But that explanation is a trifle bit weak, because it fails to explain why ordinary assignment copies values instead of addresses.

```
int x, y;
int* p;
x = y; // OK
x = p; // No good
p = y; // Also no good
```

C++ took a more formal approach to this problem by introducing *reference types* as fundamental types in the language. A reference holds a location, and assignment is defined as taking a reference type on the left and introducing some special rules:

- any mention of a variable of type T in an expression is a *reference* to that variable and has type T& (or const T& if the variable is declared as const).
- References of type T& or const T& can be freely converted to values of type T whenever the compiler finds it necessary to do so.
  - This conversion is carried out by the compiler generating the code to actually fetch the value from that referenced location.

Reference types turn out to be very useful. Among other things, they open up a useful option for functions. When we have a function with an "output parameter", that's because we actually pass in a reference type for that parameter.

For the purposes of this discussion, however, what is important is that reference types are l-values. Any operator or function whose return type is a reference type can be used to supply a location to which we can assign:

```
int a[100];
int& foo(int i) { return a[i]; }
int bar(int i) { return a[i]; }
...
foo(0) = 12; // OK: assigns to a[0]
bar(1) = 11; // compilation error - bar() does not return a reference
```

Many C++ books claim that C++ allows us to pass function parameters "by reference" as opposed to "by copy". But that's not really true. Many older programming languages did indeed have distinct parameters passing modes for input and output parameters. But that was because they lacked the idea of a reference type. If you pass a reference type value "by copy", you have given the function the *location* of your data rather than its value, which is logically equivalent to passing "by reference". C++ really passes every function parameter "by copy". It's just that we programmers sometimes decide to add a & to the parameter type, making it a reference type.

OK, so references are l-values. Each reference holds a location where data can be stored.

```
int a [100]
int k =1;
int& x = a[2*k+1]; // x holds the location of a[3]
x = 22; // stores 22 at a[3];
```

Now let's shift our attention back to the right. It's certainly possible in C++ to have references on the right as well:

```
int a [100]
int k =1;
int& x = a[2*k+1]; // x holds the location of a[3]
y = x; // copies a[3] into y
```

The thing to note here is that references denote actual memory locations where data is kept. But what about a statement like

```
x = y+1;
```

Where is the value y+1 stored prior to the actual assignment? In fact it might not be stored in memory at all. It might be a value that is computed in a CPU register and held there until we are ready to perform the assignment into x. If it does get stored in memory, it would be in some temporary storage location not directly

accessible to the programmer.

`y+1` is a “pure” r-value without a notion of “storage location”.

## 4.2 R-values and Returns

Suppose that we have function to produce a new edition of a Book:

```
Book newEdition (const Book& ofBook)
{
    Book b = ofBook;
    ++b.edition;
    return b;
}
```

We've already discussed the `return` statement triggers a call to the `Book` copy constructor, so that a copy of the book `b` is made as part of the return. That means that, if we use our new function like this:

```
Book oldBook;
:
Book newBook = newEdition(oldBook); ①
```

then line ① actually results in 2 calls to the `Book` copy constructor, one to enact the `return` statement in the `newEdition` function, and the second to copy that return value into `newBook`.

Let's assume, for the sake of example, that we are using the dynamic array version of `Book`. That's two new arrays allocated on the heap. Now, as soon as the returned value has been copied, it is no longer usable, so its destructor will be invoked. That destructor will delete one of those new arrays on the heap. So, we went to all the trouble of building it just to immediately throw it away.

The C++11 standard provides a new mechanism that enables us to avoid excess copying when working with temporary values like that.

- The “old” references, denoted by `&`, e.g., `Book&`, are now referred to as “[value references](#)”, in recognition of the fact that they describe locations where data could be stored.
- A new data type, [r-value references](#), denoted by a pair of ampersands, e.g., `Book&&`, is introduced that matches r-values of temporary expression results.

These will be used primarily in the parameter lists of two new functions:

- A [move constructor](#) is similar to a copy constructor, but used specifically to copy a temporary r-value.
- A [move assignment operator](#) is similar to an ordinary assignment operator, but used specifically to copy a temporary r-value computed on the right-hand-side into the location indicated on the left.

In some circumstances, these two “move” functions might run faster by taking advantage of the fact that, if the value being copied is known to be a temporary that's about to be destroyed, there's no penalty if we destroy the value in the course of copying it.

I know, that sounds strange. Let's look at some examples.

### Example 7: Book Copy and Move Constructors

We've already looked at a copy constructor for `Book`:

```
Book::Book (const Book& b)
: title(b.title), isbn(b.isbn),
  publisher(b.publisher), edition(b.edition)
  numAuthors(b.numAuthors), MAXAUTHORS(b.MAXAUTHORS)
{
    authors = new Author[numAuthors+1];
    for (int i = 0; i < numAuthors; ++i)
        authors[i] = b.authors[i];
}
```

In this copy constructor, we had to copy the non-pointer data members and then allocate a new array and copy the contents of the old array into the new one.

This leaves us with two perfectly valid books, the original book `b` and the newly constructed one. But suppose that we know that we will be doing a lot with functions like `newEdition` that construct and return books, which appear in the caller as temporary variables:

```
Book newBook = newEdition(oldBook);
```

We could avoid creating and copying the array by providing a move constructor:

```
Book::Book (Book&& b)
: title(b.title), isbn(b.isbn),
  publisher(b.publisher), edition(b.edition)
  numAuthors(b.numAuthors), MAXAUTHORS(b.MAXAUTHORS),
  authors(b.authors)
```

```
{
    b.authors = nullptr;
}
```

Instead of allocating a new array, we copy the address of the old array into the new book. Now, from our previous discussion of shallow copying, we know that this leads to sharing which is not what we want to do here.

*But*, this constructor will only be selected by the compiler if **b** is a temporary r-value, which means that it's going to go away as soon as this call is finished. So, sharing problem solved, right? Well, just one thing to worry about. The Book destructor deletes the authors array, so when **b** is destroyed, it will try to take its array with it. We stymie that by deliberately **breaking b's pointer to its array** by setting that to null. In effect, we are deliberately trashing **b** now that we have got what we want from it. Because **b** no longer has a pointer to that array, it won't be able to destroy it.

### Example 8: Book Assignment and Move Assignment

We can do something similar with assignment. Our old assignment operator for books was

```
bookSelfAsst.cpp +
```

```
const Book& Book::operator= (const Book& b)
{
    if (this != &b)
    {
        title = b.title;
        isbn = b.isbn;
        publisher = b.publisher;
        numAuthors = b.numAuthors;
        MAXAUTHORS = b.MAXAUTHORS;
        delete [] authors;
        authors = new Author[MAXAUTHORS];
        for (int i = 0; i < numAuthors; ++i)
            authors[i] = b.authors[i];
    }
    return *this;
}
```

We can do a special version for copying r-values:

```
Book& Book::operator= (Book&& b)
{
    if (this != &b)
    {
        title = b.title;
        isbn = b.isbn;
        publisher = b.publisher;
        numAuthors = b.numAuthors;
        delete [] authors;
        authors = b.authors;
        b.authors = nullptr;
    }
    return *this;
}
```

This saves the time that would be spent allocating and copying data from an array that would be discarded momentarily, anyway.

## 4.3 The Rule of the Big 5?

So, do we now have a “Rule of the Big 5” to replace the former “Rule of the Big 3”?

Well, yes and no.

The Rule of the Big 3 was important because violations of it almost always meant that our program would not function correctly.

The two new move operations are not nearly so crucial. Implementing them may result in a speedup of our code, but the code would still function correctly without them. So, if there is a Rule of the Big 5, it would have to be something like this:

If you provide your own version of a copy constructor, assignment operator, or destructor, you should provide your own version of all 3.

You should then at least consider providing your own version of the move constructor and move assignment operator.

Not quite as catchy, and it remains to be seen just how widely the move functions will be embraced by the C++ programming community.

## 5 Summary

The Big 3 are the destructor, copy constructor, and assignment operator.

If your class has pointers among its data members and the data is not something you want to share among different class instances, you need to implement your own versions of the Big 3.

**Rule of the Big 3:** If you implement your own version of any of the Big 3, you usually will need to implement your own versions of all 3.

- If you find that your program is leaking memory, you may need a destructor that cleans up memory allocated by your objects.
- If you find that data inserted into one variable mysteriously shows up in other variables of the same type, or that changing one variable mysteriously corrupts other, you may need to implement **deep copying** via a copy constructor and assignment operator.
- If you get messages that you are accessing data areas that have already been deleted, or deleting areas for the second time, then you may have a violation of the Rule of the Big 3.

Specifically, you may have implemented a destructor, but not provided a copy constructor and/or an assignment operator (or you have provided those but not successfully implemented deep copying using them).

# Review Sessions

Steven Zeil

## Contents:

- [1 When](#)
- [2 What](#)
- [3 Where](#)
- [4 Recordings](#)

## 1 When

**Wednesdays, 7:30PM (EDT)**

This will be a live session conducted as a network conference via Zoom. Sessions may range from 15 min. to 1 hr.

A recording will be made for anyone unable to attend. Links to the recordings will be appended to this page. There may be a delay of a day or two before recordings are available.

## 2 What

The purpose of these sessions will be to take student questions about the recent course material. In some cases, particularly if there are not many questions, I may opt to demo techniques or talk about concepts that have tripped up students in past semesters.

Most of these sessions will occur late in the time period allocated for a module in the course outline. If you are following the general practice of read during the first half of the module dates, work on the assignments during the second half, you should already have completed the bulk of the reading before coming to the review session.

## 3 Where

- [Enter the conference](#) for the review session, or go to <https://odu.zoom.us/> and join meeting number 937-7052-8143.

Etiquette for the conference:

- I will be using audio, so you should make sure you are someplace where you can listen without disturbing others (or use headphones).
- You will be able to join in by audio, if you have a microphone, or by typing in a chat interface.

If you do use a microphone, please **mute** it when you are not speaking.

- Otherwise the accumulated background noise and audio feedback from so many people will annoy everyone.

## 4 Recordings

Links to recordings of past review session will be appended here. There may be a delay of a day or two before recordings are made available.

- [May 20](#): walkthrough of preparing & submitting assignments, multiple submissions, Eclipse plugins

# Algorithms as Patterns for Code

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Same Pattern – Different Code](#)
- [2 Generalizing the Element Type](#)
- [3 Making Copies](#)

Templates describe common “patterns” for similar classes and functions that differ only in a few names.

Templates come in two varieties

- class templates
- function templates

Class templates are patterns for similar classes. Function templates are patterns for similar functions.

In this lesson, we’ll explore why the idea of a pattern for several different codes can be useful.

## 1 Same Pattern – Different Code

Let’s look at some common array manipulation functions that you have probably seen in earlier courses.

```
arrayops.h +  
  
***** arrayops.h *****/  
ifndef ARRAYOPS_H  
define ARRAYOPS_H  
  
**  
* Assume the elements of the array are already in order  
* Find the position where value could be added to keep  
*   everything in order, and insert it there.  
* Return the position where it was inserted  
* - Assumes that we have a separate integer (size) indicating how  
*   many elements are in the array  
* - and that the "true" size of the array is at least one larger  
*   than the current value of that counter  
*  
* @param array array into which to add an element  
* @param size number of data elements in the array. Must be less than  
*   the number of elements allocated for the array. Incremented  
*   upon output from this function.  
* @param value value to add into the array  
* @return the position where the element was added  
*/  
int addInOrder (int* array, int& size, int value);  
  
/*  
* Search an array for a given value, returning the index where  
* found or -1 if not found.  
*  
* From Malik, C++ Programming: From Problem Analysis to Program Design  
*  
* @param list the array to be searched  
* @param listLength the number of data elements in the array  
* @param searchItem the value to search for  
* @return the position at which value was found, or -1 if not found  
*/  
int seqSearch(const int list[], int listLength, int searchItem);  
  
/*  
* Search an ordered array for a given value, returning the index where  
* found or -1 if not found.  
* @param list the array to be searched. Must be ordered.  
* @param listLength the number of data elements in the array  
* @param searchItem the value to search for  
* @return the position at which value was found, or -1 if not found  
*/  
int seqOrderedSearch(const int list[], int listLength, int searchItem);  
  
/*  
* Removes an element from the indicated position in the array, moving  
* all elements in higher positions down one to fill in the gap.  
*  
* @param array array from which to remove an element  
*/
```

```

* @param size number of data elements in the array. Decremented
*             upon output from this function.
* @param index position from which to remove the element. Must be < size
*/
void removeElement (int* array, int& size, int index);

<**
* Performs the standard binary search using two comparisons per level.
* Returns index where item is found or -1 if not found
*
* From Weiss, Data Structures and Algorithm Analysis, 4e
* ( modified SJ Zeil)
*
* @param a array to search. Must be ordered.
* @param size number of elements in the array
* @param x value to search for
* @return position where found or -1 if not found
*/
int binarySearch( const int* a, int size, const int & x );

#endif

```

### arrayops.cpp +

```

***** arrayops.cpp *****/
#include "arrayops.h"

<**
*
* Assume the elements of the array are already in order
* Find the position where value could be added to keep
*   everything in order, and insert it there.
* Return the position where it was inserted
*   - Assumes that we have a separate integer (size) indicating how
*     many elements are in the array
*   - and that the "true" size of the array is at least one larger
*     than the current value of that counter
*
* @param array array into which to add an element
* @param size number of data elements in the array. Must be less than
*             the number of elements allocated for the array. Incremented
*             upon output from this function.
* @param value value to add into the array
* @return the position where the element was added
*/
int addInOrder (int* array, int& size, int value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;
    while (toBeMoved >= 0 && value < array[toBeMoved]) {
        array[toBeMoved+1] = array[toBeMoved];
        --toBeMoved;
    }
    // Insert the new value
    array[toBeMoved+1] = value;
    ++size;
    return toBeMoved+1;
}

/*
* Search an array for a given value, returning the index where
*   found or -1 if not found.
*
* From Malik, C++ Programming: From Problem Analysis to Program Design
*
* @param list the array to be searched
* @param listLength the number of data elements in the array
* @param searchItem the value to search for
* @return the position at which value was found, or -1 if not found
*/
int seqSearch(const int list[], int listLength, int searchItem)
{
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            return loc;

    return -1;
}

/*
* Search an ordered array for a given value, returning the index where
*   found or -1 if not found.

```

```

/*
 * @param list the array to be searched. Must be ordered.
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 * @return the position at which value was found, or -1 if not found
 */
int seqOrderedSearch(const int list[], int listLength, int searchItem)
{
    int loc = 0;

    while (loc < listLength && list[loc] < searchItem)
    {
        ++loc;
    }
    if (loc < listLength && list[loc] == searchItem)
        return loc;
    else
        return -1;
}

/*
 * Removes an element from the indicated position in the array, moving
 * all elements in higher positions down one to fill in the gap.
 *
 * @param array array from which to remove an element
 * @param size number of data elements in the array. Decremented
 *            upon output from this function.
 * @param index position from which to remove the element. Must be < size
 */
void removeElement (int* array, int& size, int index)
{
    int toBeMoved = index + 1;
    while (toBeMoved < size) {
        array[toBeMoved] = array[toBeMoved+1];
        ++toBeMoved;
    }
    --size;
}

const int NOT_FOUND = -1;

/**
 * Performs the standard binary search using two comparisons per level.
 * Returns index where item is found or -1 if not found
 *
 * From Weiss, Data Structures and Algorithm Analysis, 4e
 * (modified SJ Zeil)
 *
 * @param a array to search. Must be ordered.
 * @param size number of elements in the array
 * @param x value to search for
 * @return position where found or -1 if not found
 */
int binarySearch( const int* a, int size, const int & x )
{
    int low = 0, high = a.size() - 1;

    while( low <= high )
    {
        int mid = ( low + high ) / 2;

        if( a[ mid ] < x )
            low = mid + 1;
        else if( a[ mid ] > x )
            high = mid - 1;
        else
            return mid; // Found
    }
    return NOT_FOUND; // NOT_FOUND is defined as -1
}

```

Notice that our `binarySearch` routine operates on arrays of `int`. What would we have to do if we wanted to have a binary search over arrays of `double`, or of `string`?

We would simply need to make a new copy of the `binarySearch` declaration and body, replacing some of the occurrences of “`int`” by “`double`” or “`string`”.

In a large project, we might actually wind up with a lot of copies of `binarySearch`. (And in some projects, the original binary-search-of-ints might not even be one of the copies we would need!)

## 2 Generalizing the Element Type

One way we might generalize these functions to rewrite them to work on arrays of an imaginary type `T`.

[arrayopst.h](#) +

```

***** arrayops.h *****/
#ifndef ARRAYOPS_H
#define ARRAYOPS_H

/**
 *
 * Assume the elements of the array are already in order
 * Find the position where value could be added to keep
 *   everything in order, and insert it there.
 * Return the position where it was inserted
 *   - Assumes that we have a separate integer (size) indicating how
 *     many elements are in the array
 *   - and that the "true" size of the array is at least one larger
 *     than the current value of that counter
 *
 * @param array array into which to add an element
 * @param size number of data elements in the array. Must be less than
 *   the number of elements allocated for the array. Incremented
 *   upon output from this function.
 * @param value value to add into the array
 * @return the position where the element was added
 */
int addInOrder (T* array, int& size, T value);

/*
 * Search an array for a given value, returning the index where
 *   found or -1 if not found.
 *
 * From Malik, C++ Programming: From Problem Analysis to Program Design
 *
 * @param list the array to be searched
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 * @return the position at which value was found, or -1 if not found
 */
int seqSearch(const T list[], int listLength, T searchItem);

/*
 * Search an ordered array for a given value, returning the index where
 *   found or -1 if not found.
 * @param list the array to be searched. Must be ordered.
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 * @return the position at which value was found, or -1 if not found
 */
int seqOrderedSearch(const T list[], int listLength, T searchItem);

/*
 * Removes an element from the indicated position in the array, moving
 * all elements in higher positions down one to fill in the gap.
 *
 * @param array array from which to remove an element
 * @param size number of data elements in the array. Decremented
 *   upon output from this function.
 * @param index position from which to remove the element. Must be < size
 */
void removeElement (T* array, int& size, int index);

/**
 * Performs the standard binary search using two comparisons per level.
 * Returns index where item is found or -1 if not found
 *
 * From Weiss, Data Structures and Algorithm Analysis, 4e
 * (modified SJ Zeil)
 *
 * @param a array to search. Must be ordered.
 * @param size number of elements in the array
 * @param x value to search for
 * @return position where found or -1 if not found
 */
int binarySearch( const T* a, int size, const T & x );

#endif

```

[arrayopst.cpp](#) +

```

***** arrayops.cpp *****/
#include "arrayops.h"

/**
 *
 * Assume the elements of the array are already in order
 * Find the position where value could be added to keep
 *   everything in order, and insert it there.
 * Return the position where it was inserted
 *   - Assumes that we have a separate integer (size) indicating how
 *     many elements are in the array

```

```

/*
 * - and that the "true" size of the array is at least one larger
 *   than the current value of that counter
 *
 * @param array array into which to add an element
 * @param size number of data elements in the array. Must be less than
 *            the number of elements allocated for the array. Incremented
 *            upon output from this function.
 * @param value value to add into the array
 * @return the position where the element was added
 */
int addInOrder (T* array, int& size, T value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;
    while (toBeMoved >= 0 && value < array[toBeMoved]) {
        array[toBeMoved+1] = array[toBeMoved];
        --toBeMoved;
    }
    // Insert the new value
    array[toBeMoved+1] = value;
    ++size;
    return toBeMoved+1;
}

/*
 * Search an array for a given value, returning the index where
 * found or -1 if not found.
 *
 * From Malik, C++ Programming: From Problem Analysis to Program Design
 *
 * @param list the array to be searched
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 * @return the position at which value was found, or -1 if not found
 */
int seqSearch(const T list[], int listLength, T searchItem)
{
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            return loc;

    return -1;
}

/*
 * Search an ordered array for a given value, returning the index where
 * found or -1 if not found.
 * @param list the array to be searched. Must be ordered.
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 * @return the position at which value was found, or -1 if not found
 */
int seqOrderedSearch(const T list[], int listLength, T searchItem)
{
    int loc = 0;

    while (loc < listLength && list[loc] < searchItem)
    {
        ++loc;
    }
    if (loc < listLength && list[loc] == searchItem)
        return loc;
    else
        return -1;
}

/*
 * Removes an element from the indicated position in the array, moving
 * all elements in higher positions down one to fill in the gap.
 *
 * @param array array from which to remove an element
 * @param size number of data elements in the array. Decremented
 *            upon output from this function.
 * @param index position from which to remove the element. Must be < size
 */
void removeElement (T* array, int& size, int index)
{
    int toBeMoved = index + 1;
    while (toBeMoved < size) {
        array[toBeMoved] = array[toBeMoved+1];
        ++toBeMoved;
    }
    --size;
}

```

```

const int NOT_FOUND = -1;

/**
 * Performs the standard binary search using two comparisons per level.
 * Returns index where item is found or -1 if not found
 *
 * From Weiss, Data Structures and Algorithm Analysis, 4e
 * ( modified SJ Zeil)
 *
 * @param a array to search. Must be ordered.
 * @param size number of elements in the array
 * @param x value to search for
 * @return position where found or -1 if not found
 */
int binarySearch( const T* a, int size, const T & x )
{
    int low = 0, high = a.size( ) - 1;

    while( low <= high )
    {
        int mid = ( low + high ) / 2;

        if( a[ mid ] < x )
            low = mid + 1;
        else if( a[ mid ] > x )
            high = mid - 1;
        else
            return mid; // Found
    }
    return NOT_FOUND; // NOT_FOUND is defined as -1
}

```

Now, we don't actually have a predefined type by that name, so if were to try and use these functions this way:

```

#include "arrayops.h"
int importantNumbers[100];
int n;
:
addInOrder (importantNumbers, n, 42);

```

we would get a compilation error from somewhere inside `arrayops.h` complaining that `T` is undefined.

We *can* use any of these four functions, provided that we define `T` first:

```

typedef int T;
#include "arrayops.h" // T == int

int importantNumbers[100];
int n;
:
addInOrder (importantNumbers, n, 42);

```

and this works just great until we start writing more complicated programs and, one day, discover that we need to manipulate an array of `int` *and* an array of some other type:

```

typedef int T;
#include "arrayops.h" // T == int

typedef std::string T;
#include "arrayops.h" // T == std::string

int importantNumbers[100];
std::string favoriteNames[100];
int n1, n2;
:
addInOrder (importantNumbers, n1, 42);
addInOrder (favoriteNames, n2, "Arthur Dent");

```

Again, we will get compilation errors, this time at the second `typedef`, because we can't define `T` to mean two different things at the same time.

## 3 Making Copies

Instead of using `typedefs` to redefine `T`, we could simply make a distinct copy of `arrayops.h` and `arrayops.cpp` for each kind of array, using an ordinary text editor to replace `T` by a “real” type name:

[arrayopsdouble.h](#) +

```

#ifndef ARRAYOPSDOUBLE_H
#define ARRAYOPSDOUBLE_H

```

```

/**
 *
 * Assume the elements of the array are already in order
 * Find the position where value could be added to keep
 *   everything in order, and insert it there.
 * Return the position where it was inserted
 * - Assumes that we have a separate integer (size) indicating how
 *   many elements are in the array
 * - and that the "true" size of the array is at least one larger
 *   than the current value of that counter
 *
 * @param array array into which to add an element
 * @param size number of data elements in the array. Must be less than
 *   the number of elements allocated for the array. Incremented
 *   upon output from this function.
 * @param value value to add into the array
 * @return the position where the element was added
 */
int addInOrder (double* array, int& size, double value);

/*
 * Search an array for a given value, returning the index where
 *   found or -1 if not found.
 *
 * From Malik, C++ Programming: From Problem Analysis to Program Design
 *
 * @param list the array to be searched
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 * @return the position at which value was found, or -1 if not found
 */
int seqSearch(const double list[], int listLength, double searchItem);

/*
 * Search an ordered array for a given value, returning the index where
 *   found or -1 if not found.
 * @param list the array to be searched. Must be ordered.
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 * @return the position at which value was found, or -1 if not found
 */
int seqOrderedSearch(const double list[], int listLength, double searchItem);

/*
 * Removes an element from the indicated position in the array, moving
 * all elements in higher positions down one to fill in the gap.
 *
 * @param array array from which to remove an element
 * @param size number of data elements in the array. Decremented
 *   upon output from this function.
 * @param index position from which to remove the element. Must be < size
 */
void removeElement (double* array, int& size, int index);

/**
 * Performs the standard binary search using two comparisons per level.
 * Returns index where item is found or -1 if not found
 *
 * From Weiss, Data Structures and Algorithm Analysis, 4e
 * (modified SJ Zeil)
 *
 * @param a array to search. Must be ordered.
 * @param size number of elements in the array
 * @param x value to search for
 * @return position where found or -1 if not found
 */
int binarySearch( const double* a, int size, const double & x );

```

#endif

[arrayopsstring.h](#) [+]

```
#ifndef ARRAYOPSSTRING_H
#define ARRAYOPSTRING_H
```

```
#include <string>

/**
 *
 * Assume the elements of the array are already in order
 * Find the position where value could be added to keep
 *   everything in order, and insert it there.
 * Return the position where it was inserted
 * - Assumes that we have a separate integer (size) indicating how
 *   many elements are in the array
```

```

/*
 * - and that the "true" size of the array is at least one larger
 *   than the current value of that counter
 *
 * @param array array into which to add an element
 * @param size number of data elements in the array. Must be less than
 *             the number of elements allocated for the array. Incremented
 *             upon output from this function.
 * @param value value to add into the array
 * @return the position where the element was added
 */
int addInOrder (std::string* array, int& size, std::string value);

/*
 * Search an array for a given value, returning the index where
 * found or -1 if not found.
 *
 * From Malik, C++ Programming: From Problem Analysis to Program Design
 *
 * @param list the array to be searched
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 * @return the position at which value was found, or -1 if not found
 */
int seqSearch(const std::string list[], int listLength, std::string searchItem);

/*
 * Search an ordered array for a given value, returning the index where
 * found or -1 if not found.
 * @param list the array to be searched. Must be ordered.
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 * @return the position at which value was found, or -1 if not found
 */
int seqOrderedSearch(const std::string list[], int listLength, std::string searchItem);

/*
 * Removes an element from the indicated position in the array, moving
 * all elements in higher positions down one to fill in the gap.
 *
 * @param array array from which to remove an element
 * @param size number of data elements in the array. Decremented
 *             upon output from this function.
 * @param index position from which to remove the element. Must be < size
 */
void removeElement (std::string* array, int& size, int index);

/**
 * Performs the standard binary search using two comparisons per level.
 * Returns index where item is found or -1 if not found
 *
 * From Weiss, Data Structures and Algorithm Analysis, 4e
 * (modified SJ Zeil)
 *
 * @param a array to search. Must be ordered.
 * @param size number of elements in the array
 * @param x value to search for
 * @return position where found or -1 if not found
 */
int binarySearch( const std::string* a, int size, const std::string & x );

#endif

```

(Similar changes would be made to the .cpp files.)

With just a few additional edits to the #include's, #ifdef's, and so on, we get a perfectly workable pair of .h and .cpp files. (Don't you just love those global find-and-replace commands in editors?)

Of course, in some situations, we may wish that we had used a slightly less common dummy type name than just "T", so that we wouldn't need to worry about changing lines like:

```

string Temp = "This is a Temporary string.";

to

string doubleemp = "doublehis is a doubleemporary string.";

```

But after the first time we get burned on that, we'll never need reminding again.

A similar copy-and-edit could give us `arrayopsdouble.cpp` and `arrayopsstring.cpp`. Then we can write our code this way:

```
#include "arrayopsdouble.h"
#include "arrayopsstring.h"
```

```
double importantNumbers[100];
std::string favoriteNames[100];
int n1, n2;
:
addInOrder (importantNumbers, n1, 42.0);
addInOrder (favoriteNames, n2, "Arthur Dent");
```

Not too bad. But doing this in a large program does get to be a bit of a project management headache. Suppose we found a bug in our implementation of `addInOrder`. We would not only have to fix the problem in `arrayops.cpp`, but also in `arrayopsdouble.cpp` and `arrayopsstring.cpp`. In fact, we would have to remember *every* copy that has been made of the original files, and either insert our bug fix into those copies or else regenerate the copies from the fixed `arrayops.cpp`.

This isn't impossible. Programmers working in C, Pascal, and other programming languages have been doing just this for decades. But since it *is* a common situation, and it *does* involve a fair amount of work, some programming language designers eventually found a way to do the same thing automatically.

In C++, this is done via *templates*, the subjects of our next readings.

# Function Templates

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Function Templates](#)
- [2 First Templates: swap, min, & max](#)
- [3 Writing Function Templates](#)
- [4 Function Templates and the C++ std Library](#)
  - [4.1 fill\\_n](#)
  - [4.2 rellops](#)

We've suggested that programs may have many instances of code that are really minor variations on the same "pattern".

These code instances can be created manually by programmers by editing an existing version of the code, making changes for each desired variation on the pattern. But this process can be tedious and error-prone.

## 1 Function Templates

That's where templates come in. A C++ *template* is a kind of pattern for code. Within that pattern are certain names that, like  $\tau$  in the earlier example, we intend to replace by something "real" when we use the pattern. These names are called *template parameters*.

The compiler *instantiates* (creates an instance of) a template by filling in the appropriate replacement for these template parameter names, thereby generating the actual code for a function or class, and compiling that instantiated code.

## 2 First Templates: swap, min, & max

Let's start with a very simple pattern for code that you have probably written many times: exchanging the values of two variables.

Have you ever written code that looked like:

```
// Exchange the values of x and y
int temp = x;
x = y;
y = temp;
```

or

```
// Exchange the values of s1 and s2
string temp = s1;
s1 = s2;
s2 = temp;
```

or maybe something similar for exchanging floating point numbers, or characters, or ...? This is a perfect example of a pattern for code that pretty much remains the same regardless of what the actual data types being used might be. (More specifically, the source code that we write would look the same. The compiler may need to generate very different code for copying, say, strings, than it would generate for copying integers.)

It's a good candidate for a template. First, we think about how we might capture this code in a function:

```
void swap (int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

This function would work for swapping the values of two `ints`, but not two `strings`. But we can imagine writing a whole slew of functions of the form:

```
void swap (T& x, T& y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

thinking of  $\tau$  as a placeholder for the "real" type. This isn't legal C++, though, because,  $\tau$  doesn't stand for anything. But we can turn this pattern for a function into a *template* by adding a header that tells the compiler that we want this to be a template and that  $\tau$  is a placeholder:

```
template <typename T>
void swap (T& x, T& y)
{
```

```

T temp = x;
x = y;
y = temp;
}

```

Voila! We have a template. We can now write

```

int i = ...
int j = ...
double x = ...
double y = ...
string s = ...
string t = ...
:
swap(i, j);
swap(x, y);
swap(s, t);

```

Each time we use `swap`, the compiler looks at the actual data types of the parameters being passed, and if we have never used that particular combination of data types before, generates a new function using our swap pattern. The technical terminology here is that the compiler *instantiates* the swap template to match the actual parameter types. So the first call to `swap` will produce a function named “`swap`” that takes two `ints` as parameters, the second produces a function named “`swap`” that takes two `doubles` as parameters, and the third produces a function named “`swap`” that takes two `strings` as parameters.

Let's look at another couple of common programming patterns:

```

// Set z to the smaller of x and y
z = x;
if (y < x)
    z = y;

```

and

```

// Set z to the larger of x and y
z = x;
if (x < y)
    z = y;

```

Again, we can easily think of functions for this purpose...

```

T min (const T& x, const T& y)
{
    T z = x;
    if (y < x)
        z = y;
    return z;
}

T max (const T& x, const T& y)
{
    T z = x;
    if (x < y)
        z = y;
    return z;
}

```

or, even shorter

```

T min (const T& x, const T& y)
{
    return (x < y) ? x : y;
}

T max (const T& x, const T& y)
{
    return (x < y) ? y : x;
}

```

Now, in this case, we never said what the data types of `x` and `y` were in the original pattern, so using a placeholder `T` seems natural. And all we need to do is to add the headers to get true C++ templates:

```

typename <class T>
T min (const T& x, const T& y)
{
    return (x < y) ? x : y;
}

typename <class T>
T max (const T& x, const T& y)
{
    return (x < y) ? y : x;
}

```

These should work for any data types that support the `<` operator. So

```
string s = min("abc", "def"); // returns "abc"
double d = max(-1.0, 1.0); // returns 1.0
```

are fine but

```
struct Point {
    double x;
    double y
};
Point p1 = ...
Point p2 = ...
Point p = min(p1, p2); // error
```

will produce a compilation error because we have no ‘<’ operator on the data type `Point`.

- We don’t need to use “T” as our placeholder. For example, I like to use

```
typename <class Comparable>
Comparable min (const Comparable& x, const Comparable& y)
{
    return (x < y) ? x : y;
}
```

for placeholders that will need a comparison operator (<) to work.

- More complex templates may need multiple placeholders. We can have as many placeholders as we need by separating them with commas inside the “<>”, e.g.,

```
template <typename T, typename U>
```

The key restriction is that each placeholder must appear somewhere within the function’s formal parameter list.

The three templates we have shown here, `swap`, `min`, and `max`, are actually all part of the C++ standard library, in the `<algorithm>` header.

## 3 Writing Function Templates

For examples of writing some larger templates, let’s look at the functions for manipulating arrays that we used in the previous reading:

```
arrayops.h +
```

```
***** arrayops.h *****/
#ifndef ARRAYOPS_H
#define ARRAYOPS_H

/**
 *
 * Assume the elements of the array are already in order
 * Find the position where value could be added to keep
 *   everything in order, and insert it there.
 * Return the position where it was inserted
 * - Assumes that we have a separate integer (size) indicating how
 *   many elements are in the array
 * - and that the "true" size of the array is at least one larger
 *   than the current value of that counter
 *
 * @param array array into which to add an element
 * @param size number of data elements in the array. Must be less than
 *   the number of elements allocated for the array. Incremented
 *   upon output from this function.
 * @param value value to add into the array
 * @return the position where the element was added
 */
int addInOrder (int* array, int& size, int value);

/*
 * Search an array for a given value, returning the index where
 *   found or -1 if not found.
 *
 * From Malik, C++ Programming: From Problem Analysis to Program Design
 *
 * @param list the array to be searched
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 * @return the position at which value was found, or -1 if not found
 */
int seqSearch(const int list[], int listLength, int searchItem);

/*
 * Search an ordered array for a given value, returning the index where
 *   found or -1 if not found.
 * @param list the array to be searched. Must be ordered.
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 */
```

```

    * @return the position at which value was found, or -1 if not found
    */
int seqOrderedSearch(const int list[], int listLength, int searchItem);

/*
 * Removes an element from the indicated position in the array, moving
 * all elements in higher positions down one to fill in the gap.
 *
 * @param array array from which to remove an element
 * @param size number of data elements in the array. Decremented
 *             upon output from this function.
 * @param index position from which to remove the element. Must be < size
 */
void removeElement (int* array, int& size, int index);

<**
 * Performs the standard binary search using two comparisons per level.
 * Returns index where item is found or -1 if not found
 *
 * From Weiss, Data Structures and Algorithm Analysis, 4e
 * (modified SJ Zeil)
 *
 * @param a array to search. Must be ordered.
 * @param size number of elements in the array
 * @param x value to search for
 * @return position where found or -1 if not found
 */
int binarySearch( const int* a, int size, const int & x );

#endif

```

### arrayops.cpp +

```

***** arrayops.cpp *****/
#include "arrayops.h"

<**
 *
 * Assume the elements of the array are already in order
 * Find the position where value could be added to keep
 *   everything in order, and insert it there.
 * Return the position where it was inserted
 *   - Assumes that we have a separate integer (size) indicating how
 *     many elements are in the array
 *   - and that the "true" size of the array is at least one larger
 *     than the current value of that counter
 *
 * @param array array into which to add an element
 * @param size number of data elements in the array. Must be less than
 *             the number of elements allocated for the array. Incremented
 *             upon output from this function.
 * @param value value to add into the array
 * @return the position where the element was added
 */
int addInOrder (int* array, int& size, int value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;
    while (toBeMoved >= 0 && value < array[toBeMoved]) {
        array[toBeMoved+1] = array[toBeMoved];
        --toBeMoved;
    }
    // Insert the new value
    array[toBeMoved+1] = value;
    ++size;
    return toBeMoved+1;
}

/*
 * Search an array for a given value, returning the index where
 *   found or -1 if not found.
 *
 * From Malik, C++ Programming: From Problem Analysis to Program Design
 *
 * @param list the array to be searched
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 * @return the position at which value was found, or -1 if not found
 */
int seqSearch(const int list[], int listLength, int searchItem)
{
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            return loc;
}
```

```

        return -1;
    }

/*
 * Search an ordered array for a given value, returning the index where
 * found or -1 if not found.
 * @param list the array to be searched. Must be ordered.
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 * @return the position at which value was found, or -1 if not found
 */
int seqOrderedSearch(const int list[], int listLength, int searchItem)
{
    int loc = 0;

    while (loc < listLength && list[loc] < searchItem)
    {
        ++loc;
    }
    if (loc < listLength && list[loc] == searchItem)
        return loc;
    else
        return -1;
}

/*
 * Removes an element from the indicated position in the array, moving
 * all elements in higher positions down one to fill in the gap.
 *
 * @param array array from which to remove an element
 * @param size number of data elements in the array. Decremented
 *             upon output from this function.
 * @param index position from which to remove the element. Must be < size
 */
void removeElement (int* array, int& size, int index)
{
    int toBeMoved = index + 1;
    while (toBeMoved < size) {
        array[toBeMoved] = array[toBeMoved+1];
        ++toBeMoved;
    }
    --size;
}

const int NOT_FOUND = -1;

/**
 * Performs the standard binary search using two comparisons per level.
 * Returns index where item is found or -1 if not found
 *
 * From Weiss, Data Structures and Algorithm Analysis, 4e
 * (modified SJ Zeil)
 *
 * @param a array to search. Must be ordered.
 * @param size number of elements in the array
 * @param x value to search for
 * @return position where found or -1 if not found
 */
int binarySearch( const int* a, int size, const int & x )
{
    int low = 0, high = a.size() - 1;

    while( low <= high )
    {
        int mid = ( low + high ) / 2;

        if( a[ mid ] < x )
            low = mid + 1;
        else if( a[ mid ] > x )
            high = mid - 1;
        else
            return mid; // Found
    }
    return NOT_FOUND; // NOT_FOUND is defined as -1
}

```

These are all written to manipulate arrays of int, so we need to generalize these to manipulate arrays of any type T (or some other placeholder name).

- Be careful. Some of the “int”s in that code represent the data being stored in the array, but some really are integers (e.g., the size of the array).

Before we move forward with that step, we need to consider one other interesting aspect of templates. Think of how we are going to use, for example, the template form of

```
int seqSearch(const int list[], int listLength, int searchItem);
```

We might think that it will look something like

```
template <typename Comparable>
int seqSearch(const Comparable list[], int listLength, Comparable searchItem);
```

so that we could write code like

```
#include "arrayops.h"

...
string* words = new string[100];
...
int position = seqSearch(words, numWords, "foo");
```

When the compiler sees this call matching the seqSearch pattern, it should generate the actual code for an array-of-strings version of seqSearch.

- The pattern of a template is instantiated when we *compile* uses of that template.
- We are used to dividing our functions into *declarations*, in a header file, that are used when we *compile* code that calls those functions, and *definitions*, in a compilation unit (.cpp file) that are brought in when we *link* all of our compiled code to form an executable.
  - For normal code, the compiler doesn't need to see the bodies of the functions in order to generate the code required to call them.
- For calls to templated functions, however, the compiler needs to see the templated function body when *compiling* the calls, so that it can instantiate the code for that particular version of the template.

As a consequence,

we put the whole body of the template into a header file

so that it gets seen (via #include) during the compilation phase.

So, to templateize seqSearch, we

1. Copy the whole body into the header file.
2. Figure out how many template parameters (placeholders) we need and pick names for them.
3. Add a template header to declare those template parameter names.
4. Use those template parameter names in place of the original data type that we are trying to generalize away.

So, seqSearch becomes

```
template <typename Comparable>
int seqSearch(const Comparable list[], int listLength, Comparable searchItem)
{
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            return loc;

    return -1;
}
```

And the rest of our functions can be generalized similarly:

```
arrayops templated.h +
```

```
#ifndef ARRAYOPS_H
#define ARRAYOPS_H

/**
 *
 * Assume the elements of the array are already in order
 * Find the position where value could be added to keep
 *   everything in order, and insert it there.
 * Return the position where it was inserted
 * - Assumes that we have a separate integer (size) indicating how
 *   many elements are in the array
 * - and that the "true" size of the array is at least one larger
 *   than the current value of that counter
 *
 * @param array array into which to add an element
 * @param size number of data elements in the array. Must be less than
 *   the number of elements allocated for the array. Incremented
 *   upon output from this function.
 * @param value value to add into the array
 * @return the position where the element was added
 */
template <typename Comparable>
int addInOrder (Comparable* array, int& size, Comparable value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;
```

```

        while (toBeMoved >= 0 && value < array[toBeMoved]) {
            array[toBeMoved+1] = array[toBeMoved];
            --toBeMoved;
        }
        // Insert the new value
        array[toBeMoved+1] = value;
        ++size;
        return toBeMoved+1;
    }

/*
 * Search an array for a given value, returning the index where
 * found or -1 if not found.
 *
 * From Malik, C++ Programming: From Problem Analysis to Program Design
 *
 * @param list the array to be searched
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 * @return the position at which value was found, or -1 if not found
 */
template <typename T>
int seqSearch(const T list[], int listLength, T searchItem)
{
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            return loc;

    return -1;
}

/*
 * Search an ordered array for a given value, returning the index where
 * found or -1 if not found.
 * @param list the array to be searched. Must be ordered.
 * @param listLength the number of data elements in the array
 * @param searchItem the value to search for
 * @return the position at which value was found, or -1 if not found
 */
template <typename Comparable>
int seqorderedSearch(const Comparable list[], int listLength,
                     Comparable searchItem)
{
    int loc = 0;

    while (loc < listLength && list[loc] < searchItem)
    {
        ++loc;
    }
    if (loc < listLength && list[loc] == searchItem)
        return loc;
    else
        return -1;
}

/*
 * Removes an element from the indicated position in the array, moving
 * all elements in higher positions down one to fill in the gap.
 *
 * @param array array from which to remove an element
 * @param size number of data elements in the array. Decremented
 *           upon output from this function.
 * @param index position from which to remove the element. Must be < size
 */
template <typename T>
void removeElement (T* array, int& size, int index)
{
    int toBeMoved = index + 1;
    while (toBeMoved < size) {
        array[toBeMoved] = array[toBeMoved+1];
        ++toBeMoved;
    }
    --size;
}

/**
 * Performs the standard binary search using two comparisons per level.
 * Returns index where item is found or -1 if not found
 *
 * From Weiss, Data Structures and Algorithm Analysis, 4e
 * (modified SJ Zeil)
 *
 * @param a array to search. Must be ordered.
 * @param size number of elements in the array
 */

```

```

* @param x value to search for
* @return position where found or -1 if not found
*/
template <typename Comparable>
int binarySearch( const Comparable* a, int size, const Comparable & x )
{
    const int NOT_FOUND = -1;

    int low = 0, high = a.size( ) - 1;

    while( low <= high )
    {
        int mid = ( low + high ) / 2;

        if( a[ mid ] < x )
            low = mid + 1;
        else if( a[ mid ] > x )
            high = mid - 1;
        else
            return mid; // Found
    }
    return NOT_FOUND; // NOT_FOUND is defined as -1
}

#endif

```

In early versions of C++ templates, a template parameter that stood for a data type was introduced by the keyword “class” instead of “typename”. That’s still accepted (and fairly common).<sup>1</sup> However, some people felt that it was a bad choice because sometimes the parameter ( $T$ ) will wind up standing for a primitive non-class type (e.g., `int`). So later versions of C++ recommended the use of the newer keyword “typename” instead. Both are acceptable. Both mean exactly the same thing.

## 4 Function Templates and the C++ std Library

The C++ std library has a number of useful templates for small, common programming idioms, many in the header `<algorithm>`. We’ve already seen `swap`, `min`, and `max`.

Some others of note are

### 4.1 fill\_n

To fill an array of size  $n$  with a constant value:

```

template <typename T>
void fill_n (T* array, int n, T value)
{
    for (int i = 0; i < n; ++i)
        array[i] = value;
}

```

Not exactly earth-shaking, but often convenient.

### 4.2 relops

```

namespace relops {

template <typename T>
inline bool operator!= (const T& a, const T& b)
{
    return !(a == b);
}

template <typename T>
inline bool operator> (const T& a, const T& b)
{
    return b < a;
}

template <typename T>
inline bool operator<= (const T& a, const T& b)
{
    return !(b < a);
}

template <typename T>
inline bool operator>= (const T& a, const T& b)
{
    return !(a < b);
}
}

```

Because of these templates, if we want a new class to have a full set of relational operators, we only need to provide < and ==. The std::rellops templates take care of the other 4:

```
class PersonnelRecord {
public:
    PersonnelRecord (const std::string& name, ...);
    ...
    bool operator<
        (const PersonnelRecord&) const;
    bool operator==
        (const PersonnelRecord&) const;
};
...
using namespace std::rellops; // Without this, the rellops templates
                            // aren't visible to our code.
...
if (myRecord >= yourRecord) // OK, defined in terms of <
```

#### 4.2.1 Coming up...

We'll see some other std templates later, after we introduce the iterator ADT.

---

[1](#): You may see it a lot in my code, because I've been programming in C++ for a long time, and old habits are hard to break.

# Class Templates

Steven J. Zeil

Last modified: Feb 12, 2020

## Contents:

- [1 Class Templates](#)
- [2 Developing and Using Class Templates](#)
  - [2.1 Templates as a Pattern for Data](#)
  - [2.2 Instantiating Class Templates](#)
  - [2.3 Templates with Function Members](#)
  - [2.4 Tips for Implementing Templates](#)
- [3 Example – Matrix](#)
  - [3.1 Start with a working class...](#)
  - [3.2 Combine into .h File](#)
  - [3.3 Add Template Parameters](#)
  - [3.4 Rewrite Class Names](#)
  - [3.5 Check Related Classes](#)
- [4 Second Example – the pair class](#)
  - [4.1 Thinking in Terms of Tuples](#)
  - [4.2 The pair Template](#)
  - [4.3 Applications of pair<...>](#)
- [5 Third Example: array](#)

Just as function templates allow us to write general patterns for functions, from which the compiler then generates multiple instances, so [class templates](#) allow us to write general patterns for classes.

Class templates are especially useful for “containers”, data structures that serve mainly as collections of other, smaller data types. In our spell checker, for example, we have used two different kinds of sets and two different kinds of ordered sequence. The code for the two sets was essentially the same – only the data types of the [contained](#) data varied. The same is true if we compare the code for the two kinds of sequences.

Now, we certainly [can](#) get these different versions of the same container class by copying a file and editing it. But, as you’ve seen in prior lecture, this approach can be tedious and error-prone. It also creates project management nightmares when a bug is fixed in one copy of an often-replicated container, and we must then find and fix all the copies.

## 1 Class Templates

- A template is a “pattern” for a class or function in which certain type names and constants are left as general [template parameters](#).
- These template parameters are specified in a [template header](#) at the start of the class declaration. This template header consists of the keyword “template” followed by the list of replaceable names inside <...>.
- We [instantiate](#) a class template when we are ready to use it by supplying the parameter value:

```
classTemplateName<myFavoriteType> myVariable;
```

- the resulting class is an [instance](#) of the template.

You’ve probably *used* some class templates before. In CS250/333 you would have made good use of `std::vector` as a kind of expandable array:

```
vector<string> names;
// Read names from a file.
ifstream input ("names.txt");
string oneName;
while (input >> oneName) {
    names.push_back (oneName);
}
int numberOfNamesRead = names.size();
```

But you may or may not have *written* your own class templates yet.

## 2 Developing and Using Class Templates

### 2.1 Templates as a Pattern for Data

Here’s some basic code declaring a typical sequence of strings, together with a function that inserts a data value into the sequence in a specific position.

```
struct Sequence {
    string* data;
    int numStrings;
};
```

```

void add (Sequence& toSequence, string newValue, int position)
{
    for (int i = toSequence.numStrings-1; i >= position; --i)
        toSequence.data[i+1] = toSequence.data[i];
    toSequence.data[position] = newValue;
    ++toSequence.numStrings;
}

```

Notice that the code to insert something into this sequence would be the same if it were a sequence of `int` instead of `string`, or a sequence of `double`, or a sequence of `PayrollRecords`. That's a sign that this structure might be a good candidate for being turned into a template.

## 2.1.1 Adding a Template Header

We start the conversion of this into a template class<sup>1</sup> by adding a template header to introduce a new “placeholder” name, `Data`, to represent the data stored in the sequence.

```

seq1.cpp + 

template <class Data>
struct Sequence {
    Data* data;
    int numStrings;
};

void add (Sequence& toSequence, string newValue, int position)
{
    for (int i = toSequence.numStrings-1; i >= position; --i)
        toSequence.data[i+1] = toSequence.data[i];
    toSequence.data[position] = newValue;
    ++toSequence.numStrings;
}

```

At this point, the name “Sequence” no longer defines a class/struct, but a pattern for an infinite number of possible classes.

That's all we need to do to turn `Sequence` into a class template. Now we can declare sequences of any kinds of data that we want.

Of course, that may not be very useful if the only functions we have for manipulating sequences assume, like `add` here, that *their* sequences only contain `string` data. So we really need to turn `add` into a template function ...

```

seq2.cpp + 

template <class Data>
struct Sequence {
    Data* data;
    int numStrings;
};

template <class Data>
void add (Sequence<Data>& toSequence, Data newValue, int position)
{
    for (int i = toSequence.numStrings-1; i >= position; --i)
        toSequence.data[i+1] = toSequence.data[i];
    toSequence.data[position] = newValue;
    ++toSequence.numStrings;
}

```

Notice that, everywhere we used to just say `Sequence`, now we must instantiate the class template by saying `Sequence<...>`.

## 2.2 Instantiating Class Templates

Instantiating class templates is a little different from instantiating function templates. When we have a function template like:

```

template <class T>
inline const T& min(const T& a, const T& b) {
    return b < a ? b : a;
}

```

we instantiate it simply by using it:

```

int i, j, k;
double x, y, z;
:
k = min(i, j); // instantiated with T => int
z = min(x, y); // instantiated with T => double

```

and the compiler would infer the appropriate replacements for the template parameters by examining the data types of the actual parameters.

But with a class template, we supply the replacements ourselves when we use the class as a type name:

```
Sequence<PayrollRecord> peopleGettingRaises;
```

## 2.3 Templates with Function Members

The example we've looked at was pretty simple. In particular, the class we used had no member functions.

Now, let's start over, this time with some member functions.

```
struct Sequence {
    string* data;
    int numStrings;

    Sequence (int maxSize);

    void add (string newValue, int position);
};

Sequence::Sequence (int maxSize)
: numStrings(0)
{
    data = new string[maxSize];
}

void Sequence::add (string newValue, int position)
{
    for (int i = numStrings-1; i >= position; --i)
        data[i+1] = data[i];
    data[position] = newValue;
    ++numStrings;
}
```

In this case, I've added a constructor that initializes the data fields of the sequence and turned the add function into a member function.

Now, we start by "templatizing" the class declaration and the sequentialInsert function just as we did with sequentialSearch earlier ...

### 2.3.1 Add the Header and Update the Date Type References

```
+ seq3.cpp +
```

```
template <class Data>
struct Sequence {
    Data* data;
    int numStrings;

    Sequence (int maxSize);

    void add (Data newValue, int position);
};

Sequence::Sequence (int maxSize)
: numStrings(0)
{
    data = new string[maxSize];
}

void Sequence::add (string newValue, int position)
{
    for (int i = numStrings-1; i >= position; --i)
        data[i+1] = data[i];
    data[position] = newValue;
    ++numStrings;
}
```

But what to do about the body of the member functions?

### 2.3.2 Member Functions Become Member Function Templates

Member functions of a template class are, implicitly, templates and so when we supply their bodies, we have to add a template header to indicate what names we are using for the placeholders.

```
+ seq4.cpp +
```

```
template <class Data>
struct Sequence {
    Data* data;
    int numStrings;

    Sequence (int maxSize);

    void add (Data newValue, int position);
};
```

```

template <class Data>
Sequence<Data>::Sequence (int maxSize)
: numStrings(0)
{
    data = new string[maxSize];
}

template <class Data>
void Sequence<Data>::add (Data newValue, int position)
{
    for (int i = numStrings-1; i >= position; --i)
        data[i+1] = data[i];
    data[position] = newValue;
    ++numStrings;
}

```

So we convert the member function by writing it as a function template. (In fact, the add member function body winds up looking a lot like the earlier example of the add non-member function template.)

## 2.4 Tips for Implementing Templates

For more complicated classes, I recommend the following approach to implementing class templates:

1. Start with a working non-template class in which you have used the template parameter names but employed `typedef`'s to supply replacements for the parameter names.

(Strictly speaking, this step isn't necessary. But compilers are able to do more checking for you this way, and often will issue much clearer error messages.)

2. Combine everything from the class's .h and .cpp files into the .h file.
3. Get rid of the phony `typedef`s, and replace by a template header at the start of the class, declaring the template parameters. Add a similar template header to each member definition.
4. Change all uses of the class name, except for constructor and destructor names, to `theClassName<templateParams>`.
5. Check all closely related, nested, and friend classes to see if they need to be converted to templates as well.

## 3 Example – Matrix

Arrays in C++ may seem like familiar old friends to you. Certainly, you should be comfortable with singly-dimensioned arrays. But many problems require a two-dimensional array.

This can be done working directly with C++ arrays (by having an array of arrays) but initializing and cleaning up after such structures can be awkward and error-prone. As with many data structures, it may be worth putting in the effort to develop a class that provides this capability so that, having put in the effort once to get it working, we can use it with confidence many times in the future.

Here is the header file for a matrix-of-floating-point-numbers class.<sup>2</sup>

```

#ifndef MATRIX_H
#define MATRIX_H

#include <algorithm>
#include <cassert>

class Matrix
// 
// Provides a "2-dimensional" rectangular
// array-like structure for indexing using
// a pair of indices.
{
public:
    Matrix();

    Matrix (unsigned theLength1, unsigned theLength2);

    Matrix (const Matrix&);

    ~Matrix();

    const Matrix& operator= (const Matrix&);

    // Indexing into the matrix: What we would like to do is allow
    // myMatrix[i,j]. But C++ allows operator[] only to take a single
    // parameter. But operator() can take whatever parameters we like.
    // So we can write myMatrix(i,j).
    double& operator() (int i1, int i2);
    const double& operator() (int i1, int i2) const;

    unsigned length1() const;
    unsigned length2() const;
}

```

```

    bool operator==(const Matrix&) const;

private:
    double* data;
    unsigned _length1;
    unsigned _length2;

};

#endif

```

The class itself is pretty straightforward except for the declaration of an `operator()` function taking two integer parameters. It may seem a bit strange to think of `()` as an operator, but C++ treats it that way. So we can use this class like this:

```

Matrix m(4,3);
for (int i = 0; i < 4; ++i)
{
    m(i,0) = (double)i;
    for (int j = 1; j < 3; ++j)
        m(i,j) = m(i,j-1) + j;
}

```

The expressions `m(i,0)`, `m(i,j)`, etc., are actually invoking that odd-looking operator. It's very much like working with an array, except for the use of `()` rather than `[]`.

Here is the implementation of the matrix class:

```

#include "matrix.h"

Matrix::Matrix()
    : data(0), _length1(0), _length2(0)
{ }

Matrix::Matrix(unsigned theLength1, unsigned theLength2)
    : _length1(theLength1), _length2(theLength2)
{
    data = new double[_length1 * theLength2];
}

Matrix::Matrix(const Matrix& m)
    : _length1(m._length1), _length2(m._length2)
{
    data = new double[_length1 * theLength2];
    copy(m.data, m.data + _length1 * theLength2, data);
}

Matrix::~Matrix()
{
    delete [] data;
}

const Matrix& Matrix::operator= (const Matrix& m)
{
    if (this != &m)
    {
        if (_length1 * _length2 < m._length1 * m._length2)
        {
            delete [] data;
            data = new double[m._length1 * m._length2];
        }
        _length1 = m._length1;
        _length2 = m._length2;
        copy (m.data, m.data + _length1 * _length2, data);
    }
    return *this;
}

// Indexing into the matrix: What we would like to do is allow
// myMatrix[i,j]. But C++ allows operator[] only to take a single
// parameter. But operator() can take whatever parameters we like.
// So we can write myMatrix(i,j).
double& Matrix::operator() (int i1, int i2)
{
    assert ((i1 >= 0) && (i1 < _length1));
    assert ((i2 >= 0) && (i2 < _length2));
    return data[i1 + _length1 * i2];
}

const double& Matrix::operator() (int i1, int i2) const

```

```

{
    assert ((i1 >= 0) && (i1 < _length1));
    assert ((i2 >= 0) && (i2 < _length2));
    return data[i1 + _length1*i2];
}

inline
unsigned Matrix::length1() const
{
    return _length1;
}

inline
unsigned Matrix::length2() const
{
    return _length2;
}

bool Matrix::operator==(const Matrix& m) const
{
    return (_length1 == m._length1)
        && (_length2 == m._length2)
        && equal (data, data+_length1*_length2, m.data);
}

```

The implementation file for this class is pretty straightforward once you grasp the key idea that a two-dimensional array can be mapped onto a one-dimensional array by taking advantage of the formula  $i + j * n_i$ , which maps each possible (i,j) pair onto a unique integer position, if  $n_i$  is the number of values in each “row” (i.e., the number of possible values of i). For example, the matrix m declared as

```
Matrix m(4,3);
```

would store element (0,0) in position 0, (1,0) in position 1, (0,1) in position 4, (1,1) in position 5, etc.

So we would visualize this matrix as

0,0	1,0	2,0	3,0
0,1	1,1	2,1	3,1
0,2	1,2	2,2	3,2

but implement it as

0,0	1,0	2,0	3,0	0,1	1,1	2,1	3,1	0,2	1,2	2,2	3,2
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Now, how to convert this to a more general matrix-of-anything template? To do that, we would take the matrix-of-double code and

1. Start with a working non-template class using template parameter names but employing typedefs to supply replacements for the parameter names.
2. Combine everything from the class's .h and .cpp files into the .h file.
3. Get rid of the phony typedefs, and replace by a template header at the start of the class, declaring the template parameters. Add a similar template header to each member definition.
4. Change all uses of the class name, except for constructor and destructor names, to the `ClassName<templateParams>`.
5. Check all closely related, nested, and friend classes to see if they need to be converted to templates as well.

We'll take it in steps.

### 3.1 Start with a working class...

```

matrix0.h +
#ifndef MATRIX_H
#define MATRIX_H

#include <algorithm>
#include <cassert>

typedef double Element;

```

```

class Matrix
//
// Provides a "2-dimensional" rectangular
// array-like structure for indexing using
// a pair of indices.
{
public:
    Matrix();
    Matrix (unsigned theLength1, unsigned theLength2);
    Matrix (const Matrix&);

    ~Matrix();

    const Matrix& operator= (const Matrix&);

    // Indexing into the matrix: What we would like to do is allow
    // myMatrix[i,j]. But C++ allows operator[] only to take a single
    // parameter. But operator() can take whatever parameters we like.
    // So we can write myMatrix(i,j).
    Element& operator() (int i1, int i2);
    const Element& operator() (int i1, int i2) const;

    unsigned length1() const;
    unsigned length2() const;

    bool operator== (const Matrix&) const;

private:
    Element* data;
    unsigned _length1;
    unsigned _length2;
};

#endif

```

### [matrix0.cpp](#) +

```

#include "matrix.h"

Matrix::Matrix()
    : data(0), _length1(0), _length2(0)
{ }

Matrix::Matrix(unsigned theLength1, unsigned theLength2)
    : _length1(theLength1), _length2(theLength2)
{
    data = new Element[_length1*_length2];
}

Matrix::Matrix(const Matrix& m)
    : _length1(m._length1), _length2(m._length2)
{
    data = new Element[_length1*_length2];
    copy (m.data, m.data+_length1*_length2, data);
}

Matrix::~Matrix()
{
    delete [] data;
}

const Matrix& Matrix::operator= (const Matrix& m)
{
    if (this != &m)
    {
        if (_length1*_length2 < m._length1*m._length2)
        {
            delete [] data;
            data = new Element[m._length1*m._length2];
        }
        _length1 = m._length1;
        _length2 = m._length2;
        copy (m.data, m.data+_length1*_length2, data);
    }
    return *this;
}

// Indexing into the matrix: What we would like to do is allow
// myMatrix[i,j]. But C++ allows operator[] only to take a single
// parameter. But operator() can take whatever parameters we like.

```

```

// So we can write myMatrix(i,j).
Element& Matrix::operator() (int i1, int i2)
{
    assert ((i1 >= 0) && (i1 < _length1));
    assert ((i2 >= 0) && (i2 < _length2));
    return data[i1 + _length1*i2];
}

const Element& Matrix::operator() (int i1, int i2) const
{
    assert ((i1 >= 0) && (i1 < _length1));
    assert ((i2 >= 0) && (i2 < _length2));
    return data[i1 + _length1*i2];
}

inline
unsigned Matrix::length1() const
{
    return _length1;
}

inline
unsigned Matrix::length2() const
{
    return _length2;
}

bool Matrix::operator==(const Matrix& m) const
{
    return (_length1 == m._length1)
        && (_length2 == m._length2)
        && equal (data, data+_length1*_length2, m.data);
}

```

We expect that `Matrix` is already working. We start by changing all the instances of `double` to a name we can eventually use as a template parameter.

## 3.2 Combine into .h File

```

matrix1.h [+]

#ifndef MATRIX_H
#define MATRIX_H

#include <algorithm>
#include <cassert>

typedef double Element;

class Matrix
//
// Provides a "2-dimensional" rectangular
// array-like structure for indexing using
// a pair of indices.
{
public:
    Matrix();

    Matrix (unsigned theLength1, unsigned theLength2);

    Matrix (const Matrix&);

    ~Matrix();

    const Matrix& operator= (const Matrix&);

    // Indexing into the matrix: What we would like to do is allow
    // myMatrix[i,j]. But C++ allows operator[] only to take a single
    // parameter. But operator() can take whatever parameters we like.
    // So we can write myMatrix(i,j).
    Element& operator() (int i1, int i2);
    const Element& operator() (int i1, int i2) const;

    unsigned length1() const;
    unsigned length2() const;

    bool operator==(const Matrix&) const;

private:
    Element* data;
    unsigned _length1;
    unsigned _length2;
}

```

```

};

Matrix::Matrix()
: data(0), _length1(0), _length2(0)
{}

Matrix::Matrix(unsigned theLength1, unsigned theLength2)
: _length1(theLength1), _length2(theLength2)
{
    data = new Element[_length1*_length2];
}

Matrix::Matrix(const Matrix& m)
: _length1(m._length1), _length2(m._length2)
{
    data = new Element[_length1*_length2];
    copy (m.data, m.data+_length1*_length2, data);
}

Matrix::~Matrix()
{
    delete [] data;
}

const Matrix& Matrix::operator= (const Matrix& m)
{
    if (this != &m)
    {
        if (_length1*_length2 < m._length1*m._length2)
        {
            delete [] data;
            data = new Element[m._length1*m._length2];
        }
        _length1 = m._length1;
        _length2 = m._length2;
        copy (m.data, m.data+_length1*_length2, data);
    }
    return *this;
}

// Indexing into the matrix: What we would like to do is allow
// myMatrix[i,j]. But C++ allows operator[] only to take a single
// parameter. But operator() can take whatever parameters we like.
// So we can write myMatrix(i,j).
Element& Matrix::operator() (int i1, int i2)
{
    assert ((i1 >= 0) && (i1 < _length1));
    assert ((i2 >= 0) && (i2 < _length2));
    return data[i1 + _length1*i2];
}

const Element& Matrix::operator() (int i1, int i2) const
{
    assert ((i1 >= 0) && (i1 < _length1));
    assert ((i2 >= 0) && (i2 < _length2));
    return data[i1 + _length1*i2];
}

inline
unsigned Matrix::length1() const
{
    return _length1;
}

inline
unsigned Matrix::length2() const
{
    return _length2;
}

bool Matrix::operator== (const Matrix& m) const
{
    return (_length1 == m._length1)
        && (_length2 == m._length2)
        && equal (data, data+_length1*_length2, m.data);
}

#endif

```

Easy enough. Don't forget though, to move the #endif down to the new bottom.

### 3.3 Add Template Parameters

matrix2.h +

```
#ifndef MATRIX_H
#define MATRIX_H

#include <algorithm>
#include <cassert>

template <class Element>
class Matrix
// 
// Provides a "2-dimensional" rectangular
// array-like structure for indexing using
// a pair of indices.
{
public:
    Matrix();
    Matrix (unsigned theLength1, unsigned theLength2);
    Matrix (const Matrix&);

    ~Matrix();

    const Matrix& operator= (const Matrix&);

    // Indexing into the matrix: What we would like to do is allow
    // myMatrix[i,j]. But C++ allows operator[] only to take a single
    // parameter. But operator() can take whatever parameters we like.
    // So we can write myMatrix(i,j).
    Element& operator() (int i1, int i2);
    const Element& operator() (int i1, int i2) const;

    unsigned length1() const;
    unsigned length2() const;

    bool operator==(const Matrix&) const;

private:
    Element* data;
    unsigned _length1;
    unsigned _length2;
};

template <class Element>
Matrix::Matrix()
    : data(0), _length1(0), _length2(0)
{ }

template <class Element>
Matrix::Matrix(unsigned theLength1, unsigned theLength2)
    : _length1(theLength1), _length2(theLength2)
{
    data = new Element[_length1*_length2];
}

template <class Element>
Matrix::Matrix(const Matrix& m)
    : _length1(m._length1), _length2(m._length2)
{
    data = new Element[_length1*_length2];
    copy (m.data, m.data+_length1*_length2, data);
}

template <class Element>
Matrix::~Matrix()
{
    delete [] data;
}

template <class Element>
const Matrix& Matrix::operator= (const Matrix& m)
{
    if (this != &m)
    {
        if (_length1*_length2 < m._length1*m._length2)
        {
            delete [] data;
            data = new Element[m._length1*m._length2];
        }
    }
}
```

```

        _length1 = m._length1;
        _length2 = m._length2;
        copy (m.data, m.data+_length1*_length2, data);
    }
    return *this;
}

// Indexing into the matrix: What we would like to do is allow
// myMatrix[i,j]. But C++ allows operator[] only to take a single
// parameter. But operator() can take whatever parameters we like.
// So we can write myMatrix(i,j).
template <class Element>
Element& Matrix::operator() (int i1, int i2)
{
    assert ((i1 >= 0) && (i1 < _length1));
    assert ((i2 >= 0) && (i2 < _length2));
    return data[i1 + _length1*i2];
}

template <class Element>
const Element& Matrix::operator() (int i1, int i2) const
{
    assert ((i1 >= 0) && (i1 < _length1));
    assert ((i2 >= 0) && (i2 < _length2));
    return data[i1 + _length1*i2];
}

template <class Element>
unsigned Matrix::length1() const
{
    return _length1;
}

template <class Element>
unsigned Matrix::length2() const
{
    return _length2;
}

template <class Element>
bool Matrix::operator== (const Matrix& m) const
{
    return (_length1 == m._length1)
        && (_length2 == m._length2)
        && equal (data, data+_length1*_length2, m.data);
}

#endif

```

Template headers have been added to the class and to each function body.

## 3.4 Rewrite Class Names

```

matrix3.h +
#ifndef MATRIX_H
#define MATRIX_H

#include <algorithm>
#include <cassert>

template <class Element>
class Matrix
//
// Provides a "2-dimensional" rectangular
// array-like structure for indexing using
// a pair of indices.
{
public:
    Matrix();
    Matrix (unsigned theLength1, unsigned theLength2);
    Matrix<Element> (const Matrix<Element>&);

    ~Matrix();

    const Matrix<Element>& operator= (const Matrix<Element>&);

    // Indexing into the matrix: What we would like to do is allow
    // myMatrix[i,j]. But C++ allows operator[] only to take a single

```

```

// parameter. But operator() can take whatever parameters we like.
// So we can write myMatrix(i,j).
Element& operator() (int i1, int i2);
const Element& operator() (int i1, int i2) const;

unsigned length1() const;
unsigned length2() const;

bool operator==(const Matrix<Element>&) const;

private:
Element* data;
unsigned _length1;
unsigned _length2;

};

template <class Element>
Matrix<Element>::Matrix()
: data(0), _length1(0), _length2(0)
{ }

template <class Element>
Matrix<Element>::Matrix(unsigned theLength1, unsigned theLength2)
: _length1(theLength1), _length2(theLength2)
{
    data = new Element[_length1*_length2];
}

template <class Element>
Matrix<Element>::Matrix(const Matrix<Element>& m)
: _length1(m._length1), _length2(m._length2)
{
    data = new Element[_length1*_length2];
    copy (m.data, m.data+_length1*_length2, data);
}

template <class Element>
Matrix<Element>::~Matrix()
{
    delete [] data;
}

template <class Element>
const Matrix<Element>& Matrix<Element>::operator=(const Matrix<Element>& m)
{
    if (this != &m)
    {
        if (_length1*_length2 < m._length1*m._length2)
        {
            delete [] data;
            data = new Element[_length1*_length2];
        }
        _length1 = m._length1;
        _length2 = m._length2;
        copy (m.data, m.data+_length1*_length2, data);
    }
    return *this;
}

// Indexing into the matrix: What we would like to do is allow
// myMatrix[i,j]. But C++ allows operator[] only to take a single
// parameter. But operator() can take whatever parameters we like.
// So we can write myMatrix(i,j).
template <class Element>
Element& Matrix<Element>::operator()(int i1, int i2)
{
    assert ((i1 >= 0) && (i1 < _length1));
    assert ((i2 >= 0) && (i2 < _length2));
    return data[i1 + _length1*i2];
}

template <class Element>
const Element& Matrix<Element>::operator()(int i1, int i2) const
{
    assert ((i1 >= 0) && (i1 < _length1));
    assert ((i2 >= 0) && (i2 < _length2));
    return data[i1 + _length1*i2];
}

template <class Element>

```

```

unsigned Matrix<Element>::length1() const
{
    return _length1;
}

template <class Element>
unsigned Matrix<Element>::length2() const
{
    return _length2;
}

template <class Element>
bool Matrix<Element>::operator==(const Matrix<Element>& m) const
{
    return (_length1 == m._length1)
        && (_length2 == m._length2)
        && equal(data, data+_length1*_length2, m.data);
}
#endif

```

Uses of the class name have been replaced.

## 3.5 Check Related Classes

There are no such closely related classes, so we are done.

We can use this class like this:

```

Matrix<int> m(4,3);
for (int i = 0; i < 4; ++i)
{
    m(i,0) = i;
    for (int j = 1; j < 3; ++j)
        m(i,j) = m(i,j-1) + j;
}

```

## 4 Second Example – the pair class

There are times when the template form of a class is more useful than the original class might have been. For example, sometimes we need to “pair up” two different data items just so we can insert them together inside a container.

For example, suppose that we were writing a spellchecker.

A natural ADT in a spell checker would be the notion of a `Replacement` - a misspelled word and the correct spelling to put in its place.

Now, `Replacement` is really little more than a pair of strings. Because it's a fairly basic concept in spell-checker-land, we could give it a descriptive name and give descriptive names to its component pieces, as shown here.

```

class Replacement
{
public:
    Replacement () {}
    Replacement (const std::string& missp,
                 const std::string& repl);

    void setMisspelledWord (const std::string& mw);
    std::string getMisspelledWord() const;

    void setReplacement (const std::string& r);
    std::string getReplacement() const;

    void put (ostream&) const;

private:
    std::string _misspelledWord;
    std::string _replacement;
};

```

Another idea that arises in the spell checker is that of a “word occurrence”, a combination of a word and the location in a document where that word was found.

```

class WordOccurrence
{
public:
    typedef std::streampos Location;

    WordOccurrence();
    //post: getLexeme() == "" && getLocation() == 0

    WordOccurrence (const std::string& lexeme,

```

```

        Location location);

// A "lexeme" is the string of characters that has been identified
// as constituting a token.
std::string getLexeme() const {return _lexeme;}
void putLexeme (const std::string& lex) {_lexeme = lex;}

// The location indicates where in the original file the first character of
// a token's lexeme was found.
Location getLocation() const;
void setLocation (const Location&);

// Output (mainly for debugging purposes)
void put (std::ostream&) const;

private:
    std::string _lexeme;
    Location _location;
};

```

This ADT follows much the same pattern as the previous one. It's really just a container of two items. Unlike `Replacement`, the two items aren't of the same type. But we're basically looking at a simple pair of items, what a mathematician would call a "tuple".

## 4.1 Thinking in Terms of Tuples

What's to stop us from just writing these classes like this?

```

class Replacement {
public:
    string first;
    string second;

    Replacement () {}
    Replacement (string f, string s):
        first(f), second(s)  {}
};

class WordOccurrence {
public:
    string first;
    Location second;

    WordOccurrence () {}
    WordOccurrence (string f, Location s):
        first(f), second(s)  {}
};

```

Not much. The names aren't as nice as our originals, we generally would prefer that all member variables should be private.

So there's no good reason to really do this ...

## 4.2 The pair Template

... until we consider maybe turning this pattern for containers-of-two-things into a template. Then it becomes one of those little things that's not earth-shattering, but is still nice to have around.

```

// from std header file <utility>

template <class T1, class T2>
class pair {
public:
    T1 first;
    T2 second;

    pair () {}
    pair (T1 f, T2 s):
        first(f), second(s)  {}
};

```

In fact, this `pair` template is part of the C++ std library, in the header `<utility>`. (`<utility>` also defines operators `<` and `==` for pairs.)

Using `pair`, we could, if we wished, redefine `Replacement` and `WordOccurrence` as follows

```

typedef pair<std::string, std::string> Replacement;
typedef pair<std::string, std::streampos> WordOccurrence;

```

Whether or not this is a wise choice is a judgment call. Personally, I would probably not do so, reasoning that these two abstractions are so pervasive in a typical spell checker that the documentation value obtained by accessing their components as "`getMisspelledWord()`" and "`getReplacement()`" is greatly preferred to simply "`.first`" and "`.second`".

## 4.3 Applications of pair<...>

pair gets used in situations where we need to construct a simple container-of-two-things without a lot of fuss.

For example, suppose we wanted to modify our seqSearch template

```
search1.cpp +  
  
template <class T>  
int sequentialSearch  
(const T a[], unsigned n, const T& x)  
// Look for x within a sorted array a, containing n items.  
// Return the position where found, or -1 if not found.  
{  
    int i;  
    for (i = 0; ((i < n) && (a[i] < x)); i++) ;  
    if ((i >= n) || (a[i] != x))  
        i = -1;  
    return i;  
}  
  
template <typename T>  
int seqSearch(const T list[], int listLength, T searchItem)  
{  
    int loc;  
  
    for (loc = 0; loc < listLength; loc++)  
        if (list[loc] == searchItem)  
            return loc;  
  
    return -1;  
}
```

so that, instead of returning the integer position at which an item was found (-1 if the item was not found), we returned a reference directly to the found item:

```
template <typename T>  
const T& seqSearch(const T list[], int listLength, T searchItem)  
{  
    int loc;  
  
    for (loc = 0; loc < listLength; loc++)  
        if (list[loc] == searchItem)  
            return list[loc];  
  
    return -1;  
}
```

### 4.3.1 Returning the element that was found

That's not too hard when we actually find the value we were looking for. But if we don't find it (`return -1`), then what do we do?

We need to return, not just the reference, but also a boolean value indicating whether or not we found it. If we return false, then the calling program will know not to actually look at the reference.

But wait. How can we return the reference *and* the boolean flag? Isn't it true that a function can only return one value?

Of course it is. So we have two choices. The simplest choice is to turn one of those would-be return values into a simple output parameter:

```
template <typename T>  
bool seqSearch(const T list[], int listLength, T searchItem, T& foundValue)  
{  
    :  
}
```

### 4.3.2 Returning Two Things at Once

There are times, though, when we *really* want that information passed out as a returned value. In those cases, we can use pair.

```
template <typename T>  
pair<bool, const T&> seqSearch(const T list[], int listLength,  
                                T searchItem)  
{  
    int loc;  
  
    for (loc = 0; loc < listLength; loc++)  
        if (list[loc] == searchItem)  
            return pair<bool, const T&>(true, list[loc]);  
  
    return pair<bool, const T&>(false, list[0]);  
}
```

The rather hefty looking return expressions are, if you look closely, actually invoking the constructor for the data type `pair<bool, const T&>`, which was declared as the functions return type. That constructor takes two parameters, hence the two expressions inside the () .

So we really are returning only one value from our function, but it just so happens that that one value is a pair.

Application code to use this would look something like:

```
pair<bool, string&> p = sequentialSearch(a, n, x);
if (p.first)
    cout << "Found "
        << p.second
        << endl;
else
    cout << "Not found"
        << endl;
```

although, thanks to C++11, we can simplify that first statement:

```
auto p = sequentialSearch(a, n, x);
if (p.first)
    cout << "Found "
        << p.second
        << endl;
else
    cout << "Not found"
        << endl;
```

## 5 Third Example: array

Wait...“array”? Don’t we already have that?

Yes, but C++ has always had a kind of love-hate relationship with its arrays. It inherited from its parent language, C, the idea that an array is just a pointer that points to repeated instances of its elements instead of just to a single one.

This has some interesting implications, both positive and negative:

- Array declarations can be hard to read. It’s OK for static arrays. If we write

```
int c[3];
```

it’s clear that we want an array. But things get muddier for dynamically allocated arrays. When you see

```
int* a;
```

there’s no way to tell just from looking at it whether a is intended to hold a pointer to a single integer or to an entire array of them. We have to hunt through the code for the place where the memory is actually allocated to find out whether we see

```
a = new int;
```

or

```
a = new int[N];
```

- Arrays don’t copy naturally. If I write

```
array1 = array2;
```

that does not copy the array, but copies the address of the array (and if we aren’t careful, losing the old address in array1 and leaving ourselves with a memory leak).

This has both good and bad points.

- Copying large arrays can take a lot of time and memory, and it’s not something we want to do indiscriminately.
- On the other hand, sometimes we really **do** want to make a copy, and it’s annoying to have to write out an entire loop for an operation that is a simple assignment for almost all other data types in C++.
- Arrays can’t be directly compared for equality.
- Arrays don’t have the same parameter passing options in function calls that other data types have. With other data types, we can pass by copy, giving the function its own copy to examine, modify, and even destroy without affecting our original. When we want an output from the function, or want to avoid the time and memory cost of a copy, we can pass by reference instead.

For normal C++ arrays, we don’t have the option. We will only send an address – there is no “pass a copy” mechanism.

So, the trend in C++ is to move away from use of ordinary arrays. What we replace them with depends on the main question we *always* have to ask when working with arrays: do we know the size of the array we want at compilation time, or is the size computed at run time?

- If we know the size we want when we are writing the code (compile time), then we generally used statically allocated arrays such as

```
int c[10];
```

The newer style is to use `std::array`:

```
std::array<int, 10> c;
```

- If our code needs to compute the size we want at run time, then we generally used dynamically allocated arrays such as

```
int* c = new int[N];
```

The newer style is to use `std::vector`:

```
std::vector c;
```

You should already be familiar with the basic use of vectors. If not, go [here](#) for a quick review. We'll look at the implementation of the `vector` template in a [future lesson](#).

Right now, though, let's look at `array` as yet another example of a class template.

As shown above, we will declare standard arrays using a template instantiation:

```
std::array<int, 10> c;
```

The second template parameter **must** be a compile-time constant. It can be a name, instead of a literal constant, if the name has been declared as a constant:

```
const int MaxCustomers = 10000;
std::array<CustomerRecord, MaxCustomers> accounts;
```

We can initialize these arrays when we declare them:

```
std::array<int, 10> d = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Once we have a `std::array`, we can use it like an ordinary array:

```
c[i] = d[j] + 1;
```

but we also have some other options

```
c.at(i) = d.at(j) + 1;
```

This means the same thing as the `[ ]` form, but the index value is checked first to be sure it is in the legal bounds for this array, and a run-time error is signaled if the index is illegal.

We also have some things we can do with these that we cannot do with pointer-based arrays:

- We can fill the entire array with a value:

```
c.fill(0);
```

- We can exchange two arrays (if they have the same element type and size):

```
c.swap(d);
```

- We can ask how many elements are in the array:

```
unsigned nC = c.size();
```

- We can compare them:

```
if (c == d)
if (c != d)
if (c < d)
:
```

(`c < d` is true if the two arrays have at least one pair of corresponding elements that are not equal, and the first such pair, say `c[k]` and `d[k]`, have `c[k] < d[k]`.)

Assignment of std arrays actually makes a copy:

```
c = d;
```

and copies are made when std arrays are passed as copy parameters to functions.

Best of all, code using `std::array` should be nearly as efficient as code using pointer-based arrays. In fact, in many cases the code generated by the compiler should be nearly identical.

So how do we pull off this wonderful new class?

```
template < typename T, size_t N >           ①
class array
{
    T elements[N];                         ②
public:
    typedef T             value_type;      ③
    typedef value_type*   pointer;
    typedef const value_type* const_pointer;
    typedef value_type&   reference;
    typedef const value_type& const_reference;
    typedef std::size_t    size_type;
    ...
    void fill(const value_type& __u);        ④
    void swap(array& __other);
    ...

// Capacity.
const size_type size() const { return N; }
const size_type max_size() const noexcept { return N; }
const bool empty() const { return size() == 0; }

// Element access.
reference operator[](size_type i);          ⑤
const_reference operator[](size_type i) const;

reference at(size_type i);
const_reference at(size_type i) const;

reference front();
const_reference front() const;

reference back();
const_reference back() const;

pointer data() { return elements; }
const_pointer data() const { return elements; }
};

// Array comparisons.
template<typename T, std::size_t N>           ⑥
inline bool
operator==(const array<T, N>& left, const array<T, N>& right);

template<typename T, std::size_t N>
inline bool
operator!=(const array<T, N>& left, const array<T, N>& right);

template<typename T, std::size_t N>
inline bool
operator<(const array<T, N>& left, const array<T, N>& right);
:
```

- ① This is the first time we have seen a template header where one of the template parameters did not start with “typename”. There are actually two kinds of things that can be used as template parameters: data types and constants. If we want to pass a constant as a template parameter, we precede its name with its data type instead of “typename”.

In this case, `size_t` is a type declared elsewhere in the `std` library. It denotes some form of unsigned integer.

- ② This is the entire data structure for `std::array`. It’s just a statically allocated array. Because it’s wrapped inside a class/struct, however, copying the struct will copy the entire array.
- ③ Something else we have not seen before. Classes (whether template or not) can provide data type names as “members” much as they provide data and function members. These can be used to provide a convenient shortcut for some names or, in the case of `std`, to help provide a uniform interface and naming scheme for all the different containers of data in the library.

We access these types the same way that we access static data members and functions, via the `::` operator.

```
typedef std::array<std::string, 100> ShortWordList;
:
ShortWordList commonWords = {"a", "the", "of", "in", ... };
ShortWordList::size_type nWords = commonWords.size();
```

If you struggle with the idea of when to use `.` and when to use `::`, there are two important things to remember:

- The . always has an object or value on the left. The :: always has a data type or namespace on the left.
- We use . to get the properties of an object – a specific value of some class type. We use :: to get properties of the entire data type – things that do not vary from one object to another.
- ④ The rest of the class declaration lists the various functions supplied by std::array. You should be able to see matches for the various operations on this type that we have already discussed.
- ⑤ here you see the declaration of operator[], the square bracket function for indexing into an array. We'll look at the function body for this in a moment. For now, though, look at the very next line.

We actually have two functions named operator[]. The const at the end of the declaration tells us that one of these functions is used when we have a std::array that we are allowed to change and the other is used when we have a std::array that is a constant. The non-const version returns a reference to (address of) a data element. And code calling this function could use that address to change the value stored in the array. But the const version returns a const reference to the same element. Because it is a const reference, code calling that version could look at but not change the values stored in the array.

This idea of providing distinct, overloaded function pairs for use on const and non-const data is a *very common practice in C++*.

- ⑥ Various relational operators are then declared, outside of the class declaration itself.

So, how does all this work? Let's look at the critical functions for accessing data. For example, to support operations like

```
c[i] = d[j] + 1;
```

we use

```
template <typename T, size_t N>
array<T,N>::reference array<T,N>::operator[]
  (array<T,N>::size_type i)
{
  return elements[i];
}
```

After all the elaborate syntax of the template header and the function declaration itself, the body is amazingly simple.

And the const version looks pretty much the same:

```
template <typename T, size_t N>
array<T,N>::const_reference array<T,N>::operator[]
  (array<T,N>::size_type i) const
{
  return elements[i];
}
```

The at functions just add a little bit of checking:

```
template <typename T, size_t N>
array<T,N>::reference array<T,N>::operator[]
  (array<T,N>::size_type i)
{
  if (i >= 0 && i < N)
    return elements[i];
  else
    ...abort with a run-time error...
}
```

Now, I have left out a little bit of information on the std::array. Most of that has to do with “iterators”, which we will look at very shortly.

**1:** The **only** difference between a **class** and a **struct** in C++ is that the members of a struct are, by default, public, but the members of a class are, by default, private.

**2:** Your textbook also develops a similar matrix template, but it uses some techniques that we have not covered yet, so for illustrative purposes I've decided to keep this example. Still, after you've read through this one, you may find it interesting to return to section 1.7 of your textbook and comparing to his version.

# The Standard Library: Overview

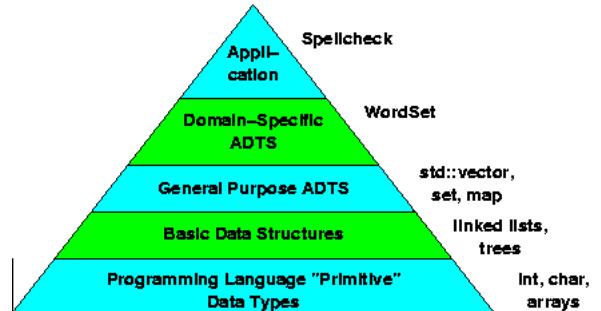
Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 std/STL Containers](#)
- [2 The std Containers](#)
  - [2.1 Sequential and Associative](#)
  - [2.2 Consistent Interfaces](#)
  - [2.3 A Matter of Style](#)

A large project builds upon libraries of ADTs built for that specific project, or that have been collected by the company for related projects (a.k.a. domain-specific ADTs). These in turn are generally built upon the library of ADTs provided by all C++ compilers. This standard C++ library is called “std”.



## 1 std/STL Containers

The key components of std are

- I/O library: e.g., streams
- Utility ADTs: e.g., string
- Containers (ADTs that hold one or more values of some other ADT): vector, set, ...
- Iterators: “positions” within a container
- Algorithms: common programming “patterns”

You are already familiar with many parts of std. You presumably were already familiar with the I/O portion. We have already looked at some utilities. Now we want to do a quick overview of the containers portion, prior to beginning to look at some of the specific containers in detail.

You may or may not have heard of the “STL” library before. When the C++ standards committee was considering what should be included in the standard library, std, the early drafts included what we now recognize as the I/O and utility sections. A group of researchers from HP then proposed addition of their STL library to std. STL consisted of a set of containers, iterators, and function templates (algorithms). “STL” stands for “Standard Template Library”, which may have been a bit presumptuous because it certainly was not part of the standard at the time. The C++ standards committee eventually accepted the STL, with some revisions, as part of std.

## 2 The std Containers

We call an ADT a “container” if its main purpose is to provide a collection of items of some other, simpler type.

### 2.1 Sequential and Associative

Collections come in two flavors

#### sequential

values are accessed in an order determined by their insertion.

The std sequential containers are `array`, `initializer_list`, `vector`, `deque`, `list`, `stack`, `queue`, and `priority_queue`.

#### associative

allow “random” access

The std associative containers are `set`, `map`, `multiset`, `multimap`, `unordered_set`, `unordered_map`, `unordered_multiset`, and `unordered_multimap`.

The first four “ordered” containers are based on trees. The four “unordered” containers are based on hashing.

We’ll look at all of these in upcoming lessons.

## 2.2 Consistent Interfaces

One goal of the `std` library is to provide consistent interfaces to many different containers.

- Many operations are common to multiple ADTS.
- These are always given the same name and formal parameter lists.

Examples:

- If a container allows you to add and remove elements from its end, those operations are always called `push_back` and `pop_back`.

```
myContainer.pop_back(); // discard last element from container
myContainer.push_back(x); // add x to end of container
```

Similarly, if a container allows you to add and remove elements from its front, those operations are always called `push_front` and `pop_front`.

- Operations to add and remove elements at arbitrary positions are always called `insert` and `erase`. These use iterators to specify the position:

```
myContainer.insert(x, position); // insert x at the indicated
                                // position, shifting other
                                // elements out of the way
myContainer.erase(position);    // discard the element at that position
myContainer.erase(start, stop); // discard all elements at positions
                                // start up to (but not including)
                                // stop.
```

- You can test to see if a container is empty with `empty()`. You can ask how many items are in the container with `size()`.

```
if (myContainer.empty())
{
    assert (myContainer.size() == 0);
}
```

- In the next reading, we will introduce the idea of an “iterator” to represent positions within the container. Nearly all containers will provide two iterator types, and these are called `iterator` and `const_iterator`. The beginning position in the container is given by `begin()`. The position just after the last element is given by `end()`. When the container is empty, `begin() == end()`.
- All containers will provide a set of data types describing their internal elements, e.g.:

```
template <typename T>
class Some_Standard_Container
{
public:
    typedef T           value_type;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef std::size_t size_type;

    typedef ...          iterator;
    typedef ...          const_iterator;
```

- `value_type` is the data type of the elements stored within the container. For the containers that you might already be familiar with, `vector` and `array`, this seems pretty obvious. `vector<int>::value_type` will be `int`, and `array<Nonsense, 100>::value_type` will be `Nonsense`. But when we get to the various forms of maps, you’ll see that the `value_type` is not always obvious.

As basic as it seems, `value_type` can be quite useful, especially when writing templates. For example, suppose that I wanted a function that would walk through two standard arrays, choosing the smaller value from each position, saving the results in another array:

```
template <typename Array>
Array collectSmallerElements (const Array& left, const Array& right)
{
    Array result;
    for (int i = 0; i < left.size(); ++i)
    {
        typename Array::value_type x = left[i];
        typename Array::value_type y = right[i];
        result[i] = min(x,y);
    }
    return result;
}
```

The `value_type` gives us a way to refer to the data types inside the containers, which would otherwise have been impossible to do.

- `pointer`, `const_pointer`, `reference` and `const_reference` are all variations on ways to get to a `value_type` element.

- What data types are actually used here depends on whether the underlying data structure could be broken by allowing the programmer to directly change element values “in place” within the container.

For example, with an array or vector, there's no real problem with writing something like:

```
Container::reference element = container[22];
element.title = "Data Structures";
```

But some of the containers we will be looking at actually use the data values to determine where an element will be stored. So changing an element value might corrupt the entire container. Such containers will want to generate a compilation error if you try the code above. And they do that by providing different types for `reference` and `pointer`:

```
typedef T           value_type;
typedef const value_type* pointer;
typedef const value_type* const_pointer;
typedef const value_type& reference;
typedef const value_type& const_reference;
```

because the compiler will not allow you to use `const` pointers and references to change the value of the data that they point to.

## 2.3 A Matter of Style

Because of this consistency in the interfaces, learning to use the `std` containers is more a matter of learning the `std` “style” than of learning 10-14 different ADT interfaces.

You can find a handy summary of the container interfaces in [my STL containers reference sheets](#). You can also find some very good overviews and documentation from [cplusplus.com](#).

# Iterators: an ADT for Positions

Steven J. Zeil

Last modified: May 24, 2020

## Contents:

- [1 The Abstraction: Positions Within a Collection of Data](#)
  - [1.1 Redesigning the Constructors](#)
  - [1.2 Providing Access to Individual Elements](#)
  - [1.3 A Related Problem – search functions.](#)
  - [1.4 What Operations Should A Position ADT Provide?](#)
- [2 The C++ iterator](#)
  - [2.1 The iterator as an ADT Interface](#)
  - [2.2 Names for iterator types](#)
  - [2.3 iterator vs. const\\_iterator](#)
  - [2.4 auto](#)
- [3 Example: Adding Iterators to the Book Interface](#)
  - [3.1 Creating an Iterator Class](#)
  - [3.2 Reusing Existing Data Types as Iterators](#)
- [4 Example: Cleaning Up the Book Constructors](#)
  - [4.1 initializer\\_list](#)
  - [4.2 Constructors with Start-Stop Ranges](#)
- [5 Iterator Variations](#)
  - [5.1 Forward Iterators](#)
  - [5.2 Bidirectional Iterators](#)
  - [5.3 Random Access Iterators](#)
  - [5.4 Input Iterators](#)
  - [5.5 Output Iterators](#)
  - [5.6 Reverse Iterators](#)
- [6 Example: Searching via Iterator Variants.](#)
  - [6.1 Ordered vs Binary Search](#)
  - [6.2 Letting the Compiler Choose](#)
- [7 Range-based for Loops](#)

In this lesson we introduce one of the fundamental building blocks of the C++ standard library: the *iterator*.

- Concept is simple
- Notation is unfamiliar
- Effective use requires a shift in how we think about many common programming tasks

## 1 The Abstraction: Positions Within a Collection of Data

Let's look back at our Book interface and focus on a couple of awkward interface decisions that we sort of glazed over in our earlier looks.

Here's what we have so far:

```
class Book {  
public:  
    Book();  
  
    Book (std::string theTitle, const Publisher* thePubl,  
          int numberofAuthors, Author* theAuthors,  
          std::string theISBN); // for books with multiple authors  
  
    Book (std::string theTitle, const Publisher* thePubl,  
          const Author& theAuthor,  
          std::string theISBN); // for books with single authors  
  
    Book(const Book&);  
    ~Book();  
    const Book& operator= (const Book&);  
  
    std::string getTitle() const;  
    void setTitle(std::string theTitle);  
  
    int getNumberOfAuthors() const;  
    Author& getAuthor (int authorNumber);  
    const Author& getAuthor (int authorNumber) const;  
  
    void addAuthor (const Author&);  
    void removeAuthor (const Author&);  
  
    Publisher& getPublisher() const;  
    void setPublisher(const Publisher& publ);
```

```

    std::string getISBN() const;
    void setISBN(std::string id);

private:
    :
};

```

What do I see as awkward about this?

## 1.1 Redesigning the Constructors

Well consider first the idea of trying to create a Book with several authors already in place. [This constructor](#) lets us do that by passing in an array. But if we don't already have an array, we would need to create one, e.g.,

```

Author jones = ...;
Author smith = ...;
Author doe = ...;
:
Author* tempArray[] = {jones, smith, doe};
Book b (theTitle, thePublisher, 3, tempArray, theISBN);
delete [] tempArray;

```

It's kind of ugly. Ugly enough to justify [another constructor](#) for the common special case of a book with only one author:

```
Book b2 (theTitle, thePublisher, jones, theISBN);
```

But if we are going to have a special case for one author, why not a special case for zero authors, or for two authors?

Or, while we're griping about that interface, what if we have our authors already in a `std::array`, or a `vector`, or ... any of the other sequential containers that we will be studying in the coming weeks? Seems a bit silly to unpack data from one data structure just to repack it into a temporary array so that we can pass that to our constructor:

```

std::array<Author, 3> authors = {jones, smith, doe};
:
Author* tempArray = new Author[3];
for (int i = 0; i < tempArray.size(); ++i)
    tempArray[i] = authors[i];
Book b (theTitle, thePublisher, 3, tempArray, theISBN);
delete [] tempArray;

```

What would be better? Well, for one thing you can see in the code examples above I have twice used the syntax `{ ...comma-separated list of values... }`. In early versions of C++, this syntax was limited to initializing arrays. But in modern C++, this syntax is used to provide initial values to all kinds of sequential ADTs. So what we would really like to be able to do is:

```
Book b (theTitle, thePublisher, {jones, smith, doe}, theISBN);
Book b2 (anotherTitle, thePublisher, {jones}, anotherISBN);
```

That gives us a very nice way to construct books when we know, at compile time, exactly how many authors we will need.

We'll come back shortly to the problem of initializing books when the number of authors is determined at run time. For now, however, let's look at another, related problem.

## 1.2 Providing Access to Individual Elements

Our current interface let's us access specific elements by index number:

```

int getNumberOfAuthors() const;
const Author& getAuthor (int authorNumber) const;

```

So if, for example, we wanted to print all of the authors in a book `b`, we could write

```

for (int i = 0; i < b.getNumberOfAuthors(); ++i)
    cout << b.getAuthor(i) << endl;

```

which seems fine. But...

Getting the  $i^{\text{th}}$  element of a sequence is efficient if the sequence is implemented using ordinary arrays or `std::array`. With just index into the underlying array, and go directly to the element we want. It takes no more time to access `authors[9999]` than it does to access `authors[0]`.

But if the sequence were implemented with a linked list, we can only get to the  $i^{\text{th}}$  element by starting at element 0 and then moving forward, one element at a time, through the sequence. `getAuthor(9999)` likely takes 10,000 times as long as `getAuthor(0)`.

### 1.2.1 Please Don't Bias the Implementers

For that reason, the moment most programmers see an operation like

```
Element get (int index) const;
```

they immediately assume that they will need to use arrays or array-like structures for the implementation. Anything else is automatically too slow.

But there are many circumstances where a non-array structure would be preferred for other operations. So, can we replace that get-by-integer-position operation with something a bit more neutral, something that would not immediately force the programmer into a array-style solution? What we **want** is a more general notion of a position-within-the-container:

```
Author getAuthor (Position authorNumber) const;
```

## 1.3 A Related Problem – search functions.

Suppose that we wanted to add a new capability to our book – an ability to check to see if a particular author is in the author list and, if so, where in the list that author can be found.

Something like this:

```
class Book
{
    ...
    ??? find (const Author& au) const;
    ...
}
```

What data type should the `find` function return?

- It can't return an `Author`, because it's possible that the author we are looking for isn't in the book, and then there is no author value we can reasonably return.
- We could return the integer index at which the author is found, which would let us return -1 or some other special value to indicate when the author was not found.

But we just finished arguing that using integer positions tends to bias the implementer into array-like structures, and we don't want to do that.

- What we **want** to return is, once again, a more general notion of a position-within-the-container:

```
Position find (const Author& au) const;
```

## 1.4 What Operations Should A Position ADT Provide?

So, what do we need our hypothetical position-within-a-collection ADT to do?

1. Given a position, we want to be able to access the data at that position in a short, fixed amount of time.
  - By "access", I mean to both examine and store/alter the data at that location.
2. Given a position, we need to be able to easily and quickly get the "next" position within the container.
  - If we do this enough times, we should eventually visit every position within the container.
3. We should be able to compare two position values to see if they denote the same position within a container.
4. For any container, we should be able to get the beginning position in that container and the ending position.
5. We should be able to quickly and easily copy a position.

If we had an ADT that supported these operations, we could imagine various ways of using it.. For example, looping through a container could look like:

```
Position p = container.getStartingPosition();
while (p != container.getEndingPosition())
{
    Data x = valueAtPosition(p);
    doSomethingWith(x);
    p = nextPositionAfter(p);
}
```

A simple sequential search through a container could look like

```
Position seqSearch (container, x)
{
    Position p = container.getStartingPosition();
    while (p != container.getEndingPosition()
        && valueAtPosition(p) != x)
    {
        p = nextPositionAfter(p);
    }
    return p;
}
```

In fact, it becomes clear that one of the common uses for such Positions is to support iterating through all or part of a container. For this reason, many past attempts to develop a useful ADT interface for positions within a container have called the resulting ADT an “iterator”.

## 2 The C++ iterator

This problem of dealing with positions and using them to iterate over collections of data is so common that a specific C++ style has evolved to deal with it. In C++, we call these position ADTs *iterators*. Iterators are deeply embedded into the C++ std:: library.

Remember that there were five things that we wanted to do with a “position” value. The table below summarizes those things and gives the C++ style operation for that purpose.

Given a container `c` and iterators `it` and `it0`, denoting positions somewhere within `c`:

access the data at that position	<code>*it, it-&gt;</code>
move <code>it</code> to the next position within <code>c</code>	<code>++it</code> or <code>it++</code>
compare two position values <code>it</code> and <code>it0</code>	<code>it == it0, it != it0</code>
get the beginning and ending positions in a container	<code>c.begin(), c.end()</code>
copy a position	<code>it0 = it</code>

One of the container types that supports this abstraction is `std::array`.

So if I have

```
std::array<int, N> arr;
:
sum = 0;
for (int i = 0; i < N; ++i)
    sum += arr[i];
```

I could rewrite that loop as

```
std::array<int, N> arr;
:
sum = 0;
for (std::array<int, N>::iterator it = arr.begin(); ①
     it != arr.end(); ++it) ②
    sum += *it; ③
```

- ① The rather lengthy type name `std::array<int, N>::iterator` refers to the [internal data type name](#) provided by nearly all `std` containers.
  - Typing that out is annoying, but we'll see later that we usually don't need to. However, when you are just starting out with iterators, it may be good to be reminded that this type exists and that this is its proper name.
- ② In the same line, you can see the use of `arr.begin()` to get the first position within `arr`.
- ③ In the next line, you see the use of `arr.end()` to get the ending position for the loop, and the use of `++` to advance our iterator to the next position.
  - Notice that we do not actually enter the body of the loop when our position is `arr.end()`. That's fine. By C++ convention, we use `end()` to denote the position just **after** the last piece of data. There is no data stored at `end()`.
- ④ In the final line, you see the use of `*` to “dereference” the iterator and get the data stored at that location.

If the data stored at a position is actually a class or struct, we can use `->` to access data or function members. For example,

```
std::array<string, N> arr;
:
sum = 0;
for (std::array<string, N>::iterator it = arr.begin(); ①
     it != arr.end(); ++it) ②
    sum += it->size(); ③
```

If the use of `*` and `->` suggest pointers to you, that's no accident. C and C++ programmers have long had a kind of second style for dealing with arrays that relied on the fact that an array is a pointer to a sequence of data, and incrementing a pointer is the same as moving it from one element of the array to the next, e.g.

```
string* arr = new string[N];
:
sum = 0;
for (string* it = arr; it != arr + N; ++it)
    sum += it->size();
```

Comparing iterators to this older style of working with arrays.

	container <code>c</code>	array <code>arr</code>
access the data at that position	<code>*it, it-&gt;</code>	<code>*it, it-&gt;</code>
move <code>it</code> to the next position within <code>c</code>	<code>++it</code> or <code>it++</code>	<code>++it</code> or <code>it++</code>
compare two position values <code>it</code> and <code>it0</code>	<code>it == it0, it != it0</code>	<code>it == it0, it != it0</code>
get the beginning position	<code>c.begin()</code>	<code>arr</code>

	<b>container c</b>	<b>array arr</b>
get the ending position	c.end()	arr+N
copy a position	it0 = it	it0 = it

You can see that, other than the problem of finding the beginning and ending positions within a container, the notation is the same.

## 2.1 The iterator as an ADT Interface

There is no single data type in C++ for iterators. Instead, “iterator” is a pattern that we adopt. Each different type of std container will provide its own class that implements this pattern. And, outside of std::, most C++ programmers who face the problem of providing access to a sequence of data values (e.g., the authors for a book) will provide something that implements the same iterator pattern.

If, however, we were to try to capture this pattern in the form of a class, it would probably look something like this.

```
template <typename T>
class MyIterator
{
public:
    typedef std::forward_iterator_tag iterator_category; ①
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;

    iterator();

    // Get the data element at this position
    reference operator*() const;                      ②
    pointer operator->() const;

    // Move position forward 1 place
    iterator& operator++();                           ③
    iterator operator++(int);

    // Comparison operators
    bool operator==(const iterator&) const;          ④
    bool operator!=(const iterator&) const;
private:
    :
};
```

- ① Like many std:: types, there is a collection of data type names that are associated with most iterators. These exist primarily to aid authors of templates that make use of iterators.

The data type std::forward\_iterator\_tag is used to indicate which of several varieties of iterator we want to provide. We'll discuss these variations [later](#).

- ② Here you have the declarations for the \* and -> operators.
- ③ Here you have the declarations for the ++ operators. The first of these provides the prefix form of ++, the other [provides the postfix form](#).
- ④ Here are the declarations for the relation operators for comparing two iterators.
- As a general rule, iterators are supposed to be small and permit fast, shallow copies. So we usually do not need explicit forms of the Big 3. The compiler-generated defaults are usually OK.
- Not provided in this interface are the begin() and end() functions. These are generally not functions of the iterator class but of the container class that the iterator works on.

Although there is no single iterator class, there is actually an iterator template that provides a little bit of aid in writing our own iterator classes. What it does is to provide a mean of writing out that block of internal data type names (① above).

```
#include <iterator>

template <typename T>
class MyIterator: public iterator<std::forward_iterator_tag, T>
{
public:
    iterator();

    // Get the data element at this position
    reference operator*() const;
    pointer operator->() const;

    // Move position forward 1 place
    iterator& operator++();
    iterator operator++(int);

    // Comparison operators
    bool operator==(const iterator&) const;
    bool operator!=(const iterator&) const;
private:
    :
};
```

## 2.2 Names for iterator types

Most containers will actually provide two different iterator types:

Suppose we have a class C that will provide iterators.

- Iterators for use with objects of type C or C& are called C::iterator.
- Iterators for use with objects of type const C or const C& are called C::const\_iterator.

- The C::const\_iterator provides the ability to examine the data at a position, but not to change that data.
- The C::iterator usually provides the ability to both examine and alter the data at a position.
  - Some containers cannot allow you to directly change the data at an internal position because doing so would break the rules by which the container inserts and locates its data. Those containers are the ones that will provide ::iterator types that can examine, but not change, the data at position.

We now have all the tools we need to show some of the more basic patterns of working with iterators. We already know three containers that supply iterators: ordinary arrays, std::array and string. Ordinary arrays and std::arrays can be containers of almost anything we like, but strings are containers of chars.

First, suppose that we want to write a function that would add up all of the values in a [std::vector](#) of doubles:

```
double sum (std::vector<double>& v) {  
    double result = 0.0;  
    for (std::vector<double>::iterator it = v.begin(); it != v.end(); ++it)  
        result += *it;  
    return result;  
}
```

or

```
double sum (std::vector<double>& v) {  
    double result = 0.0;  
    std::vector<double>::iterator it = v.begin();  
    while (it != v.end())  
    {  
        result += *it;  
        ++it;  
    }  
    return result;  
}
```

Now, those loops do exactly the same thing as these next ones:

```
double sum (std::vector<double>& v) {  
    double result = 0.0;  
    for (int i = 0; i < v.size(); ++i)  
        result += v[i];  
    return result;  
}
```

or

```
double sum (std::vector<double>& v) {  
    double result = 0.0;  
    int i = 0;  
    while (i < v.size())  
    {  
        result += v[i];  
        ++i;  
    }  
    return result;  
}
```

So why should we bother with the iterator forms? Well, for arrays, std::arrays, and std::vectors, I might would not use the iterator forms, because the loops based on an integer counter and the [ ] operator may be simpler and easier to read.

But shortly we will begin working with container types that don't provide a nice convenient [ ] operator. The iterators will be our only choice then. Also, because all of the std containers support iterators, we can write some algorithms using iterators that will work on many different kinds of containers.

Another reason to use the iterator style is we can generalize this function to work on regular arrays, std::arrays, std::vectors, and, in fact, to work on almost any container at all.

First, though, suppose that we wanted to slightly generalize this function

```
template <size_t N>  
double sum (std::vector<double>& v)  
{  
    double result = 0.0;  
    for (int i = 0; i < v.size(); ++i)  
        result += v[i];
```

```

    return result;
}

```

so that, instead of adding up every number in the array, it only added up the numbers in some range of positions. We could do that by providing a starting and ending position.

```

template <size_t N>
double sum (std::vector<double>& v, int start, int stop)
{
    double result = 0.0;
    for (int i = start; i < stop; ++i)
        result += v[i];
    return result;
}

```

I've followed the usual C++ convention of interpreting the ending position as being the position just after the last data element that we will actually use. So if we wanted to add up all of the elements in a vector, we would say

```
double theSum = sum(myVector, 0, myVector.size());
```

but if we wanted to get separate sums for the first and second halves of the vector, we could so:

```
double firstHalfSum = sum(myVector, 0, myVector.size()/2);
double secondHalfSum = sum(myVector, myVector.size()/2, myVector.size());
```

Now, let's consider replacing the start and stop integer positions with iterators:

```

template <size_t N, typename Iterator>
double sum (std::array<double,N>& v, Iterator start, Iterator stop)
{
    double result = 0.0;
    for (Iterator i = start; i < stop; ++i)
        result += *i;
    return result;
}

```

Now, an interesting thing has happened. The container `v` itself is no longer used anywhere inside the body of the function. We don't need it, and can drop it from the parameter list and then drop the size `N` from the template parameter list:

```

template <typename Iterator>
double sum (Iterator start, Iterator stop)
{
    double result = 0.0;
    for (Iterator i = start; i < stop; ++i)
        result += *i;
    return result;
}

```

Now, if we wanted to add up all of elements of a `std::array`, we could do

```
std::array<double, 100> myArray;
⋮
double theSum = sum(myArray.begin(), myArray.end());
```

But we could also use this function with a vector:

```
std::vector<double> myVector;
⋮
double theSum = sum(myVector.begin(), myVector.end());
```

And if we wanted to get separate sums for the first and second halves of a conventional array, we could so:

```
double* arr = new double[N];
⋮
double firstHalfSum = sum(arr, arr+N/2);
double secondHalfSum = sum(arr+N/2, arr+N);
```

The `sum` function template really doesn't care what kind of container the data is in, so long as that container provides working iterators.

Designing functions to work on a range of data positions by passing a pair of iterators rather than the container from which those iterators are obtained is very common in C++. It allows us to write function templates that can work on many different `std::` containers and on ordinary arrays.

## 2.3 iterator vs. const\_iterator

Now let's look a little bit more closely at those example functions.

We probably would not have actually declared them as

```
double sum (std::vector<double>& v);
```

A problem with this is that it suggests that `v` is both an input and an output parameter. But we really don't expect the operation of computing a sum to change the values in the array. To help catch buggy code that might inadvertently make such changes, the principle of const-correctness says that we should pass that array by copy (which would be potentially quite expensive) or as a const reference:

```
double sum (const std::vector<double>& v);
```

But then our code

```
double sum (const std::vector<double>& v) {
    double result = 0.0;
    for (std::array<double,N>::iterator it = v.begin(); it != v.end(); ++it)
        result += *it;
    return result;
}
```

would not compile. We would get compilation errors near the calls `v.begin()` and `v.end()` because those functions would actually return `const_iterator`s, not `iterators`. So we would be trying to assign a `const_iterator` (`v.begin()`) to an iterator variable `it` and trying to compare an iterator `it` to a `const_iterator` (`v.end()`). Neither of those operations is going to be legal.

The fix is to replace our mentions of "iterator" by "const\_iterator":

```
double sum (const std::vector<double>& v) {
    double result = 0.0;
    for (std::vector<double>::const_iterator it = v.begin(); it != v.end(); ++it)
        result += *it;
    return result;
}
```

## 2.4 auto

Mixing up iterators and const iterators has turned out to be a very common mistake for many programmers. In addition, many programmers find names like `std::array<double,N>::const_iterator` to be annoyingly long. They are easy to mis-type and can reduce the readability of the code.

C++11 introduced a feature that can help here. When you are declaring variables or parameters, you don't have to write out the data type if the declaration provides enough information for the compiler to deduce the type. Instead, you simply use the keyword "auto". For example, you could declare an integer variable like this

```
auto k = 12;
```

because the compiler can tell from the initial value being assigned what the data type of `k` would be.

You *cannot*, however, do the same here:

```
auto m;
cin >> m;
int k = m + 1;
```

because there are no clues within the declaration of `m` as to what it should be.

Now, when we call a container's `begin()` or `end()` function, the compiler can look at the container type, determine whether it is a const container or not, and determine what data type would be returned by the `begin()` call. So we can rewrite our last example as

```
double sum (const std::vector<double>& v) {
    double result = 0.0;
    for (auto it = v.begin(); it != v.end(); ++it)
        result += *it;
    return result;
}
```

which is definitely an improvement.

`auto` can, when used properly, reduce errors and enhance the readability of your code. It can be a bit of a trap for beginning programmers, however, by hiding the "real" type of a variable.

- When you are writing code and you need to do something with a variable or the result of some expression (or function call), the list of things that you **can** do to that variable depends entirely on the data type of that variable or expression.

If you don't **know** that data type, you don't know what code you can write to work with the value.

- Similarly, when reading someone else's code, one of the most important things you want to keep track of is that the data types are for the variables and expressions used in that code.

So if you, as a programmer, use `auto` because you don't want to type out a long awkward type name, that's fine. But if you use it as a crutch because you don't know what the data type really is, you're not helping yourself because you won't know what you can actually *do* with that variable or expression.

# 3 Example: Adding Iterators to the Book Interface

We can think of our Book class as a container of Authors.

Earlier, we dealt with the problem providing access to individual authors by using integer indices:

```
class Book {
public:
:
int getNumberOfAuthors() const;
Author getAuthor (int authorNumber) const;
:
```

Now, there's nothing really wrong with `getNumberOfAuthors`. It's a perfectly useful function in its own right. But, as we have noted earlier, the get-author-by-integer-index operation biases the possible implementations of Book in a way that we don't like. We will relieve this bias by replacing `getAuthor` with an iterator:

```
class Book {
public:
:
typedef ... iterator;
typedef ... const_iterator;
:
int getNumberOfAuthors() const;

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
:
```

Now, we have to choose or create a suitable data structure to use for the iterators, and implement the `begin` and `end` functions using that data type.

As always, when faced with the need to choose a data structure, we should consider options such as

1. Reuse a data type or ADT that provides the behavior we want.
2. Adapt an ADT that almost provides the behavior we want via inheritance.
3. Create an appropriate ADT "from scratch".

## 3.1 Creating an Iterator Class

As it happens, we won't need to create our `Book::iterator` from scratch. But if we did, we would start by introducing class names for that purpose:

```
#include "authorIterator.h"

class Book {
public:
:
typedef AuthorIterator iterator;
typedef AuthorConstIterator const_iterator;
:
int getNumberOfAuthors() const;

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
:
private:
:
};
```

Then, to actually declare the new iterator classes, we would fall back to the pattern for iterator classes discussed [earlier](#):

```
class AuthorIterator
{
public:
    typedef std::forward_iterator_tag iterator_category;
    typedef Author          value_type;
    typedef ptrdiff_t       difference_type;
    typedef value_type*     pointer;
    typedef value_type&    reference;

    AuthorIterator();

    // Get the data element at this position
    reference operator*() const;
    pointer operator->() const;

    // Move position forward 1 place
    AuthorIterator& operator++();
    AuthorIterator& operator++(int);

    // Comparison operators
```

```

    bool operator==(const AuthorIterator&) const;
    bool operator!=(const AuthorIterator&) const;
private:
    :
};

class AuthorConstIterator
{
public:
    typedef std::forward_iterator_tag iterator_category;
    typedef Author value_type;
    typedef ptrdiff_t difference_type;
    typedef const value_type* pointer;
    typedef const value_type& reference;

    AuthorConstIterator();

    // Get the data element at this position
    reference operator*() const;
    pointer operator->() const;

    // Move position forward 1 place
    AuthorConstIterator& operator++();
    AuthorConstIterator operator++(int);

    // Comparison operators
    bool operator==(const AuthorConstIterator&) const;
    bool operator!=(const AuthorConstIterator&) const;
private:
    :
};

```

The const and non-const iterator declarations are identical except for the highlighted items.

The implementation of these classes depends on the underlying data structure used to hold the authors within a book. For example, if we were using a dynamic array:

```

class Book {
public:
    :
private:
    std::string title;
    Publisher* publisher;
    int numAuthors;
    static const int maxAuthors = 12;
    Author* authors; // array of authors
    std::string isbn;
};

```

then we could use pointers to individual array elements as our underlying data structure for the iterator:

```

class AuthorIterator
{
public:
    :
private:
    Author* position;
    friend class Book;
};

```

The “friend” declaration means that the Book class will have access to the private members of the AuthorIterator class.

Then in Book, we implement begin() and end():

```

Book::iterator Book::begin()
{
    AuthorIterator b;
    b.position = authors;
    return b;
}

Book::iterator Book::end()
{
    AuthorIterator b;
    b.position = authors+numAuthors;
    return b;
}

```

and the operations for AuthorIterator are pretty simple:

```

AuthorIterator::reference AuthorIterator::operator*() const
{
    return *position;
}

AuthorIterator::pointer AuthorIterator::operator->() const
{
    return position;
}

```

```

// Move position forward 1 place
AuthorIterator& AuthorIterator::operator++()
{
    ++position;
    return *this;
}
AuthorIterator AuthorIterator::operator++(int)
{
    AuthorIterator saved = *this;
    position++;
    return saved;
}

// Comparison operators
bool AuthorIterator::operator==(const AuthorIterator& p) const
{
    return position == p.position;
}

bool AuthorIterator::operator!=(const AuthorIterator& p) const
{
    return position != p.position;
}

```

## 3.2 Reusing Existing Data Types as Iterators

Often, however, once we have decided on the underlying data structure, we will find that it already provides a data type that supports the \*, ->, \*, and ++ operations with behaviors identical to what we need for an iterator. In that case, we would not need to create a new class.

For example, if we're using a dynamic array to hold our authors, we could use simple pointers to array elements as our iterators

```

class Book {
public:
    typedef Author* iterator;
    typedef const Author* const_iterator;
    ...
private:
    std::string title;
    Publisher* publisher;
    int numAuthors;
    static const int maxAuthors = 12;
    Author* authors; // array of authors
    std::string isbn;
};

```

and then merely need to implement the Book's begin and end operations:

```

Book::iterator Book::begin()
{
    return authors;
}

Book::iterator Book::end()
{
    return authors+numAuthors;
}

```

On the other hand, if we used a `std::array` to hold our authors, we could use its iterators as our Book's iterators.

```

class Book {
public:
    typedef std::array<Author,maxAuthors>::iterator iterator;
    typedef std::array<Author,maxAuthors>::const_iterator const_iterator;
    ...
private:
    std::string title;
    Publisher* publisher;
    int numAuthors;
    static const int maxAuthors = 12;
    std::array<Author,maxAuthors> authors
    std::string isbn;
};

Book::iterator Book::begin()
{
    return authors.begin();
}

Book::iterator Book::end()
{
    return authors.end();
}

```

## 4 Example: Cleaning Up the Book Constructors

Another problem with our older interface for `Book` was the awkwardness of initializing a book with various number of authors.

Our old version had been:

```
Book (std::string theTitle, const Publisher* thePubl,
      int numAuthors, Author* theAuthors,
      std::string theISBN);

Book (std::string theTitle, const Publisher* thePubl,
      const Author& theAuthor,
      std::string theISBN);
```

but we had [suggested](#) that instead of creating temporary arrays every time we want to create a new `Book`:

```
Author textAuthors[] = {budd};
Book text361 ("Data Structures in C++", macmillan, 1,
              textAuthors, "0-201-10758");
Book ootext ("Introduction to Object Oriented Programming", macmillan,
             1, textAuthors, "0-201-12967");

Author recipeAuthors[] = {doe, smith};
Book recipes ("Cooking with Gas", 2, recipeAuthors, "0-124-46821");
```

we would like to be able to say things like

```
Book text361 ("Data Structures in C++", macmillan,
               {budd}, "0-201-10758");
Book ootext ("Introduction to Object Oriented Programming", macmillan,
             {budd}, "0-201-12967");

Book recipes ("Cooking with Gas", {doe, smith}, "0-124-46821");
```

in cases where the number of authors is known at compile-time, and would like to be able to use nearly any container of authors in cases where the authors are determined at run-time.

### 4.1 initializer\_list

What is the data type of an expression like `{budd}` or `{doe, smith}`?

That is an `initializer_list` (not to be confused with the [initialization lists](#)) that is part of the constructor syntax.

`initializer_list` is a container provided in the header `<initializer_list>`. It's rather like a limited form of the `std::array` in that it is template providing a simple sequence of values.

In fact, the only operations provided by `initializer_list` are

- `size()`
- `begin()`, and
- `end()`.

So we can add an ability to use initializer lists to `Book` with a constructor

```
Book (std::string theTitle, const Publisher* publ,
      std::initializer_list<Author> theAuthors,
      std::string theISBN);
```

which would be implemented as

```
Book:: Book (std::string theTitle, const Publisher* publ,
            std::initializer_list<Author> theAuthors,
            std::string theISBN)
: title(theTitle), publisher(publ),
  numAuthors(0), authors(new Author*[maxAuthors]),
  isbn(theISBN)
{
    for (auto it = theAuthors.begin(); it != theAuthors.end(); ++it)
    {
        authors[numAuthors] = *it;
        ++numAuthors;
    }
};
```

Note the use of the `size()` function to [initialize numAuthors](#), and the [iterator-style loop](#) to copy the authors into the `Book`.

### 4.2 Constructors with Start-Stop Ranges

For cases where the number of authors is not known at compile time, we can use a common pattern in C++ programming:

Pass ranges of data to function templates as a pair of iterators, denoting a starting and ending position.

This allows us to pass a whole range of values that could be stored in any container type that provides iterators (and, by modern C++ convention, most of them, even containers that are not part of the standard library, do provide iterators).

We can use this pattern to allow a Book to be constructed from a range of sequential positions from almost any container:

```
template <typename Iterator>
Book (std::string theTitle, const Publisher* publ,
      Iterator startAuthors, Iterator stopAuthors,
      std::string theISBN);
```

implemented as

```
template <typename Iterator>
Book::Book (std::string theTitle, const Publisher* publ,
           Iterator startAuthors, Iterator stopAuthors,
           std::string theISBN)
: title(theTitle), publisher(publ),
  numAuthors(0), authors(new Author*[maxAuthors]),
  isbn(theISBN)
{
    for (auto it = startAuthors; it != stopAuthors; ++it)
    {
        authors[numAuthors] = *it;
        ++numAuthors;
    }
};
```

This would allow us to write code like:

```
void readBook (std::istream& in)
{
    string title, isbn;
    Publisher pub;
    cin >> title >> isbn >> pub;
    int numAuthors;
    cin >> numAuthors;
    Author* temp = new Author[numAuthors];
    for (int i = 0; i < numAuthors; ++i)
        cin >> temp[i];
    Book b (title, pub, temp, temp+numAuthors, isbn);
    delete [] temp;
    return b;
}
```

For comparison and for the sake of completeness,, here is the std::array version of our Book class, with the same changes made to the constructors.

[abook.h](#) +

```
#ifndef Book_H
#define Book_H

#include <initializer_list>
#include <string>
#include <array>
#include "author.h"

class Publisher;

class Book {
private:
    std::string title;
    const Publisher* publisher;
    int numAuthors;
    std::string isbn;
    static const int MaxAuthors = 12;
    std::array<Author,MaxAuthors> authors;

public:
    typedef std::array<Author,MaxAuthors>::iterator iterator;
    typedef std::array<Author,MaxAuthors>::const_iterator const_iterator;

    Book();
    Book (std::string theTitle, const Publisher* publ,
          std::initializer_list<Author> theAuthors,
          std::string theISBN);

    template <typename Iterator>
    Book (std::string theTitle, const Publisher* publ,
          Iterator startAuthors, Iterator stopAuthors,
          std::string theISBN);
```

```

        std::string getTitle() const {return title;}
        void setTitle(std::string theTitle) {title = theTitle;}

        int getNumberOfAuthors() const;

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

        void addAuthor (Author);
        void removeAuthor (Author);

        const Publisher* getPublisher() const {return publisher;}
        void setPublisher(const Publisher* publ) {publisher = publ; }

        std::string getISBN() const {return isbn;}
        void setISBN(std::string id) {isbn = id; }

};

template <typename Iterator>
Book::Book (std::string theTitle, const Publisher* publ,
           Iterator startAuthors, Iterator stopAuthors,
           std::string theISBN)
: title(theTitle), publisher(publ), numAuthors(0),
  isbn(theISBN)
{
    while (startAuthors != stopAuthors)
    {
        authors[numAuthors] = *startAuthors;
        ++numAuthors;
        ++startAuthors;
    }
}

#endif

```

### [abook.cpp](#) +

```

/*
 * book.cpp
 *
 * Created on: May 23, 2018
 * Author: zeil
 */

#include "abook.h"
#include <cassert>
#include <algorithm>

using namespace std;

Book::Book()
: title(), publisher(nullptr), numAuthors(0),
  isbn()
{
}

Book::Book (std::string theTitle, const Publisher* publ,
           std::initializer_list<Author> theAuthors,
           std::string theISBN)
: title(theTitle), publisher(publ), numAuthors(theAuthors.size()),
  isbn(theISBN)
{
    int i = 0;
    for (const Author& au: theAuthors)
    {
        authors[i] = au;
        ++i;
    }
}

int Book::getNumberOfAuthors() const
{
    return numAuthors;
}

Book::iterator Book::begin()
{
    return authors.begin();
}
Book::const_iterator Book::begin() const
{
    return authors.begin();
}
```

```

}
Book::iterator Book::end()
{
    return authors.begin() + numAuthors;
}
Book::const_iterator Book::end() const
{
    return authors.begin() + numAuthors;
}

void Book::addAuthor (Author au)
{
    assert(numAuthors < MaxAuthors);
    authors[numAuthors] = au;
    ++numAuthors;
}
void Book::removeAuthor (Author au)
{
    auto pos = find(begin(), end(), au);
    if (pos != end())
    {
        copy (pos+1, end(), pos);
        --numAuthors;
    }
}

```

## 5 Iterator Variations

There are a number of common variations on the basic iterator interface.

### 5.1 Forward Iterators

- The interface we have developed so far is called a [forward iterator](#)
- we can move forward through a container, but never backwards.

[forwardIterator.h](#) +

```

#include <cstddef>
class MyIterator
{
public:
    typedef std::forward_iterator_tag iterator_category;
    typedef T                      value_type;
    typedef ptrdiff_t               difference_type;
    typedef T*                     pointer;
    typedef T&                    reference;
    ...

    // Get the data element at this position
    reference operator*() const;
    pointer operator->() const;

    // Move position forward 1 place
    MyIterator& operator++();
    MyIterator operator++(int);

    // Comparison operators
    bool operator== (const iterator&) const;
    bool operator!= (const iterator&) const;
private:
    ...
};

```

### 5.2 Bidirectional Iterators

An obvious extension is to allow iterator applications to move backwards as well. The operator `--` provides this capability.

[biIterator.h](#) +

```

class MyIterator
{
public:
    typedef std::bidirectional_iterator_tag iterator_category;
    typedef T                      value_type;
    typedef ptrdiff_t               difference_type;
    typedef T*                     pointer;
    typedef T&                    reference;
    ...

    // Get the data element at this position
    reference operator*() const;

```

```

pointer operator->() const;

// Move position forward 1 place
iterator& operator++();
iterator& operator++(int);

// Move position backward 1 place
iterator& operator--();
iterator& operator--(int);

// Comparison operators
bool operator==(const iterator&) const;
bool operator!=(const iterator&) const;
private:
:
};

```

These are called [bidirectional iterators](#).

The `std::list` provides bi-directional iterators.

## 5.3 Random Access Iterators

The `++` and `--` operators allow us to move only one position at a time. Some containers provide “random access” iterators that can move any integer number of places forward or back.

[randomIterator.h](#)

```

class MyIterator
{
public:
    typedef std::random_access_iterator_tag iterator_category;
    typedef T                               value_type;
    typedef ptrdiff_t                      difference_type;
    typedef T*                            pointer;
    typedef T&                           reference;
    :

    // Get the data element at this position
    reference operator*() const;
    pointer operator->() const;

    // Move position forward 1 place
    iterator& operator++();
    iterator& operator++(int);

    // Move position backward 1 place
    iterator& operator--();
    iterator& operator--(int);

    // Random access operations
    ptrdiff_t operator- (const iterator&) const;
        // how many positions apart are these iterators?

    iterator operator+ (ptrdiff_t k) const;
        // Get a new iterator k positions past this one.

    iterator operator- (ptrdiff_t k) const;
        // Get a new iterator k positions before this one.

    // Comparison operators
    bool operator==(const iterator&) const;
    bool operator!=(const iterator&) const;
    bool operator< (const iterator&) const;
    bool operator<= (const iterator&) const;
    bool operator> (const iterator&) const;
    bool operator>= (const iterator&) const;
private:
    :
};

```

The `std::array`, `std::initializer_list`, and `std::vector` classes provide random-access iterators. Array pointers are considered to be random-access iterators as well.

### 5.3.1 Random Access Iterator Ops

With a random access iterator, `it`, we can get a position 5 places past `it` by computing `it+5`. We can get a position 3 places in front of `it` by the expression `it-3` or `it + (-3)`.

- For example, if I had a function `find` what took a pair of iterators denoting a range of positions to search, and an array `arr` containing `N` elements, I could search the entire array like this:

```
auto position = find (arr+0, arr+N, ...)
```

because array pointers function as random-access iterators. Of course, there's not much point to adding zero to `arr`, other than to draw attention to the fact that `arr+0` denotes the same address/position as `arr[0]`, so I might just as well write

```
auto position = find (arr, arr+N, ...)
```

Subtracting one random-access iterator from another yields the number of positions apart that they are: `(it+5) - it == 5`.

- For example, if I used the `find` function from the previous example

```
auto position = find (arr, arr+N, ...)
```

I could get at the item I had located via the iterator `position` as `*position`. But if, for some reason, I wanted to know what the index of that element was within the array, I could compute it as

```
int index = position - arr;
```

Random access iterators (and *only* random access iterators) allow easy conversion between iterator-style and index-style access.

Random access iterators are needed when an algorithm needs to “jump around” in a container rather than plod methodically from one end of the container to the other. For example, the iterator-based rewrite of the binary search algorithm that would compute the next place to look this way:

```
int low, high;  
:  
int mid = ( low + high ) / 2;  
RandomAccessIterator midPos = start + mid;
```

Random-access iterators also support the relational operators `<`, `<=`, `>`, and `>=` for determining whether one position comes before or after another within the container.

## 5.4 Input Iterators

Input iterators are like forward iterators except that

- You may only look at, not assign to an input iterator (in other words, it is a `const` iterator)
- An input iterator is only good for one pass over its container. With a true forward iterator, you can use it over and over, by setting it back to `begin()` each time.

Input iterators are most commonly used to walk through the contents of an input stream, e.g., `cin`:

```
istream_iterator<string> in (cin);  
while (in != istream_iterator<string>()) {  
    cout << "Next string in input is " << *in << endl;  
    ++in;  
}
```

The restrictions of an input iterator are appropriate here because one can read from an input stream but cannot store data in it.

## 5.5 Output Iterators

Similarly, we have output iterators, which are like forward iterators except that

- You may only assign to, not look at an output iterator.
- An output iterator is only good for one pass over its container.

Output iterators are most commonly used to insert data into successive positions of an output stream, e.g., `cout`:

```
ostream_iterator<int> out (cout, "\n");  
for (int i = 0; i < 100; i *= 2) {  
    *out = i; // writes an int to cout followed by newline  
    ++out;  
}
```

The restrictions of an output iterator are appropriate here because one can write into an output stream but cannot read data from it.

## 5.6 Reverse Iterators

Some containers (including vectors and lists) provide reverse iterators, which behave like the container's normal iterators except that

- Instead of using `begin()` to find the position of the *first* element in the container, we use `rbegin()` to get the position of the *last* element in the container.
- Instead of using `end()` to find the position *just after the last* element in the container, we use `rend()` to get the position *just before the first* element in the container.
- The `++` operator moves us one step *backwards* in the container, while the `--` operator moves us one step *forward* in the container.

For example:

```
vector<int> v {1, 2, 3, 4};
for (vector<int>::iterator i = v.begin();
     i != v.end(); ++i)
{
    cout << *i << ' ';
}
cout << endl;
for (vector<int>::reverse_iterator i = v.rbegin();
     i != v.rend(); ++i)
{
    cout << *i << ' ';
}
cout << endl;
```

would print

```
1 2 3 4
4 3 2 1
```

Some functions, such as `vector::insert`, will accept an iterator as a parameter but will not accept a reverse iterator. Luckily, a reverse iterator can be converted to a forward iterator via `base()`. However, `base()` returns the position just after the position denoted by the reverse iterator.

```
vector<int>::reverse_iterator rit1
    = myVector.rbegin(); // points to last element in myVector
vector<int>::iterator it1 = rit1.base(); // == myVector.end()

vector<int>::reverse_iterator rit2
    = myVector.rend(); // points just before first element in myVector
vector<int>::iterator it2 = rit2.base(); // == myVector.begin()
```

Technically, reverse iterators are not categories of iterators in the same sense as forward, bidirectional, etc. A reverse iterator itself can be forward, bidirectional, or random access. The “reverse” simply refers to which end of the container it starts from and the direction in which `++` moves it.

## 6 Example: Searching via Iterator Variants.

In this example, let's develop an efficient utility function for searching through data that is being kept in ascending order. We would like create a function template `search`:

```
/**
 * Search through a range of data in positions [start..stop) (i.e.,
 * positions beginning at start and going up to, but not including, stop)
 * looking for the value key. All data in the array being searched
 * should be in non-descending order (i.e., for any two adjacent values x
 * and y, x <= y).
 *
 * Return the position containing key or, if key is not in the range, the
 * position at which key should be inserted in order to maintain the
 * non-descending order of the data.
 *
 * @param start position of the first data element to be considered.
 * @param stop position just after the last data element to be considered.
 * @param key the value we are searching for
 * @return The position containing key of, if key is not in the range,
 *         the position at which key could be inserted to preserve the
 *         data ordering.
 */
template <typename Iterator, typename Value>
Iterator lower_bound (Iterator start, Iterator stop, const Value& key)
```

For example, if we had

```
std::array<string, 5> names = {"Adams", "Baker", "Clarke", "Davis", "Evans"};
int arr[] = {0, 2, 4, 8};
```

then

```
lower_bound (names.begin(), names.end(), "Baker");
```

would return the position of the second string, and

```
lower_bound (arr, arr+4, 3)
```

would return the position in `arr` currently occupied by the number 4.

```
lower_bound (names.begin(), names.end(), "Zeil");
```

would return `names.end()`, and

```
lower_bound (arr, arr+4, 10)
```

would return the position arr+4.

## 6.1 Ordered vs Binary Search

A simple way to search data that is ordered is to start at the beginning of the sequence, and move forward one step at a time until we either find the data we are looking for or we encounter a value larger than what we are looking for. We can call this an ordered search. For example,

```
template <typename Comparable>
int seqOrderedSearch(const Comparable list[], int listLength,
                     Comparable key)
{
    int loc = 0;

    while (loc < listLength && list[loc] < key)
    {
        ++loc;
    }
    return loc;
}
```

This code finds the integer position of key within an array or the position of the first item larger than key. We can rewrite that into iterator style, so that it can work on any container, like this:

```
template <typename Iterator, typename Value>
Iterator orderedSearch (Iterator start, Iterator stop, const Value& key)
{
    while ((start != stop) && (*start < key))
        ++start;
    return start;
}
```

You can see that this is the same underlying algorithm, just using the start...stop range convention instead of passing the array and using \*start in place of indexing into the array.

Although fairly simple, this function can be slow when applied to large amounts of data. For example, if we applied this to an array of 10000 items, on average it would look though 5000 of them before stopping. If we applied this to an array of 100000 items, on average it would look though 50000 of them before stopping.

For larger amounts of data, a binary search works better. The binary search works by considering a range [low...high] of possible positions where the key might be located. At each step we look at the value in the middle of that range. If that middle value is the key, we are done. If the key is smaller than the middle value, then we know the key can only be found in the lower half of the [low...high] range. If the key is larger than the middle value, then we know the key can only be found in the upper half of the [low...high] range. Either way, we can immediately cut the range we are considering in half.

```
/** 
 * Performs the standard binary search using two comparisons per level.
 * Returns index where item is found or the index where it could
 * be inserted if not found
 *
 * From Weiss, Data Structures and Algorithm Analysis, 4e
 * (modified SJ Zeil)
 */
template <typename Comparable>
int binarySearch( const Comparable* a, int size, const Comparable & x )
{
    int low = 0, high = size - 1;

    while( low <= high )
    {
        int mid = ( low + high ) / 2;

        if( a[ mid ] < x )
            low = mid + 1;
        else if( a[ mid ] > x )
            high = mid - 1;
        else
            return mid; // Found
    }
    return low;
}
```

Suppose, for example, that we apply this function to an array of 10000 items. \* On the first pass through the loop, low = 0 and high = 9999. We are looking at a total of 10000 possible positions.

- On the second pass through the loop, either low = 0 and high = 4998, or low = 5000 and high=9999. Either way we are looking at a total of 4999 possible positions.
- On the third pass through the loop, we would be looking at a total of 2499 possible positions.

And so on:

Pass #	Possible positions
1	10000

Pass #	Possible positions
2	4999
3	2499
4	1249
5	624
6	312
7	156
8	78
9	39
10	19
11	9
12	4
13	2
14	1

We might, of course, get lucky and find the key earlier, but with this algorithm we can search through 10000 elements while examining at most 14 of them. That's clearly much faster than the simpler ordered search.

- [Run this algorithm](#) until you are comfortable with your understanding of how it works.

We can rewrite this to work with iterators:

```
template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0;
    auto high = stop - start - 1;                                ①
    while( low <= high )
    {
        auto mid = ( low + high ) / 2;
        RandomAccessIterator midPos = start + mid;    ②
        if( *midPos < key )
            low = mid + 1;
        else if( key < *midPos )
            high = mid - 1;
        else
            return midPos; // Found
    }
    return start + low;                                         ③
}
```

Now, this code makes use of two special properties that are only available on random access iterators. At ①, I used the ability to find how many spaces are between two iterators by subtracting them. At ② and ③, I used the ability to add an integer to an iterator to get the position that integer number of spaces away.

So this code will only work on ranges of data that are described using random access iterators. But I'm OK with that restriction. The very nature of binary search is to jump around within the range of data being searched. We can't expect that to work well (or at all) with iterators that only allow us to move one step at a time.

## 6.2 Letting the Compiler Choose

The rest of this example begins to delve into some real C++ wizardry. Consider the remainder of this section to be optional reading.

However, this does not get us our all-purpose `lower_bound` function that we described at the start of this example. Instead, we have two different functions to choose from.

We could leave this choice up to the programmer, but, interestingly enough, it is possible to have the compiler choose the appropriate search function by looking at what type of iterator we are giving it.

The key here is in the idea of an “iterator category”. When we looked at the various data types provided by the [typical iterator ADT interface](#), one of the data types we provided was:

```
template <typename T>
class MyIterator
{
public:
    typedef std::forward_iterator_tag iterator_category;
```

We declared the “category” of `MyIterator` to be `std::forward_iterator_tag`. Now, `std::forward_iterator_tag` is a class type. It does not provide any data or function members of particular interest. It simply serves as an identification of what kind of iterator we are providing. There are several of these tag functions:

Iterator Variant	Tag	operations provided
input iterator	<code>std::input_iterator_tag</code>	examine data, <code>++</code>
output iterator	<code>std::out_iterator_tag</code>	store data, <code>++</code>

Iterator Variant	Tag	operations provided
forward iterator	<code>std::forward_iterator_tag</code>	read/store data, <code>++</code>
bidirectional iterator	<code>std::bidirectional_iterator_tag</code>	read/store data, <code>++, --</code>
random access iterator	<code>std::random_access_iterator_tag</code>	read/store data, <code>++, --, -, + int</code>

All of these tags are data types. The values of these types are pretty much useless, but the types themselves can be used just like other types. In particular, suppose that I were to require programmers who wanted to use our search function to supply a pointer to one of these tag values as a parameter. I could write different, overloaded versions of our search function associated with these different tags:

```
template <typename Iterator, typename Value>
Iterator lower_bound (Iterator start, Iterator stop, const Value& key,
                     const std::random_access_iterator_tag*)
{
    return binarySearch(start, stop, key);
}

template <typename Iterator, typename Value>
Iterator lower_bound (Iterator start, Iterator stop, const Value& key,
                     const std::input_iterator_tag*)
{
    return orderedSearch(start, stop, key);
}

template <typename Iterator, typename Value>
Iterator lower_bound (Iterator start, Iterator stop, const Value& key,
                     const std::forward_iterator_tag*)
{
    return orderedSearch(start, stop, key);
}

template <typename Iterator, typename Value>
Iterator lower_bound (Iterator start, Iterator stop, const Value& key,
                     const std::bidirectional_iterator_tag*)
{
    return orderedSearch(start, stop, key);
}
```

So, if I were to do

```
std::array<string, 5> names = {"Adams", "Baker", "Clarke", "Davis", "Evans"};
auto position = lower_bound (names.begin(), names.end(), "Baker",
                           new std::random_access_iterator_tag);
```

the compiler would choose the version of `lower_bound` that uses `binarySearch`, but if I change that call to

```
auto position = search (names.begin(), names.end(), "Baker",
                      new std::forward_iterator_tag);
```

the compiler would choose one of the search functions that uses `orderedSearch`.

OK, that use of the tag is getting ugly, but I'm not done yet. Since we don't actually *use* the tag value for anything, any old pointer will do. It doesn't even need to point at a real object:

```
std::array<string, 5> names = {"Adams", "Baker", "Clarke", "Davis", "Evans"};
auto position = lower_bound (names.begin(), names.end(), "Baker",
                           (std::random_access_iterator_tag)null_ptr);
```

The null pointer will do the job, so long as we cast it to the correct tag type.

Now comes the clever bit. We don't need to have the programmer supply the tag type at all. The compiler can do that for us as well. We just defined the 3-parameter version of `lower_bound`, the thing that we wanted all along as:

```
template <typename Iterator, typename Value>
Iterator lower_bound (Iterator start, Iterator stop, const Value& key)
{
    typedef std::iterator_traits<Iterator> traits; ①
    return lower_bound (start, stop, key,
                       (typename traits::iterator_category*)nullptr); ②
}
```

- ① The `iterator_traits` retrieves that block of internal data types associated with this `Iterator`.
  - It's defined in the `std`: library in such a way that, if presented with a pointer to a conventional array, it returns the traits for a random access iterator.
- ② Here we pull out the tag representing the category for `Iterator`, and create a null pointer of that type.

So now our programmer can write

```
auto position = lower_bound (names.begin(), names.end(), "Baker");
```

This 3-parameter version of `lower_bound` will compile into a call to one of our 4-parameter versions of `lower_bound`. The compiler will choose the one that matches the traits for whatever iterator we are working.

The net result is that the 3-parameter `lower_bound` function will choose to do binary search when it can, and fall back to ordered search when given anything other than a random access iterator.

The `lower_bound` function that we have just described, that switches between binary search and ordered search based on the iterator category, is already in the `std::` library in the `<algorithm>` header.

## 7 Range-based for Loops

Perhaps my favorite change introduced in C++11 is a simplified form of `for` loop called [range-based for loops](#). (In other programming languages, these are called [for-each](#) loops.)

These loops take advantage of the fact that all `std` containers support iterators, that most C++ programmers who design their own containers (e.g. our Book as a container of authors) will provide iterators following the same convention, and that containers/iterators look like arrays/pointers.

If you have an array or an iterator-providing container and if you want to loop through all of the elements of that array or container, you can write:

```
for (ElementType variableName: containerOrArray)
{
    :
}
```

For example, we can rewrite our earlier example:

```
double sum (const std::vector<double>& v) {
    double result = 0.0;
    for (auto it = v.begin(); it != v.end(); ++it)
        result += *it;
    return result;
}
```

as

```
double sum (const std::vector<double>& v) {
    double result = 0.0;
    for (double d: v)
        result += d;
    return result;
}
```

which is really quite nice.

It's not a universal replacement for all `for` loops, however. Sometimes we *need* the iterator because we are really interested in the position information. There's no way, for example, to use this new loop in

```
template <typename T, size_t N>
std::array<T,N>::const_iterator search (const std::array<T,N>& v, T x)
{
    for (auto it = v.begin(); it != v.end(); ++it)
    {
        if (x == *it)
            return it;
    }
    return v.end();
}
```

because if we tried to write the loop as

```
for (T y: v)
{
    if (x == y)
        return it; // oops!
}
```

we would no longer have the position `it`, which is the thing that we actually wanted to return.

# Analysis of Algorithms: Motivation

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Comparison of Spellcheck Solutions](#)
- [2 Why Not Just Time the Programs?](#)
- [2.1 Better than Timing](#)

An important theme throughout this semester will be viewing the process of developing software as an *engineering* process. Now, engineers in traditional engineering disciplines, civil engineers, electrical engineers, and the like, face *trade offs* in developing a new product, trade offs in cost, performance, and quality. One of the jobs of an engineer is to look at competing proposals for the design of a new product and to estimate ahead of time the cost, the speed, the strength, the quality, etc. of products that would result from these competing designs.

Software developers face the same kinds of choices. Early on, you may have several alternative designs and need to make a decision about which of those designs to actually pursue. It's no good waiting until the program has already been implemented, written down in code. By then you've already committed to one design and invested significant resources into it.

How do we make these kinds of choices? In this course, we'll be looking at mathematical techniques for analyzing algorithms to determine what their speed will be. It will be important that we do this both with real algorithms already written into code and with proposed algorithms that have been given a much sketchier description, probably written in "pseudocode".

Suppose that we worked for a company that was producing word processors and other text manipulation programs. They have decided to add an automatic spell-check feature to the product line. Our designers have considered the process of how to check a document for spelling errors (i.e., any words not in a "dictionary" of known words). They have proposed two different algorithms for finding the set of misspelled words within a target file.

### Spell-Checker, version 1

```
collectMisspelledWords /* inputs */ targetFile, dictionaryFile,
                      /* outputs */ misspellings)
{
    read dictionaryFile into dictionary;
    open targetFile;
    misspellings = empty;
    while more words in targetFile {
        read word w from targetFile;
        if w is not in dictionary {
            add w to misspellings;
        }
    }
    close targetFile;
}
```

In the first alternative, we read words, one at a time, from the target file. Each word that is not in the dictionary gets added to the set of misspellings.

Some of the designers, however, have objected that the first algorithm will waste time by repeatedly looking up common words like "a", "the", etc., in the dictionary.

They suggest an alternative algorithm.

### Spell-Checker, version 2

```
collectMisspelledWords /* inputs */ targetFile, dictionaryFile,
                      /* outputs */ misspellings)
{
    concordance = empty;
    open targetFile;
    while more words in targetFile {
        read word w from targetFile;
        add w to concordance;
    }
    close targetFile;

    open dictionaryFile;
    for each word w in the concordance {
        while (last word read from dictionaryFile < w) {
            read another word from the dictionaryFile;
        }
        if (w != last word read from dictionaryFile) {
```

```

        add all occurrences of w to misspellings;
    }
}
close targetFile;
}

```

This works by first collecting all words from the document to form a *concordance*, an index of all the words taken from a document together with the locations where they were found. Then each word is checked just once against the dictionary, no matter how many times that word actually occurs within the target document.

So, which of these algorithms is likely to run faster?

## 1 Comparison of Spellcheck Solutions

We can make plausible arguments in either direction:

- Searching the dictionary for “random” words in solution 1 may cause many words to be examined multiple times.
  - In solution 2, each word is checked against the dictionary at most 1 time.
- Solution 2 has the extra cost of building the concordance.
  - but the concordance is probably much smaller than the full document.

Overall, it’s not obvious which is faster. Thinking more deeply about the question, we might ask:

- Does the choice of the faster solution depend upon the relative sizes of the dictionary and the document?
- or the size of the concordance?
- or the number of misspelled words?

*In the lessons that follow, we will develop the mathematical tools for answering these kinds of questions.*

## 2 Why Not Just Time the Programs?

So, why the fuss? Why don’t we simply sit down with a stopwatch, run both programs on some test data, time them, and adopt the one that runs faster?

Of course, if we’re still at the design level, we can’t time the programs because we haven’t written them yet. But even if we actually had the code for both programs in hand, a simple timing experiment might yield different results depending on who ran it and how.

Why should there be such big difference? Well, it turns out that the kind of results we get with timing experiments like that will very considerably because of

- differences in code quality:

Different programmers typically produce codes that run at very different speeds. That’s hardly surprising. Different programmers often have different skill levels. Interestingly enough, even if the same programmer were to code both algorithms, that programmer would probably wind up producing more efficient code for the algorithm that was more familiar to him or her.

- differences in the machines on which we are running the programs:

It is obvious that if you take an algorithm and run it on, let’s say, a 33MHZ machine and then take it and run it on a 350MHZ machine , it’s going to run faster on the second machine.

What may be less obvious is that if you take two different processors that are rated at roughly the same speed and run both the algorithms on those processors, you may find that one of those algorithms runs faster on one processor and the other will run faster in other processor. Why? Because even processors that are roughly rated at the same speed will have different speed when we get down to the details. One may be faster at addition and the other at multiplication, so the actual speed that you get when timing an algorithm will depend upon how many addition instructions versus how many multiplication instructions you have.

- differences in the compiler and compiler settings that we are using:

For similar reasons, you see different possibilities when you switch compilers. It is possible, if you switch compilers , one algorithm would behave better on compiler A and other algorithm would behave better on compiler B. The reason: because different compilers have different approaches for compiling certain kinds of instruction. A simple instruction like  $a=2*b$  might be encoded by one compiler as a multiplication by 2 but by another compiler as a addition of B to itself and by still another compiler as a shift-the-bits-left operation. Consequently the same algorithms run on different compilers (or maybe using different settings on the same compiler) may wind up giving very different results when you try to compare them.

- differences in the choice of input data that we actually use for an experiment:

Finally we have problems in terms of choosing input data. The choice of input data may very well be biased (even if unintentionally so) to one algorithm or the other. Consequently different people choosing different input data are likely to come up with very different results on this experiment.

### 2.1 Better than Timing

Now how we are going to overcome these problems?

- differences in code quality:

To minimize the impact of differences in code quality, we will learn to use [\*order of magnitude analysis\*](#) to ask the question: "How quickly does running time of each algorithm increase as we increase problem size?"

The answer to this question typically is determined more by the choice of algorithm than by the incidental details in coding it up, and the analysis is mathematical rather than experimental to further focus attention on the algorithm rather than on the details of the code.

- differences in the machines on which we are running the programs:

This same analysis will help overcome problems arising from differences in machines because this analysis is going to be on the algorithm rather than on the specific code running on specific machine.

- differences in the compiler and compiler settings that we are using:

That also helps with differences in compilers that because we are not going to be dealing with the compilers' encoding.

- differences in the choice of input data that we actually use for an experiment:

The choice of input data remains a problem. Depending upon what kind of inputs we are actually choosing to analyze, we might windup with very different results.

We will deal with this in two ways:

- We can ask the question: "For all possible inputs to this program, what would be the average behavior?" (That is, which algorithm on average would run faster than the other?)
- More often, the question we will ask is: "For all possible inputs to this program, which input gives us the worst behavior?" (That is, which one will make us wait the longest before we actually complete the processing?) Then we will compare these worst case times for the two algorithms.

In order to make our choices, we tend to use worst case more often than average case analysis because

- it is often easier to do and
- because in many circumstances, it better expresses users' frustration in terms of waiting and waiting for an actual answer to come out.

# Analysis of Algorithms: Worst Case Complexity

Steven J. Zeil

Last modified: Feb 12, 2020

## Contents:

- [1 Brute Force Timing Analysis](#)
  - [1.1 Timing Analysis Example](#)
  - [1.2 Do We Need That Much Detail?](#)
- [2 Best, Worst, and Average Cases](#)
  - [2.1 Varying Times](#)
  - [2.2 Size of the Input](#)
- [3 Worst Case Complexity](#)
  - [3.1 Time Proportional To](#)
  - [3.2 Big-O](#)
  - [3.3 Big-O == Worst Case Complexity](#)
- [4 Testing a  \$O\(f\(N\)\)\$  Hypothesis](#)
- [5 Detailed Timing Example Revisited](#)
- [6 Pitfalls to Avoid](#)
  - [6.1 Loose Bounds and Tight Bounds](#)
  - [6.2 "n" is "n"othing special](#)

Our primary tool in trying to predict the speed of an algorithm is going to be an idea we call the “complexity” or more specifically the “worst case complexity” of the algorithm.

The English language word “complexity” may mean many things, and two people might not necessarily agree which of two algorithms is more complex in the generic sense. But in this lesson we are going to explore a specific definition of complexity that will, as it turns out, be directly related to the running speed of the algorithm.

I'll warn you right up front that the initial work we are going to do, both giving this definition and showing how it applies, is going to be little bit tedious. But as we go through the remainder of the lessons in this section of the course, we will come up with faster and more elegant ways to apply this idea of complexity to actual coded algorithms.

## 1 Brute Force Timing Analysis

### 1.1 Timing Analysis Example

```
1: for (i = 0; i < N; ++i) {  
2:     a[i] = 0;  
3:     for (j = 0; j < N; ++j)  
4:         a[i] = a[i] + i*j;  
5: }
```

Let's look at how we might analyze a simple algorithm to determine its run time before implementation:

We do this by tallying up all of the primitive operations that would be performed by this code. (This requires a certain level of insight into how a compiler will translate non-trivial operations, such as array indexing.)

---

```
1: for (i = 0; i < N; ++i) {  
2:     a[i] = 0;  
3:     for (j = 0; j < N; ++j)  
4:         a[i] = a[i] + i*j;  
5: }
```

Analysis:

- line 1 is executed  $N$  times.
  - line 2 is executed  $N$  times.
  - line 3 is executed  $N^2$  times.
  - line 4 is executed  $N^2$  times.
-

Now let's break down line 1 into more detail:

```
1: for (i = 0; i < N; ++i) {  
2:     a[i] = 0;  
3:     for (j = 0; j < N; ++j)  
4:         a[i] = a[i] + i*j;  
5: }
```

- line 1 is executed  $N$  times.
  - More precisely, we have 1 assignment when  $i$  is initialized, plus an addition, comparison, and assignment to  $i$  each time the loop is repeated. There is also a final comparison just before exiting the loop.
- line 2 is executed  $N$  times.
- line 3 is executed  $N^2$  times.
- line 4 is executed  $N^2$  times.

We'll tally these into a table like this:

Line	Iterations	Assignments	Additions	Multiplications	Comparisons
1	$N$	$N+1$	$N$	0	$N+1$

Now let's look in detail at line 2:

```
1: for (i = 0; i < N; ++i) {  
2:     a[i] = 0;  
3:     for (j = 0; j < N; ++j)  
4:         a[i] = a[i] + i*j;  
5: }
```

The 2nd line contributes 1 addition, 1 multiplication and 1 assignment each time it is executed, because an array indexing op like  $a[i]$  gets translated by the compiler into

$$\text{address} = \text{addr}_a + i * s_a$$

where  $\text{addr}_a$  is the starting address of the array  $a$  and  $s_a$  is the size (in bytes) of a single element of  $a$ .

Line	Iterations	Assignments	Additions	Multiplications	Comparisons
1	N	N+1	N	0	N+1
2	N	N	N	N	0

Moving on to line 3,

```
1: for (i = 0; i < N; ++i) {  
2:     a[i] = 0;  
3:     for (j = 0; j < N; ++j)  
4:         a[i] = a[i] + i*j;  
5: }
```

The 3rd line contributes 1 assignment when  $j$  is initialized. This happens  $N$  times. It also has an addition, comparison, and assignment to  $j$  each time the loop is repeated, plus one other comparison at the start of the loop.

Line	Iterations	Assignments	Additions	Multiplications	Comparisons
1	$N$	$N+1$	$N$	0	$N+1$
2	$N$	$N$	$N$	$N$	0
3	$N^2$	$N^2 + N$	$N^2$	0	$N(N + 1)$

And, finally, line 4:

```
1: for (i = 0; i < N; ++i) {  
2:   a[i] = 0;  
3:   for (j = 0; j < N; ++j)  
4:     a[i] = a[i] + i*j;  
5: }
```

The 4th line produces 3 additions, 3 multiplications (2 of these additions and multiplications are from the array indexing) and one assignment each time it is executed.

- Some compilers are smart enough to recognize that “`a[i]`” is the same expression in two places, and must refer to the same address each time. These compilers would only do the calculation once, saving an addition and a multiplication.

Recognition of common subexpressions like that is one of the more common *optimizations* that you are likely to see if you compile your code with the appropriate flags (e.g., `-O2` for `g++`) to request optimization.

Line	Iterations	Assignments	Additions	Multiplications	Comparisons
1	N	N+1	N	0	N+1
2	N	N	N	N	0
3	$N^2$	$N^2 + N$	$N^2$	0	$N(N + 1)$
4	$N^2$	$N^2$	$3N^2$	$3N^2$	0

## Totals

And the total is ...

```
1: for (i = 0; i < N; ++i) {  
2:   a[i] = 0;  
3:   for (j = 0; j < N; ++j)  
4:     a[i] = a[i] + i*j;  
5: }
```

Line	Iterations	Assignments	Additions	Multiplications	Comparisons
1	N	N+1	N	0	N+1
2	N	N	N	N	0
3	$N^2$	$N^2 + N$	$N^2$	0	$N(N + 1)$
4	$N^2$	$N^2$	$3N^2$	$3N^2$	0
<b>Totals:</b>		$2N^2 + 3N + 1$	$4N^2 + 2N$	$3N^2 + N$	$N^2 + 2N + 1$

The total run time,  $T(N)$ , for this algorithm is

$$T(N) = (2N^2 + 3N + 1)t_{\text{asst}} + (N^2 + 2N + 1)t_{\text{comp}} \\ + (4N^2 + 2N)t_{\text{add}} + (3N^2 + N)t_{\text{mult}}$$

where  $t_{\text{asst}}$  is the time required by our CPU to do one assignment,  $t_{\text{comp}}$  is the time required by our CPU to do one comparison,  $t_{\text{add}}$  is the time required to do one addition, and  $t_{\text{mult}}$  is the time required to do one multiplication.

### 1.1.1 Limitations of Detailed Analysis

- This process is tedious.
- Actual value depends upon how fast your CPU does assignments, additions, ...
- Actual value depends on how your compiler translates the source code into lower-level instructions and upon the settings you use when invoking the compiler.

## 1.2 Do We Need That Much Detail?

The total run time,  $T(N)$ , for this algorithm is

$$T(N) = (2N^2 + 3N + 1)t_{\text{asst}} + (N^2 + 2N + 1)t_{\text{comp}} \\ + (4N^2 + 2N)t_{\text{add}} + (3N^2 + N)t_{\text{mult}}$$

Suppose that we group together terms that involve different powers of  $N$ :

$$T(N) = N^2(2t_{\text{asst}} + t_{\text{comp}} + 4t_{\text{add}} + 3t_{\text{mult}}) \\ + N(3t_{\text{asst}} + 2t_{\text{comp}} + 2t_{\text{add}} + t_{\text{mult}}) \\ + 1(t_{\text{asst}} + t_{\text{comp}})$$

Each of the parenthesized terms is, for any given CPU, a constant.

### 1.2.1 CPU Constants

Define

$$c_1 = 2t_{\text{asst}} + t_{\text{comp}} + 4t_{\text{add}} + 3t_{\text{mult}} \\ c_2 = 3t_{\text{asst}} + 2t_{\text{comp}} + 2t_{\text{add}} + t_{\text{mult}} \\ c_3 = t_{\text{asst}} + t_{\text{comp}}$$

Then  $T(N) = c_1 N^2 + c_2 N + c_3$

- The  $c_i$  will be different for different CPUs and even for different compilers on the same CPU.
- But this formula for  $T(N)$  still allows us to describe the behavior of this algorithm for different input (array) sizes.
- Shortly, we will see that's all that *really* matters.

## 2 Best, Worst, and Average Cases

### 2.1 Varying Times

Many algorithms will run in different amounts of time depending upon which input data they are given, even when given the same “amount” of data.

Consider, for example, a basic sequential search:

```
int seqSearch (int[] arr, int N, int key)
{
    for (int i = 0; i < N; ++i)
    {
        if (arr[i] == key)
            return i;
    }
    return -1;
}
```

Obviously, if we increase  $N$ , the number of elements in the array, this algorithm will slow down. But even if we look at different possibilities for some fixed value of  $N$ , there can be very different run times.

- If we search for a key that happens to be in  $arr[0]$ , we return after a single iteration of the loop.
- If we search for a key that happens to be in  $arr[N-1]$ , we return after  $N$  iterations of the loop.
- If we search for a key that isn't anywhere within  $arr$ , we return after  $N$  iterations of the loop, but  $N+1$  evaluations of the loop condition. This is actually slightly slower than the previous case.
- If we make a random selection of a string from among the ones filled into  $arr$ , we will take something between the two extremes of 1 iteration and  $N$  iterations of the loop. Intuitively, we might guess that we will average around  $N/2$  iterations.

For a fixed “size” of input,

- we call the minimum run-time we can obtain, over all possible inputs of that size, the *best case* time of that algorithm,
- we call the maximum run-time we can obtain, over all possible inputs of that size, the *worst case* time of that algorithm, and
- we call the average run-time we can obtain, over all possible inputs of that size, the *average case* time of that algorithm.

Now, the notion of what constitutes an “average” is a surprisingly slippery one, so we will defer discussion of that till a [later lesson](#).

Even the idea of “size of input” is a bit problematic, but we need to come to grips with that almost immediately.

First though, let's point out that we are only rarely interested in the best case behavior. It would be absurdly overly-optimistic to make decisions about what algorithm to use based on it's best-case behavior.

We might think that the average case behavior would be a much more reasonable basis for choice. And, sometimes it is. But if the worst case behavior is much, much slower than the average, then Murphy's law says that we will hit that slow worst case just when it is most critical, when we really *need* that output quickly or when we are demoing the code to upper management or potential customers.

- Worst case behavior is particularly crucial for interactive programs. If you have a button in front of you, and when you click on it, you are used to getting a response in an average of a tenth of a second, what do you do when you suddenly get a “worst case” delay of 5 seconds? Why you click it again, and again, and... (Come on, be honest, you *know* you do this!) And suddenly instead of performing a function once, the program finally finishes handling the first click and then immediately tries to honor all of those other clicks as well.

So, for now, we're going to focus on worst-case behavior.

## 2.2 Size of the Input

What do we mean by “size”? Often, that's pretty obvious. In our sequential search function, the array size  $N$  is clearly an indicator of how much data we have to work with. If, as another example, we had a program that read a series of numbers and computed the average of them all, the “size” is probably how many numbers are in the input file. If we had a program that read personnel records for preparing a company's payroll, the “size” is probably the number of people working for the company.

For some code, there may be multiple possibilities for a “size” measure. If our program reads and processes lines of text, the best “size” measure might be the number of lines in the input file. But if we actually scan through each character in each line, the total number of characters might be a more appropriate “size” measure.

Sometimes, we will need to use multiple, separate numbers to describe the size of the input. Let's change our sequential search function just slightly:

```
int seqSearch (string[] arr, int N, string key)
{
    for (int i = 0; i < N; ++i)
    {
        if (arr[i] == key)
            return i;
    }
    return -1;
}
```

Now, the time required to do the comparison  $arr[i] == key$  depends upon how many characters are in our strings. Obviously, strings of a few dozen characters can be compared quickly. Strings of thousands or millions of characters may take a long time to compare. So we might need many numbers to measure the size  $\bar{n}$  of the input for this function:

$$\bar{n} = (N, w_0, w_1, \dots, w_{N-1}, w_{key})$$

where  $N$  is the parameter denoting the array size, the  $w_i$  are the widths of the strings in  $arr$ , and  $w_{key}$  is the width of the string in  $key$ .

Well, that got ugly real fast!

Now, given that we have said that we are going to focus on worst case behavior, we can guess that we can probably reduce that a bit to

$$\bar{n} = (N, w)$$

where  $N$  is the parameter denoting the array size, and  $w$  is the maximum width of any string in  $\text{arr}$  and  $\text{key}$ .

In some cases we might be able to reduce the our size  $\bar{n}$  to a single number. If someone were to walk in on our design meeting and say “Oh, by the way, we never have more than 25 elements in the array.”, we might conclude that  $w$  is the only thing we need to worry about. Or, hearkening back to our earlier example of a spellchecker design, if we believed that our array was filled with English language words, we could argue that there is a reasonably small limit on  $w$  and that  $N$  is the only thing we *really* need to worry about.

- Reputedly, “antidisestablishmentarianism” is the longest word recognized in English dictionaries. While quite a mouthful, that’s still not long enough to be a major concern in most text processing applications.)
- On the other hand, if we were working in genetics, our “strings” might represent chromosomes and a “word” might be many thousands of characters long, in which case we might expect the length of the strings being processed would be a major factor in the algorithm performance.

## 3 Worst Case Complexity

### 3.1 Time Proportional To

#### Definition

We say that an algorithm requires time  $\text{proportional to } f(\bar{n})$  if there are constants  $c$  and  $\bar{n}_0$  such that the algorithm requires no more than  $c * f(\bar{n})$  time units to process an input set of size  $\bar{n}$  whenever  $\bar{n} \geq \bar{n}_0$ .

The definition you see here is going to be fundamental to our idea of complexity of an algorithm. This definition explains what we mean when we say that an algorithm requires time proportional to some function of  $n$ , where  $n$  is the measure of the amount of input data.

Let's take this definition apart, piece by piece, and see what it actually tells us.

- First, we are talking about *an input set of size  $\bar{n}$* . We've already talked about this idea. It's not always clear what the components of  $\bar{n}$  are, but, as it happens, the techniques that we will be developing will help to reveal the appropriate “size” measure(s) as a side effect of the analysis that we will be doing anyway. So we don't need to sweat over this question too much for now.
- Second, we are talking about algorithms taking *time proportional to  $f(\bar{n})$* . The “ $f$ ” here describes the rate at which the time required by this algorithm goes up as you change the size of the input for particular program or algorithm.

Many (most?) authors write the definition of “time proportional to” in terms of  $f(n)$  instead of  $f(\bar{n})$  and assume that you implicitly understand that  $n$  can have multiple components.

I think that, when we are starting out, it's good to remember that this is not restricted to single numbers, but don't let it phase you if the bar over  $\bar{n}$  disappears from time to time.

For example, if I were to tell you that an algorithm runs in time *proportional to  $N$*  (i.e.,  $f(|\text{bar}(n)|)=N$ ), then I am telling you that the running time of this algorithm is directly proportional to the size of the input set.

- Doubling the size of input set (running twice as much data we did previously) may double the running time. More specifically, it will double the maximum amount of time we were expecting this algorithm will take.
- On the other hand if I run 10 times as many inputs through this algorithm, I will expect that, for an algorithm with time proportional to  $n$ , it would take 10 times as much time to run.

Now, suppose I tell you that this algorithm runs in time proportional to  $N^2$ . Then the running time of algorithm in the worst case should go up with the square of the input size.

- So if I run twice as much input than before, I should see that the maximum running time goes up by the factor of 4.  
◦ If I run 10 times as many inputs as I did previously, I could see the running time will go up by as much as 100 times.

Keep in mind that we are talking about *worst case* situations here. If an algorithm takes time proportional to  $N^2$  and I double the amount of input I put through it, I *might* see the running time go up as much as a factor of 4. It doesn't have to be that bad, but it could be.

- The next component of the definition I want to look at is the *multiplier c*. The multiplier  $c$  is used so that we can talk about the algorithm requiring no more than “ $c * f(\bar{n})$  time units”. In that sense,  $c$  is the multiplier that converts this abstract function  $f(\bar{n})$  into a real time.

If I tell you, let's say, that this algorithm runs in time proportional to  $f(\bar{n}) = N$  and also tell you that constant  $c$  is 0.25 seconds, then you can take any particular value of  $N$ , plug it into this formula, and get the exact value for the maximum run time of this algorithm. For example, when  $N=10$ , then I would expect this algorithm would take no more than 2.5 seconds.

- The final component of this definition is  $\$|\text{bar}(n)|\$$ .

When we say  $\bar{n} \geq \bar{n}_0$ , we mean that each component of  $\bar{n}$  is greater than or equal to the corresponding component of  $\bar{n}_0$ .

$\bar{n}_0$  is used to place a lower limit on the inputs that we are really worried about. The reason for doing this is that

1. many algorithms behave rather differently on small input sets than they do on large sets, and
2. algorithms are often fast enough on small input sets that we just don't care about their speed. It's only when we have large amounts of data to process that speed becomes an issue.

Suppose that algorithm A is faster than B on small input sets, but B is faster than A on large input sets. Odds are that, on the small sets, both algorithms are “fast enough” and any differences between them are of little practical consequence. What we are usually interested in is talking about what happens when the input set size is big enough to actually be troublesome. That’s what  $n_0$  does for us. The definition says that we are only worried about this when  $\bar{n} \geq n_0$ .  $n_0$  represents the limit of “being large enough to worry about”. When  $\bar{n}$  is smaller than  $n_0$  we *don’t care* about the speed.

Sometimes an algorithm is able to achieve good run times for large input sizes because it does a lot of extra preparations at the beginning of the run. For example, some algorithms compute a bunch of intermediate results and store them away in data structures from which they can be retrieved later. In circumstances like that, you often find that the algorithm speed suffers for the very small input set sizes because there is certain amount of overhead involved in doing that preparation. And so a simpler algorithm that doesn’t do that kind of extra preparation may run faster for small input set sizes. But when you start giving larger and larger sets of inputs, the algorithm that does the extra preparation starts to go faster and faster, taking advantage of that extra work that it did early.

Given a choice between these two algorithms, we probably want to go with more elaborate algorithm, not because it is more elaborate but because, when it really counts, that algorithm actually runs faster for us.

This is what  $n_0$  does for us in that definition. It sets a certain threshold below which we don’t really care.

### 3.1.1 Getting back to “complexity”

What does “time proportional to” have to do with the idea of complexity?

When I started off this lesson, I said what we are trying to do was to come up with the way to discuss the speed of an algorithm. Then I told you that we would not measure the speed directly, but instead use this notion of “complexity”. Then I turned around and gave you this definition of “time proportional to”. Is this some sort of bait and switch?

There are two steps remaining to justify the connections between “time proportional to” and “complexity”.

## 3.2 Big-O

First, we introduce a shorthand notation so we don’t have to keep writing out “time proportional to”.

### Definition

$O(f(N))$  is the set of all functions that are proportional to  $f(N)$ .

If  $t(N)$  is the time required to run a given program on an input set of size  $N$ , we write

$$t(N) = O(f(N))$$

to mean “there exist constants  $c$  and  $n_0$  such that  $t(N) \leq c * f(N)$  when  $N > n_0$ .”

This is often called “big-O” notation. For example, if a program runs in time  $T(N)$  and we can show that  $T(N)$  is proportional to  $f(N) = N^2$ , then we would say that “ $T(N)$  is in  $O(N^2)$ ”. Informally, we often simply say that the program is in  $O(N^2)$ . Some people will shorten that phrase even further and simply say that the program “is  $O(N^2)$ ”, but that tends to hide the fact that  $O(N^2)$  is actually the name of a whole set of programs.

The “O” in this notation stands for “order”, so people will sometimes talk about  $O(N)$  functions as having “linear order” or  $O(N^2)$  functions as having “quadratic order”.

We’re abusing the “=” sign here. That’s not really equality. Since  $O(f(N))$  is a set of functions, we should probably be writing

$$t(N) \in O(f(N))$$

instead, and the original definitions of  $O(f(n))$  did exactly that. But somewhere along the line, it became tradition to use the “=” symbol instead of “ $\in$ ”. [Donald Knuth](#) has noted that mathematicians often write “=” to mean “is”, which in turn can sometimes mean “is the same as” (“ $1+1$  is  $2$ ”) and other times means “is an example of” (“ $2$  is an integer”). If we were to treat these as “true” equality, “we could deduce ridiculous things like  $n = n^2$  from the identities  $n = O(n^2)$  and  $n^2 = O(n^2)$ ”.

## 3.3 Big-O == Worst Case Complexity

Second, we assert that this quantity,  $O(f(n))$ , is in fact what we call the *worst case complexity* of an algorithm, and that this will be our measure of algorithm speed in most cases.

### 3.3.1 Big-O and Expectations for Run Times

Suppose that we have run a program all day, every day for a year.

- In all that time, we have never managed to process more than 1,000 data in one day.
- We are considering buying a new CPU that runs twice as fast as our current one

How many data records should we expect to process per day with the new CPU?

If we know the complexity of the program, we can answer that question.

Program Complexity	Max Records per Day
$O(\log n)$	1,000,000

Program Complexity	Max Records per Day
$O(n)$	2,000
$O(n \log n)$	1,850
$O(n^2)$	1,414
$O(n^3)$	1,259
$O(2^n)$	1,001

All the big-O expressions shown above are ones that we will encounter in the course of this semester, so you can see that there is a very noticeable and practical difference in the answers we would offer to the question, “Is it worth it to buy the faster CPU?” Certainly, if we knew that our program was in  $O(2^n)$ , then buying a faster CPU would not actually boost our processing capability by much at all. We would probably do better by devoting our resources to redesign the program to use a much lower-complexity algorithm.

## 4 Testing a $O(f(N))$ Hypothesis

For example, suppose that we run a program on different sizes of input. Suppose, furthermore, that we have observed this program for a long time and noted that, for any given size of the input, it always takes approximately the same amount of time. In other words, whatever the worst case time is, we are reasonably sure that we have seen it or that it’s not significantly worse than the other times we have been observing.

If this is what we have observed:

Input Set Size	Average Time (seconds)
1.0	10.0
2.0	40.0
3.0	80.0
4.0	165.0

We would suspect that this algorithm is in  $O(N^2)$  (with  $c = 10$ ).

To defend this suspicion, refer back to our definition:

We say that an algorithm requires time proportional to  $f(n)$  if there are constants  $c$  and  $n_0$  such that the algorithm requires no more than  $c*f(n)$  time units to process an input set of size  $n$  whenever  $n \geq n_0$ .

Divide each running time by  $f(N) = N^2$ . This would give us an approximate value for the constant  $c$  from our definition of complexity.

Input Set Size (N)	Avg. Time (sec)	Time / ( $N^2$ )
1.0	10.0	10.0
2.0	40.0	10.0
3.0	80.0	8.9
4.0	165.0	10.3

If our guess is correct, the quotients should stay roughly constant.

Some variation is normal in any experiment, but this looks pretty good. One important limitation, however, is that in this example we’re only dealing with averages over a finite number of observations. If that sample of input cases does not, in fact, include the input case responsible for the worst time of the algorithm and if that worst case is significantly different from the average, then then our numbers don’t mean much at all about the worst case complexity.

Another thing to consider: the following is also probably  $O(N^2)$ :

Input Set Size (N)	Time (seconds)
1.0	2000.0
2.0	8000.0
3.0	16000.0
4.0	32000.0

It just has a significantly larger value for  $c$ .

So two programs can both be in  $O(f)$ , but have very different run times. As we’ll see shortly, however, when two programs have different complexities  $O(f)$  and  $O(g)$ , no matter what constants they might involve, the function part of the complexities will eventually dominate any comparison.

## 5 Detailed Timing Example Revisited

We left this earlier example

```
1: for (i = 0; i < N; ++i) {
2:     a[i] = 0;
3:     for (j = 0; j < N; ++j)
```

```

4:     a[i] = a[i] + i*j;
5: }

```

after concluding that  $T(N) = c_1N^2 + c_2N + c_3$

### Conjecture

$$T(N) = c_1N^2 + c_2N + c_3 = O(N^2)$$

(i.e., the algorithm has a worst-case complexity of  $N^2$ )

### Proof

$$T(N) = c_1N^2 + c_2N + c_3$$

Let  $c = c_1 + c_2 + c_3$  and let  $n_0 = 1$ . It should be clear that

$$c_1N^2 + c_2N + c_3 \leq c_1N^2 + c_2N^2 + c_3N^2$$

because  $c_2N \leq c_2N^2$  and  $c_3 \leq c_3N^2$  whenever  $N \geq 1$ .

We know, therefore, that

$$T(N) \leq c_1N^2 + c_2N^2 + c_3N^2$$

Now define  $c$  as  $c = c_1 + c_2 + c_3$ , and we have

$$T(N) \leq cN^2$$

which means, by the [definition of “proportional to”](#) that  $T(N)$  is proportional to  $N^2$  and therefore is in  $O(N^2)$ .

## 6 Pitfalls to Avoid

### 6.1 Loose Bounds and Tight Bounds

The tallest building in the world is said to be the Burj Khalifa, at 2,717 ft. So if I told you that “ $h(b)$  is the height of a building  $b$ ”, you might suggest that  $h(b) \leq 2717$  ft. And that’s a reasonable bound.

Of course it’s also true that  $h(b) \leq 4000$  ft. And that  $h(b) \leq 10000$  mi. None of those are false statements, but 2717 ft is a *tight bound*, 4000 ft is a *loose bound*, and 10000 mi is a ridiculously loose bound.

What is the complexity of  $t(n) = 0.5n$ ?

- We could say that  $t(n) \in O(n)$ , and that’s a tight bound.
- We could say that  $t(n) \in O(n^2)$ , and that’s true but is a loose bound.
- We could say that  $t(n) \in O(\infty)$ , and that’s also true but is such a loose bound that it’s completely useless.

In general, you should always try to give as tight a bound for the complexity as you can.

Sometimes we will settle for slightly looser bounds because proving anything tighter would be difficult. But being off by one or more powers of  $n$  will be considered a failure to provide a proper bound.

### 6.2 “n” is “n”othing special

Recall our definition:

We say that an algorithm requires time [proportional to  \$f\(\bar{n}\)\$](#)  if there are constants  $c$  and  $\bar{n}_0$  such that the algorithm requires no more than  $c * f(\bar{n})$  time units to process an input set of size  $n$  whenever  $\bar{n} \geq \bar{n}_0$ .

In our definition of complexity,  $\bar{n}$  is simply some numeric measure of the “size” of the algorithm’s inputs. Now, that doesn’t mean the algorithm will actually contain a variable or parameter named “ $n$ ”, or that there will even be a single program variable that we can substitute for  $n$  in this definition. Sometimes, our “ $n$ ” is an expression that, somehow, defines a measure of size of the input.

For example, consider the function

```
ostream& operator<< (ostream& out, const std::string& str);
```

for writing strings to an output stream. Now, intuitively, we can guess that if were to double the number of characters in the string, `str`, to be written out, that the amount of time taken by this function would also double. That suggests that this function's complexity is probably proportional to the length of the string `str`.

But how do we write that? We can't simply say:

- `operator<<` is in  $O(n)$

because *there is no “n” here*.

You know that, when you are programming, you aren't allowed to use variables that you have not properly declared. Surprise! The same rules applies to mathematics: you can't use undeclared variables in mathematics!

We also can't say

- `operator<<` is in  $O(out)$

or

- `operator<<` is in  $O(str)$

because, although `out` and `str` *are* defined variables in this context, neither one is a numeric quantity, so the expressions  $O(out)$  and  $O(str)$  don't make sense. For example,  $O(str)$  would, ultimately, mean that  $t(n) < c * str$ , and that simply makes no sense if `str` is a string.

There's two ways to express the idea that this function's complexity is probably proportional to the length of the string `str`. The first is to simply define the symbol we want:

- `operator<<` is in  $O(n)$ , where  $n$  is the number of characters in `str`.

The added definition makes all the difference — it allows a reader to ascertain that  $n$  is a property of `str` (and not of some other variable that might be mentioned nearby, such as, for example, `out`) and exactly what the nature of that property is. This assumes, of course, that the idea of “number of characters in” is sufficiently obvious to be easily and unambiguously understood.

The other way to express the same idea is to replace “ $n$ ” by an appropriate expression that captures the concept of “number of characters in `str`”. Taking advantage of the public interface to `std::string`, we can write:

- `operator<<` is in  $O(str.size())$

which is perfectly understandable to anyone familiar with `std::string` in C++.

Writing a big-O expression with undefined variables (especially “ $n$ ”) is one of the most common, and least forgivable, mistakes in students answers on assignments and tests.

# The Algebra of Big-O

Steven J. Zeil

Last modified: Feb 12, 2020

## Contents:

- [1 Basic Manipulation](#)
- [2 Dropping Constant Multipliers](#)
  - [2.1 Intuitive Justification](#)
  - [2.2 Proof:  \$O\(c \* f\(N\)\) = O\(f\(N\)\)\$](#)
- [3 Larger Terms Dominate a Sum](#)
  - [3.1 Intuitive Justification](#)
  - [3.2 Proof: Larger Terms Dominate a Sum](#)
- [4 Logarithms are Fast](#)
- [5 Summary of big-O Algebra](#)
  - [5.1 The Tao of N](#)
- [6 Always Simplify!](#)

In the previous lesson we saw the definition of “time proportional to” (big-O), and we saw how it could be applied to simple algorithms. I think you will agree with me that the approach we took was rather tedious, and probably you wouldn’t want to use it in practice.

We are going to start working towards more usable approach to match the complexity of algorithm. We will start by looking at rather peculiar algebraic rules that can be applied for manipulating big-O expressions. In the next lesson, we shall then look at the process of taking an algorithm the way you and I actually write it in typical programming languages and analyzing that to produce the original big-O expressions.

For now, though, let us start by discussing how we might manipulate big-O expressions once we’ve actually got them. The first thing we have to do is to recognize that the algebra of big-O expressions is not the same “good old fashioned” algebra you learned back in high school. The reason is that when we say something like  $O(f(N)) = O(g(N))$ , we are not comparing two numbers to one another with that ‘=’, nor are we claiming that  $f(N)$  and  $g(N)$  are equal. We are instead comparing two sets of programs (or functions describing the speed of programs) and we are stating that any program in the set  $O(f(N))$  is also in the set  $O(g(N))$  and vice-versa.

We will now explore the peculiar algebra of big-O, and, through that algebra, will see why big-O is appropriate for comparing algorithms.

## 1 Basic Manipulation

We start with a pair of simple, almost self-evident rules:

**Algebraic Rule 1:**  $f(N) + g(N) \in O(f(N) + g(N))$

This rule comes into play most often when we discover that a program has to do two (or more) things before it is finished. The rule simply says that the running time of the entire program *might* be no faster than the sum of the running times of its constituent parts.

Because this is only an upper bound, the program might actually be faster than that (e.g., if the first part of the program prepares the data in some fashion that allows the second part to run faster than it would have in isolation). But it will be no slower, in the worst case, than some constant times the sum of the two run time bounds.

**Algebraic Rule 2:**  $f(N) * g(N) \in O(f(N) * g(N))$

This rule comes into play most often when we discover that a program has a loop that runs through  $f(N)$  iterations, and each iteration of the loop takes  $O(g(N))$  time. Then the program will be no slower, in the worst case, than some constant times the product of those two functions.

## 2 Dropping Constant Multipliers

Our next rule is a bit less intuitive:

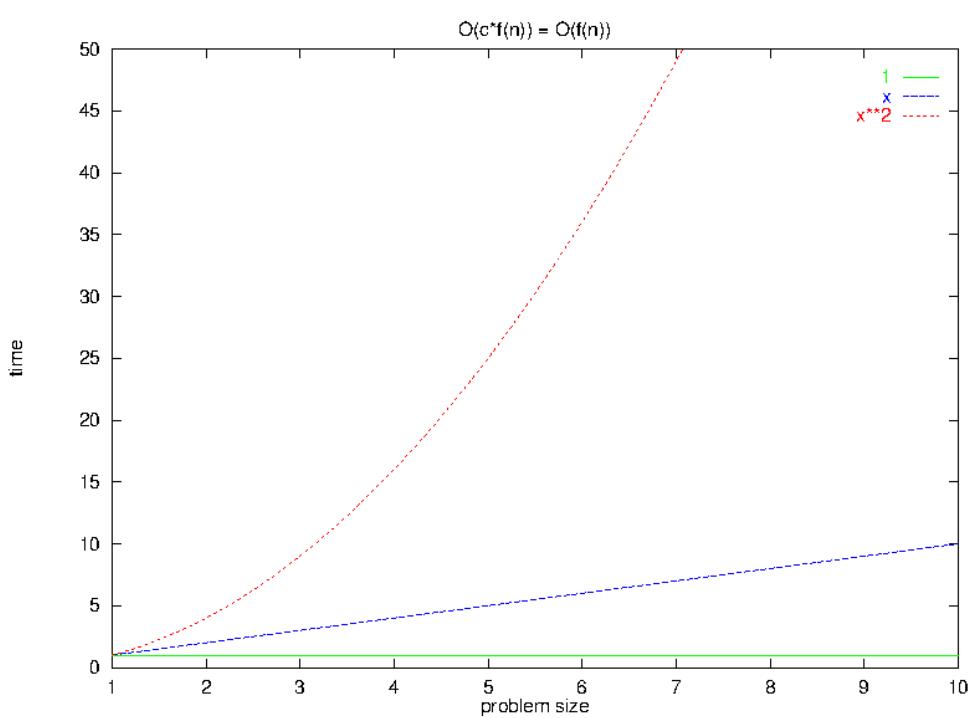
**Algebraic Rule 3:**  
 $O(c * f(N)) = O(f(N))$

This rule suggests that, for sufficiently large  $N$ , constant multipliers “don’t count”.

### 2.1 Intuitive Justification

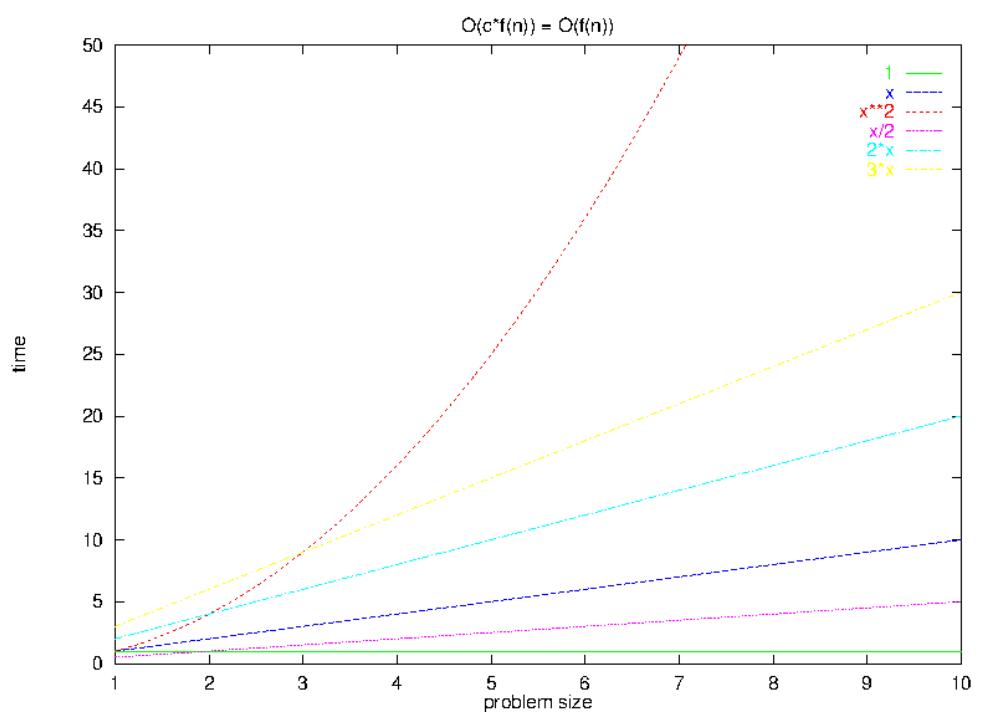
To see why this might be so, suppose we have three programs that run in time  $O(1)$ ,  $O(x)$ , and  $O(x^2)$ , where  $x$  is the size of the input set.

Notice from the plot of these three times, that, for sufficiently large  $x$ , the  $O(x)$  time falls between the other two.



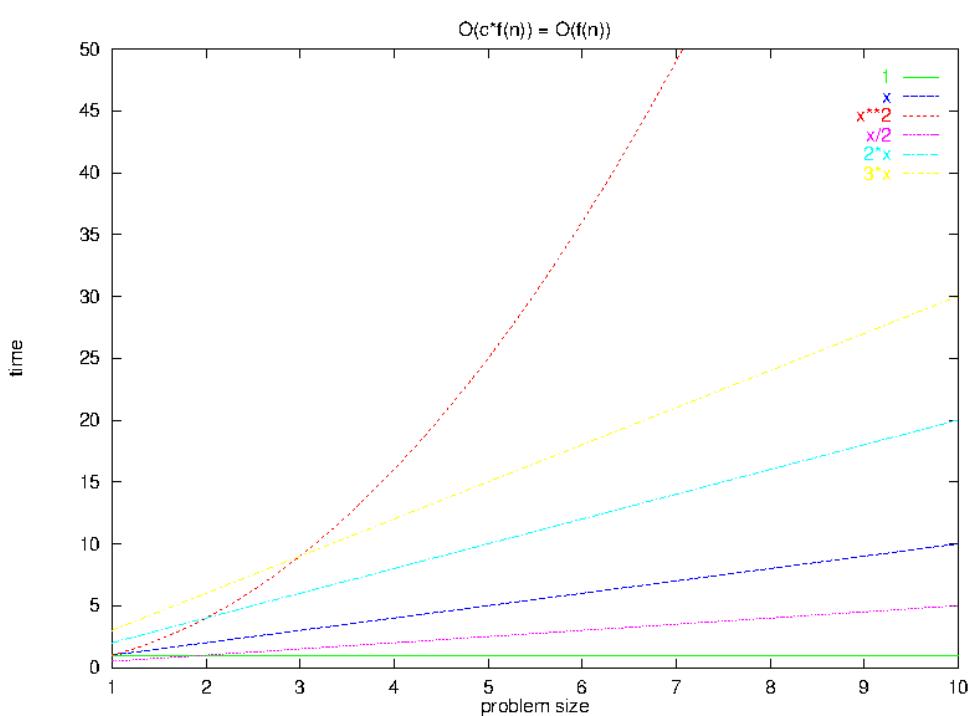
Now add some more programs, whose running times differ from the  $O(x)$  program only by constant multipliers.

- Note that the new  $O(c * x)$  programs all run faster than the  $O(x^2)$  program (for sufficiently large  $x$ ).
    - Note especially that, as  $x$  gets bigger, the differences between the  $O(x^2)$  program and *any* of the  $O(c * x)$  programs is far, far bigger than the differences among the  $O(c * x)$  programs.
    - This suggests that knowing whether the “main” function is  $x$  or  $x^2$  is far more important than knowing what  $c$  is.
- 



- Note, similarly, that the  $O(c * x)$  programs all run slower than the  $O(1)$  program.

- And again, note that, as  $x$  gets bigger, the differences between the  $O(1)$  program and *any* of the  $O(c * x)$  programs is far, far bigger than the differences among the  $O(c * x)$  programs.
  - Again, this suggests that knowing whether the “main” function is  $x$  or 1 is far more important than knowing what  $c$  is.
- 

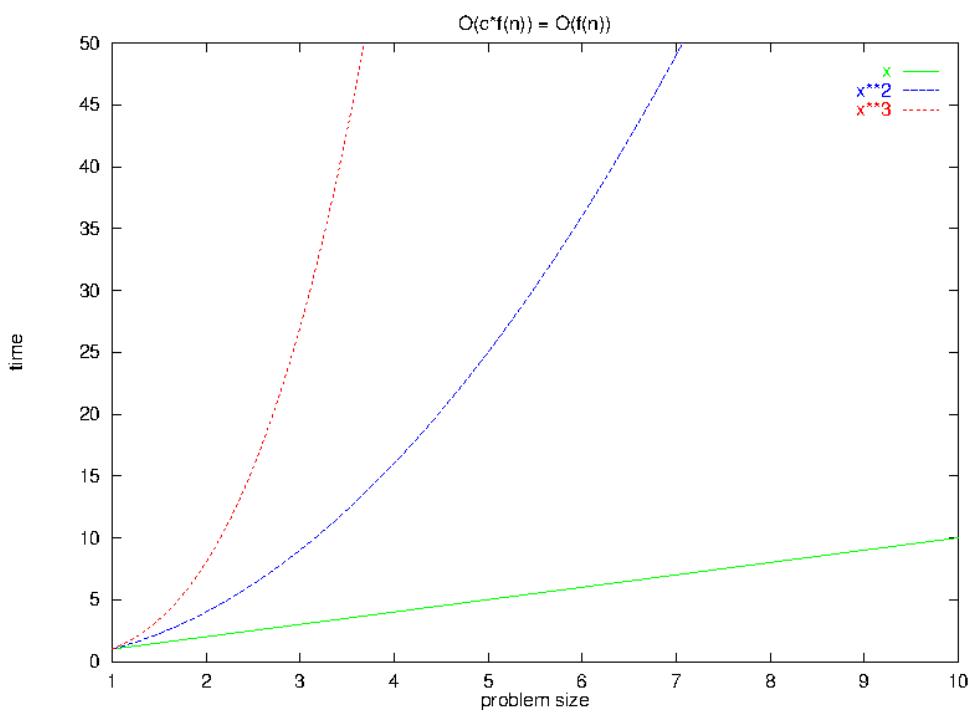


We can see a similar pattern with even larger functions.

Suppose we start with three programs that run in time  $O(x)$ ,  $O(x^2)$ , and  $O(x^3)$ , where  $x$  is the size of the input set.

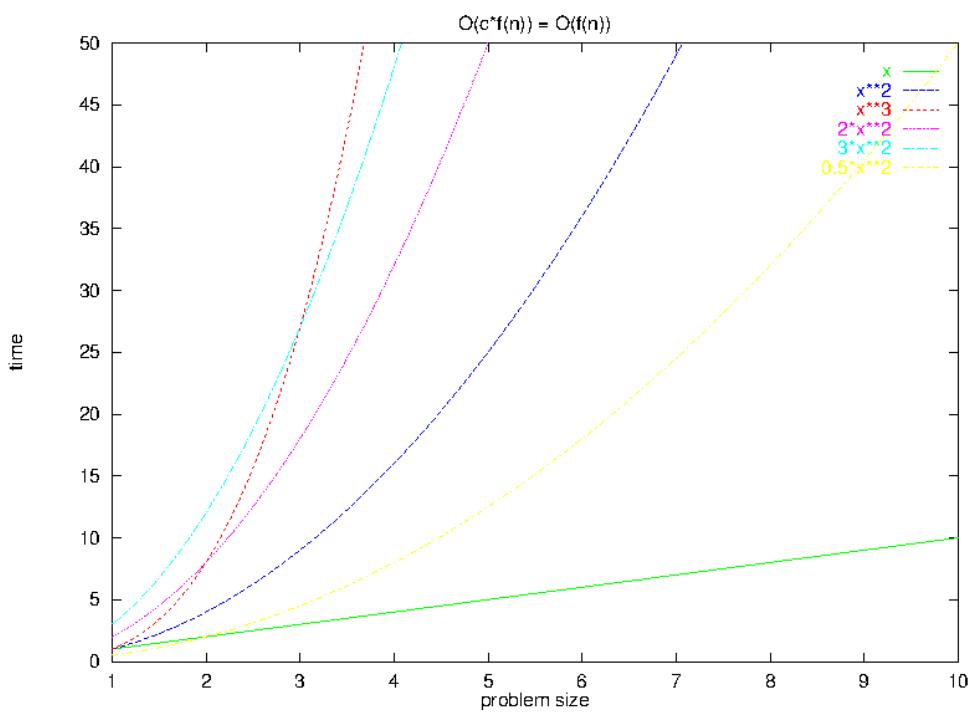
Notice from the plot of these three times, that, for sufficiently large  $x$ , the  $O(x^2)$  time falls between the other two.

---

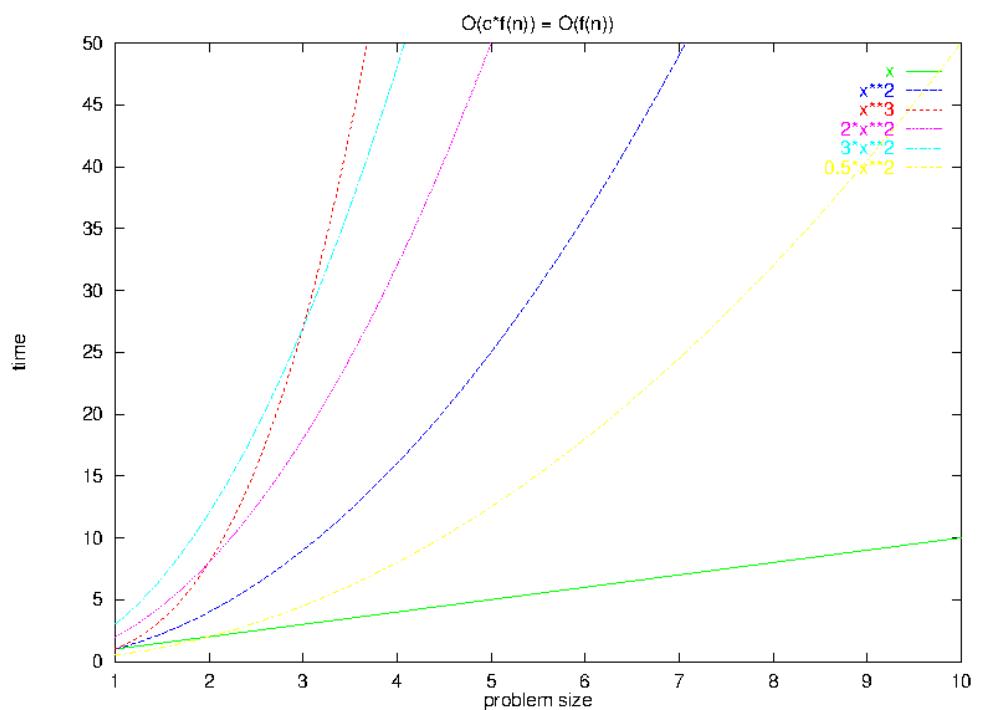


Now add some more programs, whose running times differ from the  $O(x^2)$  program only by constant multipliers.

- Note that the new  $O(c * x^2)$  programs all run faster than the  $O(x^3)$  program (for sufficiently large  $x$ ).
    - Note especially that, as  $x$  gets bigger, the differences between the  $O(x^3)$  program and *any* of the  $O(c * x^2)$  programs is far, far bigger than the differences among the  $O(c * x^2)$  programs.
    - This suggests that knowing whether the “main” function is  $x^2$  or  $x^3$  is far more important than knowing what  $c$  is.
- 



- Note, similarly, that the  $O(c * x^2)$  programs all run slower than the  $O(x)$  program.
  - And again, note that, as  $x$  gets bigger, the differences between the  $O(x)$  program and *any* of the  $O(c * x^2)$  programs is far, far bigger than the differences among the  $O(c * x^2)$  programs.
  - Again, this suggests that knowing whether the “main” function is  $x^2$  or  $x$  is far more important than knowing what  $c$  is.



### 2.1.1 So What Good are Constants?

Does this mean that constant multipliers aren’t relevant at all?

- No. Suppose we had two programs A and B with times

$$t_A(N) = N * (1\text{ second})$$

$$t_B(N) = N * (1\text{ minute})$$

- For  $N = 60$ , this is the difference between waiting 1 minute and waiting for 1 hour. No one will argue that this difference is insignificant.

- But now suppose that we consider a third program C with an even smaller constant:

$$t_C(N) = N^2 * (0.01\text{ second})$$

Even though this program has a much smaller constant factor, and will perform much faster on small  $N$ , it will be slower than program A whenever  $N > 100$  and slower than B when  $N > 6000$ .

- So the constant can play a role in choosing the faster algorithm only when the function parts of the complexity are equal.

## 2.2 Proof: $O(c * f(N)) = O(f(N))$

All we have done so far is to give examples that this rule holds. No number of examples can constitute a proof. To prove that it holds in general, consider any program with running time  $t(N) = O(c * f(N))$ .

By the definition of  $O(\dots)$ , we have:

$$\exists c_1, n_0 | n > n_0 \Rightarrow t(N) \leq c_1(c * f(N))$$

But, grouping the multiplication slightly differently gives

$$\exists c_1, n_0 | n > n_0 \Rightarrow t(N) \leq (c_1 * c)f(N)$$

Now, since  $c$  and  $c_1$  are both constants, we can introduce a new constant  $c_2$ :

$$c_2 = c_1 * c$$

and then we can claim that

$$\exists c_1, n_0 | n > n_0 \Rightarrow t(N) \leq c_2 * f(N)$$

But this is just the definition of  $O(f(N))$ , so we conclude that

$$t(N) = O(f(N))$$

Therefore any program in  $O(c * f(N))$  is also in  $O(f(N))$ .

It's easy enough to modify the above argument to show that any program in  $O(f(N))$  is also in  $O(c * f(N))$  (e.g., replace  $c$  by  $1/c$ ).

So the sets  $O(f(N))$  and  $O(c * f(N))$  are, in fact, the same.

Q.E.D.

This rule can be applied across sums, by the way. If we have two functions  $f(N)$  and  $g(N)$  and two constants  $c_1$  and  $c_2$ , then it's easy to modify the above proof to show that

$$O(c_1 * f(N) + c_2 * g(N)) = O(f(N) + g(N))$$

## 3 Larger Terms Dominate a Sum

**Algebraic Rule 4:** If  $\exists n_0 \mid \forall N > n_0, f(N) \geq g(N)$ , then  $O(f(N) + g(N)) = O(f(N))$

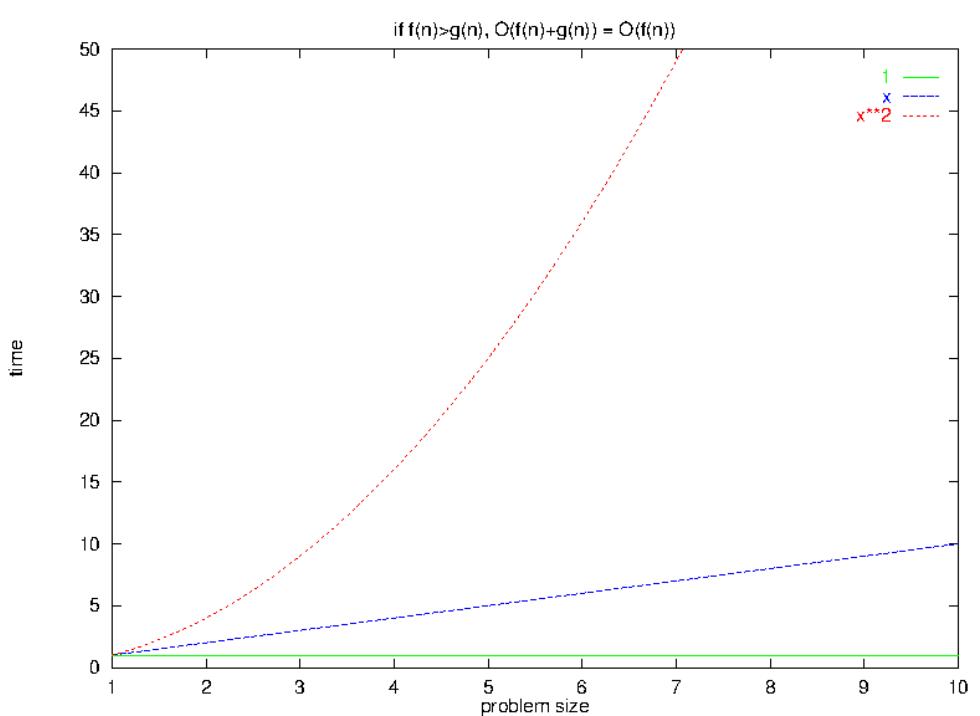
This rule states that, if we have a program that does two different things before it finishes, then for large input sets the slower of these two will dominate the overall time.

### 3.1 Intuitive Justification

To see why this might be so, suppose we have three programs that run in time  $O(1)$ ,  $O(x)$ , and  $O(x^2)$ , where  $x$  is the size of the input set.

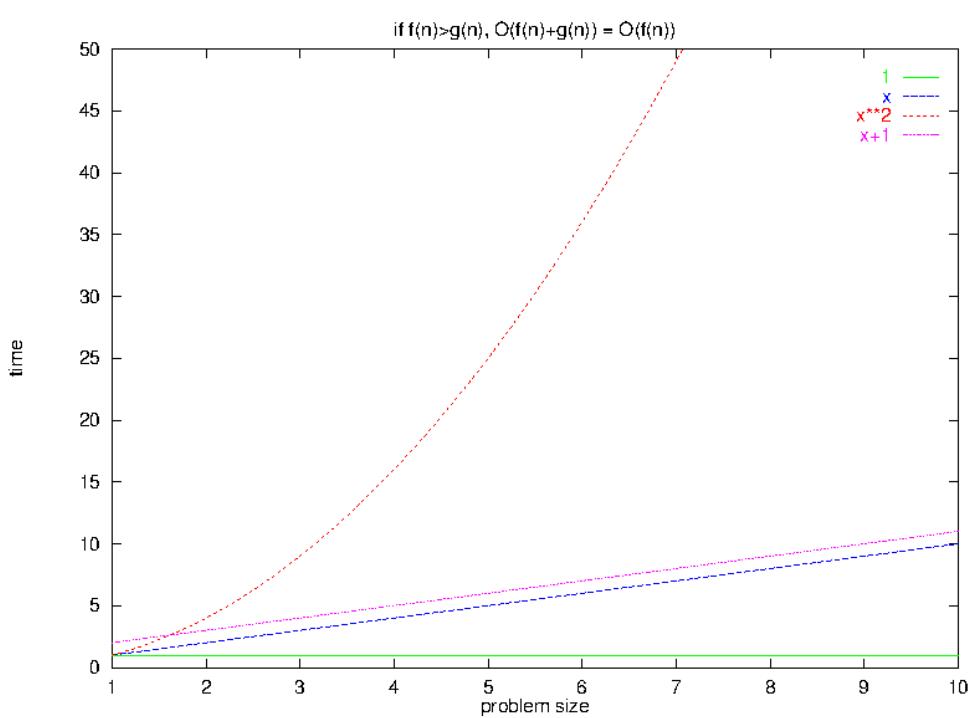
As before, we note that, for sufficiently large  $x$ , the  $O(x)$  time falls between the other two.

---

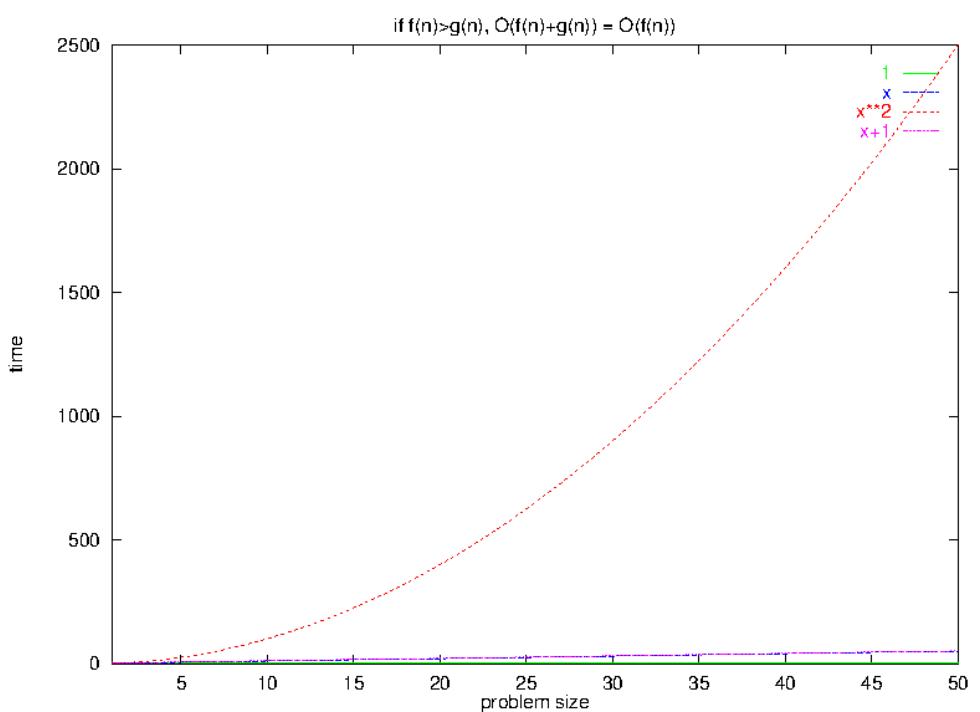


Now we add another program, that does the same work as the  $O(x)$  program plus some additional  $O(1)$  work, for a total of  $O(x + 1)$ .

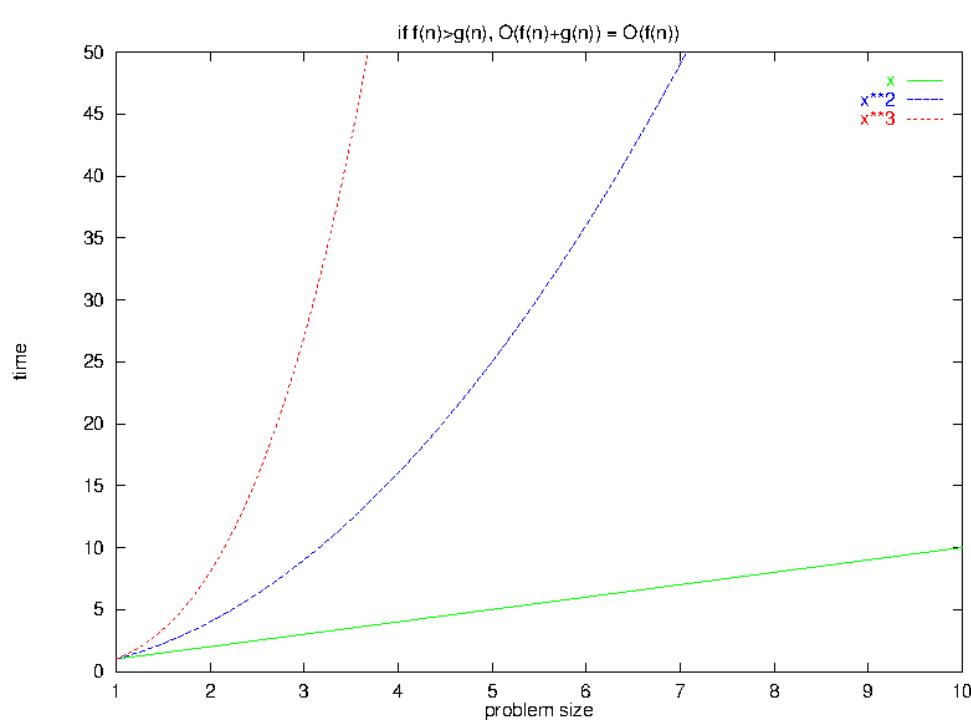
- Note that the new  $O(x + 1)$  program still runs faster than the  $O(x^2)$  program and slower than the  $O(1)$  program (for sufficiently large  $x$ ).
- 



- In fact, if we “pull back” our perspective and look at a larger range of input sizes, we see that the difference between the  $O(x)$  and  $O(x + 1)$  programs virtually disappears. (Depending upon your browser settings, you may not even be able to tell the two lines apart.)
- 



We can show that this same effect holds for even larger functions.

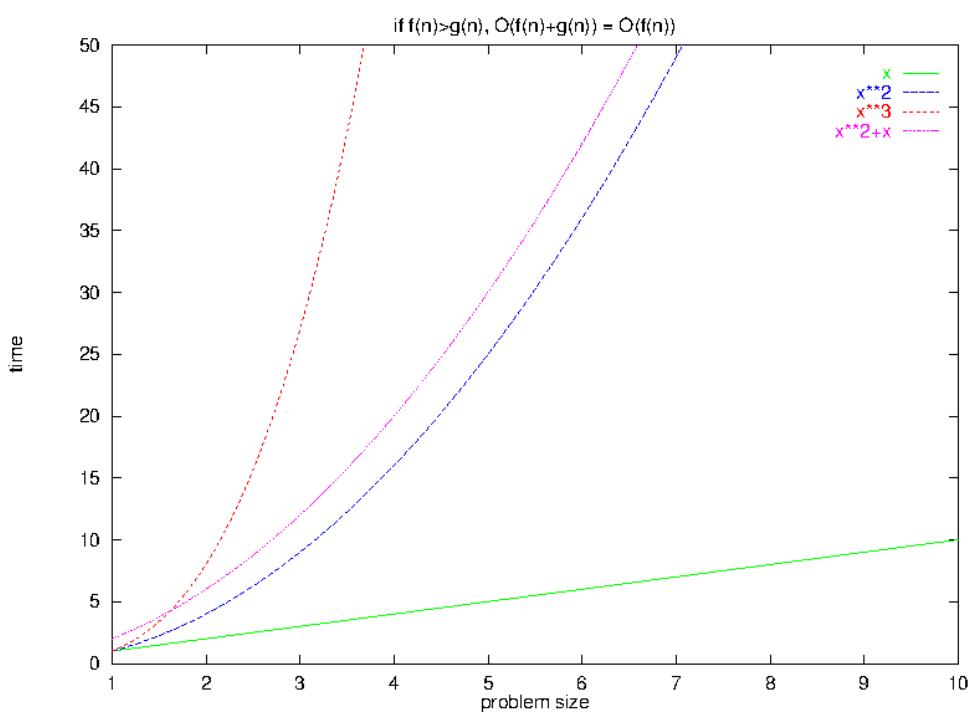


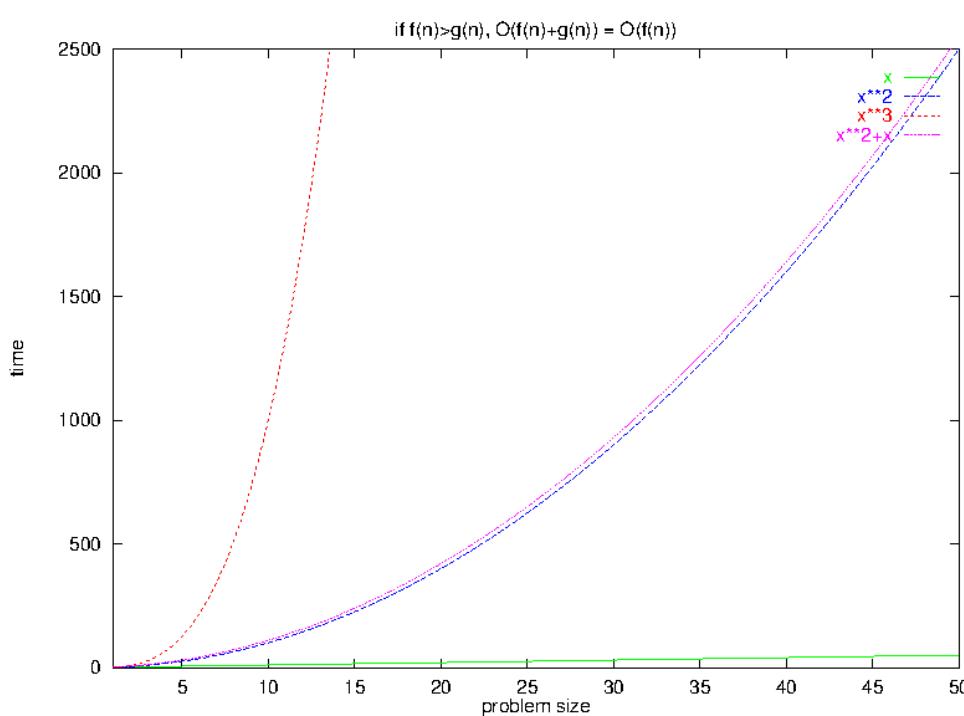
Suppose we start with three programs that run in time  $O(x)$ ,  $O(x^2)$ , and  $O(x^3)$ , where  $x$  is the size of the input set.

Notice from the plot of these three times, that, for sufficiently large  $x$ , the  $O(x^2)$  time falls between the other two.

Now add another program that runs in time  $O(x^2 + x)$ .

- Note that the new  $O(x^2 + x)$  program runs faster than the  $O(x^3)$  program and slower than the  $O(x)$  program (for sufficiently large  $x$ ).
- 





And, again, if we pull back our perspective to consider even larger input set sizes we again see that the difference between the  $O(x^2)$  and  $O(x^2 + x)$  programs virtually disappears.

In essence, adding a “lower-order” function (e.g.,  $x$ ) to a “higher-order” one (e.g.,  $x^2$ ) does not sufficiently alter the *curvature* of the new higher-order function enough to change its big-O relationship to other even higher- (e.g.,  $x^3$ ) and lower-order (e.g.,  $x$ ) functions.

### 3.2 Proof: Larger Terms Dominate a Sum

To prove that, if  $\forall n > n_0. f(n) \geq g(n)$ , then  $O(f(n) + g(n)) = O(f(n))$ , consider any program with running time  $t(n) = O(f(n) + g(n))$ .

By the definition of big-O, we know that, for some  $c$  and  $n_1$ , we have:

$$n > n_1 \Rightarrow t(n) \leq c(f(n) + g(n))$$

But assume that  $\forall n > 0, f(n) \geq g(n)$

Then we can claim that the following are also true:

$$n > \max(n_0, n_1) \Rightarrow t(n) \leq c(f(n) + g(n)) \quad (1)$$

$$n > \max(n_0, n_1) \Rightarrow f(n) > g(n) \quad (2)$$

But equation (2) suggests that we can, for those values of  $n$ , replace the  $g(n)$  by  $f(n)$  in equation (1) and, because we are replacing one quantity by an even larger one, the “ $\leq$ ” relation will still hold:

$$n > \max(n_0, n_1) \Rightarrow t(n) \leq c(f(n) + f(n))$$

$$n > \max(n_0, n_1) \Rightarrow t(n) \leq 2c * f(n)$$

But  $2c$  and  $\max(n_0, n_1)$  are just constants, so that final equation simply expresses the big-O definition:  $t(n) = O(f(n))$

Q.E.D.

## 4 Logarithms are Fast

Our final rule doesn’t come into play as often as the others, but is still useful on occasion:

**Algebraic Rule 5:**  $\forall k \geq 0, O(\log^k(n)) \subset O(n)$

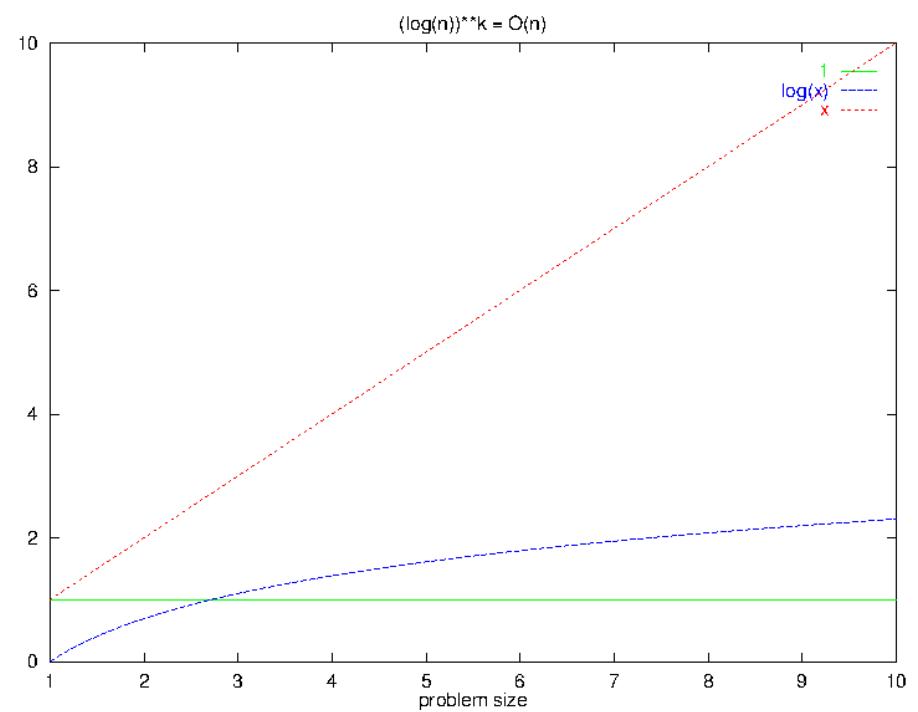
Remember that big-O expressions are really describing sets of functions, so the  $\subset$  in this rule says that [logarithms](#) are faster than linear functions.

The **best** time we could ask for from any algorithm is  $O(1)$ , meaning that the algorithm never exceeds some constant run time, no matter how large the input set.

An algorithm that works in logarithmic time is often nearly as good as one that works in constant time.

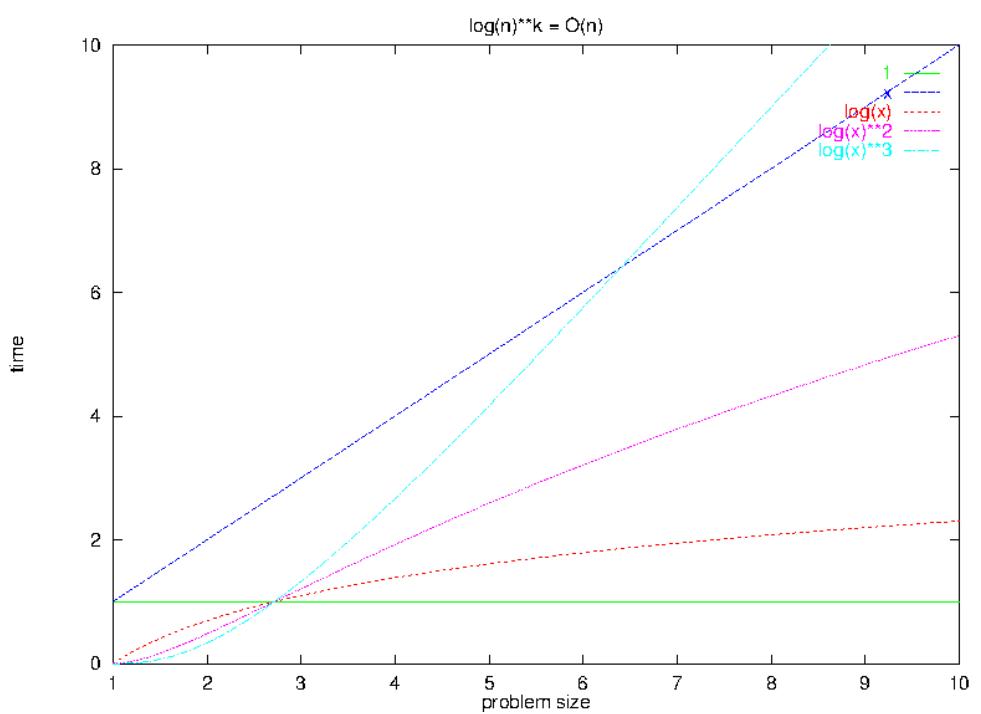
Notice how the tail of the log curve bends over nearly horizontally, almost but not quite ever becoming parallel to the flat  $O(1)$  plot.

So  $O(\log N)$  is just slightly slower, for large  $N$ , than  $O(1)$  and significantly faster than  $O(N)$ .

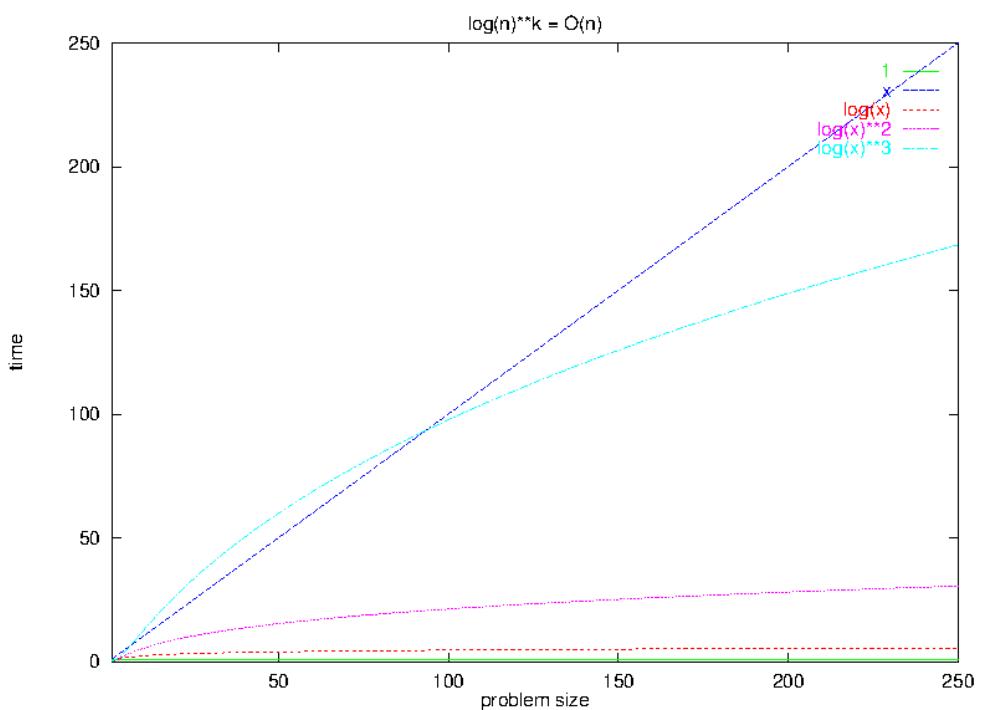


If we look at larger powers of the logarithm function, we see that they also remain slower than  $O(1)$  but faster than  $O(N)$ .

Now, in this plot, it may appear that  $O(\log^3(n))$  is slower than  $O(x)$ , ...



... but if we consider larger values of  $x$ , we see that even  $O(\log^3(n))$  eventually bends over and crosses beneath the  $O(x)$  function.



## 5 Summary of big-O Algebra

The five rules we have presented:

1.  $f(N) + g(N) \in O(f(N) + g(N))$
2.  $f(N) * g(N) \in O(f(N) * g(N))$
3.  $O(c * f(N)) = O(f(N))$
4. If  $\exists n_0 | \forall N > n_0, f(N) \geq g(N)$ , then  $O(f(N) + g(N)) = O(f(N))$
5.  $\forall k \geq 0, O(\log^k(n)) \subset O(n)$

allow us to simplify complicated analyses without resorting to a full-blown proof based upon the definition of big-O.

For example, we earlier looked at a piece of code that we felt ran in time

$$\begin{aligned} t(N) = & N^2(2t_{\text{asst}} + t_{\text{comp}} + 4t_{\text{add}} + 3t_{\text{mult}}) \\ & + N(3t_{\text{asst}} + 2t_{\text{comp}} + 2t_{\text{add}} + t_{\text{mult}}) \\ & + 1(t_{\text{asst}} + t_{\text{comp}}) \end{aligned}$$

Since this is the exact running time, it is also an upper bound. Therefore,

$$\begin{aligned} t(N) \in & O(N^2(2t_{\text{asst}} + t_{\text{comp}} + 4t_{\text{add}} + 3t_{\text{mult}})) \\ & + O(N(3t_{\text{asst}} + 2t_{\text{comp}} + 2t_{\text{add}} + t_{\text{mult}})) \\ & + O(1(t_{\text{asst}} + t_{\text{comp}})) \end{aligned}$$

By rule 4, the higher-order  $N^2$  term will dominate this sum:

$$t(N) \in O(N^2(2t_{\text{asst}} + t_{\text{comp}} + 4t_{\text{add}} + 3t_{\text{mult}}))$$

By rule 3, we can discard the constant multipliers:

$$t(N) \in O(N^2)$$

This is, I hope you'll agree, much simpler than the proof we engaged in when we first analyzed that code.

### 5.1 The Tao of N

Finally, keep in mind that  $N$  is merely a placeholder here for what may be an expression involving multiple measures of the input set size. Only the last of these rules is limited to a single variable (because the expression “ $\log(n)$ ” only makes sense if  $n$  is a single quantity, although it can still be any single expression, e.g.,  $\log(n^3 + 1/n)$ ).

For example, if we were presented with a function that was  $O(30.0x^2 + 15.0y + 2x)$ , we could simplify it as follows:

$$\begin{aligned} O(30.0x^2 + 15.0y + 2x) \\ = O(30.0x^2 + 15.0y) &\quad (\text{by rule 4}) \\ = O(x^2 + y) &\quad (\text{by rule 3}) \end{aligned}$$

and would have to stop there. Although  $x^2$  has a larger exponent than  $y$ , we can't assume that it dominates  $y$  unless we have some other information about the relative sizes of  $x$  and  $y$ .

If someone were to come by later and inform us, “Oh, by the way, it's always true that  $y \leq x$ ”, then we could indeed simplify the above to  $O(x^2)$ . On the other hand, if that same mysterious font of information were to instead say, “Oops, my mistake. Actually  $y \geq x^3$ ”, then we would simplify  $O(x^2 + y)$  to  $O(y)$ .

## 6 Always Simplify!

Whenever you analyze an algorithm in an assignment, quiz or exam in this course, you should employ these algebraic rules to present your answer in the simplest form possible. If you don't, your answer will be considered wrong!

# Worst-Case Complexity Analysis

Steven J. Zeil

Last modified: Feb 12, 2020

## Contents:

### [1 Some Overall Guidelines](#)

#### [1.1 It All Boils Down to Addition](#)

#### [1.2 Nothing Happens in O\(0\) Time](#)

#### [1.3 All of Your Variables Must be Defined](#)

#### [1.4 Complexity is Written in Terms of the Inputs](#)

#### [1.5 The Complexity of Any Block of Code Must be Numeric](#)

#### [1.6 Surprises Demand Explanation](#)

### [2 The Complexity of Expression Evaluation](#)

#### [2.1 Arithmetic & Relational Operations on Primitive Types are O\(1\)](#)

#### [2.2 Assignments of Primitive Types are O\(1\)](#)

#### [2.3 Basic Address Calculations are O\(1\)](#)

#### [2.4 Function Calls](#)

#### [2.5 Annotating Expression Statements](#)

### [3 The Complexity of Compound Statements](#)

#### [3.1 Sequences of Statements](#)

#### [3.2 Conditional statements](#)

#### [3.3 Loops](#)

#### [3.4 Recursive Functions](#)

### [4 The Importance of Understanding](#)

Now that we have seen how to manipulate big-O expressions, the next step is to figure out where we get them from in the first place. We are going to consider common programming language constructs and how we can determine the worst-case behavior for each of them, starting with simple expressions and building up to entire function bodies.

As we do this, we will be looking at two things:

1. How do we derive the worst-case expression for each kind of programming language construct?
2. How can we annotate our code to express those worst-case behaviors in a way that amounts to a proof of your results?

A worst-case analysis is really a kind of mathematical proof, and people's styles of writing mathematical proofs vary so much that the results are often unreadable by anyone other than the author.

So in this lesson I am going to show you my own *copy-and-paste style* for conducting and documenting a worst-case analysis. This is the technique that I will use through this course whenever I am giving an analysis. And it is also the technique that I will expect you to use when you do assignments in which I ask you to show me your work on the algorithm analysis.

This technique takes advantage of the fact that these days we don't usually write down our algorithms using paper and pencil. Instead we use text editors which are very good in doing copy and paste. Consequently the technique relies on your ability to make copies of an algorithm over and over again, making small changes to the algorithm at each stage that reflect the analysis that you have actually done.

## 1 Some Overall Guidelines

First, some basic observations.

### 1.1 It All Boils Down to Addition

If you want to know how much time a complicated process takes, you figure that out by *adding up* the times of its various components.

Now, we will have a lot of situations where we use multiplication and other operations, but those are always "special cases". For example, if I tell you that a certain component of a program takes time  $t_0$  and we later discover that this component is inside a loop that repeats 1000 times, we may write the total as  $1000t_0$ , but that's only because

$$1000t_0 = \sum_{i=1}^{1000} t_0 = t_0 + t_0 + \dots \text{ (997 times)} \dots + t_0$$

It's just a simplification for a lot of additions.

When in doubt, just add things up.

### 1.2 Nothing Happens in O(0) Time

Even doing nothing takes time. “Not doing” a block of code still requires at least a clock cycle to move the address counter forward to the next block of code. So, even doing nothing is  $O(1)$ , not  $O(0)$ .

So if your math comes out to zero (or even worse, negative time values), it’s time to go back and check your math.

## 1.3 All of Your Variables Must be Defined

In math, as in programming, all of Your variables must be declared/defined before you can use them.

If you reach a conclusion that a certain C++ function is  $O(N + M^2)$ , then  $N$  and  $M$  have to actually *mean* something. That meaning may come because they are actual variables in the code:

```
void foo(double[] array, int N, int M) // O(N + M^2)
```

or because, in the style of mathematical proofs everywhere, we have supplied our own definition, e.g.,

```
void foo (vector<double> v)
Let N be the number of elements in the vector v and let M
be the number of those elements that are greater than 1.0.
```

But, one way or another, every symbol in your big-O expressions needs to be properly declared.

- Don’t fall in love with “n”. Even though our definition of complexity talks about  $\bar{n}$ , that doesn’t mean that there has to be a variable named “n” in the code, or that “n” means anything at all in a specific analysis problem.
- And, don’t forget that , in mathematics as in C++, n and N are *not* the same variable.

The possibility that we may use variables from the code in our mathematical description of complexity leads to a closely related rule:

## 1.4 Complexity is Written in Terms of the Inputs

The complexity of a block of code must be a function of the inputs (only!) to that block.

If you are looking at a block of code like this:

```
int x = y + z;
int k = foo (x, y, z);
bar (k);
```

your big-O expression to describe this code *cannot* involve  $k$ . That’s because  $k$  does not even exist when we start the block of code. And, in many cases, e.g.:

```
void baz(int y, int z)
{
    int x = y + z;
    int k = foo (x, y, z);
    bar (k);
}
```

$k$  will not exist after we have completed the block of code. In this case, because  $k$  is computed from  $x$ ,  $y$ , and  $z$ , and  $x$  was computed from  $y$  and  $z$ , we would need to say, instead of  $O(k)$ , the complexity could be  $O(\text{foo}(y+z, y, z))$ .

## 1.5 The Complexity of Any Block of Code Must be Numeric

Our definition of  $t \in O(f(\bar{n}))$  says that “...  $t < c * f(\bar{n})$  ...”.

The “ $<$ ” and multiplication by a constant  $c$  only makes sense if  $f(\bar{n})$  is a numeric quantity of some kind. So if we were looking at

```
void foo (vector<string> v, string str)
```

it might be valid to say that the complexity of `foo` is  $O(1)$ ,  $O(v.size())$ ,  $O(N)$  where  $N$  is the number of vowels in `str`, etc., because all of those are numeric quantities. But you cannot possibly say the complexity is  $O(v)$  or  $O(str)$  because `v` and `str` are not numeric. The can’t be multiplied by a constant  $c$  and cannot be compared to a numeric time value.

## 1.6 Surprises Demand Explanation

In many scientific and engineering fields, students are taught that any complicated calculations should be subjected to “sanity checks” that may

- check parts of the calculation for internal consistency
  - Many of the rules we’ve already been discussing in this section would fall into this category
- compare the answer against a simpler, rough calculation (often called a “back-of-the-envelope” method, after the stereotypical image of an engineer scratching a few numbers on a handy piece of paper and suddenly announcing an approximate answer to the problem that everyone else has been struggling with).

Back-of-the-envelope calculations are not a substitute for doing the “real” problem solution, but they do provide an important function as a sanity check. It’s entirely possible that, when we do the real calculation, we will come up with a different net answer than the back-of-the-envelope answer. If so, it’s worthwhile to take a moment to understand *why* the two answers are different.

1. Maybe we made a mistake in one computation or the other. If so, we need to find that mistake and correct.
2. On the other hand, maybe we did everything correctly, and there is some deep feature of this problem that isn’t obvious to a quick back-of-the-envelope approach, something that accounts for the surprise of the final result of the full calculation.

If so, we need to *identify* and *explain* that surprise before we can have full confidence in our final answer.

For example, most of the time of typical code is spent in loops, and so loops tend to dominate the worst-case complexity. A reasonable back-of-the-envelope technique is to

- Look at the most deeply nested loops in the algorithm. If the loops are nested  $k$  deep, and each appears to execute up to  $N$  times, then  $N^k$  is a reasonable first guess for the algorithm complexity.

It’s only a guess, but if you come up with something else, you should be sure that you understand where the difference is coming from.

## 2 The Complexity of Expression Evaluation

Most of the actual executable part of a typical C++ program is made up of expression evaluation. Even many things that we refer to as statements, such as assignment statements or procedure/function calls, e.g.

```
x = y + z;  
doSomethingWith(x, y, z);
```

are just larger expressions in C++.

### 2.1 Arithmetic & Relational Operations on Primitive Types are O(1)

If we restrict our attention to the primitive types (integers, floating point numbers, booleans, characters, pointers, and references, then all operators provided by the language are O(1)).

Think about something like:

```
int x = ...  
int y = ...  
:  
x + y // What is the complexity of this?
```

The time required to compute the sum is independent of how large the values in  $x$  and  $y$  might be. It takes no more time to compute  $1000000 + 987654321$  than it takes to compute  $0 + 1$ . It might take a bit longer to add a pair of 64-bit long integers than a pair of 8-bit short integers, but the available bit widths is fixed by the environment (CPU and compiler) and, for each available choice of widths, is a constant no matter what the actual numbers involved might be.

The same is true for subtraction, multiplication, boolean operations, relational operators, etc.

When we combine these operators into more complicated expressions, we are simply instructing the machine to do one of these operations, then another, and so on. For example, if we had integer variables  $x$ ,  $y$ , and  $z$ , then an expression like  $x + y * z$  is  $O(1)$ . There are two operations in that expression, a multiplication and an addition, and those operations each run in  $O(1)$  time (i.e., they each take a constant amount of time regardless of the amount of data being manipulated by the program). So the expression has complexity  $O(c_+ + c_*)$ , which, by our rules for big-O algebra, simplifies to  $O(1)$ .

On the other hand the complexity of an expression like  $\text{foo}(i) + \text{bar}(j)$  would be the sum of the complexities of the calls to `foo` and to `bar`. We’ll talk about how those calls are evaluated shortly.

**Question:** What is the worst-case complexity of the expression  $(x < 100) \& (y + 1 > 0)$ ?

*Click to reveal* +

We do have to be a little bit cautious here. Because C++ allows programmers to *overload* operators and to write their own code for many common operators, we can’t say that **all** uses of the plus sign are  $O(1)$ .

For example, `+` applied to strings denotes the concatenation of two strings, and intuitively you can guess that the amount of time required to join two strings of a million characters each will be about a hundred thousand times longer than the amount of time required to join two strings of 10 characters each.

But `string` is not a primitive type in C++, and that’s why I said that “arithmetic & relational operations **on primitive types** are  $O(1)$ .”

This expression feeds the results of one operator into others, but that really doesn't matter. Remember "when in doubt, just add things up". We have in this expression an addition, two relational operators comparing integers, and a single boolean operating on a pair of booleans (the results from the two relational sub-expressions). Every one of those is  $O(1)$ , so the total is

$$\begin{aligned} & t_{\text{add}} + t_{\text{less}} + t_{\text{greater}} + t_{\text{and}} \\ &= O(1) + O(1) + O(1) + O(1) = O(1+1+1+1) \\ &= O(4) \\ &= O(1) \end{aligned}$$

## 2.2 Assignments of Primitive Types are $O(1)$

Really, assignments of any data type that has a fixed number of bytes (ints, doubles, chars, etc., but not strings, vectors, etc.) are  $O(1)$ .

The explanation is that it takes the same amount of time to copy a byte that contains 127 as a byte that contains 0. The actual value stored there is irrelevant. The number of bytes making up the total does matter, but if that is fixed, then copying that number of bytes takes the same amount of time for all data types of that size.

- Again, this does not mean that assignment of non-primitive types is always  $O(1)$ . C++ allows programmers to write their own assignment operators, and until we actually examine the algorithm used for such an operator, we can't guess at what its complexity would be.

## 2.3 Basic Address Calculations are $O(1)$

We have [previously](#) discussed the fact that array indexing is just an address calculation and shown that it boils down to an integer multiplication and addition. So

- Array indexing  $a[i]$  is  $O(1)$ .

- That doesn't mean that everything we *do* with array elements will be  $O(1)$ . For example, if I write

```
int array[100]; /*...*/ a[i] = a[i-1]; // O(1)
```

the whole expression, including the assignment, is composed of  $O(1)$  components, so the whole thing is  $O(1)$ .

But if I had

```
string array[100]; /*...*/ a[i] = a[i-1]; // Not O(1)
```

the final statement is probably not  $O(1)$ . But that's because assignment of strings  $s = t$  is generally  $O(t.length())$ . It has nothing to do with the time required to compute the addresses  $a[i]$  and  $a[i-1]$ .

Similarly,

- Pointer dereferencing  $*ptr$  is  $O(1)$ . This operation simply grabs an address out of memory and waits to see what we want to do with it.

Again, this does not mean that the uses we make of those addresses will be  $O(1)$ . If I write, for example,  $*ptr1 = *ptr2$ , I really need to focus on the complexity of assignment for whatever data type those pointers point to.

- Struct/class member selection `myStruct.member` is  $O(1)$ . The '.' is just an address calculation in which the base address of the entire `myStruct` structure is added together with a compiler-assigned integer offset for the beginning of `member` within each structure of that same type. It's just an addition.
- The combined pointer dereference and member select `ptr->member` is  $O(1)$ .

It's just the combination of two  $O(1)$  components.

## 2.4 Function Calls

So far, everything we might write into an expression has been pretty quick. That stops once we start to consider function calls (including programmer-supplied operator overloads).

Addition is still king. When we have an expression like

```
int k = foo(x, y, z);
```

we still add the complexity of the function call to the complexity of the other operations (in this case, integer assignment). But what is the complexity of the function call?

To answer that, someone needs to have analyzed the body of that function and figured out its complexity.

- Sometimes someone else will have already done that for us. In particular, many functions in the `std` library are required by the [C++ standard](#) to have a certain worst-case complexity. So in that case we can simply look it up.
- If no one else has done the job for us, we will have to set our current problem aside for the moment and analyze the function body ourselves. A fair number of functions form the `std` library are so simple that, by a few weeks from now, you'll be able to analyze them at a glance. Other functions may require a substantial effort.

For now, let's assume that, one way or the other, we have found the complexity of a function that we are calling. What does that mean? What will we actually "know"?

Remember that “complexity is written in terms of the inputs”. So what we will get is a big-O expression written in terms of the possible inputs to the function.

For example, the worst-case complexity of the assignment operator for strings, `const string& operator= (const string& t)`, is  $O(t.length())$ .

- Take note that `operator=` is a *member function* of class `string`. Like all member functions, it has a [hidden parameter named “this”](#). So, in general, “this” is an input to member functions and we might expect a member function’s complexity to include either explicit or implicit references to `this->`.

Now the input variables that are listed in the complexity of a function will be the [formal parameters](#) of the function. We will need to replace those by the actual parameters that we actually use in the function call.

For example, earlier we looked at

```
string array[100];
⋮
a[i] = a[i-1]; // Not O(1)
```

What is the complexity of the expression in the last line of code there?

- We have said that the complexity of the address calculations `a[i]` and `a[i-1]` are  $O(1)$ .
- We have said that the complexity of the function `const string& operator= (const string& t)`, is  $O(t.length())$ .
  - But `t` is a formal parameter, for which this particular line of code substitutes `a[i-1]`.
  - Therefore the complexity of that function call is  $O(a[i-1].length())$ .

The total complexity of that line of code, therefore, is  $O(1) + O(1) + O(a[i-1].length())$ , which simplifies to  $O(a[i-1].length())$ .

**Question:** Let  $L$  denote the length of the longest string in `array`. Would it then be valid to state that the above line of code is  $O(L)$ ?

Click to reveal +

Yes, sort of. According to the definition of  $L$ ,

$a[i-1].length \leq L$

so

$O(a[i-1].length) \subseteq O(L)$

$O(L)$  is a looser bound than  $O(a[i-1].length)$ , and normally we would prefer to keep the bound as tight as possible. But, looking ahead a bit, if we discovered that we had this block of code inside a loop

```
string array[100];
:
for (int i = N; i > 0, --i)
    a[i] = a[i-1]; // O(a[i-i].length())
```

so that we knew that  $i-1$  would eventually index every position currently in use in the array, we might well feel justified in using the simpler-though-looser bound  $O(L)$ .

- Keep in mind that, because we are looking at worst-case behavior, a “worst case” for that loop might be when all of the strings are of length  $L$ .

## 2.5 Annotating Expression Statements

As promised at the beginning of this lesson, we will be recording our conclusions about the complexity of the various components of the code by annotating the code itself with comments.

For statement-level expressions (e.g., assignments or void function calls) the annotation is very simple. We simply write the big-O expression at the end of the line as a  $//$  comment. In fact, you have already seen me do this:

```
int array[100];
:
a[i] = a[i-1]; // O(1)
```

and

```
string array[100];
:
for (int i = N; i > 0, --i)
    a[i] = a[i-1]; // O(a[i-i].length())
```

Now, typing mathematics in plain text like this has some limitations that we will need to work around.

- The most common is the need to use superscripts to denote raising something to a power. We will use the caret ( $\wedge$ ) character for that purpose, for example, typing  $O(N^2)$  as  $O(N\wedge 2)$ .  
Not only does  $\wedge$  suggest raising something up, but in some programming languages, it is actually used as the “raise to a power” operator.
- Less often, we will want to denote subscripts. We will use the underscore ( $_$ ) character for that purpose. For example, we would type  $O(x_0 + x_1)$  as  $O(x_0 + x_1)$ .

As we conduct an analysis of complexity, we will wind up with these kinds of annotations sprinkled throughout the code, e.g.,

```
template <typename Comparable>
int seqOrderedSearch(const Comparable list[], int listLength,
                     Comparable key)
{
    int loc = 0; // O(1)

    while (loc < listLength && list[loc] < key)
    {
        ++loc; // O(1)
    }
    return loc; // O(1)
}
```

(A return statement is essentially a copy or assignment to a special variable in the caller, so for primitive types it, like assignment, will be  $O(1)$ .)

## 3 The Complexity of Compound Statements

Of course, expression evaluation and assignment may account for the bulk of single statements in a program, but what allows us to combine single statements into a full algorithm is the idea of compound statements: sequences (a.k.a. blocks), conditionals (e.g., `if`), and loops.

Compound statements are single statements that are built around one or more simpler component statements. For example, a while loop is build around a loop body, which is itself a statement. Some of the components of a compound statement may also be smaller compound statements, as when we nest loops inside one another.

There is a general observation we can make about analyzing compound statements. In order to compute an upper bound on the time required by a compound statement, we will almost always need to know the bounds on the times of its components. Consequently

We usually analyze compound statements in an inside-out fashion, starting with the most deeply nested components and working our way outward from there.

## 3.1 Sequences of Statements

The simplest form of compound statement is the sequence or block, usually written in C++ between {} brackets. Examples include function bodies, loop bodies, and the “then” and “else” parts of if statements.

A sequence tells the machine to process statements one after the other. So,

The time for a sequence of statements is the sum of the times of the individual statements.

More formally, for a sequence of statements  $s_1, s_2, \dots, s_k$

$$t_{\text{seq}} = \sum_{i=1}^k t_{s_i}$$

This rule applies when we have a block of statements in a straight line – no way to jump into the middle of the block, no way to jump out. When we have sequential arrangements of statements like that, then the complexity of the sequence as a whole is sum of the complexities of the statements

### Example 1: Statement Sequence

```
{  
    a[10*i+j] = foo(i,j);  
    bar(a[10*i+j], i, j);  
}
```

In the code shown here, if I look at the first assignment statement, its complexity depends on the complexity of the body of the function foo.

Let's just say for the sake of example that the function  $\text{foo}(x, y)$  has a body of complexity  $O(x^2)$ . Substituting the actual parameter  $i$  from the call for the formal parameter  $x$ , we conclude that the first statement has complexity  $O(i^2)$ .

We annotate the first statement accordingly:

```
{  
    a[10*i+j] = foo(i,j); // O(i^2)  
    bar(a[10*i+j], i, j);  
}
```

The second statement involves a call to bar. For the sake of example, let's assume that  $\text{bar}(x, y, z)$  has complexity of  $O(y + z)$ . Again, to get the complexity of the call we must substitute the actual parameters for the formals, and doing so indicates that the second statement has complexity  $O(i + j)$ .

We would record our conclusion by annotating that statement:

```
{  
    a[10*i+j] = foo(i,j); // O(i^2)  
    bar(a[10*i+j], i, j); // O(i + j)  
}
```

That would mean that the complexity for the sequence of statements consisting these two individual statements would be sum of those two complexities which would be  $O(i^2 + i + j)$  and since  $i^2$  is greater than  $i$ , this simplifies to  $O(i^2 + j)$ .

### 3.1.1 Annotating Statement Sequences

In some cases we won't bother annotating the total complexity of a statement sequence, particularly when it is short or the answer is obvious (e.g., because all of the component statements have the same complexity).

When we do want to record the total for a statement sequence, we will do so by adding an annotation // total: O(...) at the top of the sequence.

In our example above, we would record our conclusion like this:

```
{ // total: O(i^2 + j)  
    a[10*i+j] = foo(i,j); // O(i^2)  
    bar(a[10*i+j], i, j); // O(i + j)  
}
```

As we proceed through a proof, we may decide that we don't need the details inside the sequence any more. In that case, we may opt to make a separate copy of the code, replace the entire sequence with its total complexity, and add a brief text in between explaining what we are doing, e.g.,

```

{ // total: O(i^2 + j)
  a(10*i+j) = foo(i,j); // O(i^2)
  bar(a(10*i+j), i, j); // O(i + j)
}

Collapsing that block,

{
  // O(i^2 + j)
}

```

## 3.2 Conditional statements

When we have an `if` statement, we know that we will execute either the `then` part *or* the `else` part, but not both. Since we could take either one, the worst case time for the `if` is the slower of the two possibilities.

$$t_{\text{if}} = t_{\text{condition}} + \max(t_{\text{then}}, t_{\text{else}})$$

- A missing `else` clause (or, for that matter, any “empty” statement list) is  $O(1)$ .

### 3.2.1 Annotating if statements

In analyzing an `if` statement, we will start by getting the complexity of its condition expression, its “then” part, and its “else” part.

We will record the complexity of the condition by adding an annotation of the form `cond: O(...)` in a comment on the line(s) where the condition is written.

Optionally, we may record the complexity of the “then” part and the “else” part in the same place as `then: O(...)` and `else: O(...)` annotations. This is optional, since, if we have clearly annotated the “then” and “else” statements, we should be able to get this information at a glance anyway.

The total complexity is recorded in the same general location using the same `total: O(...)` mark that we used for sequences. (In general, we use the `total:` mark for the total complexity of a compound statement.).

#### Example 2: Analyzing an If

Suppose that we have the code

```

if (i < j)
{ // total: O(i^2 + j)
  a(10*i+j) = foo(i,j); // O(i^2)
  bar(a(10*i+j), i, j); // O(i + j)
}
else
  bar(0, i, j);

```

for which we have already analyzed the “then” part.

Before we can analyze the entire `if` statement, we must complete the analysis of its component pieces. First, we can do the “else” part. In the earlier example, we established that the complexity of `bar` was the sum of its final two parameters:

```

if (i < j)
{ // total: O(i^2 + j)
  a(10*i+j) = foo(i,j); // O(i^2)
  bar(a(10*i+j), i, j); // O(i + j)
}
else
  bar(0, i, j); // O(i + j)

```

We can, if we wish, collapse the “then” part:

```

if (i < j)
{
  // O(i^2 + j)
}
else
  bar(0, i, j); // O(i + j)

```

and then turn our attention to the condition. Relational operations on integers are  $O(1)$ :

```

if (i < j) // cond: O(1)
{
  // O(i^2 + j)
}
else
  bar(0, i, j); // O(i + j)

```

We are now ready to tackle the total complexity for the `if`. According to our rule above,

$$t_{\text{if}} = t_{\text{condition}} + \max(t_{\text{then}}, t_{\text{else}}) \\ \in O(1) + \max(O(i^2 + j), O(i + j))$$

For sufficiently large values of  $i$  and  $j$ ,  $i^2 + j > i + j$ , so

$$t_{\text{if}} \in O(1) + \max O(i^2 + j), O(i + j) \\ = O(1) + O(i^2 + j) \\ = O(1 + i^2 + j) \\ = O(i^2 + j)$$

and so

```
if (i < j) // cond: O(1) total: O(i^2 + j)
{
    // O(i^2 + j)
}
else
    bar(0, i, j); // O(i + j)
```

If we were analyzing a still larger block of code within which this `if` were a single component, we might, in a separate step, collapse it, replacing it by its total complexity annotation.

### 3.2.2 Special Cases

The rule we have given above for the time of an `if` is a general rule. It's certainly correct, but is not always going to give us the tightest bound we could get.

Sometimes we have extra knowledge about the behavior of the program that we can exploit to obtain a tighter bound. For example, if we had

```
i = j + 1;
: (no changes to i or j)
if (i < j) // cond: O(1)
{
    O(i^2 + j)
}
else
    bar(0, i, j); // O(i + j)
```

we would be safe in concluding that the “then” part would never be executed, and that a better description of the behavior of this particular `if` would be

$$t_{\text{if}} = t_{\text{condition}} + t_{\text{else}}$$

in which case we would conclude that the total complexity of this `if` would be  $O(i + j)$  instead of  $O(i^2 + j)$ .

Now, you might wonder if that sort of thing would ever happen in real programming. Surprisingly, it does happen.

- Programmers sometimes engage in [defensive programming](#), writing conditions into code to defend against possible coding errors.
- Coding standards, particularly when enforced by automatic tools, can force programmers to insert code that they believe will never be executed, just to get the standards enforcer off their backs. For example, an automated standards tool might flag a line of code

```
a[i] = 0;
```

as a “possible out-of-range error”. To get rid of that message, the programmer might write

```
if ((i >= 0) && (i < N))
    a[i] = 0;
else
    exit_with_message("This should never happen.");
```

- When we put `if` statements inside loops, we will sometimes have enough knowledge of how the algorithm works to say things like “if this loop executes  $N$  times, then on  $\sqrt{N}$  iterations the `if` inside will take the ‘else’ branch, but on  $N - \sqrt{N}$  iterations it will take the ‘then’ branch.”

If we have that kind of knowledge, we are allowed, indeed, urged to exploit it when doing so give us a tighter bound.

Complexity analysis is not a recipe to be followed exactly the same way every time we apply it. It is, like programming itself, a creative mathematical process that calls for us to use the knowledge at hand to get the best result we can.

We have general rules for compound statements that will give us a valid upper bound when applied “inside-out” from the most deeply nested statements. But sometimes we can get tighter bounds by applying our knowledge and understanding of the code to identify special cases.

## 3.3 Loops

When we analyze a loop, we need to add up the time required for all of its iterations.

When we encounter a loop of the form

```
while (condition)
{
    body
}
```

we can say that

$$t_{\text{while}} = \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{body}}) + t_{\text{condition}}^{\text{final}}$$

where

- $t_{\text{condition}}$  is the time to evaluate the loop condition,
- $t_{\text{condition}}^{\text{final}}$  is the time to evaluate the loop condition the final time (when we exit the loop), and
- $t_{\text{body}}$  is the time required to do the loop body.

The summation is taken over all iterations of the loop. The number of iterations is likely to depend on the input size  $\bar{n}$  and, in many cases, the particular set of inputs that constitutes the worst case input over all possible inputs of that size. Similarly, the values of  $t_{\text{condition}}$ ,  $t_{\text{condition}}^{\text{final}}$ , and  $t_{\text{body}}$  may depend on the size of the input, on the particular values of the worst case input, and on which iteration we are performing.

If we encounter a `for` loop:

```
for (init; condition; increment)
{
    body
}
```

we know that this is equivalent to

```
init;
while (condition)
{
    body
    increment
}
```

and so we can substitute into our while-loop rule to get

$$t_{\text{for}} = t_{\text{init}} + \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{increment}} + t_{\text{body}}) + t_{\text{condition}}^{\text{final}}$$

where

- $t_{\text{init}}$  is the time required to do the loop initialization,
- $t_{\text{condition}}$  is the time to evaluate the loop condition,
- $t_{\text{condition}}^{\text{final}}$  is the time to evaluate the loop condition the final time (when we exit the loop), and
- $t_{\text{body}}$  is the time required to do the loop body.

### 3.3.1 Annotating Loops

Again, we can't analyze an entire compound statement until we have analyzed its components. For loops, this means we will need to know the complexity of the body, the condition, and, for `for` loops, the initialization and increment portions.

We will annotate while loops like this:

```
while (condition) // cond: O(...) body: O(...) #: ... total: O(...)

{
    body
}
```

- The `cond:` part is the complexity of the condition.
- The `body:` part is the complexity of the loop body, and may be omitted if it is easily read by simply looking down a couple of lines to the annotated body.
- The `#:` is the number of iterations in the worst case. (This is usually not a big-O expression. It can be, but most of the time we can give an exact expression for this.)
- The `total:` is, again, the overall complexity of the loop, combining all of its components and summing over all of the iterations.

We will annotate for loops like this:

```
// init: O(...)
for (init; condition; increment) // cond: O(...) body: O(...) incr: O(...) #: ... total: O(...)

{
```

```
    body
}
```

which adds the two for-loop specific components:

- `init`: is the complexity of the initialization. This is written before the loop to highlight the fact that it happens only once. It can be omitted if it is  $O(1)$ .
- `incr`: part is the complexity of the increment. It can be omitted if it is  $O(1)$ .

### 3.3.2 Examples

#### Example 3: A simple loop

```
for (int j = 0; j < i; ++j)
  a[i+10*j] = i + j;
```

For the loop shown here, we can see that the complexity of the loop body is  $O(1)$  (by our assignment statement rule).

```
for (int j = 0; j < i; ++j)
  a[i+10*j] = i + j; // O(1)
```

Similarly, the time to do the loop initialization, to evaluate the loop condition ( $j < i$ ), and to increment  $j$  are also  $O(1)$ .

```
// init: O(1)
for (int j = 0; j < i; ++j) // cond: O(1)  incr: O(1)
  a[i+10*j] = i + j; // O(1)
```

How many times does this loop execute in the worst case? Well, as is often the case with `for` loops, that's actually pretty easy to read out of the loop header:

```
// init: O(1)
for (int j = 0; j < i; ++j) // cond: O(1)  incr: O(1)  #: i
  a[i+10*j] = i + j; // O(1)
```

And now we are in a position to apply our rule for for-loops:

$$\begin{aligned} t_{\text{loop}} &= O \left( t_{\text{init}} + \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{increment}} + t_{\text{body}}) + t_{\text{final}}^{\text{condition}} \right) \\ &\in O(1) + \sum_{j=0}^{i-1} (O(1) + O(1) + O(1)) + O(1) \\ &= O(1) + \sum_{j=0}^{i-1} O(1 + 1 + 1) + O(1) \\ &= O(1) + \sum_{j=0}^{i-1} O(1) + O(1) \\ &= O(1) + O \left( \sum_{j=0}^{i-1} 1 \right) + O(1) \\ &= O(1) + O(i) + O(1) \\ &= O(1 + i + 1) \\ &= O(i) \end{aligned}$$

And we write our conclusion that

```
// init: O(1)
for (int j = 0; j < i; ++j) // cond: O(1)  incr: O(1)  #: i  total: O(i)
  a[i+10*j] = i + j; // O(1)
```

We could have reached the same conclusion by a slightly different path:

$$\begin{aligned} t_{\text{loop}} &= t_{\text{init}} + \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{increment}} + t_{\text{body}}) + t_{\text{final}}^{\text{condition}} \\ &\in O(1) + \sum_{j=0}^{i-1} (O(1) + O(1) + O(1)) + O(1) \end{aligned}$$

Now, just looking at the values of the times inside the summation, we can observe that none of them depend in the value of  $j$  (the particular iteration that we are in). A general special case for summations is that

If  $f(\dots)$  is independent of  $i$ , then  $\sum_{i=1}^n f(\dots) = nf(\dots)$ .

So we can immediately simplify to

$$\begin{aligned} t_{\text{loop}} &\in O(1) + i * (O(1) + O(1) + O(1)) + O(1) \\ &= O(1) + i * O(1) + O(1) \\ &= O(1) + O(i) + O(1) \\ &= O(i) \end{aligned}$$

#### Example 4: Nested Loops

Suppose we have:

```
for (int i = 0; i < n; ++i)
    for (int j = 0; j < i; ++j)
        a[i+10*j] = i + j;
```

We have already analyzed the inner loop:

```
for (int i = 0; i < n; ++i)
{
    // init: O(1)
    for (int j = 0; j < i; ++j) // cond: O(1)  incr: O(1)  #: i  total: O(i)
        a[i+10*j] = i + j; // O(1)
}
```

and this might be a good time to collapse the inner loop:

```
for (int i = 0; i < n; ++i)
{
    // O(i)
}
```

The remaining outer loop has components

```
// init: O(1)
for (int i = 0; i < n; ++i) // cond: O(1)  incr: O(1)  #: n
{
    // O(i)
}
```

So now we can analyze the outer loop as

$$\begin{aligned} t_{\text{loop}} &= t_{\text{init}} + \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{increment}} + t_{\text{body}}) + t_{\text{condition}}^{\text{final}} \\ &\in O(1) + \sum_{i=0}^{n-1} (O(1) + O(i) + O(1)) + O(1) \\ &= O(1) + \sum_{i=0}^{n-1} O(i) + O(1) \end{aligned}$$

We can't apply the same replace-summation-by-multiplication shortcut this time, because the expression inside the sum depends on the iteration number ( $i$ ).

So, continuing on,

$$t_{\text{loop}} \in O(1) + O\left(\sum_{i=0}^{n-1} i\right)$$

Now, that summation is actually very well known, and one that we will encounter a lot during this course. You can find it and several others in the [FAQ](#)

$$\begin{aligned} t_{\text{loop}} &\in O(1) + O\left(\sum_{i=0}^{n-1} i\right) \\ &= O(1) + O\left(\frac{(n-1)n}{2}\right) \\ &= O(1) + O((n-1)n) \\ &= O(n^2 - n + 1) \\ &= O(n^2) \end{aligned}$$

And so we conclude that

```
// init: O(1)
for (int i = 0; i < n; ++i) // cond: O(1)  incr: O(1)  #: n  total: O(n^2)
{
    // O(i)
}
```

## 3.4 Recursive Functions

For recursive routines, we must take the number of recursive calls times the complexity of each call.

Our final rule for combining statements is one which we won't use for little while, but I'll go ahead and list it here for the sake of completeness. When we have recursive functions (functions that call themselves), things get little bit complicated. Our general approach to first find the complexity of each call while ignoring the part that calls itself, and then trying to figure out how many recursive calls are made, then add up the complexity of the individual calls across all the number of calls this function actually makes.

## 4 The Importance of Understanding

The rules I've discussed here will cover most of the circumstances you will encounter. But they aren't a complete list of all programming language constructs and they don't amount to a recipe that can be followed without thinking.

You need to always be aware of what the code you are analyzing actually *does*. If you see a statement that you just know has to manipulate K pieces of information, then any answer that isn't O(K) or slower should be suspicious. And if you don't know what a particular statements or group of statements actually does, you need to figure that out before you try to figure out how long it will take to do that thing.

*You can't analyze the complexity of code that you don't understand.*

In fact, no one can.

For example, [no one has been able to prove](#) if this code

```
unsigned collatz (unsigned n)
{
    while (n != 1)
    {
        if (n % 2 == 0) // n is even
            n = n/2;
        else           // n is odd
            n = 3*n + 1;
    }
    return 1;
}
```

actually comes to a halt for all values of n. It's been checked for all values of n up to some absurdly high value, and has eventually halted each time, but no one has been able prove that it halts for all inputs. So it's entirely possible that this code has worst-case complexity  $O(\infty)$ . But we don't *know* that is true, because no one actually understands the behavior of this function well enough to answer the question of how many repetitions it does for arbitrary values of *n*.

# Case Studies: Analyzing Standalone Functions

Steven J. Zeil

Last modified: Feb 12, 2020

## Contents:

- [1 Some Simple Examples](#)
  - [1.1 Diagonal Matrix](#)
  - [1.2 Working with a Matrix](#)
  - [1.3 Books and Authors](#)
- [2 Ordered Insert](#)
  - [2.1 Analysis of addInOrder](#)
  - [2.2 Special Case Behavior](#)
  - [2.3 addInOrder as a template](#)
  - [2.4 addInOrder as an Iterator-style Template](#)
- [3 Sequential Search](#)
  - [3.1 Analysis](#)
- [4 Ordered Sequential Search](#)
- [5 Binary Search](#)
  - [5.1 Starting the Analysis](#)
  - [5.2 Logarithmic Behavior](#)
  - [5.3 Back to the Analysis](#)
- [6 Appendix: std Functions Analyzed in This Lesson](#)
  - [6.1 std::find](#)
  - [6.2 std::lower\\_bound](#)

We'll illustrate the techniques we've learned with some common functions, including some of the array manipulation functions that we [converted to iterator style](#).

## 1 Some Simple Examples

### 1.1 Diagonal Matrix

This block of code might be found in an early step in a linear algebra application. It sets up an NxN matrix with 1.0 on the diagonals and 0.0 on all of the off-diagonal elements:

```
double** matrix = new double*[N];
for (int i = 0; i < N; ++i)
{
    matrix[i] = new double[N];
    for (int j = 0; j < N; ++j)
    {
        if (i == j)
            matrix[i][j] = 1.0;    ①
        else
            matrix[i][j] = 0.0;    ②
    }
}
```

The most deeply nested statements in this code are ① and ②. Looking at them the involve address calculations for the array indexing, and assignment of a single double. All of those are  $O(1)$ .

```
double** matrix = new double*[N];
for (int i = 0; i < N; ++i)
{
    matrix[i] = new double[N];
    for (int j = 0; j < N; ++j)
    {
        if (i == j)
            matrix[i][j] = 1.0; // O(1)
        else
            matrix[i][j] = 0.0; // O(1)
    }
}
```

The `if` statement condition is  $O(1)$ ,

```
double** matrix = new double*[N];
for (int i = 0; i < N; ++i)
{
    matrix[i] = new double[N];
    for (int j = 0; j < N; ++j)
    {
        if (i == j)           // cond: O(1)
            matrix[i][j] = 1.0; // O(1)
        else
    }
}
```

```

        matrix[i][j] = 0.0; // O(1)
    }
}

```

which means that

$$\begin{aligned}
 t_{\text{if}} &= t_{\text{condition}} + \max(t_{\text{then}}, t_{\text{else}}) \\
 &= O(1) + \max(O(1), O(1)) \\
 &= O(1)
 \end{aligned}$$

```

double** matrix = new double*[N];
for (int i = 0; i < N; ++i)
{
    matrix[i] = new double[N];
    for (int j = 0; j < N; ++j)
    {
        if (i == j) // cond: O(1) total: O(1)
            matrix[i][j] = 1.0; // O(1)
        else
            matrix[i][j] = 0.0; // O(1)
    }
}

```

Collapsing,

```

double** matrix = new double*[N];
for (int i = 0; i < N; ++i)
{
    matrix[i] = new double[N];
    for (int j = 0; j < N; ++j)
    {
        // O(1)
    }
}

```

The inner for loop has initialization, condition, and increment all  $O(1)$ . It executes  $N$  times.

```

double** matrix = new double*[N];
for (int i = 0; i < N; ++i)
{
    matrix[i] = new double[N];
    // init: O(1)
    for (int j = 0; j < N; ++j) // cond: O(1) incr: O(1) #: N
    {
        // O(1)
    }
}

```

So by our rule for for loops,

$$\begin{aligned}
 t_{\text{for}} &= t_{\text{init}} + \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{increment}} + t_{\text{body}}) + t_{\text{final}}^{\text{condition}} \\
 &= O(1) + \sum_{j=0}^{N-1} (O(1) + O(1) + O(1)) + O(1) \\
 &= O(1) + N * (O(1) + O(1) + O(1)) + O(1) \\
 &= O(1) + N * (O(1)) + O(1) \\
 &= O(1) + O(N) + O(1) \\
 &= O(N)
 \end{aligned}$$

```

double** matrix = new double*[N];
for (int i = 0; i < N; ++i)
{
    matrix[i] = new double[N];
    // init: O(1)
    for (int j = 0; j < N; ++j) // cond: O(1) incr: O(1) #: N total: O(N)
    {
        // O(1)
    }
}

```

Collapsing:

```

double** matrix = new double*[N];
for (int i = 0; i < N; ++i)
{
    matrix[i] = new double[N];
    // init: O(1)
    // O(N)
}

```

Now look at the first statement in the remaining loop body. We have talked a bit [already](#) about what happens when arrays are allocated. First a block of memory is obtained, then the array elements are each initialized by the data type's default constructor — unless it is a primitive type in which case no such initialization occurs.

In this case we are dealing with an array of `double`, a primitive type. So no initialization occurs. The only time spent is to get a block of memory from the operating system, which, interestingly enough, does not depend upon the size of the block requested.

```
double** matrix = new double*[N];
for (int i = 0; i < N; ++i)
{
    matrix[i] = new double[N]; // O(1)
    // init: O(1)
    // O(N)
}
```

So the statement sequence making up this function body adds up to  $O(N)$

```
double** matrix = new double*[N];
for (int i = 0; i < N; ++i)
{
    // O(N)
}
```

The remaining for loop has initialization, condition, and increment all  $O(1)$ . It executes  $N$  times.

```
double** matrix = new double*[N];
for (int i = 0; i < N; ++i) // cond: O(1) incr: O(1) #: N
{
    // O(N)
}
```

And so the time for this loop is

$$\begin{aligned} t_{\text{for}} &= t_{\text{init}} + \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{increment}} + t_{\text{body}}) + t_{\text{condition}}^{\text{final}} \\ &= O(1) + \sum_{i=0}^{N-1} (O(1) + O(1) + O(N)) + O(1) \\ &= O(1) + N * (O(1) + O(1) + O(N)) + O(1) \\ &= O(1) + N * (O(1)) + O(1) \\ &= O(1) + O(N^2) + O(1) \\ &= O(N^2) \end{aligned}$$

```
double** matrix = new double*[N];
for (int i = 0; i < N; ++i) // cond: O(1) incr: O(1) #: N total: O(N^2)
{
    // O(N)
}
```

Collapsing:

```
double** matrix = new double*[N];
// O(N^2)
```

The first statement allocates an array of pointers. Pointer are primitive types and are not initialized, so we have only the  $O(1)$  time to get the block of memory:

```
double** matrix = new double*[N]; // O(1)
// O(N^2)
```

and the entire statement sequence adds up to  $O(N^2)$ .

There are a few things that I hope you take notice of here:

- Note the general flow of the analysis from most deeply nested towards the outermost statements.
- If we had not been able to guess that the “size” measure for this code would be  $N$ , nonetheless  $N$  emerged naturally from the analysis when we considered the natural behavior of the loops.
- If we had done a [sanity check](#) of “If the loops are nested  $k$  deep, and each appears to execute up to  $N$  times, then  $N^k$  is a reasonable first guess”, we would have guessed at  $O(N^2)$ . The fact that the full analysis came out to that same value is reassuring.

## 1.2 Working with a Matrix

Continuing with our  $N \times N$  matrix, suppose that later in the code we saw:

```
for (int i = 0; i < N; ++i)
{
    for (int j = 0; j < N; ++j)
    {
        matrix[i][j] += matrix[i][i];
```

```

        matrix[j][i] += matrix[i][i];
        if (i == j)
        {
            for (int k = 0; k < N; ++k)
                matrix[i][i] *= 2.0;
        }
    }
}

```

Here we have three nested loops, each apparently repeating  $N$  times, so we might guess that this will be  $O(N^3)$ . let's see if this guess holds up.

Again, we can start by marking simple statements that involve only arithmetic and assignment of primitive types.

```

for (int i = 0; i < N; ++i)
{
    for (int j = 0; j < N; ++j)
    {
        matrix[i][j] += matrix[i][i];      // O(1)
        matrix[j][i] += matrix[i][i];      // O(1)
        if (i == j)
        {
            for (int k = 0; k < N; ++k)
                matrix[i][i] *= 2.0;        // O(1)
        }
    }
}

```

Looking at the innermost loop,

```

for (int i = 0; i < N; ++i)
{
    for (int j = 0; j < N; ++j)
    {
        matrix[i][j] += matrix[i][i];      // O(1)
        matrix[j][i] += matrix[i][i];      // O(1)
        if (i == j)
        {
            for (int k = 0; k < N; ++k) //cond: O(1) iter: O(1) #: N
                matrix[i][i] *= 2.0;      // O(1)
        }
    }
}

```

and the loop time is

$$\begin{aligned}
 t_{\text{for}} &= t_{\text{init}} + \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{increment}} + t_{\text{body}}) + t_{\text{final}}^{\text{condition}} \\
 &= O(1) + \sum_{k=0}^{N-1} (O(1) + O(1) + O(1)) + O(1) \\
 &= O(1) + N * (O(1) + O(1) + O(1)) + O(1) \\
 &= O(1) + N * (O(1)) + O(1) \\
 &= O(1) + O(N) + O(1) \\
 &= O(N)
 \end{aligned}$$

```

for (int i = 0; i < N; ++i)
{
    for (int j = 0; j < N; ++j)
    {
        matrix[i][j] += matrix[i][i];      // O(1)
        matrix[j][i] += matrix[i][i];      // O(1)
        if (i == j)
        {
            for (int k = 0; k < N; ++k) //cond: O(1) iter: O(1) #: N total: O(N)
                matrix[i][i] *= 2.0;      // O(1)
        }
    }
}

```

Collapsing:

```

for (int i = 0; i < N; ++i)
{
    for (int j = 0; j < N; ++j)
    {
        matrix[i][j] += matrix[i][i];      // O(1)
        matrix[j][i] += matrix[i][i];      // O(1)
        if (i == j)
        {
            // O(N)
        }
    }
}

```

The if condition is  $O(1)$ .

```
for (int i = 0; i < N; ++i)
{
    for (int j = 0; j < N; ++j)
    {
        matrix[i][j] += matrix[i][i];      // O(1)
        matrix[j][i] += matrix[i][i];      // O(1)
        if (i == j) // cond: O(1)
        {
            // O(N)
        }
    }
}
```

And we can then evaluate the if as

$$\begin{aligned} t_{\text{if}} &= t_{\text{condition}} + \max(t_{\text{then}}, t_{\text{else}}) \\ &= O(1) + \max(O(N), O(1)) \\ &= O(N) \end{aligned}$$

```
for (int i = 0; i < N; ++i)
{
    for (int j = 0; j < N; ++j)
    {
        matrix[i][j] += matrix[i][i];      // O(1)
        matrix[j][i] += matrix[i][i];      // O(1)
        if (i == j) // cond: O(1) total: O(N)
        {
            // O(N)
        }
    }
}
```

Now, commonly we would, at this point, collapse the if code and just retain the total complexity. But there are three things that hint to me that this might not be the best approach:

1. The if is inside a loop.
2. The then and else part complexities are different, and
3. The if condition seems highly selective (will take one path much more often than the other).

Whenever I see those two characteristics together, I think it's worth asking whether the more expensive of the two if parts gets executed often enough, in the worst case, to actually dominate the overall loop.

Ignore the outermost loop for the moment.

```
for (int j = 0; j < N; ++j)
{// O(N)
    matrix[i][j] += matrix[i][i];      // O(1)
    matrix[j][i] += matrix[i][i];      // O(1)
    if (i == j) // cond: O(1) total: O(N)
    {
        // O(N)
    }
}
```

**Question: How many times, total, is the if statement going to be executed by this loop?**

Click to reveal +

It will be executed a total of  $N$  times.

**Question:** How many times, total, will the `if` statement take the “then” (true) branch?

*Click to reveal* +

Only one time.

**Question:** How many times, total, will the `if` statement take the “then” (true) branch?

*Click to reveal* +

•  $N - 1$  times.

OK, now the rest of that loop is easy enough to handle:

```
for (int i = 0; i < N; ++i)
{
    for (int j = 0; j < N; ++j) // cond: O(1) iter: O(1) #: N
    { // O(N)
        matrix[i][j] += matrix[i][i]; // O(1)
        matrix[j][i] += matrix[i][i]; // O(1)
        if (i == j) // cond: O(1) total: O(N)
        {
            // O(N)
        }
    }
}
```

so that

$$\begin{aligned}
 t_{\text{for}} &= t_{\text{init}} + \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{increment}} + t_{\text{body}}) + t_{\text{final}}^{\text{condition}} \\
 &= O(1) + \sum_{j=0}^{N-1} (O(1) + O(1) + t_{\text{body}}) + O(1) \\
 &= O(1) + \sum_{j=0}^{N-1} (O(1) + O(1)) + \sum_{j=0}^{N-1} t_{\text{body}} + O(1) \\
 &= O(1) + N * O(1) + \sum_{j=0}^{N-1} t_{\text{body}} \\
 &= O(1) + O(N) + \sum_{j=0}^{N-1} t_{\text{body}} \\
 &= O(N) + \sum_{j=0}^{N-1} t_{\text{body}}
 \end{aligned}$$

OK, now let's think about what our earlier questions told us about the sum of the loop body times over all of the iterations. That sum will add together a total of  $N$  executions of the body. Exactly one of those will take  $O(N)$  time and  $N - 1$  of those will take  $O(1)$  time. So

$$\begin{aligned}
 t_{\text{for}} &= O(N) + \sum_{j=0}^{N-1} t_{\text{body}} \\
 &= O(N) + O(N) + (N - 1)O(1) \\
 &= O(N) + O(N) + O(N) - O(1) \\
 &= O(N)
 \end{aligned}$$

So what we can see is that the expensive option of the `if` is done so rarely that it does not dominate the rest of the sum – it kind of disappears into the overall sum.

```
for (int i = 0; i < N; ++i)
{
    for (int j = 0; j < N; ++j) // cond: O(1) iter: O(1) #: N total: O(N)
    { // O(N)
        matrix[i][j] += matrix[i][i]; // O(1)
        matrix[j][i] += matrix[i][i]; // O(1)
        if (i == j) // cond: O(1) total: O(N)
        {
            // O(N)
        }
    }
}
```

Now we can collapse that “j” loop:

```
for (int i = 0; i < N; ++i)
{
    // O(N)
}
```

The final loop follows the familiar form:

```
for (int i = 0; i < N; ++i) // cond: O(1) iter: O(1) #: N
{
    // O(N)
}
```

$$\begin{aligned}
t_{\text{for}} &= t_{\text{init}} + \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{increment}} + t_{\text{body}}) + t_{\text{condition}}^{\text{final}} \\
&= O(1) + \sum_{i=0}^{N-1} (O(1) + O(1) + O(N)) + O(1) \\
&= O(1) + N * (O(1) + O(1) + O(N)) + O(1) \\
&= O(1) + N * (O(1)) + O(1) \\
&= O(1) + O(N^2) + O(1) \\
&= O(N^2)
\end{aligned}$$

And the entire block of code is  $O(N^2)$ .

That is not what we predicted by just looking at the depth of nesting of the loops. *Surprises demand explanation.* In this case we have the explanation: the innermost loop is executed only  $(1/N)$ th of the time, so we wind up being a factor of  $N$  faster than we had originally predicted.

## 1.3 Books and Authors

Consider this code from our early [Book ADT](#)

```

void Book::addAuthor (const Author& anAuthor)
{
    authors[numAuthors] = anAuthor;
    ++numAuthors;
}

void Book::removeAuthor (Author* anAuthor)
{
    int i = 0;
    while (i < numAuthors &&
           authors[i]->getID() != anAuthor->getID())
        ++i;
    while (i < numAuthors-1)
    {
        authors[i] = authors[i+1];
        ++i;
    }
    --numAuthors;
}

```

Let's analyze each of the functions, in turn.

First, look at `addAuthor`. We expect any function's complexity to be written [in terms of its inputs](#), so **what are the inputs to this function?**

*Click to reveal* +

The inputs are `anAuthor` and `this`, the pointer to the current value of the `Book` that we want to modify. *Don't forget "this"!*

We expect the answer to be written in terms of some numerical properties of that author and/or that book.

That said, there are no loops or conditionals in `addAuthor`, so the analysis is pretty straightforward.

**What is the complexity of `authors[numAuthors] = anAuthor;` ? How do we annotate it?**

*Click to reveal* +

The statement does some address calculations for the array indexing, but those are all  $O(1)$ . There is an assignment of an object of the Author class. To see the complexity of that, we would have to visit [that class](#) and look at its code. It turns out that we chose to use the compiler-proved assignment operator, and so this assignment involves assignment of one string (the author name), one Address (itself a collection of 4 strings), and one long. I'm going to assume that all the strings in this application have a fairly small upper bound (a few hundred characters at most), in which case this assignment is  $O(1)$ .

```
void Book::addAuthor (const Author& anAuthor)
{
    authors[numAuthors] = anAuthor; // O(1)
    ++numAuthors;
}
```

Clearly the second statement, `++numAuthors;`, is  $O(1)$ .

**What is the overall complexity of the statement sequence that makes up the function body?**

*Click to reveal* +

```

void Book::addAuthor (const Author& anAuthor)
{
    authors[numAuthors] = anAuthor; // O(1)
    ++numAuthors; // O(1)
}

```

For a statement sequence, we add up the complexities of the component statements, and  $O(1) + O(1) = O(1)$ .

```

void Book::addAuthor (const Author& anAuthor)
{//total: O(1)
    authors[numAuthors] = anAuthor; // O(1)
    ++numAuthors; // O(1)
}

```

So the complexity of the entire function is  $O(1)$ .

The removeAuthor function looks a bit more challenging.

```

void Book::removeAuthor (Author* anAuthor)
{
    int i = 0;
    while (i < numAuthors &&
           authors[i]->getID() != anAuthor->getID())
        ++i;
    while (i < numAuthors-1)
    {
        authors[i] = authors[i+1];
        ++i;
    }
    --numAuthors;
}

```

To speed things up just a tad, let's mark all the simple statements that are clearly  $O(1)$ .

```

void Book::removeAuthor (Author* anAuthor)
{
    int i = 0; // O(1)
    while (i < numAuthors && // O(1)
           authors[i]->getID() != anAuthor->getID())
        ++i; // O(1)
    while (i < numAuthors-1)
    {
        authors[i] = authors[i+1]; // O(1) // O(1)
        ++i;
    }
    --numAuthors; // O(1)
}

```

Now let's look at the first loop. As occasionally happens, all the hard work in this loop seems to be done in the condition. But  $i < numAuthors$  is just an integer comparison, which is  $O(1)$ .  $authors[i]->$  is a simple address calculation, also  $O(1)$ . For the `getID()` calls, we need to go back to the `Author` class and see that this just retrieves a long data member, in  $O(1)$  time. That means that the `!=` is just comparing two long ints, and that's  $O(1)$ . Finally, the `&&` boolean operator is  $O(1)$ . So, messy as that condition is, it's still just  $O(1)$ .

```

void Book::removeAuthor (Author* anAuthor)
{
    int i = 0; // O(1)
    while (i < numAuthors && // cond: O(1)
           authors[i]->getID() != anAuthor->getID()) // cond: O(1)
        ++i; // O(1)
    while (i < numAuthors-1)
    {
        authors[i] = authors[i+1]; // O(1) // O(1)
        ++i;
    }
    --numAuthors; // O(1)
}

```

How often does that loop repeat in the worst case? It uses `i`, which starts at 0 and goes up as high as `numAuthors` before we force an exist. So, in the worst case, the author we are looking for is not in the array, and the loop repeats a total of `numAuthors` times.

```

void Book::removeAuthor (Author* anAuthor)
{
    int i = 0; // O(1)
    while (i < numAuthors && // cond: O(1) #: numAuthors
           authors[i]->getID() != anAuthor->getID()) // cond: O(1) #: numAuthors
        ++i; // O(1)
    while (i < numAuthors-1)
    {
        authors[i] = authors[i+1]; // O(1) // O(1)
        ++i;
    }
    --numAuthors; // O(1)
}

```

So the time for the first loop is

$$\begin{aligned}
 t_{\text{while}} &= \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{body}}) + t_{\text{final}}^{\text{condition}} \\
 &= \sum_{i=0}^{\text{numAuthors}} (O(1) + O(1)) + O(1) \\
 &= \text{numAuthors} * (O(1) + O(1)) + O(1) \\
 &= \text{numAuthors} * (O(1)) + O(1) \\
 &= O(\text{numAuthors}) + O(1) \\
 &= O(\text{numAuthors})
 \end{aligned}$$

```

void Book::removeAuthor (Author* anAuthor)
{
    int i = 0; // O(1)
    while (i < numAuthors &&
           authors[i]->getID() != anAuthor->getID()) //cond: O(1) #: numAuthors
        ++i; // O(1)
    while (i < numAuthors-1)
    {
        authors[i] = authors[i+1]; // O(1)
        ++i; // O(1)
    }
    --numAuthors; // O(1)
}

```

and, collapsing,

```

void Book::removeAuthor (Author* anAuthor)
{
    int i = 0; // O(1)
    // O(numAuthors)
    while (i < numAuthors-1)
    {
        authors[i] = authors[i+1]; // O(1)
        ++i; // O(1)
    }
    --numAuthors; // O(1)
}

```

The second loop also has an  $O(1)$  condition:

```

void Book::removeAuthor (Author* anAuthor)
{
    int i = 0; // O(1)
    // O(numAuthors)
    while (i < numAuthors-1) //cond: O(1)
    {
        authors[i] = authors[i+1]; // O(1)
        ++i; // O(1)
    }
    --numAuthors; // O(1)
}

```

How often does this loop execute? Interestingly enough, if the first loop executed  $k$  times, the second one executes  $\text{numAuthors} - k - 1$  times. In essence, the second loop visits all of the elements in the array that the first loop doesn't. In the worst case, then, this loop executes  $\text{numAuthors} - 1$  times.

```

void Book::removeAuthor (Author* anAuthor)
{
    int i = 0; // O(1)
    // O(numAuthors)
    while (i < numAuthors-1) //cond: O(1) #: numAuthors-1
    {
        authors[i] = authors[i+1]; // O(1)
        ++i; // O(1)
    }
    --numAuthors; // O(1)
}

```

and so we have

$$\begin{aligned}
 t_{\text{while}} &= \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{body}}) + t_{\text{final condition}} \\
 &= \sum_{j=0}^{\text{numAuthors}-1} (O(1) + O(1)) + O(1) \\
 &= (\text{numAuthors} - 1) * (O(1) + O(1)) + O(1) \\
 &= (\text{numAuthors} - 1) * (O(1)) + O(1) \\
 &= O(\text{numAuthors} - 1) + O(1) \\
 &= O(\text{numAuthors})
 \end{aligned}$$

```

void Book::removeAuthor (Author* anAuthor)
{
    int i = 0;                                // O(1)
    // O(numAuthors)
    while (i < numAuthors-1) //cond: O(1) #: numAuthors-1 total: O(numAuthors)
    {
        authors[i] = authors[i+1]; // O(1)
        ++i;                      // O(1)
    }
    --numAuthors;                            // O(1)
}

```

Collapsing,

```

void Book::removeAuthor (Author* anAuthor)
{
    int i = 0;                                // O(1)
    // O(numAuthors)
    // O(numAuthors)
    --numAuthors;                            // O(1)
}

```

we now are reduced to a statement sequence with four components. We add them up:

$$t_{\text{removeAuthor}} = O(1) + O(\text{numAuthors}) + O(\text{numAuthors}) + O(1) = O(\text{numAuthors})$$

## 2 Ordered Insert

We start with a function for inserting an element into a sorted array.

```

/**
 *
 * Assume the elements of the array are already in order
 * Find the position where value could be added to keep
 * everything in order, and insert it there.
 *
 * Return the position where it was inserted
 * - Assumes that we have a separate integer (size) indicating how
 *   many elements are in the array
 * - and that the "true" size of the array is at least one larger
 *   than the current value of that counter
 *
 * @param array array into which to add an element
 * @param size number of data elements in the array. Must be less than
 *            the number of elements allocated for the array.
 *            Incremented upon output from this function.
 * @param value value to add into the array
 * @return the position where the element was added
 */
int addInOrder (int* array, int& size, int value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;                ①
    while (toBeMoved >= 0 && value < array[toBeMoved]) { ②
        array[toBeMoved+1] = array[toBeMoved];      ③
        --toBeMoved;
    }
    // Insert the new value
    array[toBeMoved+1] = value;                ④
    ++size;
    return toBeMoved+1;
}

```

- We start from the high end of the array (①) and check to see if that's where we want to insert the data (②).
  - If so, fine. We exit from the loop.
  - If not, we move the preceding element up one (③) and then check to see if we want to insert value into the “hole” left behind. We repeat this step as necessary.
- Once we exit from the loop (④), we insert the value into the hole at our chosen position.
- You can [run this algorithm](#) until you are comfortable with your understanding of how it works.

- From the “Algorithm” menu, select “generate an array”. Create an array with 5-10 elements.
- Again, from the “Algorithm” menu, select “add in order” to try the algorithm. Press the forward button to step through the execution.
- Try inserting values that should fall near the middle of the array.
- Try inserting values that are smaller than any already in the array.
- Try inserting values that are larger than any already in the array.

Then we can move on to the analysis.

## 2.1 Analysis of addInOrder

We start our analysis with the easy stuff – mark all of the non-compound statements.

```
int addInOrder (int* array, int& size, int value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;                                // O(1)
    while (toBeMoved >= 0 && value < array[toBeMoved]) {      // cond: O(1) #: size
        array[toBeMoved+1] = array[toBeMoved];                // O(1)
        --toBeMoved;                                         // O(1)
    }
    // Insert the new value
    array[toBeMoved+1] = value;                            // O(1)
    ++size;                                              // O(1)
    return toBeMoved+1;                                    // O(1)
}
```

Next, looking at the `while` loop, we see that its condition can be evaluated in  $O(1)$  time, and that the loop repeats at most `size` times.

```
int addInOrder (int* array, int& size, int value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;                                // O(1)
    while (toBeMoved >= 0 && value < array[toBeMoved]) {      // cond: O(1) #: size
        array[toBeMoved+1] = array[toBeMoved];                // O(1)
        --toBeMoved;                                         // O(1)
    }
    // Insert the new value
    array[toBeMoved+1] = value;                            // O(1)
    ++size;                                              // O(1)
    return toBeMoved+1;                                    // O(1)
}
```

The loop body is  $O(1)$ , so

$$\begin{aligned} t_{\text{while}} &= \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{body}}) + t_{\text{final condition}} \\ &= \sum_{i=1}^{\text{size}} (O(1) + O(1)) + O(1) \\ &= O\left(\sum_{i=1}^{\text{size}} 1\right) + O(1) \\ &= O(\text{size}) + O(1) \\ &= O(\text{size}) \end{aligned}$$

```
int addInOrder (int* array, int& size, int value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;                                // O(1)
    while (toBeMoved >= 0 && value < array[toBeMoved]) {      // cond: O(1) #: size [total: O(size)]
        array[toBeMoved+1] = array[toBeMoved];                // O(1)
        --toBeMoved;                                         // O(1)
    }
    // Insert the new value
    array[toBeMoved+1] = value;                            // O(1)
    ++size;                                              // O(1)
    return toBeMoved+1;                                    // O(1)
}
```

Replacing the loop by this quantity ... ,

```
int addInOrder (int* array, int& size, int value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;                                // O(1)
    // O(size)
    // Insert the new value
    array[toBeMoved+1] = value;                            // O(1)
```

```
    ++size;
    return toBeMoved+1;
}
```

// O(1)  
// O(1)

... we get down to a statement sequence composed of  $O(1)$  and  $O(\text{size})$  terms. The total run time is therefore

$$t_{\text{addInOrder}} = O(1) + O(\text{size}) + O(1) + O(1) + O(1) = O(\text{size})$$

```
template <typename int>
int addInOrder (int* array, int& size, int value)
{
    // O(size)
}
```

Note that, because none of the inputs to this function are actually named “n”, it is *not* proper to say the function is “O(n)” unless we *explicitly* define “n” to be equal to size.

## 2.2 Special Case Behavior

```
int addInOrder (int* array, int& size, int value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;
    while (toBeMoved >= 0 && value < array[toBeMoved]) {
        array[toBeMoved+1] = array[toBeMoved];
        --toBeMoved;
    }
    // Insert the new value
    array[toBeMoved+1] = value;
    ++size;
    return toBeMoved+1;
}
```

A special point worth noting:

- If we are adding a value that is greater than all elements already in `array`, this algorithm does 0 iterations of the loop.
  - Suppose we are given a series of values to insert into an initially empty array, and that these values are already sorted
    - Then each new value will be greater than all the ones already inserted into the array.
    - Each call to `addInOrder` will use 0 iterations.
    - and so each call runs in  $O(1)$  time (for this special case of inserting sorted elements)
  - We'll make use of this special case later when we incorporate this function into more complex algorithms.

## 2.3 addInOrder as a template

Suppose that, instead of working on arrays of integers, we wrote `addInOrder` as a template that would work on arrays of any data type that supported assignment and comparison with `<`:

```
template <typename Comparable>
int addInOrder (Comparable* array, int& size, Comparable value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;
    while (toBeMoved >= 0 && value < array[toBeMoved]) {
        array[toBeMoved+1] = array[toBeMoved];
        --toBeMoved;
    }
    // Insert the new value
    array[toBeMoved+1] = value;
    ++size;
    return toBeMoved+1;
}
```

Again, we might start by marking certain “obvious” lines as  $O(1)$ :

```

template <typename Comparable>
int addInOrder (Comparable* array, int& size, Comparable value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;
    while (toBeMoved >= 0 && value < array[toBeMoved]) {
        array[toBeMoved+1] = array[toBeMoved];
        toBeMoved;                                // O(1)
    }
    // Insert the new value
    array[toBeMoved+1] = value;                  // O(1)
    ++size;
}

```

```

    return toBeMoved+1;           // O(1)
}

```

But look at the lines ① and ②. What are the complexity of these? It's actually hard to say.

- The address calculation involved in indexing into the array is  $O(1)$ .
- But the assignment operation is another matter. This is an assignment of one Comparable value to another. We have no idea, really, what kinds of data types might be supplied for Comparable, other than the fact that they will all need to support operator= and operator<. Not all assignments are  $O(1)$ . In fact, we know that programmers can write their own assignment operators, so these could be arbitrarily complicated.

So the best that we can do is to introduce a symbol for the time to perform an assignment of a Comparable value. Let's call that  $t_a$ .

```

template <typename Comparable>
int addInOrder (Comparable* array, int& size, Comparable value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;           // O(1)
    while (toBeMoved >= 0 && value < array[toBeMoved]) {
        array[toBeMoved+1] = array[toBeMoved];           // O(t_a)
        --toBeMoved;                                // O(1)
    }
    // Insert the new value
    array[toBeMoved+1] = value;           // O(t_a)
    ++size;                            // O(1)
    return toBeMoved+1;                // O(1)
}

```

That means that the entire loop body is  $O(t_a)$  (i.e.,  $O(t_a + 1) = O(t_a)$ ), because nothing can be smaller than  $O(1)$ .

```

template <typename Comparable>
int addInOrder (Comparable* array, int& size, Comparable value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;           // O(1)
    while (toBeMoved >= 0 && value < array[toBeMoved]) {
        // O(t_a)
    }
    // Insert the new value
    array[toBeMoved+1] = value;           // O(t_a)
    ++size;                            // O(1)
    return toBeMoved+1;                // O(1)
}

```

Now, let's look at the while loop. We have another problem in determining the complexity of the loop condition. We don't know what the complexity of operator< will be for an arbitrary Comparable type. So let  $t_c$  denote the time required to compare two Comparable values using <. Then the condition is  $O(1)$  for the integer comparison and the boolean  $\&&$  plus  $t_c$  for the Comparable comparison:

```

template <typename Comparable>
int addInOrder (Comparable* array, int& size, Comparable value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;           // O(1)
    while (toBeMoved >= 0 && value < array[toBeMoved]) { // cond: O(t_c)
        // O(t_a)
    }
    // Insert the new value
    array[toBeMoved+1] = value;           // O(t_a)
    ++size;                            // O(1)
    return toBeMoved+1;                // O(1)
}

```

At worst, the loop repeats  $size$  times:

```

int addInOrder (Comparable* array, int& size, Comparable value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;           // O(1)
    while (toBeMoved >= 0 && value < array[toBeMoved]) { // cond: O(t_c) #: size
        // O(t_a)
    }
    // Insert the new value
    array[toBeMoved+1] = value;           // O(t_a)
    ++size;                            // O(1)
    return toBeMoved+1;                // O(1)
}

```

That makes the total time for the while loop:

$$\begin{aligned}
 t_{\text{while}} &= \sum_{\text{iterations}} (t_{\text{condition}} + t_{\text{body}}) + t_{\text{final condition}} \\
 &= \sum_{i=1}^{\text{size}} (O(t_c) + O(t_a)) + O(t_c) \\
 &= O(\text{size} * t_a + (\text{size} + 1) * t_c) \\
 &= O(\text{size} * t_a + \text{size} * t_c)
 \end{aligned}$$

We would summarize this in English by stating that in the worst case, `addInOrder` performs  $O(\text{size})$  assignments and  $O(\text{size})$  comparisons of the Comparable type.

## 2.4 addInOrder as an Iterator-style Template

A more modern style of C++ would urge writing it in terms of iterators so that it could be used with containers other than conventional arrays:

```


/*
 * Assume the elements of a container are already in order
 * Find the position where value could be added to keep
 * everything in order, and insert it there.
 *
 * Return the position where it was inserted. This assumes that
 * we have room in the container for one more element at the
 * given stop position.
 *
 * @param start beginning position of the sequence within which
 *               to insert a value.
 * @param stop  Position just after the last element to be examined
 *             in determining where to place value. In other words,
 *             the range (start,stop] is considered to contain
 *             already ordered data.
 * @param value value to add into the container
 * @return the position where the element was added.
 *         Upon exit from this function, all data in the range
 *         (start,stop] will be ordered.
 */
template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop, const Comparable& value)
{
    Iterator preStop = stop;
    --preStop;
    while (stop != start && value < *preStop) {
        *stop = *preStop;
        --stop;
        --preStop;
    }
    // Insert the new value
    *stop = value;
    return stop;
}


```

- Instead of introducing a separate position `toBeMoved` as in the earlier array-based version, we here use `stop`, decrementing it until we find the right position at which to insert the value.
- The variable `preStop` in this code fulfills the same purpose as the expression `array[toBeMoved-1]` – it holds the position just in front of the one we are considering as a candidate insertion position.

In all analyses in this course, we will assume, unless stated otherwise, that iterator operations have the same complexity as pointer operations. Thus `*stop`, `--stop`, iterator assignment, and iterator comparison are all  $O(1)$ . So we can, gain, start by marking some “obvious” statements:

```


template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop, const Comparable& value)
{
    Iterator preStop = stop;      // O(1)
    --preStop;                  // O(1)
    while (stop != start && value < *preStop) {
        *stop = *preStop;        ①
        --stop;                  // O(1)
        --preStop;                // O(1)
    }
    // Insert the new value
    *stop = value;              ②
    return stop;                // O(1)
}


```

As before, the assignments (①, ②) of Comparable objects are of unknown complexity, so we introduce a symbol to denote the time required for such assignments:

```


template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop, const Comparable& value)
{
    Iterator preStop = stop;      // O(1)
    --preStop;                  // O(1)
    while (stop != start && value < *preStop) {
        *stop = *preStop;        // O(t_a)
        --stop;                  // O(1)
        --preStop;                // O(1)
    }
}


```

```

    }
    // Insert the new value
    *stop = value;           // O(t_a)
    return stop;             // O(1)
}

```

and introduce a symbol  $t_c$  for the time to perform a comparison:

```

template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop, const Comparable& value)
{
    Iterator preStop = stop;      // O(1)
    --preStop;                  // O(1)
    while (stop != start && value < *preStop) { // comp: O(t_c)
        *stop = *preStop;          // O(t_a)
        --stop;                   // O(1)
        --preStop;                // O(1)
    }
    // Insert the new value
    *stop = value;              // O(t_a)
    return stop;                // O(1)
}

```

We can collapse the loop body:

```

template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop, const Comparable& value)
{
    Iterator preStop = stop;      // O(1)
    --preStop;                  // O(1)
    while (stop != start && value < *preStop) { // comp: O(t_c)
        // O(t_a)
    }
    // Insert the new value
    *stop = value;              // O(t_a)
    return stop;                // O(1)
}

```

How do we express the worst-case number of iterations of the loop? We no longer have the parameter `size` to tell us how many elements were in the container, but that's the *idea* that we need.

There is a way to express that notion within C++. There is a `std` function `distance(iterator1, iterator2)` that, for a pair of iterators, returns the number of distinct positions in the range starting at `iterator1` and going up to, but not including, `iterator2`.

So we could annotate the loop as

```
while (stop != start && value < *preStop) { // comp: O(t_c) #: distance(start,stop)
```

I like `distance(...)` as a notation for discussing iterators, but would **strongly discourage** you from ever using it in actual code. The reason is that `distance` is one of those functions that is implemented differently (using the same iterator traits trick that we used [earlier](#)) depending on what variety of iterators it is given.

If the two iterators are random-access iterators, `distance` is implemented as

```
template <typename Iterator>
size_t distance (Iterator start, Iterator stop)
{
    return stop - start;
}
```

which is  $O(1)$ .

But if the two iterators are not random access, then

`distance` is implemented as

```
template <typename Iterator>
size_t distance (Iterator start, Iterator stop)
{
    size_t counter = 0;
    while (start != stop)
    {
        ++counter;
        ++stop;
    }
    return counter;
}
```

which is  $O(\text{distance}(\text{start}, \text{stop}))$ .

So I have seen some *horrible* code like

```

template <typename Iterator>
void doSomething (Iterator start, Iterator stop)
{
    Iterator pos = start;
    for (int i = 0; i < distance(start, stop); ++i)
    {
        doSomethingWith(*pos);
        ++pos;
    }
}

```

which looks like it should be linear in the number of elements being processed, but is often quadratic instead.

but the expression `distance(start,stop)` has two problems:

- First, the value of `stop` is constantly changing inside the loop. What we really want to say is `distance(start,value_of_stop_when_we_first_entered_the_function)`. This is actually a common enough problem that we'll adopt a convention in this course:

For any parameter  $x$  to a function that is reassigned new values as the function runs, we will use the notation  $x_0$  to denote its value upon entry to the function. Inside the code, we write this as `x_0`.

So we could annotate that loop as

```
while (stop != start && value < *preStop) { // comp: O(t_c) #: distance(start,stop_0)
```

- But even that is more than I want to be typing out within mathematical expressions, so...

Let  $N$  denote `distance(start,stop_0)`. Then we have:

```

template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop, const Comparable& value)
{
    Iterator preStop = stop;      // O(1)
    --preStop;                  // O(1)
    while (stop != start && value < *preStop) { // comp: O(t_c) #: N
        // O(t_a)
    }
    // Insert the new value
    *stop = value;              // O(t_a)
    return stop;                // O(1)
}

```

We can then quickly conclude that

$$t_{\text{while}} = O(Nt_a + Nt_c)$$

```

template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop, const Comparable& value)
{
    Iterator preStop = stop;      // O(1)
    --preStop;                  // O(1)
    while (stop != start && value < *preStop) { // comp: O(t_c) #: N total: O(N t_a + N t_c)
        // O(t_a)
    }
    // Insert the new value
    *stop = value;              // O(t_a)
    return stop;                // O(1)
}

```

and eventually that the entire function is  $O(Nt_a + Nt_c)$  where  $N$  is `distance(start,stop)` and  $t_a$  and  $t_c$  denote the time required to assign and compare Comparable values, respectively.

## 3 Sequential Search

Another common utility function is to search through a range of data looking for a “search key”, returning the position where it is found and some “impossible” position if it cannot be found.

You might have seen code like this, for example,:

```

/*
 * Search an array for a given value, returning the index where
 * found or -1 if not found.
 *
 * From Malik, C++ Programming: From Problem Analysis to Program Design
 */

```

```

* @param list the array to be searched
* @param listLength the number of data elements in the array
* @param searchItem the value to search for
* @return the position at which value was found, or -1 if not found
*/
template <typename T>
int seqSearch(const T list[], int listLength, T searchItem)
{
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            return loc;

    return -1;
}

```

You may [run this algorithm](#) if you wish.

But we're going to look at the standard, iterator-based version. The std library has a function `find` that could be implemented as

```

template <typename Iterator, typename T>
Iterator find (Iterator start, Iterator stop, const T& key)
{
    while (start != stop && !(key == *start))
        ++start;
    return start;
}

```

- We check to see if start points to a value equal to key. If so, we return the start position. Otherwise, we increment it and keep going.
- If we run out of items to check (start has become equal to stop), we exit the loop and return start, which is equal to stop.

Note that we never examine the data at position stop, because ranges in the C++ iterator style are always “up to but not including” the second iterator. So we could not possibly find the key at position stop because we will never look there. Hence returning a value of stop is the “impossible” value return that says we could not find the key.

## 3.1 Analysis

Again, we can start the analysis by marking the “easy” stuff – the non-compound statements.

```

template <typename Iterator, typename T>
Iterator find (Iterator start, Iterator stop, const T& key)
{
    while (start != stop && !(key == *start))
        ++start; // O(1)
    return start; // O(1)
}

```

Looking at the while loop condition, `start != stop` and the `&&`, `!`, and `*` operations are  $O(1)$ . But `key == *start` depends on the underlying type `T`.

Let  $t_c$  denote the complexity of comparing two `T` values of equality. Then

```

template <typename Iterator, typename T>
Iterator find (Iterator start, Iterator stop, const T& key)
{
    while (start != stop && !(key == *start)) // cond: O(t_c)
        ++start; // O(1)
    return start; // O(1)
}

```

Let  $N$  denote `distance(start, stop)`, the number of elements we can search through.

```

template <typename Iterator, typename T>
Iterator find (Iterator start, Iterator stop, const T& key)
{
    while (start != stop && !(key == *start)) // cond: O(t_c) #: N
        ++start; // O(1)
    return start; // O(1)
}

```

The while loop is therefore  $O(Nt_c)$ :

```

template <typename Iterator, typename T>
Iterator find (Iterator start, Iterator stop, const T& key)
{
    while (start != stop && !(key == *start)) // cond: O(t_c) #: N total: O(N t_c)
        ++start; // O(1)
    return start; // O(1)
}

template <typename Iterator, typename T>
Iterator find (Iterator start, Iterator stop, const T& key)
{
    // O(N t_c)
}

```

```

    return start; // O(1)
}

```

and the entire function is  $O(Nt_c)$  where  $N$  denotes `distance(start, stop)` (the number of elements we can search through) and  $t_c$  denotes the complexity of comparing two  $\text{T}$  values of equality.

## 4 Ordered Sequential Search

In [our introduction to iterators](#) we looked at the interesting function `std::lower_bound`, which makes a compile-time choice of doing either an ordered search or a binary search depending on what kinds of iterators it is passed.

We wrap up these case studies by analyzing those two search functions, starting with the ordered sequential search.

```

template <typename Iterator, typename Value>
Iterator orderedSearch (Iterator start, Iterator stop, const Value& key)
{
    while ((start != stop) && (*start < key))
        ++start;
    return start;
}

```

We can mark the simple statements that are  $O(1)$ :

```

template <typename Iterator, typename Value>
Iterator orderedSearch (Iterator start, Iterator stop, const Value& key)
{
    while ((start != stop) && (*start < key))
        ++start; //O(1)
    return start; //O(1)
}

```

Looking at the while loop condition, the `(start != stop)` part will be  $O(1)$ , as will the applications of the `&&` and `*` operators. The `<` operator, however, is an operation provided by the unknown `Value` type, so

- Let  $t_c$  denote the time required to compare to objects of the `Value` type.

The loop condition is then  $O(t_c)$ .

```

Iterator orderedSearch (Iterator start, Iterator stop, const Value& key)
{
    while ((start != stop) && (*start < key)) //cond: O(t_c)
        ++start; //O(1)
    return start; //O(1)
}

```

**Question:** What is the worst case input for this function (for any fixed input size)?

Click to reveal +

The worst case occurs when key is nowhere in the range being searched. In that case, the loop repeats until start is advanced all the way forward to stop.

**Question: How many times does the loop repeat in that worst case?**

*Click to reveal*

distance( $\text{start}_0, \text{stop}$ ), a.k.a., the number of positions between (the initial value of) start and stop.

- Let  $N$  denote the number of positions between (the initial value of) start and stop.

Then we have

```
Iterator orderedSearch (Iterator start, Iterator stop, const Value& key)
{
    while ((start != stop) && (*start < key)) //cond: O(t_c) #: N
        ++start; //O(1)
    return start; //O(1)
}
```

So we have a loop that repeats  $N$  times, and the times of its body and condition do not depend on which iteration they are evaluated in. We've seen that pattern enough times to know how it works:

$$t_{\text{while}} = N * O(t_c) = O(N * t_c)$$

```
Iterator orderedSearch (Iterator start, Iterator stop, const Value& key)
{
    while ((start != stop) && (*start < key)) //cond: O(t_c) #: N total: O(N*t_c)
        ++start; //O(1)
    return start; //O(1)
}
```

Collapsing the loop:

```
Iterator orderedSearch (Iterator start, Iterator stop, const Value& key)
{
    // O(N*t_c)
    return start; //O(1)
}
```

and the total complexity is  $O(Nt_c)$ .

This function's time is linear in the number of elements being searched.

## 5 Binary Search

Finally, let's look at binary search.

```
template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0;
    auto high = stop - start - 1;

    while( low <= high )
    {
        auto mid = ( low + high ) / 2;
        RandomAccessIterator midPos = start + mid;
        if( *midPos < key )
            low = mid + 1;
        else if( key < *midPos )
            high = mid - 1;
        else
            return midPos; // Found
    }
    return start + low;
}
```

### 5.1 Starting the Analysis

Let's mark the simple  $O(1)$  statements to start:

```
template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0; // O(1)
    auto high = stop - start - 1; // O(1)

    while( low <= high )
    {
        auto mid = ( low + high ) / 2; // O(1)
        RandomAccessIterator midPos = start + mid; // O(1)
        if( *midPos < key ) // O(1)
            low = mid + 1; // O(1)
```

```

        else if( key < *midPos )
            high = mid - 1;                      // O(1)
        else
            return midPos; // Found           // O(1)
    }
    return start + low;                      // O(1)
}

```

Now look at the innermost if (the “else if”). The then parts and else parts are already marked as  $O(1)$ . The condition, however, depends on the time required to compare two elements of the unknown value type.

- Let  $t_c$  denote the time required to compare to objects of the value type.

Then the condition is  $O(t_c)$ :

```

template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0;                                // O(1)
    auto high = stop - start - 1;                 // O(1)

    while( low <= high )
    {
        auto mid = ( low + high ) / 2;           // O(1)
        RandomAccessIterator midPos = start + mid; // O(1)
        if( *midPos < key )
            low = mid + 1;                      // O(1)
        else if( key < *midPos )                // cond: O(t_c)
            high = mid - 1;                     // O(1)
        else
            return midPos; // Found           // O(1)
    }
    return start + low;                          // O(1)
}

```

So that if takes time  $O(t_c) + \max O(1), O(1) = O(t_c)$ .

```

template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0;                                // O(1)
    auto high = stop - start - 1;                 // O(1)

    while( low <= high )
    {
        auto mid = ( low + high ) / 2;           // O(1)
        RandomAccessIterator midPos = start + mid; // O(1)
        if( *midPos < key )
            low = mid + 1;                      // O(1)
        else if( key < *midPos )                // cond: O(t_c) total: O(t_c)
            high = mid - 1;                     // O(1)
        else
            return midPos; // Found           // O(1)
    }
    return start + low;                          // O(1)
}

```

Collapsing:

```

template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0;                                // O(1)
    auto high = stop - start - 1;                 // O(1)

    while( low <= high )
    {
        auto mid = ( low + high ) / 2;           // O(1)
        RandomAccessIterator midPos = start + mid; // O(1)
        if( *midPos < key )
            low = mid + 1;                      // O(1)
        else
            // O(t_c)
    }
    return start + low;                          // O(1)
}

```

The remaining if also has a condition that takes time  $O(t_c)$ :

```

template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,

```

```

        RandomAccessIterator stop,
        const Value& key)
{
    auto low = 0;                                // O(1)
    auto high = stop - start - 1;                 // O(1)

    while( low <= high )
    {
        auto mid = ( low + high ) / 2;           // O(1)
        RandomAccessIterator midPos = start + mid; // O(1)
        if( *midPos < key )          //cond: O(t_c)
            low = mid + 1;                // O(1)
        else
            // O(t_c)
    }
    return start + low;                         // O(1)
}

```

which means that this `if` takes time  $O(t_c) + \max(O(1), O(t_c))$ .

Despite the more expensive `else` part than in the previously analyzed `if`, this still simplifies to  $O(t_c)$ .

```

template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0;                                // O(1)
    auto high = stop - start - 1;                 // O(1)

    while( low <= high )
    {
        auto mid = ( low + high ) / 2;           // O(1)
        RandomAccessIterator midPos = start + mid; // O(1)
        if( *midPos < key )          //cond: O(t_c) total: O(t_c)
            low = mid + 1;                // O(1)
        else
            // O(t_c)
    }
    return start + low;                         // O(1)
}

```

Collapsing,

```

template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0;                                // O(1)
    auto high = stop - start - 1;                 // O(1)

    while( low <= high )
    {
        auto mid = ( low + high ) / 2;           // O(1)
        RandomAccessIterator midPos = start + mid; // O(1)
        // O(t_c)
    }
    return start + low;                         // O(1)
}

```

which means that the loop body is  $O(1) + O(1) + O(t_c) = O(t_c)$ .

```

template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0;                                // O(1)
    auto high = stop - start - 1;                 // O(1)

    while( low <= high )
    { //total: O(t_c)
        auto mid = ( low + high ) / 2;           // O(1)
        RandomAccessIterator midPos = start + mid; // O(1)
        // O(t_c)
    }
    return start + low;                         // O(1)
}

```

Collapsing,

```

template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0;                                // O(1)

```

```

auto high = stop - start - 1;           // O(1)
while( low <= high )
{
    // O(t_c)
}
return start + low;                  // O(1)
}

```

The loop condition is  $O(1)$ :

```

template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0;                      // O(1)
    auto high = stop - start - 1;      // O(1)

    while( low <= high ) //cond: O(1)
    {
        // O(t_c)
    }
    return start + low;              // O(1)
}

```

Now, how many times does this loop repeat?

To answer this question let's go back to the original listing.

```

template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0;
    auto high = stop - start - 1;

    while( low <= high )
    {
        auto mid = ( low + high ) / 2;
        RandomAccessIterator midPos = start + mid;
        if( *midPos < key )
            low = mid + 1;
        else if( key < *midPos )
            high = mid - 1;
        else
            return midPos; // Found
    }
    return start + low;
}

```

Remember how this function actually works.

- $\text{low} \dots \text{high}$  define our current search area. Initially, there are  $\text{distance}(\text{start}, \text{stop})$  items in this area.
- The two values  $\text{low}$  and  $\text{high}$  define our current search range. If the value we are looking for is somewhere in the array, it is going to be at a position  $\geq \text{low}$  and  $\leq \text{high}$ . So the difference,  $\text{high} - \text{low} + 1$ , defines how many values we have left in the search range.

On any given iteration of the loop:

- there are  $\text{high} - \text{low} + 1$  items in the search area
  - Each time around the loop, we cut this area in half.
  - We stop when the search area has been reduced to a single item.
- Let  $N$  denote  $\text{distance}(\text{start}, \text{stop})$ , the number of positions being searched.

How many times can we divide  $N$  things into 2 equal parts before getting down to only 1?

That's the interesting question!

## 5.2 Logarithmic Behavior

If I start with  $N$  things in this search area, how many times can I keep cutting that search area of  $N$  things in half till I get to only a single item? (An item that *must* be the value we are looking for, if that value is anywhere in the array at all.)

Let's assume for the sake of our argument that  $N$  is some power of 2 to start with. So we cut  $N$  in half – we get  $N/2$ . Next time we cut that half which makes  $N/4$ , then  $N/8$  and then  $N/16$  and so on. And the question is, how often can we keep doing that until we reduce the number down to just 1?

The answer may be a bit clearer if we turn the problem around. Start at 1 and keep doubling until we get  $N$ . So we proceed

1, 2, 4, ...

and we keep going until we actually get up to  $N$ . The number of steps we did in doubling is same as number of steps when we start at  $N$  and kept dividing by 2.

How many steps is that? Well, what power of 2 do we reach when we finally get to  $N$ ? Suppose we took  $k$  steps. Then we are saying that  $N = 2^k$ . Solving for  $k$ , we get  $k = \log_2 N$ .

The use of  $\log_2$  is so common in Computer Science that we generally assume that any logarithm written without its base is to the base 2. In other words, computer scientists would usually write this as  $k = \log N$ . (That's in contrast to other mathematical fields, where a logarithm with no base is generally assumed to be base 10.)

What if  $N$  is not an exact power of 2? In that case,  $\log_2 N$  is not an integer, and we would need to divide or double

$$\lceil \log_2 N \rceil$$

times, where  $\lceil \cdot \rceil$  denotes the *ceiling* or “next integer higher than”.

In practice, we won't need to worry about that fractional part, because we are going to use this inside a big-O expression, where the fractional part will be dominated in any sum by the logarithm itself.

## 5.3 Back to the Analysis

```
template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0;                                // O(1)
    auto high = stop - start - 1;                 // O(1)

    while( low <= high ) //cond: O(1) #: log N
    {
        // O(t_c)
    }
    return start + low;                         // O(1)
}
```

Again, the condition and body times are independent of which iteration we are in, so

$$t_{\text{while}} = (\log N) * (O(1) + O(t_c)) + O(1) = O(t_c \log N)$$

```
template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0;                                // O(1)
    auto high = stop - start - 1;                 // O(1)

    while( low <= high ) //cond: O(1) #: log N  total: O(t_c log(N))
    {
        // O(t_c)
    }
    return start + low;                         // O(1)
}
```

Collapsing,

```
template <typename RandomAccessIterator, typename Value>
RandomAccessIterator binarySearch (RandomAccessIterator start,
                                  RandomAccessIterator stop,
                                  const Value& key)
{
    auto low = 0;                                // O(1)
    auto high = stop - start - 1;                 // O(1)

    // O(t_c log(N))

    return start + low;                         // O(1)
}
```

and the total complexity of the function is  $O(t_c \log(N))$  where  $t_c$  is the time required to compare two elements and where  $N$  is the number of positions being searched.

## 6 Appendix: std Functions Analyzed in This Lesson

The following functions and function templates have been analyzed in this lesson and are now available for use in quizzes, assignments, and later lessons:

### 6.1 std::find

```
template <typename Iterator, typename T>
Iterator find (Iterator start, Iterator stop, const T& key)
```

- $O(n * t_c)$  where  $n$  is `distance(start, stop)` (the # of positions between `start` and `stop`) and  $t_c$  is the time required to compare two items of type `T` for equality.

## 6.2 std::lower\_bound

```
template <typename Iterator, typename Value>
Iterator lower_bound (Iterator start, Iterator stop, const Value& key)
```

Let  $n$  be `distance(start, stop)` (the # of positions between `start` and `stop`) and let  $t_c$  is the time required to compare two items of type `T` using `operator<`.

- If `Iterator` is a random access iterator, this function uses binary search and is  $O(t_c \log n)$ .
- If `Iterator` is not a random access iterator (e.g., a forward or bi-directional iterator), this function uses ordered sequential search and is  $O(t_c n)$ .

# Vectors

Steven J. Zeil

Last modified: Apr 24, 2019

## Contents:

<a href="#">1 The Vector Interface</a>
<a href="#">1.1 Constructors</a>
<a href="#">1.2 Copying</a>
<a href="#">1.3 Size &amp; Capacity</a>
<a href="#">1.4 Access to Elements</a>
<a href="#">1.5 Inserting Elements</a>
<a href="#">1.6 Removing Elements</a>
<a href="#">2 Using Vectors</a>
<a href="#">2.1 Keeping Authors in a Book</a>
<a href="#">2.2 Vectorizing the book.h file</a>
<a href="#">2.3 Vectorizing the book.cpp file</a>
<a href="#">2.4 Vectors copy nicely</a>
<a href="#">2.5 Completing the Book operations</a>
<a href="#">3 Required Performance</a>

A vector is similar to an array, except that it can “grow” to accommodate as many items as you actually try to put into it:

- A very useful structure.
  - We’ll use it to implement many other ADTs.
- Your text introduces a somewhat simplified version of the `std` interface (called `vector` to avoid clashing with the `std` class.)

We’ll use that simplified form when we look at how to implement vectors, but will start by studying the `std::vector` interface so that you know what you can do when programming applications of it.

First, however, a slight digression into C++-isms. I stated earlier that “A vector is similar to an array”. The similarity comes from the fact that vectors keep their elements in numeric order and allow you to access them via indexing:

```
int a[];  
vector<int> v;  
:  
a[i] = v[i+1];
```

```
string b[];  
vector<string> w;  
:  
w[23] = "Hello";
```

```
Folderol c[];  
vector<Folderol> x;  
:  
x[1] = c[1];
```

Because vectors are templates, we can have vectors containing almost any other type.

Why “almost any type”? You might remember that not every type can be used in an array. A type can only be used as array elements if that type provides a default constructor, because that constructor is used to initialize the elements of the array.

Similarly, types that are to be used as vector elements have a special requirement as well. This requirement is different from, but no more stringent than the requirement for arrays. Types to be used in a vector must supply a *copy constructor*.

## 1 The Vector Interface

Let’s take a quick look at how to work with vectors.

### 1.1 Constructors

```
'vector<int> v1;'
```

`v1` is a vector of integers. Initially, it has 0 elements.

```
'vector<string> v2(10, "abc");'
```

`v2` is a vector of strings initialized to hold 10 copies of “abc”.

```
'vector<string> v3 (containerOfStrings.begin(),  
                     containerOfStrings.end());'
```

```
template <class T>  
class vector {  
public:  
  // types:  
  typedef T& reference;  
  typedef const T& const_reference;  
  typedef ... iterator;  
  typedef ... const_iterator;  
  typedef ... size_type;  
  typedef ... difference_type;  
  typedef T value_type;  
  typedef T* pointer;
```

v3 is initialized to hold a copy of the contents of some other container. That container does not have to be another vector.

```
vector<int> v4 = {2, 3, 5, 999};
```

v4 is initialized using an initializer list.

```
typedef const T* const_pointer
typedef std::reverse_iterator<iterator>
    reverse_iterator;
typedef std::reverse_iterator<const_iterator>
    const_reverse_iterator;

// construct / copy / destroy
vector();
explicit vector(size_type n, const T& value = T());

template <class InputIterator>
vector(InputIterator first, InputIterator last);

vector(initializer_list<value_type> il);

vector(const vector<T>& x);

~vector();
```

## 1.2 Copying

These operations replace the entire contents of a vector by something new:

Two ways to copy an entire vector from another vector:

```
vector<double> v, w;
⋮
v = w;
vector<double> z(v);
```

Other options:

```
v.assign(container.begin(), container.end());
```

You can **assign** all or part of another container to a vector.

```
v.assign(24, "abc");
```

You can also **assign** some number of copies of a single value to a vector.

```
vector(const vector<T>& x);

~vector();

vector<T>& operator=(const vector<T>& x);

template <class InputIterator>
void assign(InputIterator first, InputIterator last);

void assign(size_type n, const T& u);
```

## 1.3 Size & Capacity

```
vector<int>::size_type n;
n = v1.size();
```

How many elements currently in v1?

```
v1.resize(n, x);
```

If v1 has more than n elements, remove the last  $v1.size() - n$  elements. If it has fewer than n elements, add  $n - v1.size()$  copies of x to the end. (x may be omitted, in which case the added elements are formed using their data type's default constructor.)

```
vector<int>::size_type n;
n = v1.capacity();
```

How many elements total could v1 hold without requesting more memory? (We'll see later that this affects performance.)

```
v1.reserve(n);
```

Add memory, if necessary, to make sure v1 will have room to hold at least n elements. Note that this does not actually change the `size()` of the vector. It just helps make sure there is enough space for future growth.

```
// capacity
size_type size() const;
size_type max_size() const;
void resize(size_type sz, T c = T());
size_type capacity() const;
bool empty() const;
void reserve(size_type n);
```

## 1.4 Access to Elements

```
v1[0] = v1[1] + 1;
```

You can index into a vector just like with arrays. Note that you can only access `v[i]` if the `size()` of the vector is already at least `i+1`.

```
// element access:
reference operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference at(size_type n);
```

You can also use the at functions:

```
v1.at(0) = v1.at(1) + 1;
```

The difference between at and the square bracket indexing is that access via at is checked to be sure that the index is legal, between 0 and size() - 1, inclusive. Normally, this is not checked for square bracket style indexing (though the g++ compiler allows you to [request such checking](#) through the use of appropriate compiler flags).

```
x = v1.front();  
v1.front() = v1.front()+1;
```

Provides access to the first element in the vector.

```
x = v1.back();  
v1.back() = v1.back()+1;
```

Provides access to the last element in the vector.

## 1.5 Inserting Elements

```
v2.push_back("def");
```

Adds a new element, "def" to the end of the vector v2. v2.size() increases by 1. This is the "normal", most common way to add elements to a vector.

Although not as efficient, the std::vector also allows:

```
vector<std::string>::iterator pos;  
:  
v2.insert("def", pos);
```

Inserts a new element, "def", at the indicated position within the vector v2. Any elements already in v2 at that position or higher are moved back one position to make room. v2.size() increases by 1.

```
reference front();  
const_reference front() const;  
reference back();  
const_reference back() const;
```

```
// modifiers:  
void push_back(const T& x);  
void pop_back();  
iterator insert(iterator position, const T& x);  
void insert(iterator position, size_type n, const T& x);  
  
template <class InputIterator>  
void insert(iterator position,  
           InputIterator first, InputIterator last);  
  
iterator erase(iterator position);  
iterator erase(iterator first, iterator last);  
void swap(vector<T>&);  
void clear();
```

## 1.6 Removing Elements

```
v2.pop_back();
```

Removes the element at the end of the vector, decreasing the size() by 1.

Although not as efficient, the std::vector also allows:

```
vector<std::string>::iterator pos;  
:  
v2.erase(pos);
```

Removes the element at the indicated position within the vector v2. Any elements already in v2 at higher positions are moved forward one position to "take up the slack". v2.size() decreases by 1.

```
vector<std::string>::iterator start, stop;  
:  
v2.erase(start, stop);
```

Removes the elements at the indicated positions from start up to, but not including stop, within the vector v2. Any elements already in v2 at higher positions are moved forward to "take up the slack". v2.size() decreases by the number of elements removed.

You can remove all elements from a vector like this:

```
vector<std::string>::iterator pos;  
:  
v2.clear();
```

```
// modifiers:  
void push_back(const T& x);  
void pop_back();  
iterator insert(iterator position, const T& x);  
void insert(iterator position, size_type n, const T& x);  
  
template <class InputIterator>  
void insert(iterator position,  
           InputIterator first, InputIterator last);  
  
iterator erase(iterator position);  
iterator erase(iterator first, iterator last);  
void swap(vector<T>&);  
void clear();
```

# 2 Using Vectors

## 2.1 Keeping Authors in a Book

[Earlier](#), we developed a Book class that used std::array to hold all of the authors in a book.

[abook.h](#) +

```
#ifndef Book_H
#define Book_H

#include <initializer_list>
#include <string>
#include <array>
#include "author.h"

class Publisher;

class Book {
private:
    std::string title;
    const Publisher* publisher;
    int numAuthors;
    std::string isbn;
    static const int MaxAuthors = 12;
    std::array<Author,MaxAuthors> authors;

public:
    typedef std::array<Author,MaxAuthors>::iterator iterator;
    typedef std::array<Author,MaxAuthors>::const_iterator const_iterator;

    Book();
    Book (std::string theTitle, const Publisher* publ,
          std::initializer_list<Author> theAuthors,
          std::string theISBN);

    template <typename Iterator>
    Book (std::string theTitle, const Publisher* publ,
          Iterator startAuthors, Iterator stopAuthors,
          std::string theISBN);

    std::string getTitle() const {return title;}
    void setTitle(std::string theTitle) {title = theTitle;}

    int getNumberOfAuthors() const;

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

    void addAuthor (Author);
    void removeAuthor (Author);

    const Publisher* getPublisher() const {return publisher;}
    void setPublisher(const Publisher* publ) {publisher = publ;}

    std::string getISBN() const {return isbn;}
    void setISBN(std::string id) {isbn = id;}
};

template <typename Iterator>
Book::Book (std::string theTitle, const Publisher* publ,
           Iterator startAuthors, Iterator stopAuthors,
           std::string theISBN)
: title(theTitle), publisher(publ), numAuthors(0),
  isbn(theISBN)
{
    while (startAuthors != stopAuthors)
    {
        authors[numAuthors] = *startAuthors;
        ++numAuthors;
        ++startAuthors;
    }
}

#endif
```

[abook.cpp](#) +

```
/*
 * book.cpp
 *
 * Created on: May 23, 2018
 * Author: zeil
 */

#include "abook.h"
#include <cassert>
```

```

#include <algorithm>
using namespace std;

Book::Book()
: title(), publisher(nullptr), numAuthors(0),
isbn()
{
}

Book::Book (std::string theTitle, const Publisher* publ,
std::initializer_list<Author> theAuthors,
std::string theISBN)
: title(theTitle), publisher(publ), numAuthors(theAuthors.size()),
isbn(theISBN)
{
    int i = 0;
    for (const Author& au: theAuthors)
    {
        authors[i] = au;
        ++i;
    }
}

int Book::getNumberOfAuthors() const
{
    return numAuthors;
}

Book::iterator Book::begin()
{
    return authors.begin();
}
Book::const_iterator Book::begin() const
{
    return authors.begin();
}
Book::iterator Book::end()
{
    return authors.begin() + numAuthors;
}
Book::const_iterator Book::end() const
{
    return authors.begin() + numAuthors;
}

void Book::addAuthor (Author au)
{
    assert(numAuthors < MaxAuthors);
    authors[numAuthors] = au;
    ++numAuthors;
}
void Book::removeAuthor (Author au)
{
    auto pos = find(begin(), end(), au);
    if (pos != end())
    {
        copy (pos+1, end(), pos);
        --numAuthors;
    }
}

```

Because standard arrays must be constructed with a fixed size, we have to have a separate data member `numAuthors` to track how many of those array elements are actually in use.

We may be able to change that if we switch to a vector-based implementation.

## 2.2 Vectorizing the book.h file

Changing the data structure to vector is surprisingly simple:

Here is what we had:

```

class Book {
private:
    std::string title;
    const Publisher* publisher;
    int numAuthors;
    std::string isbn;
    static const int MaxAuthors = 12;
    std::array<Author,MaxAuthors> authors;
}

```

```
public:
    typedef std::array<Author,MaxAuthors>::iterator iterator;
```

and here is what we change it to:

```
class Book {
private:
    std::string title;
    const Publisher* publisher;
    std::string isbn;
    std::vector<Author> authors;

public:
    typedef std::vector<Author>::iterator iterator;
    typedef std::vector<Author>::const_iterator const_iterator;
```

- Obviously, the `std::array` uses will be replaced by `std::vector`.
- `MaxAuthors` disappears. Because vectors can grow, we don't need a pre-set maximum.
- `numAuthors` disappears. We don't need to know how many elements out of the vector actually contain author data. All of them will.

The last point highlights the difference in style between using arrays and using vectors. Most applications of arrays divide the array into used portions and portions available for future expansion. When we work with vectors, however, we only add as many "slots" to the vectors as we have data elements to fill them. (As we will see, the underlying implementation of vector may still use arrays, and those arrays will be divided into used and unused portions. But when we are *using* vectors that division is almost invisible.)

## 2.3 Vectorizing the book.cpp file

Probably nothing highlights the difference in styles between these two approaches than the following two functions:

With arrays:

```
int Book::getNumberOfAuthors() const
{
    return numAuthors;
}

void Book::addAuthor (Author au)
{
    assert(numAuthors < MaxAuthors);
    authors[numAuthors] = au;
    ++numAuthors;
}
```

With vectors:

```
int Book::getNumberOfAuthors() const
{
    return authors.size();
}

void Book::addAuthor (Author au)
{
    authors.push_back(au);
}
```

We use `push_back` operations to add new authors to the end of the vector-based collection, and the collection tracks its own size.

You can also see some differences in the constructors:

Arrays:

```
Book::Book (std::string theTitle, const Publisher* publ,
           std::initializer_list<Author> theAuthors,
           std::string theISBN)
: title(theTitle), publisher(publ), numAuthors(theAuthors.size()),
  isbn(theISBN)
{
    int i = 0;
    for (const Author& au: theAuthors)
    {
        authors[i] = au;
        ++i;
    }
}
```

and vectors:

```
Book::Book (std::string theTitle, const Publisher* publ,
           std::initializer_list<Author> theAuthors,
           std::string theISBN)
: title(theTitle), publisher(publ),
  isbn(theISBN), authors(theAuthors.begin(), theAuthors.end())
```

```
{  
}
```

For vectors, we no longer have a **separate data member** to track the number of authors. And we can **initialize a vector from a range of iterators** instead of needing to copy those elements one by one into the array.

## 2.4 Vectors copy nicely

We don't provide our own implementation of the Big 3 in either of these versions, because both `std::array` and `std::vector` can be copied & assigned freely, and vectors clean up after themselves with an appropriate destructor.

## 2.5 Completing the Book operations

The only other difference of note is the provision of the iterators, particularly, the `end` function.

For arrays the "end" position is the first unused slot in the array:

```
Book::iterator Book::begin()  
{  
    return authors.begin();  
}  
Book::const_iterator Book::begin() const  
{  
    return authors.begin();  
}  
Book::iterator Book::end()  
{  
    return authors.begin() + numAuthors;  
}  
Book::const_iterator Book::end() const  
{  
    return authors.begin() + numAuthors;  
}
```

For vectors, all slots are used, so the end position is the position after the last slot:

```
Book::iterator Book::begin()  
{  
    return authors.begin();  
}  
Book::const_iterator Book::begin() const  
{  
    return authors.begin();  
}  
Book::iterator Book::end()  
{  
    return authors.end();  
}  
Book::const_iterator Book::end() const  
{  
    return authors.end();  
}
```

The full implementation of the vector-based `Book` is below:

```
book.h +  
  
#ifndef BOOK_H  
#define BOOK_H  
  
#include <initializer_list>  
#include <string>  
#include <vector>  
#include "author.h"  
  
class Publisher;  
  
class Book {  
private:  
    std::string title;  
    const Publisher* publisher;  
    std::string isbn;  
    std::vector<Author> authors;  
  
public:  
    typedef std::vector<Author>::iterator iterator;  
    typedef std::vector<Author>::const_iterator const_iterator;  
  
    Book();  
  
    Book (std::string theTitle, const Publisher* publ,  
          std::initializer_list<Author> theAuthors,  
          std::string theISBN);
```

```

template <typename Iterator>
Book (std::string theTitle, const Publisher* publ,
      Iterator startAuthors, Iterator stopAuthors,
      std::string theISBN);

    std::string getTitle() const {return title;}
    void setTitle(std::string theTitle) {title = theTitle;}

    int getNumberOfAuthors() const;

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

    void addAuthor (Author);
    void removeAuthor (Author);

    const Publisher* getPublisher() const {return publisher;}
    void setPublisher(const Publisher* publ) {publisher = publ; }

    std::string getISBN() const {return isbn;}
    void setISBN(std::string id) {isbn = id; }

};

template <typename Iterator>
Book::Book (std::string theTitle, const Publisher* publ,
           Iterator startAuthors, Iterator stopAuthors,
           std::string theISBN)
: title(theTitle), publisher(publ),
  isbn(theISBN), authors(startAuthors, stopAuthors)
{
}

#endif

```

### [book.cpp](#) +

```

/*
 * book.cpp
 *
 * Created on: May 23, 2018
 * Author: zeil
 */

#include "book.h"
#include <cassert>
#include <algorithm>

using namespace std;

Book::Book()
: title(), publisher(nullptr),
  isbn(), authors()
{
}

Book::Book (std::string theTitle, const Publisher* publ,
           std::initializer_list<Author> theAuthors,
           std::string theISBN)
: title(theTitle), publisher(publ),
  isbn(theISBN), authors(theAuthors.begin(), theAuthors.end())
{
}

int Book::getNumberOfAuthors() const
{
    return authors.size();
}

Book::iterator Book::begin()
{
    return authors.begin();
}

Book::const_iterator Book::begin() const
{
    return authors.begin();
}

Book::iterator Book::end()
{
    return authors.end();
}

Book::const_iterator Book::end() const
{
}

```

```

        return authors.end();
    }

void Book::addAuthor (Author au)
{
    authors.push_back(au);
}
void Book::removeAuthor (Author au)
{
    auto pos = find(begin(), end(), au);
    if (pos != end())
    {
        authors.erase(pos);
    }
}

```

## 3 Required Performance

The C++ standard specifies that a legal (i.e., standard-conforming) implementation of `vector` must satisfy the following performance requirements:

Operation	Speed
<code>vector()</code>	$O(1)$
<code>vector(n, x)</code>	$O(n)$
<code>size()</code>	$O(1)$
<code>v[ i ]</code>	$O(1)$
<code>push_back(x)</code>	$O(1)$
<code>pop_back</code>	$O(1)$
<code>insert</code>	$O(\text{size}())$
<code>erase</code>	$O(\text{size}())$
<code>front, back</code>	$O(1)$

Next, we will look at how `vector` actually manages to achieve this performance.

# Implementing the Vector Class

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

<a href="#">1 Implementing a vector class</a>
<a href="#">1.1 Retrieving elements by index</a>
<a href="#">1.2 push_back: Adding to the end of a vector</a>
<a href="#">1.3 Coding push_back</a>
<a href="#">2 Performance</a>
<a href="#">2.1 Looking at push_back</a>
<a href="#">2.2 Using reserve() to get a True O(1) Worst Case</a>
<a href="#">2.3 Summary</a>

Now that we have a sense of how to use vectors, let's talk about how we can implement a vector.

- Your text introduces a somewhat simplified version (vector) of the std interface.

## Required Performance

The C++ standard specifies that a legal (i.e., standard-conforming) implementation of vector must satisfy the following performance requirements:

Operation	Speed
vector()	O(1)
vector(n, x)	O(n)
size()	O(1)
v[i]	O(1)
push_back(x)	O(1)
pop_back	O(1)
insert	O(size())
erase	O(size())
front, back	O(1)

## 1 Implementing a vector class

### 1.1 Retrieving elements by index

- We want to support *operator[]* in  $O(1)$  time.
- Natural choice is to use an array.

For example, we could declare the data members for a vector like this:

```
template <typename Object>
class Vector
{
    :
    private:
        int theSize;
        int theCapacity;
        Object* objects;
};
```

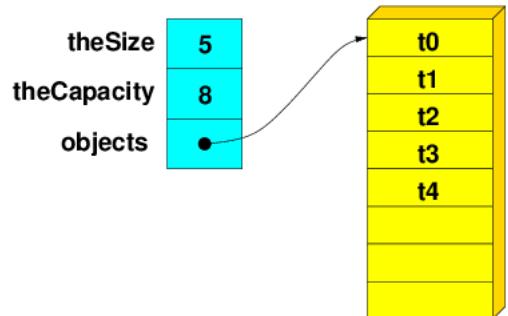
with objects being a pointer to a dynamically allocated array.

The theSize and theCapacity variables track how many elements are in the array and how large the array is, respectively.

This makes many of the operations very easy to implement. For example, indexing into the vector is done by applying the same index to the array vArr.

```
Object & operator[]( int index )
{
    return objects[ index ];
}

const Object & operator[]( int index ) const
{
    return objects[ index ];
}
```



We could, by the way, make this a bit safer:

```
Object & operator[]( int index )
{
    if( index < 0 || index >= size( ) )
        throw ArrayIndexOutOfBoundsException{ };
    return objects[ index ];
}

const Object & operator[]( int index ) const
{
    if( index < 0 || index >= size( ) )
        throw ArrayIndexOutOfBoundsException{ };
    return objects[ index ];
}
```

by checking to be sure the index is in a valid range. Beware, however, because most implementations of `std::vector` do not do this kind of checking for you.

## 1.2 push\_back: Adding to the end of a vector

But the real question is, how do we allow vectors to grow to arbitrary size?

We'll split this into cases:

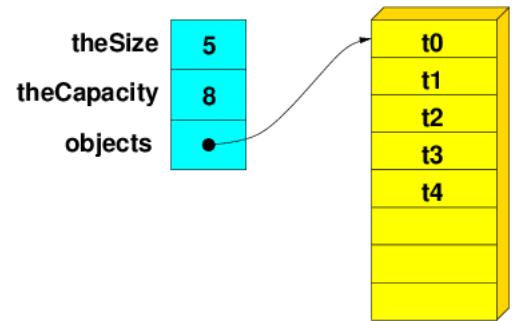
### 1.2.1 Adding to an Array That Has Unused Space

Let's consider the operation of adding a new element to the end of this vector,

```
v.push_back(t5);
```

If there is room in the array, we just add our new data element.

```
void push_back( const Object & x )
{
    :
    objects[ theSize++ ] = x;
}
```

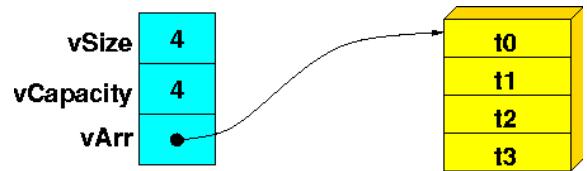


### 1.2.2 Adding to an Array That is Full

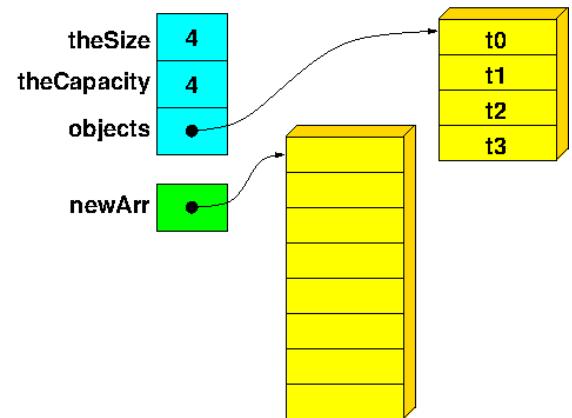
If the data array is already filled to capacity, and we try to add more to it,

```
v.push_back(t4);
```

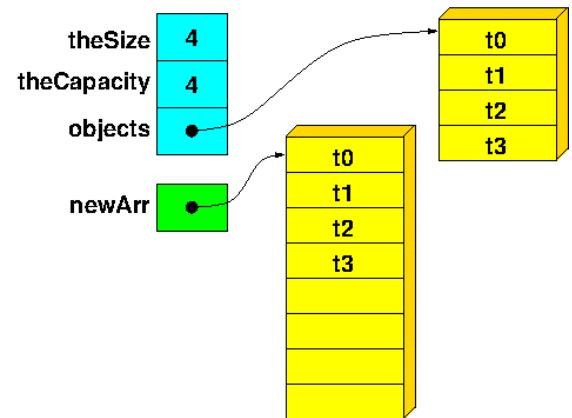
we make another that is twice as big ...



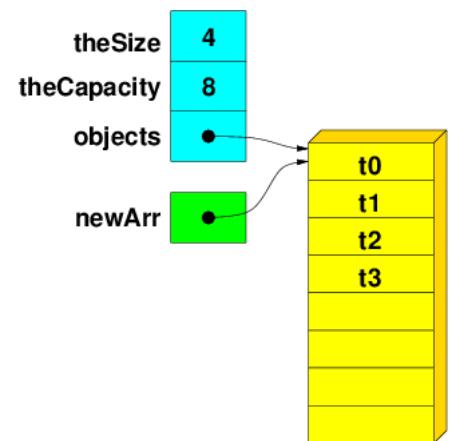
First the new data area is allocated.



Then the old data is copied into the new array.

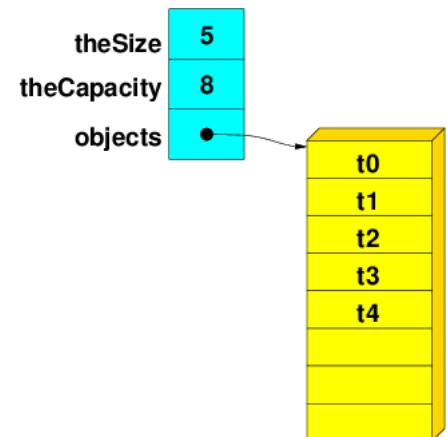


The old array is deleted, and the vArr and theCapacity fields updated.



Now we have room add the new element to the end of the vector.

We'll see later what this does to the big-O complexity for `push_back()`.



## 1.3 Coding `push_back`

Now, let's look at the code to accomplish all this.

Remember the basic steps:

- If the data array is already filled to capacity, and we try to add more to it, we make another that is twice as big ...
  - First the new data area is allocated.
  - Then the old data is copied into the new array.
  - The old array is deleted, and the `vArr` and `theCapacity` fields updated.
- Now we can add the new element to the end of the vector.

### 1.3.1 Are we Full?

```
void push_back( const Object & x )
{
    if( theSize == theCapacity ) ①
        reserve( 2 * theCapacity + 1 ); ②
    objects[ theSize++ ] = x;
}
```

- If the data array is already filled to capacity ①, and we try to add more to it,
- then we need to create a new array that is twice as large ②
  - this is accomplished via the function `reserve`, which is part of the public interface of `vector`.

`v.reserve(n)` guarantees that `v` has enough storage to contain at least `n` elements without requiring any additional memory allocation.

### 1.3.2 Reserve: reserving space for future growth

```
void reserve( int newCapacity )
{
    if( newCapacity < theSize ) ①
        return;

    Object *newArray = new Object[ newCapacity ]; ②
    for( int k = 0; k < theSize; ++k ) ③
        newArray[ k ] = std::move( objects[ k ] );

    theCapacity = newCapacity;
    std::swap( objects, newArray ); ④
    delete [ ] newArray; ⑤
}
```

- If the vector's array is already large enough ① to hold at least `newCapacity` elements, then `reserve` returns immediately without doing anything.
- Otherwise, a new array is allocated ②
- And the existing data is copied from the old array to the new one ③
- The vector's own recorded maximum capacity is updated, and the two array pointers are swapped ④ so that the vector itself now points to the new array, while `newArray` now points to the older, smaller array.
- And finally we can clean up by deleting the original array ⑤

### 1.3.3 push\_back: add to an array that has space

```
void push_back( const Object & x )
{
    if( theSize == theCapacity )
        reserve( 2 * theCapacity + 1 );
    objects[ theSize++ ] = x; ①
}
```

When we return to push\_back from the reserve call ①, we know that we have enough room to insert our new element x onto the end of the array.

You can [run this algorithm](#) and other vector operations until you are familiar with how it works.

## 2 Performance

A reminder: the standard promises this:

Operation	Speed
vector()	O(1)
vector(n, x)	O(n)
size()	O(1)
v[i]	O(1)
push_back(x)	O(1)
pop_back	O(1)
insert	O(size())
erase	O(size())
front, back	O(1)

Did we deliver?

### 2.1 Looking at push\_back

```
pushback.cpp +
```

```
void reserve( int newCapacity )
{
    if( newCapacity < theSize )
        return;

    Object *newArray = new Object[ newCapacity ];
    for( int k = 0; k < theSize; ++k )
        newArray[ k ] = std::move( objects[ k ] );

    theCapacity = newCapacity;
    std::swap( objects, newArray );
    delete [ ] newArray;
}

// Stacky stuff
void push_back( const Object & x )
{
    if( theSize == theCapacity )
        reserve( 2 * theCapacity + 1 );
    objects[ theSize++ ] = x;
}
```

The only apparent problem is push\_back().

- When we add to the end of a vector that is already filled to capacity(), we

- allocate a new array with twice the capacity() O(1)

Actually, allocating an array the way it is done here is really  $O(\text{newCapacity})$ , because the default constructor would be invoked on each of the new elements. But “real” implementations of vector use a more primitive system function to allocate space without initializing it, avoiding that cost.

Even at  $O(\text{newCapacity})$ , though, this would not change our final result.  $\text{newCapacity} == 2 * \text{size}()$ , so we would simplify this to say the allocation is  $O(\text{size}())$ , and when we add up all the other steps described on this page, the final result would be unchanged.

- copy the old elements into the new array  $O(\text{size}())$
  - discard the old array O(1)

Again, this is a simplification. “really” deleting the array would be  $O(\text{size}())$  because the destructor would be invoked on each array element. But real implementations have ways around that as well. In fact, that’s the reason that `std::move` is used in the copy above.

- add the new element to the end O(1)

- Total is  $O(\text{size}())$

### 2.1.1 push\_back is, worst-case, O(size())

Total is  $O(\text{size}())$ .

That would seem to violate the C++ standard's requirement that `push_back` run in  $O(1)$  time.

But not *every* `push_back()` takes  $O(\text{size}())$  time.

- only when we have filled the array
- otherwise, it takes  $O(1)$  time

Let's look at the issue from a slightly different point of view:

How long does it take to do a total of  $n$  `push_back()` operations, starting with an empty vector?

### 2.1.2 Doing N push\_backs

Let  $k$  be the smallest integer such that  $n \leq 2^k$ . For the sake of simplicity, we'll do the analysis as if we were actually going to do  $2^k$  pushes.

- Let  $m$  be the current `size()` of the vector.
  - If  $m$  is a power of 2, say,  $2^i$  then the array is full and we do  $O(m)$  work on the next call to `push_back()`.
  - If  $m \neq 2^i$ , then we do  $O(1)$  work on next call.

We can then add up the total effort as

$$\begin{aligned} T(n) &= \sum_{i=1}^k \left( O(2^i) + \sum_{j=1}^{2^i-1} O(1) \right) \\ &= O\left(\sum_{i=1}^k (2^i)\right) \\ &= O(1 + 2 + 4 + \dots + 2^k) \\ &= O(2^{k+1} - 1) \\ &= O(2^{k+1}) \end{aligned}$$

The total effort is  $O(2^{k+1})$ .

But we started with the definition of  $n$  saying that  $n = 2^k$ , so this total effort is  $O(2n) = O(n)$ .

### 2.1.3 push\_back has an Amortized Worst Case of O(1)

So even though

- an individual call to `push_back()` may be  $O(n)$ ,
- the *total* effort for all  $n$  `push_back`'s used to build a vector "from scratch" is also  $O(n)$

We say that the *amortized* worst-case time of `push_back()` is therefore  $O(1)$ .

#### Definition

*amortize*: to decrease (on average) over an extended period of time.

This term comes from the world of finance, where the cost of an initial high investment in equipment or facilities is often assessed (e.g., for tax purposes) at its equivalent annual cost over all the years that the equipment is in operation. For example, a \$10,000 computer expected to have a working lifetime of 5 years may be said to have an amortized cost of \$2,000 per year.

Similarly, if we consider the total work necessary to actually get a vector of  $n$  elements, we say that the total cost is  $O(n)$  and therefore we have an *amortized worst-case complexity* of  $O(1)$  per `push_back` call.

Whether or not the amortized cost is really what we want depends upon what kind of performance is important to us. If we are mainly interested in how some algorithm involving many `push_backs` performs in totality, the amortized cost is appropriate. If, however, we are dealing with an interactive algorithm that does one `push_back` in between each prompt for user input, then the "real"  $O(n)$  worst case is more appropriate because it indicates the amount of time that the user might have to wait after submitting an input.

## 2.2 Using `reserve()` to get a True O(1) Worst Case

If we knew ahead of time how many elements would be placed into the vector, we can make *all* the push\_back's O(1) time. We would do this by calling `reserve` to make sure there are enough slots without requesting more memory:

```
int n;
vector<std::string> names;
cout << "How many names? " << flush;
cin >> n;
names.reserve (n);
for (int i = 0; i < n; ++i)
{
    cout << "\nEnter name #" << i << ": " << flush;
    std::string aName;
    cin >> aName;
    names.push_back(aName);
}
```

This is the same `reserve` function that we looked at as part of the implementation of `push_back`.

## 2.3 Summary

So the true answer is that `vector::push_back` does have a worst case of  $O(n)$ , but in special circumstances that cost may average (amortize) to  $O(1)$  over a sequence of  $n$  calls.

In fact, if you were to look at the required behavior for `vector::push_back` listed in the [C++ language standard](#), you would find that the required  $O(1)$  behavior, is, indeed, a requirement for amortized time, not a requirement on the worst-case time.

# Linked Lists

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

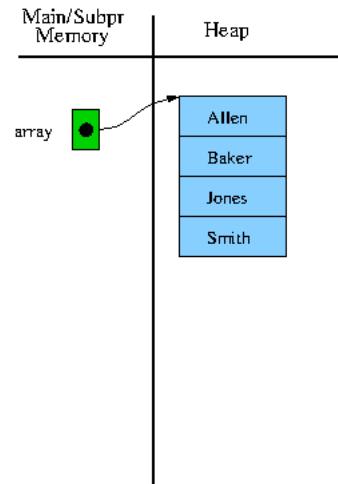
- [1 Linked Lists: the Basics](#)
  - [1.1 Traversing Linked Lists](#)
  - [1.2 Inserting into Linked Lists](#)
  - [1.3 Removing from Linked Lists](#)
- [2 Coding for Linked Lists](#)
  - [2.1 Example: Books with linked lists of Authors](#)
  - [2.2 Traversing a Linked List](#)
  - [2.3 Searching a Linked List](#)
  - [2.4 Walking Two Lists at Once](#)
  - [2.5 Adding to a Linked List](#)
  - [2.6 Removing from a Linked List](#)
- [3 Variations: Headers with First and Last](#)
  - [3.1 Adding a Last Pointer](#)
- [4 Variations: Doubly-Linked Lists](#)
  - [4.1 addBefore: Singly and Doubly Linked](#)
- [5 Variations: Sentinel Nodes](#)
- [6 Review](#)

## 1 Linked Lists: the Basics

### The data abstraction: a sequence of elements

Arrays (and vectors) work by storing elements of a sequence contiguously in memory

- Easy to access elements by number
- Inserting things into the middle is slow and awkward

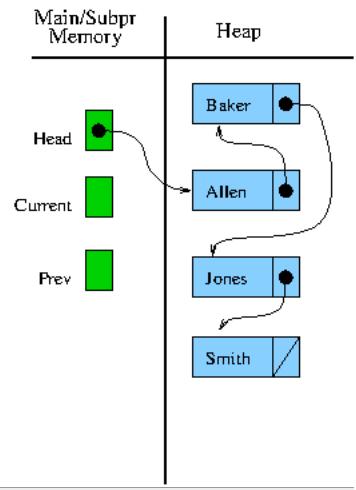


---

### An Alternate View

We can approach the same abstraction differently:

- Linked lists store each element in a distinct [node](#).
- Nodes are [linked](#) by pointers.
  - Accessing elements by number is slow and awkward
  - Easy to insert things into the middle



## List Nodes

```
template <typename T>
class node {
public:
    T nodeValue;
    node<T>* next;
    node() : next(nullptr)
    {}
    node (const T& item, node<T>* nextNode = nullptr)
        : nodeValue(item), next(nextNode)
    {}
};
```

- A linked list consists of a number of nodes.
- Each node provides a data field and a next pointer.

I've also added a couple of constructors for convenient initialization of those fields.

The `nullptr` keyword used in the code here is new to C++, part of the updated C++11 standard approved in August 2011. It replaces the use of '0' and `NULL` to denote a null pointer, both of which sometimes caused odd compilation errors because they are actually integers, not pointers.

The compilers are still catching up with all of the C++11 changes, so the new features aren't available by default, just yet. But they are used extensively in your textbook, and I'm going to use them when I feel they simplify the code or make it easier to understand.

You can activate these when compiling with `g++` by

- adding the flag `-std=c++11` to your compilation commands for `g++ 4.7` or later
- adding the flag `-std=c++0x` to older `g++` versions (starting with 4.3).

If you are using an IDE (e.g., Code::Blocks, Eclipse, etc.), you will probably have to add these to your project settings manually.

## 1.1 Traversing Linked Lists

We can move from node to node by tracing the pointers:

```
node<string> *current = head; // assuming that head points
                                // to the first node
while (current != nullptr)
{
    doSomethingWith (current->data);
    current = current->next; // move forward one step
}
```

or

```
for (node<string> *current = head; current != nullptr; current = current->next)
{
    doSomethingWith (current->data);
}
```

### 1.1.1 Demo

## SLLtraversal



You can run this demo yourself [here](#).

## 1.2 Inserting into Linked Lists

We insert by working from the node prior to the insertion point:

```
//  
// Insert value after node p  
//  
node<string> *newNode = new node<string>;  
newNode->info = value;  
newNode->link = p->link;  
p->link = newNode;
```

We can also shorten the code above by taking advantage of the constructors we added to the node class:

```
//  
// Insert value after node p  
//  
node<string> *newNode = new node<string> (value, p->link);  
p->link = newNode;
```

### 1.2.1 Demo

SLLinsert



You can run this demo yourself [here](#).

## 1.3 Removing from Linked Lists

We delete by moving the previous pointer “around” the unwanted node.

```
//  
//Remove value after node p  
//  
node<string> *q = p->link;  
p->link = q->link;  
delete q;
```

### 1.3.1 Demo

SLLerase



You can run this demo yourself [here](#).

## 2 Coding for Linked Lists

We need two data types to build a linked list:

- The linked list node, which we have already looked at
- and a header for the entire list:

```
template <typename Data>
struct LListHeader {
    node<Data>* first;
    LListHeader();
    ...
};
```

The header for the entire linked list. In practice, this is sometimes not done as a separate type, but by simply declaring the appropriate *first* (a.k.a., *front* or *head*) data member as a private member of an ADT.

I'm treating both of these as utilities, not full-fledged ADTs

- So I'm not going to be as picky about encapsulation and providing operators as I would usually be
- We would use these to define private date members within “proper” ADTs

## 2.1 Example: Books with linked lists of Authors

```
class Book {
public:
    Book (Author);           // for books with single authors
    Book (const Author[], int nAuthors); // for books with multiple authors

    Book (const Book& b);
    const Book& operator= (const Book& b);
    ~Book();

    std::string getTitle() const { return title; }
    void setTitle(std::string theTitle) { title = theTitle; }
    ...

private:
    std::string title;
    int numAuthors;
    std::string isbn;
    Publisher publisher;

    node<Author>* first;
    node<Author>* last;

    friend std::ostream& operator<< (const std::ostream& out, const Book& book);
    // The "friend" declaration gives this function access to private members of this class
};

std::ostream& operator<< (const std::ostream& out, const Book& book);
```

## 2.2 Traversing a Linked List

```
bookOut.cpp +
```

```
std::ostream& operator<< (const std::ostream& out, const Book& book)
{
    out << book.title << "\nby: ";
    const node<Author>* current = book.first;
    while (current != nullptr)
    {
        if (current != book.first)
            out << ", ";
        out << current->nodeValue;
        current = current->next;
    }
    out << "\n" << book.publisher << ", " << book.isbn << endl;
    return out;
}
```

The highlighted portion is a “typical” linked list traversal.

I often find that when writing this style of while loop, I am so focused on actions required to process the data in the current node that I forget to add the update of *current* in the final line of the loop body. So, actually, I prefer to write my traversals more like this:

```
bookOut2.cpp +
```

```

std::ostream& operator<< (const std::ostream& out, const Book& book)
{
    out << book.title << "\nby: ";
    for (const node<Author>* current = book.first; current != nullptr; current = current->next)
    {
        if (current != book.first)
            out << ", ";
        out << current->nodeValue;
    }
    out << "\n" << book.publisher << ", " << book.isbn << endl;
    return out;
}

```

which helps alleviate the problem of forgetting to update (or to initialize) the current pointer. It also has the advantage of making `current` a variable that is local to the loop, so that in a more elaborate algorithm, if I have other loops that traverse a linked list, I can re-use the name “`current`” without worrying about interfering with its use in other parts of the algorithm. Try [running this algorithm](#) to be sure that you understand it.

It’s a general rule of good C++ style that variables should be declared in such a way as to make them as local as possible, and that they should always be initialized in their declaration. So the second loop in this section is preferred stylistically over the first (because `current` is localized to a smaller region of the code). But both of those are preferred to the following, which I often see in student coding:

```

std::ostream& operator<< (const std::ostream& out, const Book& book)
{
    out << book.title << "\nby: ";
    const node<Author>* current;
    for (current = book.first; current != nullptr; current = current->next)
    {
        if (current != book.first)
            out << ", ";
        out << current->nodeValue;
    }
    out << "\n" << book.publisher << ", " << book.isbn << endl;
    return out;
}

```

which neither localizes the variable `current` nor initializes it immediately. You might think that I’m being picky about the latter – after all the variable gets initialized in the very next statement. The danger, however, is that your code is not already completed when you are writing these declarations. You might come along later and try to change the code (e.g., switching to the while-loop version) or inserting debugging output and then deleting it again. As changes get made, it’s all too easy to lose or move the initialization code, leaving you with a mess of code that is depending on an uninitialized variable. Bugs caused by uninitialized variables can be some of the hardest to debug.

On the other hand, if you declare and initialize your variables in one step, then if you do something that deletes that statement or that moves it to a later point in the algorithm, you will get an error from the compiler because the various references to it will not compile. You will therefore know immediately that something is wrong and will be given a pretty explicit message explaining the problem.

## 2.3 Searching a Linked List

To support:

```

class Book {
    ...
    void removeAuthor (const Author& au);
}

```

we would need to first locate the indicated author within our list. For that, we do a traversal but stop early if we find what we are looking for.

```

void Book::removeAuthor (const Author& au)
{
    node<Author>* current = book.first;
    while (current != nullptr && current->nodeValue != au)
    {
        current = current->next;
    }
    if (current != nullptr)
    {
        // We found the author we were looking for.
        ...
    }
}

```

In this case, I use a while loop instead of the for loop specifically because I *need* access to the value of `current` upon leaving the loop - it can’t be local to the loop body.

We’ll fill in the missing non-search portion of this function later.

## 2.4 Walking Two Lists at Once

Although not really a search, the relational operator code is similar, in that we do a traversal with a possible early exit. But now we have two walk two lists:

```

// Comparison operators
bool Book::operator== (const Book& b) const
{
    if (title != b.title || isbn != b.isbn || publisher != b.publisher)
        return false;
    if (numAuthors == b.numAuthors)
    {
        const node<Author>* current = first;
        const node<Author>* bcurrent = b.first;
        while (current != nullptr)
        {
            if (!(current->nodeValue == bcurrent->nodeValue))
                return false;
            current = current->next;
            bcurrent = bcurrent->next;
        }
        return true;
    }
    else
        return false;
}

```

## 2.5 Adding to a Linked List

```
void Book::add (const Author& au)
```

could be implemented differently depending on whether we want to keep the authors *in* the order *in* which they were added, *or* in alphabetical order by author name.

```
void Book::add (const Author& au)
{
    addToEnd (au);
    ++numAuthors;
}
```

or

```
void Book::add (const Author& au)
{
    addInOrder (au);
    ++numAuthors;
}
```

It might actually be a bit of overkill to treat `addToEnd` and `addInOrder` as separate functions, but I'm going to do so in order that we can discuss variants on these later. Note that these functions would need to be declared as (private) function members of the *Book* class.

### 2.5.1 addToEnd

```
void Book::addToEnd (const Author& au)
{
    node<Author>* newNode = new node<Author>(au, nullptr);
    if (first == nullptr)
    {
        first = newNode;
    }
    else
    {
        // Move to last node
        node<Author>* current = first;
        while (current->next != nullptr)
            current = current->next;

        // Link after that node current->next = newNode; } }
```

Notice how much time and effort is spent just “getting to” the end. (We’ll see a fix for this later.)

### 2.5.2 addInOrder

```
void Book::addInOrder (const Author& au)
{
    if (first == nullptr)
        first = new node<Author>(au, nullptr);
    else
    {
        node<Author>* current = first;
        node<Author>* prev = nullptr;
        while (current != nullptr && current->data < au)
        {
            prev = current;
            current = current->next;
        }
    }
}
```

```

    // Add between prev and current
    if (prev == nullptr)
        first = new node<Author>(au, first);
    else
    {
        addAfter (prev, au);
    }
}

```

By the way, it's only fair to point out that getting all the pointer switching correct that we need to do for these algorithms is not simple. You have to take particular care with the "special cases" of inserting at the start and end of the list, and of inserting into an empty list. Even experienced programmers can struggle with linked list manipulation, particularly with some of the still more complicated variations that we will introduce later.

The key, I find, to getting the code correct is to do a variation on desk checking – draw pictures of the data. In particular, before and after pictures of the list before you make any changes and of how you want it to look when you are done can show you how many different values in your data need to be altered. Then you can draw intermediate pictures to figure out a sensible sequence of steps that will get you from the beginning to the end. Once you have those, it's much easier to write the code that takes you from one picture to the next.

## 2.6 Removing from a Linked List

Earlier, we started looking at the idea of removing a node, and saw that we needed to start by searching for the node to be removed:

```

void Book::removeAuthor (const Author& au)
{
    node<Author>* prev = nullptr;
    node<Author>* current = book.first;
    while (current != nullptr && current->data != au )
    {
        prev = current;
        current = current->next;
    }
    if (current != nullptr)
    {
        // We found the author we were looking for.
        if (prev == nullptr)
        {
            // We are removing the first node in the list
            first = current->next;
            delete current;
        }
        else
            removeAfter (prev);
    }
}

```

```

void Book::removeAfter (node<Author>* afterThis)
{
    node<Author>* toRemove = afterThis->next;
    afterThis->next = toRemove->next;
    delete toRemove;
}

```

## 3 Variations: Headers with First and Last

With the header we have been using so far, ...

```

template <typename Data>
struct LListHeader {

    node<Data>* first;

    LListHeader();
    ;
};

```

Adding to (either) end of a list is very common, but compare the amount of work required to add to the front versus adding to the end.

Adding to the front of a list is clearly O(1). But when adding to the end, we have to traverse the entire list, making this O(N) where N is the length of the list.

```
template <typename Data>
void LListHeader<Data>::addToFront (const Data& value)
{
    node<Data>* newNode = new node<Data>(value, first);
    first = newNode;
}

template <typename Data>
void LListHeader<Data>::addToEnd (const Data& value)
{
    node<Data>* newNode = new node<Data>(value, nullptr);
    if (first == nullptr)
    {
        first = newNode;
    }
    else
    {
        // Move to last node
        node<Data>* current = first;
        while (current->next != nullptr)
            current = current->next;

        // Link after that node
        current->next = newNode;
    }
}
```

### 3.1 Adding a Last Pointer

We can speed things up by adding a second pointer in the header:

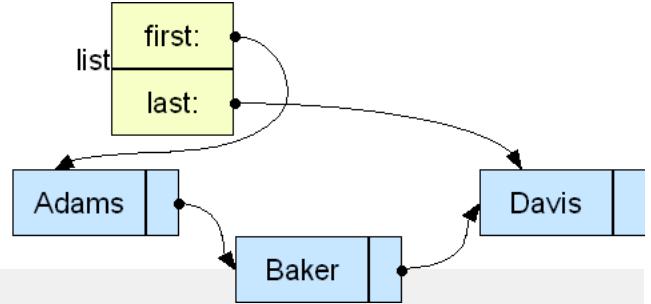
```
template <typename Data>
struct LListHeader {
    node<Data>* first;
    node<Data>* last;

    LListHeader();
    :
}
```

This enables (and requires) some change to our operations for adding to the ends.

```
template <typename Data>
void LListHeader<Data>::addToFront (const Data& value)
{
    node<Data>* newNode = new node<Data>(value, first);
    first = newNode;
    if (last == nullptr)
        last = first;
}

template <typename Data>
void LListHeader<Data>::addToEnd (const Data& value)
{
    node<Data>* newNode = new node<Data>(value, nullptr);
    if (last == nullptr)
    {
        first = last = newNode;
    }
    else
    {
        last->next = newNode;
        last = newNode;
    }
}
```



Because we now have direct access to the final element in the list, `addToEnd` now becomes O(1). The drawback is that all of the add and remove functions become just a little more complicated, because we must now watch for any special cases that affect the last node in the list, and must add code to update `last` when we do so.

## 4 Variations: Doubly-Linked Lists

By modifying the node structure:

```
template <typename Data>
struct dnode
```

```

{
    Data data;
    dnode<Data>* prev;
    dnode<Data>* next;

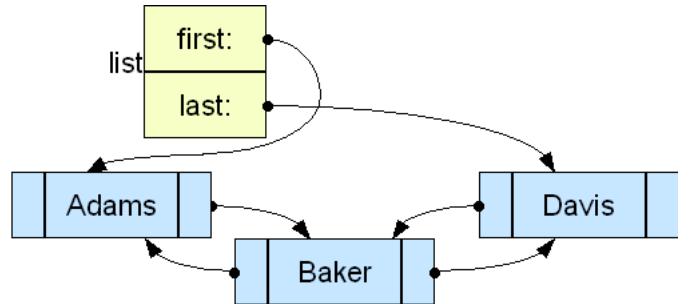
    dnode() {next = prev = 0;}
    dnode (const Data& d, dnode<Data>* prv = 0, dnode<Data>* nxt = 0)
        : data(d), next(nxt), prev(prv)
    {}
};

}

```

we can

- Move backwards as well as forward in the list
  - by following the `prev` pointers
- Easily add in front of a node



## 4.1 addBefore: Singly and Doubly Linked

Let's compare the problem of adding a value in front of a known position using both single and doubly-linked lists.

Singly linked:

```

template <typename Data>
void LListHeader<Data>::addBefore (node<Data>* beforeThis,
                                    const Data& value)
{
    if (beforeThis == first)
        addToFront (value);
    else
    {
        // Move to front of beforeThis
        node<Data>* current = first;
        while (current->next != beforeThis)
            current = current->next;

        // Link after that node
        addAfter (current, value);
    }
}

```

Doubly linked:

```

template <typename Data>
void DListHeader<Data>::addBefore (DListNode<Data>* beforeThis,
                                    const Data& value)
{
    if (beforeThis == first)
        addToFront (value);
    else
    {
        // Move to front of beforeThis
        DListNode<Data>* current = beforeThis->prev;
        // Link after that node
        addAfter (current, value);
    }
}

```

The singly linked variant is  $O(N)$ , where  $N$  is the length of the list, because in order to add in front of a given node, we have to traverse the list from the beginning up to the node just before the one where we want to insert.

But the doubly linked version needs no such traversal, because we can use the `prev` link to move backwards in  $O(1)$  time.

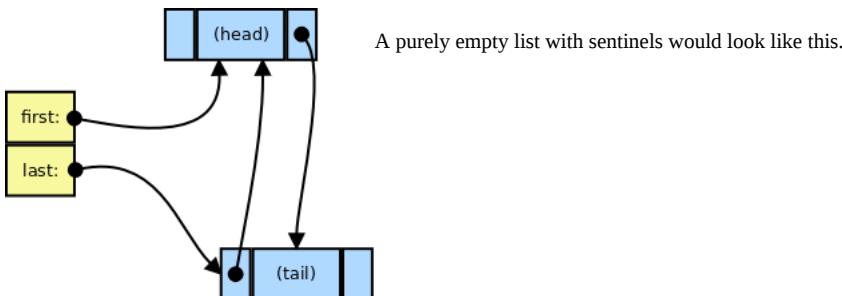
Of course, now we have almost twice as many pointers to update whenever we add or remove a node, so the code becomes a bit messier and difficult to get correct.

## 5 Variations: Sentinel Nodes

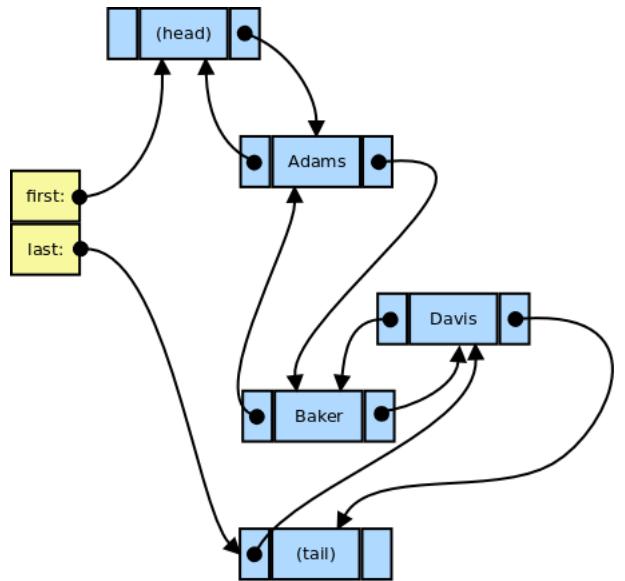
It may be clear by now that much of the difficulty in coding linked lists lies in the special care that must be taken with special cases:

- adding or removing the first node
- adding or removing the last node
- removing the only node (which is simultaneously first and last)
- doing *anything* with an empty list

Your textbook suggests a way to simplify this coding, by introducing *sentinel* nodes. A *sentinel* in a data structure is a reserved position at one end or the other that doesn't hold "real" data, but is used to help keep algorithms from running off the end of the structure.



A list with data keeps the data between the sentinels. The `(head)` and `(tail)` nodes do not hold data, and we will have to craft the code so that traversals never actually visit those nodes. But when adding or removing “real” data, this structure pretty much eliminates all of those special cases I listed above. A new node or a node to be removed is never going to be first/last/only because the sentinels sit in the first and last position.



## 6 Review

You can explore the use of singly linked lists for a variety of insertion and removal patterns by running [these algorithm](#).

Weiss' implementation of `std::list` using doubly linked lists with sentinels can be run [here](#).

# Linked List Applications

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

[1 Using Linked Lists in ADTs](#)

[1.1 The Header](#)

[1.2 Adding Authors](#)

[1.3 Initializing](#)

[1.4 Clean-Up](#)

[1.5 Copying](#)

[2 Free Lists](#)

[2.1 Freelist Overview](#)

[2.2 Adding a Freelist to the Node Type](#)

[2.3 Use dispose\(\) and alloc\(\) instead of delete and new](#)

[3 Operators new and delete](#)

[3.1 Replace dispose\(\) and alloc\(\) by operators new and delete](#)

[3.2 Implementing operator new](#)

[3.3 Implementing operator delete](#)

[3.4 Application code uses new and delete](#)

[4 Low-level Storage Management](#)

[4.1 Fragmentation](#)

Linked lists and related structures form the basis of many data structures, so it's worth looking at some applications that aren't implemented via the `std::list` that we will be examining shortly.

We'll look at issues that arise in ADT implementation, then in storage management, first in our own code, then at the operating system level.

## 1 Using Linked Lists in ADTs

We have previously explored implementations of a Book ADT that used various forms of array to store the list of authors. Now let's consider the possibility of using a linked list.

### 1.1 The Header

```
book0.h +  
  
class Book {  
public:  
    typedef const AuthorNode* AuthorPosition;  
  
    Book (Author); // for books with single authors  
    Book (const Author[], int nAuthors); // for books with multiple authors  
  
    Book (const Book& b);  
    ~Book();  
    const Book& operator= (const Book& b);  
  
    std::string getTitle() const { return title; }  
    void setTitle(std::string theTitle) { title = theTitle; }  
    int getNumberOfAuthors() const { return numAuthors; }  
  
    std::string getISBN() const { return isbn; }  
    void setISBN(std::string id) { isbn = id; }  
  
    Publisher getPublisher() const { return publisher; }  
    void setPublisher(const Publisher& publ) { publisher = publ; }  
  
    AuthorPosition find(const Author& au) const;  
    AuthorPosition next (AuthorPosition pos) const;  
    AuthorPosition getFirst () const;  
  
    void addAuthor (AuthorPosition at, const Author& author);  
    void removeAuthor (AuthorPosition at);  
  
private:  
    struct AuthorNode {  
        Author data;  
        AuthorNode* next;  
  
        AuthorNode (const Author& au, AuthorNode* nxt = 0)  
        : data(au), next(nxt)  
        {}  
    };
```

```

    };

    std::string title;
    int numAuthors;
    std::string isbn;
    Publisher publisher;

    AuthorNode* first;
    AuthorNode* last;

    void clear();
};

```

To use a linked list, we need to start by **declaring the data type for the linked list nodes**. In this case, there's no reason to ever show this type to code outside of this class, so I have chosen to nest the `AuthorNode` type within the `Book` class and to make it private.

We will also need pointers to (at least) the start of the linked list. I have opted for a **first/last header** instead, so that adding to the end of the list of authors can be implemented efficiently.

We can't expect to add and retrieve authors by numeric position if we are going to use a linked list, so I have added various **position-based functions** that manipulating and retrieving authors. This is far from an ideal solution to this problem. Ideally, we should be able to come up with a way of manipulating authors that does not change the public interface of `Book` just because we change the internal data structure. In a later lesson, we'll see a more elegant approach to this design when we introduce [iterators](#).

Finally, we note that we definitely have a data structure containing pointers to data that we don't really want to share. For this reason, we know that we will need to **declare the Big 3** and will need to provide an appropriate deep-copy implementation.

## 1.2 Adding Authors

To add an author at a given position, we consider 4 distinct cases:

```

void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
    if (first == 0)
        // List is empty - adding to both first and last position
        first = last = new AuthorNode (author, 0);
    else if (at == 0)
    {
        // Adding to the end of the list
        last->next = new AuthorNode (author, 0);
        last = last->next;
    }
    else
    {
        // Adding somewhere not at the end
        AuthorNode* newNode = new AuthorNode(author, (AuthorNode*)at);
        if (at == first)
            first = newNode;
        else
        {
            AuthorNode* prev = first;
            while (prev->next != at)
                prev = prev->next;
            prev->next = newNode;
        }
    }
    ++numAuthors;
}

```

- **Adding to an empty list:** We create one node and mark it as both the front and end of our list.
- **Adding to the end of a non-empty list:** We create a node and attach it to the end of the list, then make that our new list end.
- **Adding to the front of a non-empty list:** We create a node and attach it to the front of the list, then make that our new list start.
- **Adding somewhere in the middle:** We want to add in front of the position denoted by `at`. We therefore need to start with a loop to search for the node preceding the one pointed to by `at`. Once we have that, we go ahead with typical linked list pointer swapping.

I do find that getting code like this correct can be tricky, because of the number of pointers that need to be juggled and the number of special cases that often need to be treated separately. The trick, for me, is to draw before-and-after pictures of each case. If I draw a picture of how I expect the linked list to look before an operation and again after it, I can examine the picture to see just which pointers need to be altered to go from one picture to the other. Then I can start to worry about where, in the old picture, I would look to get the values that need to be put into those pointers that are going to change.

## 1.3 Initializing

The code for the constructors is not complicated.

```

// for books with single authors
Book::Book (Author a)
{
    numAuthors = 1;
    first = last = new Book::AuthorNode (a, nullptr);
}

```

```

}

// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
    numAuthors = 0;
    first = last = nullptr;
    for (int i = 0; i < nAuthors; ++i)
    {
        addAuthor(nullptr, au[i]);
    }
}

```

Note that, in the second constructor, where we are adding multiple authors from an array, I have opted to use the `addAuthor` function that we just implemented rather than writing the pointer manipulation code directly. A lot of programmers would have written the pointer and node manipulation directly into that loop. But we're going to have to get the code for adding an author to a list written correctly in `addAuthor` anyway, so why write it (and debug it) twice?

## 1.4 Clean-Up

It's expected that an ADT that hides a linked list would clean up all the nodes in that list when required. So here our destructor for a `Book` clears the list of authors.

```

void Book::clear()
{
    AuthorNode* nxt = nullptr;
    for (AuthorNode* current = first; current != nullptr; current = nxt)
    {
        nxt = current->next;
        delete current;
    }
    numAuthors = 0;
    first = last = nullptr;
}

Book::~Book()
{
    clear();
}

```

The `clear()` function was declared as a private function in the class declaration. Effectively, this is just a "helper" function for us to use here and, as will be seen, one other place. Basically this function runs through the whole linked list of nodes, deleting each one to return its memory to the run-time system for later re-use. It then sets the remaining author-related data members to values indicating that we now have no authors in the list.

The thing to look closely at in this function is the local variable `nxt`. You can't clear out a linked list like this:

```

for (AuthorNode* current = first; current != nullptr;
     current = current->next)
{
    delete current;
}

```

because we would have already deleted `current` *before* trying to fetch the value `current->next`. By the time we try to get the value of the next field, it may have already been garbled. It's a nasty little intermittent error – you might get away with it on 99 tests, then have it fail on the 100th test just when you are demonstrating your code to someone else.

## 1.5 Copying

We have [previously](#) explained why it is important to do a deep copy for this implementation of the `Book` ADT. This copying will be carried out by the copy constructor and the assignment operator, shown here.

```

Book::Book (const Book& b)
: title(b.title), numAuthors(0), isbn(b.isbn),
  publisher(b.publisher),
  first(nullptr), last(nullptr)
{
    for (AuthorNode* p = b.first; p != nullptr; p=p->next)
        addAuthor(nullptr, p->data);
}

const Book& Book::operator= (const Book& b)
{
    if (this != &b)
    {
        title = b.title;
        isbn = b.isbn;
        publisher = b.publisher;
        clear();
        for (AuthorNode* p = b.first; p != nullptr; p = p->next)
            addAuthor(nullptr, p->data);
    }
    return *this;
}

```

The core of each of these functions is a loop that walks the linked list of the book being copied, adding the authors found there to the end of the book we are copying into. Note the re-use, again, of the `addAuthor` function for this purpose.

The major differences between these two functions are

- The copy constructor copies several data members in an initialization list. Because initialization lists are not available outside of constructors, the assignment operator uses ordinary assignments to copy those data members.

This is pretty much a stylistic difference. We could have used assignment of those data members in both functions. The initialization list is slightly more efficient.

- In the assignment operator, when we copy authors into a book, we want those authors to replace the old list of authors, not to be tacked onto the end of the old list of authors. Hence you can see a call to our helper function `clear()` used to empty out the list before we start copying into it.

## 2 Free Lists

How should we deal with nodes that have been removed from a list?

- Obvious choice: delete them

This is what we have already done in our implementation of `Book`.

```
void Book::clear()
{
    AuthorNode* nxt = nullptr;
    for (AuthorNode* current = first; current != nullptr; current = nxt)
    {
        nxt = current->next;
        delete current;
    }
    numAuthors = 0;
    first = last = nullptr;
}
```

But we can sometimes get better performance by collecting all deleted nodes into a *free list* — a linked list of nodes that no longer belong to any list.

### 2.1 Freelist Overview

```
void Book::removeAuthor (Book::AuthorPosition at)
{
    if (at == first)
    {
        if (first == last)
            first = last = nullptr;
        else
        {
            first = first->next;
        }
    }
    else
    {
        AuthorNode* prev = first;
        while (prev->next != at)
            prev = prev->next;
        prev->next = at->next;
        if (at == last)
        {
            last = prev;
        }
    }
    delete at; ①
    --numAuthors;
}
```

Implementing a free list:

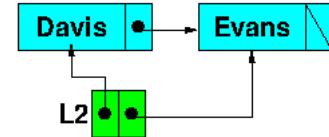
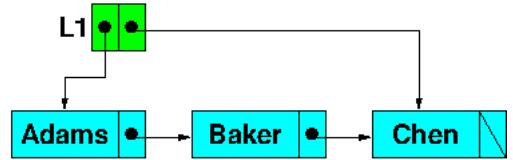
- Where we used to `delete` (①), we instead call a `dispose` routine that adds the node to a freelist.
- Where we used to allocate nodes via `new`, we instead call an `alloc` routine that
- returns a node from the freelist, if possible
- uses `new` if the freelist is empty

For example, suppose that we are executing the code shown here.

```
Book::AuthorPosition p = L1.find("Baker");
L1.removeAuthor(p);
L2.addAuthor ("Lewis");
L2.addAuthor ("Moore");
L1.addAuthor ("Zeil");
```

Assume that we starting with this data state:

(Book data members unrelated to the linked lists have been omitted to simplify the picture.)

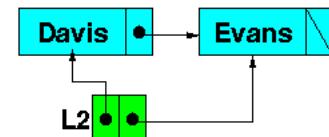
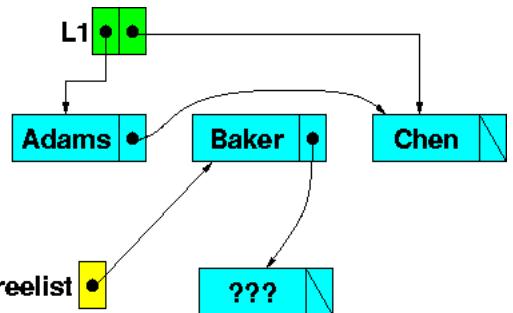


### Freelist: "deleting"

Now suppose we execute the first two lines of code. The "Baker" node will be removed (unlinked) from the list  $L_1$  and added to the free list.

```
Book::AuthorPosition p = L1.find("Baker");
L1.removeAuthor(p);
L2.addAuthor ("Lewis");
L2.addAuthor ("Moore");
L1.addAuthor ("Zeil");
```

List  $L_1$  now contains only "Adams" and "Chen". The "Baker" node has been placed on the free list. (Of course, given the way that linked lists work, the node itself hasn't actually moved anywhere. It's still at the same address it was before. All that has happened is that the links have been rearranged so that it now can be reached from a different list header.)

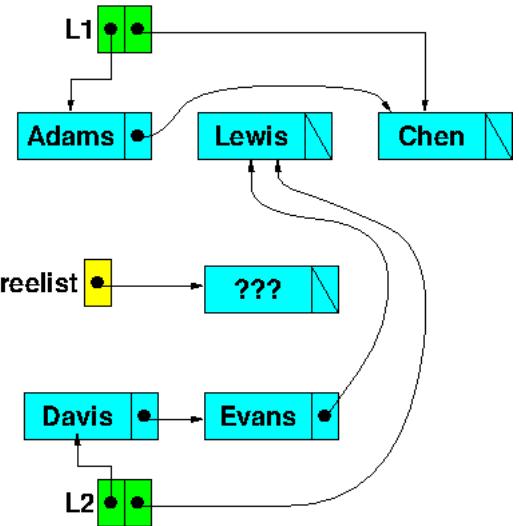


### Freelist: allocation

When we execute the third line, we add a new node ("Lewis") to the end of list  $L_2$ , the actual memory for the new node is taken from the front of the freelist.

```
Book::AuthorPosition p = L1.find("Baker");
L1.removeAuthor(p);
L2.addAuthor ("Lewis");
L2.addAuthor ("Moore");
L1.addAuthor ("Zeil");
```

The list L2 now contains 3 nodes. The last of these nodes used to be in L1.

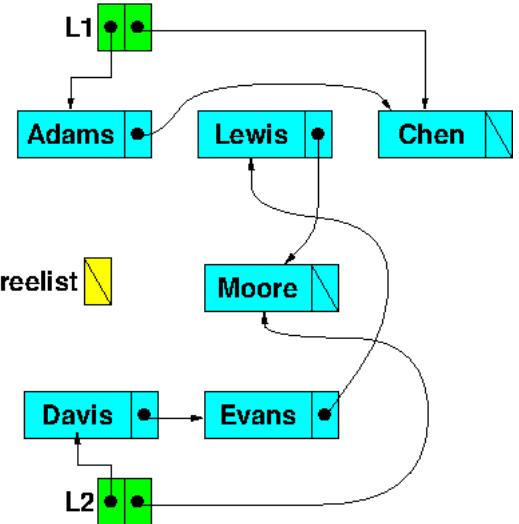


#### Freelist: allocation

If we again add a node ("Moore") to a list, the new node is again taken from the free list, and the new data written into it.

```
Book::AuthorPosition p = L1.find("Baker");
L1.removeAuthor(p);
L2.addAuthor ("Lewis");
L2.addAuthor ("Moore");
L1.addAuthor ("Zeil");
```

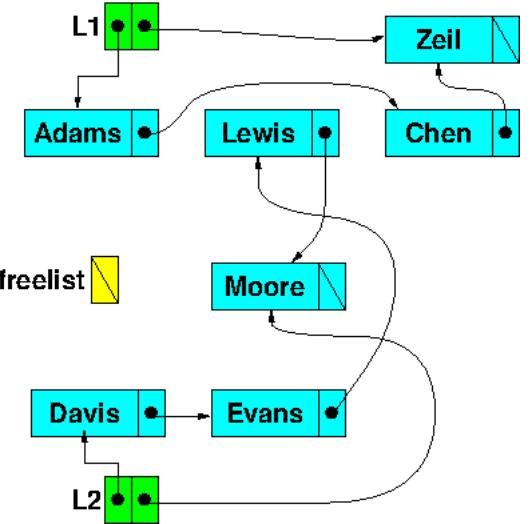
The free list is now empty ...



Adding the final node to L1 requires allocating a brand new node since we can't reuse any from the free list.

```
Book::AuthorPosition p = L1.find("Baker");
L1.removeAuthor(p);
L2.addAuthor ("Lewis");
L2.addAuthor ("Moore");
L1.addAuthor ("Zeil");
```

OK, that's the idea. How do we implement it?



## 2.2 Adding a Freelist to the Node Type

We go back to the Book::AuthorNode class and ask this class to provide a freelist.

```
class Book {
    struct AuthorNode {
        Author data;
        AuthorNode* next;

        AuthorNode (const Author& au, AuthorNode* nxt = nullptr)
            : data(au), next(nxt)
        {}

        static AuthorNode* freelist;           ①
        static void dispose (AuthorNode* p);  ②
        static AuthorNode* alloc (const Author& au, AuthorNode* nxt); ③
    };
};
```

We add 3 new declarations:

- ① : the data member *freelist*, a pointer to the first node in the free list
- ② : the *dispose* function, that adds a node to the free list, and
- ③ : the *alloc* function, that returns a pointer to a node that can be added to a list.

Notice that the *freelist* and *alloc* are declared as *static*:

- *static* applied to a data member means that this data value is not a member of each individual record, but is shared by all values of this data type.
- *static* applied to a function member means the function is not a member of each individual object.

Because static members belong to the entire class rather than to individual objects, they are not called like normal function members:

```
object.foo(x);
```

Static members are called like non-member functions:

```
foo(x);
```

or

```
Class::foo(x);
```

when calling *foo* from code outside of the class that contains it.

Why does “static” make sense here? The freelist must obviously be shared, so we only want one freelist for all objects of this class.

As for alloc, it is used to get a node. Since we don’t have a node yet, we could not apply alloc to it, so we can’t make alloc a “normal” member function that operates on existing nodes (i.e., node.alloc() makes no sense).

The implementation of these functions are pretty straightforward.

```
Book::AuthorNode* Book::AuthorNode::freelist = nullptr; ①

void Book::AuthorNode::dispose ()
{
    // Add *p to the freelist
    next = freelist; ②
    freelist = this;
}

Book::AuthorNode* Book::AuthorNode::alloc (const Author& au, AuthorNode* nxt)
{
    // Do we have a reusable node on the freelist?
    if (freelist != nullptr)
    {
        // Use the first node in the free list
        AuthorNode* result = freelist; ③
        freelist = freelist->next;
        result->data = au;
        result->next = nxt;
        return result;
    }
    else
    {
        // Allocate a new node
        return new AuthorNode(au, nxt); ④
    }
}
```

① : The freelist header is initialized to null (the list is initially empty).

② : When we dispose of an author node, it simply gets put onto the front of the freelist. This is an ordinary linked list manipulation.

③ : When allocating a new author node, we try to pull the first node off of the freelist and re-use it.

④ : If, however, the freelist has been emptied, then we fall back on a good old-fashioned new to create a new node.

## 2.3 Use dispose() and alloc() instead of delete and new

Now, let’s look at how we use the free list functions.

```
void Book::clear()
{
    AuthorNode* nxt = 0;
    for (AuthorNode* current = first; current != 0; current = nxt)
    {
        nxt = current->next;
        current->dispose ();
    }
    numAuthors = 0;
    first = last = 0;
}
```

We search through our entire class. First, we replace any deletes of AuthorNode pointers by calls to dispose.

Then we make a similar search and replace all allocations of nodes via new to calls to the alloc function.

[addWithAlloc.cpp](#) +

```
void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
    if (first == 0)
        // List is empty - adding to both first and last position
        first = last = AuthorNode::alloc (author, 0);
    else if (at == 0)
    {
        // Adding to the end of the list
        last->next = AuthorNode::alloc (author, 0);
        last = last->next;
    }
    else
    {
        // Adding somewhere not at the end
        AuthorNode* newNode = AuthorNode::alloc(author, (AuthorNode*)at);
        if (at == first)
            first = newNode;
        else
        {
```

```

        AuthorNode* prev = first;
        while (prev->next != at)
            prev = prev->next;
        prev->next = newNode;
    }
}
++numAuthors;
}

```

And that's all there is to using these routines!

## 3 Operators new and delete

This is still not the ideal solution.

- When we “dispose” of a node by putting it on the `freelist`, its `value` field is not destroyed (i.e., its destructor is never called, so any cleanup it needs to do may never happen).
- When we allocate a node by taking it from the `freelist`, its `value` field is not initialized (constructed).

We faked that in the example above by passing the two fields as parameters and assigning to them, but there are situations where that would not work or would be rather unwieldy (e.g., if we had a lot more data fields and a lot of different constructors with different combinations of parameters).

This things happened automatically when we used `delete` & `new`. We'd like to continue to have that happen.

Solution: C++ lets us define our own `new/delete` operations for a class.

- After any call to `new`, object is constructed-initialized
- Before any call to `delete`, object is destroyed.

### 3.1 Replace `dispose()` and `alloc()` by operators `new` and `delete`

So we will replace our `dispose` and `alloc` functions by `delete` and `new` operators that manipulate the free list in much the same way. (Yes, `new` and `delete` are considered operators in C++.)

```

class Book {

    struct AuthorNode {
        Author data;
        AuthorNode* next;

        AuthorNode (const Author& au, AuthorNode* nxt = 0)
        : data(au), next(nxt)
        {}

        static AuthorNode* freelist;
        void operator delete(void*);
        void* operator new(size_t);
    };
    ...
};

```

The `delete` function receives a pointer to the object to be deleted. The `new` function receives the number of bytes expected of the object and must return a pointer to the newly allocated object.

### 3.2 Implementing operator new

Implementation is much the same as before.

```

void* Book::AuthorNode::operator new (size_t sz)
{
    // Do we have a reusable node on the freelist?
    if (freelist != nullptr)
    {
        // Use the first node in the free list
        AuthorNode* result = freelist;
        freelist = freelist->next;
        return result;
    }
    else
    {
        // Allocate a new node
        return malloc(sz);
    }
}

```

The `new` operator works by returning a node from the `freelist` if possible. If the `freelist` is empty, we allocate space for a new node from the heap. We do this by calling the system routine `malloc`, which requires the number of bytes to be allocated. (Isn't it fortunate that `new` gets that number as a parameter!)

### 3.3 Implementing operator delete

The operator `delete` works, as did `dispose` before it, by placing the node at the front of the free list.

```
void Book::AuthorNode::operator delete (void* p)
{
    // Add *p to the freelist
    AuthorNode* a = (AuthorNode*)p;
    a->next = freelist;
    freelist = a;
}
```

The only catch here is that the pointer being deleted is passed as a `void*`, so we need to typecast it to `AuthorNode*` if we want to use any data members of `AuthorNode` (such as the `next` field).

### 3.4 Application code uses new and delete

The code that actually uses the author nodes goes back to the way it was when we used the regular `new/delete`.

```
bookClear2.cpp +
```

```
void Book::clear()
{
    AuthorNode* nxt = 0;
    for (AuthorNode* current = first; current != 0; current = nxt)
    {
        nxt = current->next;
        delete current;
    }
    numAuthors = 0;
    first = last = 0;
}

void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
    if (first == 0)
        // List is empty - adding to both fitst and last position
        first = last = new AuthorNode (author, 0);
    else if (at == 0)
    {
        // Adding to the end of the list
        last->next = new AuthorNode (author, 0);
        last = last->next;
    }
    else
    {
        // Adding somewhere not at the end
        AuthorNode* newNode = new AuthorNode (author, (AuthorNode*)at);
        if (at == first)
            first = newNode;
        else
        {
            AuthorNode* prev = first;
            while (prev->next != at)
                prev = prev->next;
            prev->next = newNode;
        }
    }
    ++numAuthors;
}
```

The difference is that now this code will be calling our own implementations of those functions.

## 4 Low-level Storage Management

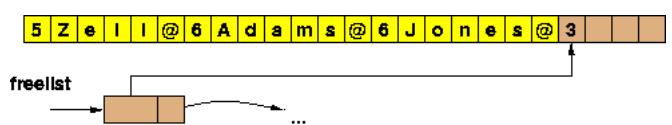
When you call the “normal” `new` in C++, it calls the routine `malloc` in the underlying operating system. When you call the “normal” `delete` in C++, it calls the routine `free` in the underlying operating system.

If you were to examine the operating system’s code for `malloc` and `free`, you would find that they maintain a free list of blocks of deleted memory.

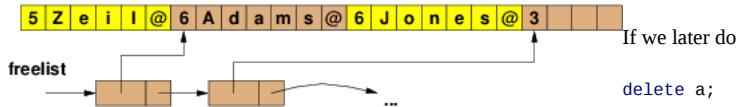
- The `malloc/free` free list problem is more complicated because blocks are of different sizes.

Suppose a program repeatedly allocates and deletes objects of varying sizes on the heap (e.g., strings):

```
string *z = new string("Zeil");
string *a = new string("Adams");
string *z = new string("Jones");
```



The operating system maintains a free list of unallocated blocks of memory on the heap.



the `delete` adds a block of memory to the freelist.

The blocks of memory don't actually move around. They are just managed using linked list nodes that point to the freed blocks of memory. [^This is a bit of an oversimplification. What usually happens is that the opening bytes of each freed block of memory is overwritten by the pointer to the next block in the free list, making it unnecessary to actually maintain a separate list.]

## 4.1 Fragmentation

A new allocation request, e.g.,

```
string* t = new string("ABC");
```

requires the OS to search the free list for a block of appropriate size

- Common schemes are
- first fit — choose the first block in the free list that is big enough
- best fit — choose the block in the free list that is closest to the requested size from among those that are no smaller than the requested size.
- Oversized blocks are split up

When blocks are split up, the remainder is placed back on the free list. Over time, a program that does many allocations and deletes may find more and more storage wasted on small fragments left on the free list, too small to be useful.

This is called fragmentation.

- Free list length could be  $O(k)$  where  $k$  is the total number of deletes
- Since each new requires a search of that list, new becomes  $O(k)$ 
  - instead of  $O(1)$ , as usually assumed
- `delete` is  $O(1)$ .

This explains why, on occasion, you will find a program that has been running for a long time seems to be getting slower and slower. The free list is being choked with small fragments, so new allocation requests are taking longer and longer. If such a program is run long enough, it may crash when an allocation request can no longer be satisfied, even though there is more than enough free memory in total.

Some malloc/free systems try to reduce or eliminate fragmentation:

- One way is to use compaction: As regions are freed, sort them by address so that adjacent regions of memory are also adjacent within the free list.
  - Adjacent free regions can then be merged to form a single, larger region.
  - Now `delete` is worst case  $O(k)$ .
  - but  $k$  *might* be smaller
- Another idea: only allocate regions in selected sizes: 1, 2, 4, 8, 16, 32, 64, ...
  - E.g., If a request is made for 10 elements, a 16-element regions is actually returned.
  - A separate free list is kept for each size of region.
  - Larger regions can be split in half to form two smaller regions, when necessary.
  - `new` and `delete` remain  $O(1)$ 
    - but storage utilization may suffer — on average wastes 25% of memory

Even with all this, you can see why sometimes we would prefer to handle our own storage management. Implementing our own freelist let's us do allocation and freeing of memory in  $O(1)$  time, because all of the data objects on our own freelist will be of uniform size. The more general problem of allocating and freeing memory of different data types of many varying sizes is much harder, and will either have a complexity proportional to the number of prior deletions, or will waste a substantial fraction of all memory.

# Standard Lists

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 The Standard list](#)
  - [1.1 Constructors](#)
  - [1.2 Copying](#)
  - [1.3 Size](#)
  - [1.4 Inserting Elements](#)
  - [1.5 Removing Elements](#)
  - [1.6 Access to Elements](#)
- [2 Implementing the List ADT](#)
  - [2.1 iterators](#)
- [3 Required Performance](#)

The [list](#) abstraction is closely related to the [linked list](#) data structure that we have already exemplified.

## 1 The Standard list

From the point of view of the public interface, `std::list` is very similar to `std::vector`. The primary differences are:

- In addition to `push_back` and `pop_back`, we also gain `push_front` and `pop_front` operations.  
All are  $O(1)$ .
- We cannot access arbitrary positions by number (e.g., `v[i]`, `v.at(23)`). Access to internal elements is exclusively via iterators.
  - Compensating for the difficulty of *getting* to an internal position, operations like `insert(position, data)` and `erase(position)` are now  $O(1)$  for lists. (They are  $O(\text{size}())$  for vectors.)
- A standard list's iterators are bi-directional. (A vector's are random access.)

The standard list is shown here.

```
namespace std {
template <class T>
class list {
public:
    typedef T&           reference;
    typedef const T&      const_reference;
    typedef ...            iterator;
    typedef ...            const_iterator;
    typedef ...            size_type;
    typedef ...            difference_type;
    typedef T              value_type;
    typedef T*             pointer;
    typedef const T*       const_pointer;
    typedef std::reverse_iterator<iterator>      reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

// construct/copy/destroy:
explicit list(const Allocator& = Allocator());
explicit list(size_type n, const T& value = T());
template <class InputIterator>
    list(InputIterator first, InputIterator last);
list(const list<T>& x);
~list();
list<T>& operator=(const list<T>& x);
template <class InputIterator>
    void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);

// iterators:
iterator           begin();
const_iterator     begin() const;
iterator           end();
const_iterator     end() const;
reverse_iterator   rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator   rend();
const_reverse_iterator rend() const;

// __lib.list.capacity_ capacity:
bool   empty() const;
size_type size() const;
size_type max_size() const;
void   resize(size_type sz, T c = T());
```

```

// element access:
reference front();
const_reference front() const;
reference back();
const_reference back() const;
// _lib.list.modifiers:
void push_front(const T& x);
void pop_front();
void push_back(const T& x);
void pop_back();
iterator insert(iterator position, const T& x);
void insert(iterator position, size_type n, const T& x);
template <class InputIterator>
void insert(iterator position, InputIterator first,
            InputIterator last);
iterator erase(iterator position);
iterator erase(iterator position, iterator last);
void swap(list<T>&);
void clear();

// _lib.list.ops_ list operations:
void splice(iterator position, list<T>& x);
void splice(iterator position, list<T>& x, iterator i);
void splice(iterator position, list<T>& x, iterator first,
            iterator last);
void remove(const T& value);
template <class Predicate> void remove_if(Predicate pred);
void unique();
template <class BinaryPredicate>
void unique(BinaryPredicate binary_pred);
void merge(list<T>& x);
template <class Compare> void merge(list<T>& x, Compare comp);
void sort();
template <class Compare> void sort(Compare comp);
void reverse();
};

template <class T>
bool operator==(const list<T>& x, const list<T>& y);
template <class T>
bool operator< (const list<T>& x, const list<T>& y);
template <class T>
bool operator!=(const list<T>& x, const list<T>& y);
template <class T>
bool operator> (const list<T>& x, const list<T>& y);
template <class T>
bool operator>=(const list<T>& x, const list<T>& y);
template <class T>
bool operator<=(const list<T>& x, const list<T>& y);

// specialized algorithms:
template <class T, class Allocator>
void swap(list<T>& x, list<T>& y);
}

```

It is, for the most part, identical to the `vector` interface, except that `list` does not allow you to access elements by indexing (e.g., `v[i]`). Instead, any access to the contents of a `std::list` will almode always be provided via iterators.

`list` provides a few specialized operations for shifting nodes from one list to another (`splice`). These aren't used all that often, but can sometimes significantly reduce the complexity of an algorithm by avoiding the need to copy long sequences of data.

`list` also provides some utility functions for sorting, merging and other list operations. In many cases, these same functions exist as separate function templates in `std`, but may not be usable with lists. For example, `std` has a `sort` function, but `std::sort` usually employs a variation of quicksort, and the quicksort algorithm requires random access iterators. `list` provides bi-directional iterators, but not random-access, so the `std::sort` function will not work with lists. Instead, therefore, `list` provides its own `sort` function as a member function of the `list<T>` class.

Let's look at this declaration, piece by piece.

## 1.1 Constructors

```
list<int> l1;
```

l1 is a list of integers. Initially, it has 0 elements

```
list<string> l2(10, "abc");
```

l2 is a list of strings initialized to hold 10 copies of "abc"

```

template <class T>
// construct / copy / destroy
list();
explicit list(size_type n, const T& value = T());

template <class InputIterator>
list(InputIterator first, InputIterator last);

list(const list<T>& x);

~list();

```

The **last constructor** is rather curious. It's actually a template function, and is designed to initialize the list with a sequence of values taken from some other container. The desired sequence is indicated via a pair of iterators, one for the beginning of the sequence, the other for the position just after the last desired element.

```
char[] greeting = "Hello";
list<char> l3(greeting+1, greeting+4);
```

l3 is a list of characters containing 'e', 'l' and another 'l'.

— In fact, a similar constructor was available for vectors as well, but I didn't mention it then because we had not yet introduced iterators. All the standard containers support this type of constructor template. For vectors and lists, this is an  $O(n)$  operation, where  $n$  is the number of elements in the indicated range.

## 1.2 Copying

```
list<double> v, w;
⋮
v = w;
list<double> z(v);
```

```
template <class InputIterator>
list(InputIterator first, InputIterator last);
list(const list<T>& x);

~list();

list<T>& operator=(const list<T>& x);

template <class InputIterator>
void assign(InputIterator first, InputIterator last);

void assign(size_type n, const T& u);
```

## 1.3 Size

```
list<int>::size_type n;
n = l1.size();
```

How many elements currently in l1?

```
l1.resize(n, x);
```

Add or remove elements, as necessary, at the end of the list to make l1 have exactly n elements. If elements need to be added, the value of the new elements will be x.

```
l1.resize(n);
```

Same as the last example, but if any elements need to be added to the list, the new elements are formed using their data type's default constructor.

```
// capacity
size_type size() const;
size_type max_size() const;
void resize(size_type sz, T c = T());
bool empty() const;
```

## 1.4 Inserting Elements

```
l2.push_back("def");
```

Adds a new element, "def" to the end of the list l2. l2.size() increases by 1.

---

```
l2.push_front("def");
```

Adds a new element, "def" to the front of the list l2. l2.size() increases by 1.

---

```
list<std::string>::iterator pos;  
:  
l2.insert(pos, "def");
```

Inserts a new element, "def", at the indicated position within the list l2. Any elements already in l2 at that position or higher are moved back one position to make room. l2.size() increases by 1.

```
// modifiers:  
void push_back(const T& x);  
void push_front(const T& x);  
void pop_back();  
void pop_front();  
iterator insert(iterator position, const T& x);  
void insert(iterator position, size_type n, const T& x);  
  
template <class InputIterator>  
void insert(iterator position,  
           InputIterator first, InputIterator last);  
  
iterator erase(iterator position);  
iterator erase(iterator first, iterator last);  
void swap(list<T>&);  
void clear();
```

## 1.5 Removing Elements

```
l2.pop_back();
```

Removes the element at the end of the list, decreasing the `size()` by 1.

```
l2.pop_front();
```

Removes the element at the front of the list, decreasing the `size()` by 1.

```
list<std::string>::iterator pos;  
⋮  
l2.erase(pos);
```

Removes the element at the indicated position within the list `l2`. Any elements already in `l2` at higher positions are moved forward one position to “take up the slack”. `l2.size()` decreases by 1.

```
list<std::string>::iterator start, stop;  
⋮  
l2.erase(start, stop);
```

Removes the elements at the indicated positions from `start` up to, but not including `stop`, within the list `l2`. Any elements already in `l2` at higher positions are moved forward to “take up the slack”. `l2.size()` decreases by the number of elements removed.

You can remove all elements from a list like this:

```
list<std::string>::iterator pos;  
⋮  
l2.clear();
```

```
// modifiers:  
void push_back(const T& x);  
void push_front(const T& x);  
void pop_back();  
void pop_front();  
iterator insert(iterator position, const T& x);  
void insert(iterator position,  
            InputIterator first, InputIterator last);  
  
iterator erase(iterator position);  
iterator erase(iterator first, iterator last);  
void swap(list<T>&);  
void clear();
```

## 1.6 Access to Elements

```
x = l1.front();  
l1.front() = l1.front()+1;
```

Provides access to the first element in the list.

```
x = l1.back();  
l1.back() = l1.back()+1;
```

Provides access to the last element in the list.

```
// element access:  
reference front();  
const_reference front() const;  
reference back();  
const_reference back() const;
```

### 1.6.1 Access to Interior Elements

None of the operations described so far give access to elements other than the front and back.

- Interior elements are accessed via [iterators](#), which follow the usual C++ conventions. For example, a typical loop through an entire list looks like:

```
list<string> names;
{
    for (list<string>::iterator pos = names.begin();
        pos != names.end(); ++pos)
    {
        cout << "One of the names is " << *pos << endl;
    }
}
```

or, in C++11,

```
list<string> names;
{
    for (auto pos = names.begin(); pos != names.end(); ++pos)
    {
        cout << "One of the names is " << *pos << endl;
    }
}
```

or, using the range-based for loop:

```
list<string> names;
{
    for (string aName: names)
    {
        cout << "One of the names is " << aName << endl;
    }
}
```

## 2 Implementing the List ADT

The `std::list` is usually implemented as pointers to the first and last nodes in a fairly conventional doubly-linked list of nodes like these:

```
template <class T>
struct listNode {
    T data;
    listNode<T>* prevLink;
    listNode<T>* nextLink;
};
```

Once that's been established, the individual functions are pretty much straightforward variations on the [linked list algorithms](#) that we have already looked at.

Your text adds the additional wrinkle of using sentinel nodes at each end of the list. You can read an explanation of the implementation there and you can run [that implementation](#) for insight into how it works.

### 2.1 iterators

The iterators are declared as classes that present the typical iterator interface, implemented using a pointer to the list node representing the current position and a pointer to the list object itself.

```
class const_iterator
{
public:
    // Public constructor for const_iterator.
    const_iterator( ) : current{ nullptr }
    { }

    // Return the object stored at the current position.
    // For const_iterator, this is an accessor with a
    // const reference return type.
    const Object & operator*( ) const
    { return retrieve( ); }

    const_iterator & operator++( )
    {
        current = current->next;
        return *this;
    }

    const_iterator operator++( int )
    {
        const_iterator old = *this;
        +( *this );
        return old;
    }

    const_iterator & operator--( )
    {
        current = current->prev;
        return *this;
    }
}
```

```

const_iterator operator-- ( int )
{
    const_iterator old = *this;
    --( *this );
    return old;
}

bool operator== ( const const_iterator & rhs ) const
{ return current == rhs.current; }

bool operator!= ( const const_iterator & rhs ) const
{ return !( *this == rhs ); }

protected:
Node *current;

// Protected helper in const_iterator that returns the object
// stored at the current position. Can be called by all
// three versions of operator* without any type conversions.
Object & retrieve( ) const
{ return current->data; }

// Protected constructor for const_iterator.
// Expects a pointer that represents the current position.
const_iterator( Node *p ) : current{ p }
{ }

friend class List<Object>;
};

```

You can see, for example, that incrementing one of these iterators involves simply following a “next” pointer by one hop.

## 3 Required Performance

The C++ standard specifies that a legal (i.e., standard-conforming) implementation of `list` must satisfy the following performance requirements:

Operation	Speed
<code>list()</code>	$O(1)$
<code>list(n, x)</code>	$O(n)$
<code>size()</code>	$O(size())$
<code>push_back(x)</code>	$O(1)$
<code>push_front(x)</code>	$O(1)$
<code>pop_back</code>	$O(1)$
<code>pop_front</code>	$O(1)$
<code>insert</code>	$O(1)$
<code>erase</code>	$O(1)$
<code>front, back</code>	$O(1)$
<code>splice</code>	$O(1)$

Perhaps the only surprise here is that `size()` is  $O(size())$ . The relatively seldom-used `splice` operation allows one to transfer arbitrarily long ranges of elements from one list to another. It is supposed to take place in  $O(1)$  time. But there’s no way for a list to know how many elements have been removed or added in a `splice` in that amount of time. Consequently, lists cannot keep track of their sizes as things are added and removed, the way that were able to do with `vector`. Hence every call to `list::size()` must actually walk the entire list, counting the nodes.

Beware, therefore, of writing loops like this:

```

void process (const list<T>& aList)
{
    int numProcessed = 0;
    auto pos = aList.begin();
    while (numProcessed < aList.size())
    {
        doSomethingWith(*pos);
        ++pos;
        ++numProcessed;
    }
}

```

(I often see code like this from clumsy conversions of code from arrays to lists.)

Because `size()` is  $O(size())$ , and because it is being called on each iteration of the loop, this function winds up being  $O(aList.size()^2)$ .

This is better, being  $O(aList.size())$ .

```

void process (const list<T>& aList)
{
    int numProcessed = 0;
    int numToBeProcessed = aList.size();
    auto pos = aList.begin();
    while (numProcessed < numToBeProcessed)

```

```
{  
    doSomethingWith(*pos);  
    ++pos;  
    ++numProcessed;  
}
```

Of course, this is both  $O(aList.size())$  and simpler:

```
void process (const list<T>& aList)  
{  
    auto pos = aList.begin();  
    while (pos != aList.end())  
    {  
        doSomethingWith(*pos);  
        ++pos;  
    }  
}
```

and this is the same complexity and still simpler:

```
void process (const list<T>& aList)  
{  
    for (const T& t: aList)  
    {  
        doSomethingWith(t);  
    }  
}
```

# Generic Programming

Steven J. Zeil

Last modified: May 12, 2016

## Contents:

- [1 Iterators + Templates = Generic Programming](#)
- [2 Copying](#)
  - [2.1 Using std::copy](#)
  - [2.2 copy and I/O iterators](#)
  - [2.3 Copying, Overwriting, and Inserting](#)
- [3 Other Useful Generic Functions in the std:: Library](#)
  - [3.1 equal](#)
  - [3.2 find and lower\\_bound](#)
  - [3.3 count](#)
  - [3.4 fill](#)
- [4 std:: Library Generics That Take Functions](#)
  - [4.1 Passing Functions as Parameters](#)
  - [4.2 for each](#)
  - [4.3 transform](#)
  - [4.4 " if" Generics and Predicates](#)
  - [4.5 all\\_of, any\\_of, none\\_of](#)
  - [4.6 all\\_of, any\\_of, none\\_of](#)
- [5 C++11 Lambda Expressions](#)
- [6 Functors](#)
  - [6.1 Example: Functors and User Interface Programming](#)
  - [6.2 operator\(\)](#)
  - [6.3 A Predicate Functor](#)
  - [6.4 Why Do Functors Work Where Functions Are Expected?](#)
  - [6.5 Functors Can Have Memory](#)
  - [6.6 std Functors for Comparisons](#)
  - [6.7 Substituting Your Own Comparison Functions](#)
- [7 References](#)

When we combine iterators with template functions, we get a powerful tool for writing programs. Because the iterator interface is the same no matter what kind of container the data is really in, many algorithms can be written as function templates to work with data in almost any kind of container.

This combination is called *generic programming*.

## 1 Iterators + Templates = Generic Programming

One benefit of designing our own classes to follow the “standard” form of iterators is that the C++ standard library is packed with small function templates for using iterators to do common tasks. These can be found in the header file `<algorithm>`.

For example, we can search any range of data for a particular element using `std::find`:

```
#include <algorithm>
:
pos = find(startingPosition, stoppingPosition, x);
```

This searches a sequence of data, beginning at `startingPosition`, up to but not including `stoppingPosition`, for the value `x`. If it finds `x`, it returns the position where it was found. If it doesn’t find it, it returns `stoppingPosition` (which, I was careful to note, is *not* one of the positions actually searched, so we can unambiguously determine whether we found `x` or not):

```
#include <algorithm>
:
pos = find(startingPosition, stoppingPosition, x);
if (pos != stoppingPosition)
{
    cout << "Found it!" << endl;
}
else
{
    cout << "It's not in there." << endl;
}
```

Now, `pos`, `startingPosition`, and `stoppingPosition` are all iterators of some kind. They must all be of the same iterator type, and that type has to be a position of whatever the type of `x` is”. But the `std::find` function will work with iterators taken from an array, a vector, a list, …, or whatever.

How does this happen? `std::find` is implemented as a template:

```
template <typename Iterator, typename T>
Iterator find (Iterator start, iterator stop, T x)
```

```
{
    while (start != stop && !(x == *start))
        ++start;
    return start;
}
```

This template makes no assumptions about the iterators passed to it, except that they support the operations `!=`, `*`, and `++`, which *all iterators, no matter what container they come from*, are supposed to support.

It also makes minimal assumptions about the type of `x`. It simply assumes that `x` will be from a data type that supports comparison via `==`.

So this template can be applied to iterators from arrays, vectors, lists, Books, PersonnelRecords, MyFavoriteDataTypeNumber241, or whatever. As long as we can give a starting position and a stopping position, the code in `find` is valid.

One of the hallmarks of the generic style of programming is that we always try to work on ranges of positions (iterators) with no explicit references to the container that those positions were drawn from.

We've already seen another such generic function template when we looked at [Searching via Iterator Variants](#):

```
template <typename Iterator, typename Value>
Iterator lower_bound (Iterator start, Iterator stop, const Value& key);
```

as well as a handful of [other useful function templates](#) that are, strictly speaking, not “generic” because they are not based on ranges of iterator positions.

In this lesson, we want to look at more such generic functions and get a little better feel for how they can influence C++ programming style. If you have taken CS333 Principles of Programming Languages or a similar course, you may also recognize that generic programming shares a lot of ideas with “functional programming” as well.

## 2 Copying

All of our `std::` containers support copying via copy constructors or assignment. But both of those cases involve copying between two containers of exactly the same type, e.g., a `vector<int>` to another `vector<int>`, or perhaps a `list<string>` to another `list<string>`.

But what if we wanted to copy a `vector` of strings to a `list` of strings? We can use `std::copy` for that.

For example, we can copy one container into another this way:

```
vector<string> ws(50);
std::string str[50];
...
copy (ws.begin(), ws.end(), str);
```

(copies `ws` into `str`)

This works because `copy` is written entirely in terms of iterator operations, and iterators can be applied to almost any container.

```
template <class InputIterator,
         class OutputIterator>
OutputIterator copy(InputIterator first,
                    InputIterator last,
                    OutputIterator result)
{
    while (first != last)
    {
        *result = *first;
        result++; first++;
    }
    return result;
}
```

Notice how we have two template parameters, `InputIterator` and `OutputIterator`, that get replaced when `copy` is used. Of course, the names for these parameters are arbitrary. We could just as well have called them `George` and `Martha` instead of `InputIterator` and `OutputIterator` (at least, if we ignore documentation quality). How, then, does the compiler know that this `copy` algorithm is supposed to work with iterators?

It doesn't, really. But the `copy` operation is written in terms of `operator*`, `operator++` and `operator!=`, all of which are part of the conventional iterator interface. So the compiler will allow us to use `copy` with any data type that is sufficiently iterator-like to provide those operations.

### 2.1 Using `std::copy`

So, for example, in one version of `Book` (using a dynamically allocated array), we implemented the `Book` constructor and assignment operator as shown here:

`oldBook.cpp` +

```
class Book {
public:
```

```

Book (int nAuthors, Author* a,
      string theTitle, string theID);
Book (const Book& b);
:
private:
    std::string title;
    int numAuthors;
    Author* authors; // dynamic array of authors
    std::string identifier;
};

:

Book::Book (int nAuthors, Author* a,
            string theTitle, string theID);
{
    authors = new Author[nAuthors];
    numAuthors = nAuthors;
    identifier = theID;
    for (int i = 0; i < numAuthors; ++i)
        authors[i] = a[i];
}

Book::Book& operator= (const Book& b)
{
    delete [] authors;
    authors = new Author[b.numAuthors];
    numAuthors = b.numAuthors;
    identifier = b.identifier;
    for (int i = 0; i < numAuthors; ++i)
        authors[i] = b.authors[i];
    return *this;
}

```

With the standard copy function, this can be written:

[bookCopy.cpp](#) +

```

Book::Book (int nAuthors, Author* a,
            string theTitle, string theID);
{
    authors = new Author[nAuthors];
    numAuthors = nAuthors;
    identifier = theID;
    copy (a, a+nAuthors, authors);
}

Book::Book& operator= (const Book& b)
{
    delete [] authors;
    authors = new Author[b.numAuthors];
    numAuthors = b.numAuthors;
    identifier = b.identifier;
    copy (b.authors, b.authors+b.numAuthors, authors);
    return *this;
}

```

OK, big deal. We saved two whole lines of code. Is that worth worrying about?

Well, that's really not the point. What we *did* gain is the instant recognition that what is going on is a “copy”. Someone reading the original version would need to study the loop and recognize the pattern of a copy. That might only take a few seconds, but multiply that by all the places in a real program where copies and similar common, trivial programming patterns occur. The total gain in readability may be substantial. And, as you gain experience using these standard algorithms, you will probably find that you avoid a lot of diddy little programming mistakes that you would have made in endlessly rewriting the more detailed explicit loops.

## 2.2 copy and I/O iterators

We can use the standard template `ostream_iterator` to get an output iterator that stores items in an output stream. All iterators represent a position within a container. In this case, the container is the output stream and the “position” is the place where we are set to write our next output. In effect, then, copying to that position will let us write a whole series of items.

```

#include <algorithm>
#include <iostream>
#include <iterator>
#include <string>

using namespace std;

int main ()
{
    int v[5] = {-1, 5, 5, 5, 8};
    string s[4];
    s[0] = "zero";
    s[1] = "one";
    s[2] = "two";
    s[3] = "three";

    copy (v, v+5, ostream_iterator<int>(cout, "\n"));
}

```

```
// writes -1 5 5 5 8, each number on a separate line
copy (s, s+4, ostream_iterator<string>(cout, "\n"));
// writes zero!=one!=two!=three!=
return 0;
}
```

Similarly, there is an input iterator called `istream_iterator`, which can be used to read from an input stream. Try compiling and running the program shown here.

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

using namespace std;

int main ()
{
    copy (istream_iterator<string>(cin), // input iterator reading strings from cin
          istream_iterator<string>(), // end-of-file position
          ostream_iterator<string>(cout, "\n") // output iterator writing to cout
    );
    return 0;
}
```

Note that it doesn't quite exactly copy its input to its output. Can you figure out why?

[Answer](#) +

The obvious difference is that we are separating words by newline characters on output. In the input, they could be separated by any whitespace characters. Hence the input

```
abc def  
ghi
```

will generate the output

```
abc  
def  
ghi
```

The `istream\_iterator` uses ordinary `>>` to read things. One of the characteristics of `>>` is that it skips over leading whitespace before it begins actually processing characters. So another, more subtle difference is that “extra” spaces and other whitespace will disappear.

The input

```
abc      def  
  
ghi
```

will generate the output

```
abc  
def  
ghi
```

## 2.3 Copying, Overwriting, and Inserting

This is dangerous:

```
int a[5] = {1, 2, 3, 4, 5};  
int b[3];  
copy (a, a+5, b);
```

It is the generic equivalent of

```
int a[5] = {1, 2, 3, 4, 5};  
int b[3];  
for (int i = 0; i < 5; ++i)  
    b[i] = a[i];
```

which “obviously” writes past the end of the array `b`.

For similar reasons, this does not work:

```
int a[5] = {1, 2, 3, 4, 5};  
vector<int> v;  
copy (a, a+5, v.begin());
```

copy writes data into existing positions - you need to be sure that the data slots actually exist.

Pretty much the same is true for the output range of any generic function - they all assume that the data positions you name as the output range already exist.

### 2.3.1 Making room

This works:

```
vector<int> v;  
:  
int a[5] = {1, 2, 3, 4, 5};  
v.resize(5);  
copy (a, a+5, v.begin());
```

because `resize` actually creates the data positions.

This does not:

```
vector<int> v;  
:  
int a[5] = {1, 2, 3, 4, 5};
```

```
v.reserve(5);
copy (a, a+5, v.begin());
```

because `reserve` merely sets things up so that, if we later try to create some data slots, we won't need to allocate new memory for them. The data slots may exist in the sense that the space for them has been allocated, but they have not actually been initialized and the vector's `size()` indicates that those positions are still unused.

### 2.3.2 copying and expanding

`resize` is a useful trick, but it's specific to vectors. There's a more general solution to this problem — special iterators that are used to create new data slots as they are being filled.

- The special iterators are
  - `back_inserter`
  - `front_inserter`
  - `inserter`
- All three are contained in `<iterator>`

### 2.3.3 back\_inserter

`back_inserter(container)` returns an iterator on that container

- This is an [output iterator](#) - you can assign to it but not look at it
- Each assignment to the element at this iterator results in a `push_back()` call on the container

```
int a[5] = {1, 2, 3, 4, 5};
vector<int> v;
assert (v.size() == 0);
copy (a, a+5, back_inserter(v));
assert (v.size() == 5); // Five push_back's were done
```

### 2.3.4 front\_inserter

`front_inserter(container)` returns an iterator on that container

- This is an output iterator - you can assign to it but not look at it
- Each assignment to the element at this iterator results in a `push_front()` call on the container

```
int a[5] = {1, 2, 3, 4, 5};
list<int> ll;
assert (ll.size() == 0);
copy (a, a+5, front_inserter(ll));
assert (ll.size() == 5);
assert (ll.front() == 5);
assert (ll.back() == 1); // note that the order fo the data is reversed
```

Obviously, this only works for containers that actually provide the `push_front` operation, so we can't do this, for example, with vectors.

### 2.3.5 inserter

`inserter(container,iter)` returns an iterator on that container denoting the same position as `iter`

- This is an output iterator - you can assign to it but not look at it
- Each assignment of `foo` to the element at this iterator results in an `insert(iter,foo)` call on the container

```
int a[5] = {1, 2, 3, 4, 5};
list<int> ll(3, 0);
list<int>::iterator pos = ll.begin();
++pos;
copy (a, a+5, inserter(pos,ll));
// ll contains 0 1 2 3 4 5 0 0
```

## 3 Other Useful Generic Functions in the `std::` Library

### 3.1 equal

Another useful function is `equal`, which tests to see if corresponding elements in two position ranges are equal:

```

int v[5] = {-1, 5, 5, 5, 8};
int w[3] = {5, 5, 5};

assert (!equal(w, w+3, v));
assert (equal(w, w+3, v+1));
assert (equal(v+1, v+4, w));

```

For example, suppose that we wanted to know if two books have the same list of authors. We could have written it this way:

```

bool sameAuthors (const Book& left, const Book& right)
{
    if (left.numberOfAuthors() == right.numberOfAuthors())
    {
        auto lpos = left.begin();
        auto rpos = right.begin();

        for (int i = 0; i < left.numberOfAuthors(); ++i)
        {
            if (*lpos != *rpos)
                return false;
            ++lpos; ++rpos;
        }
        return true;
    }
    else
        return false;
}

```

but we can simplify this to

```

bool sameAuthors (const Book& left, const Book& right)
{
    if (left.numberOfAuthors() == right.numberOfAuthors())
    {
        return equal(left.begin(), left.end(), right.begin());
    }
    else
        return false;
}

```

or

```

bool sameAuthors (const Book& left, const Book& right)
{
    return (left.numberOfAuthors() == right.numberOfAuthors())
        && equal(left.begin(), left.end(), right.begin());
}

```

(Note: it's a good practice to always do the cheap  $O(1)$  test first before diving in to a test that requires looping through the whole container.)

Similarly, we could easily implement a comparison operator for sorting books by author lists via the function `lexicographical_compare`. See the references at the end of these notes for details.

## 3.2 find and lower\_bound

The `find` function performs an unordered sequential search for a value in some range of positions.

```
p = find(ws.begin(), ws.end(), "foobar");
```

searches a container for the indicated string.

If `find` cannot locate the indicated string, it returns the end position of the search range (e.g., `ws.end()` in the example above). Remember that iterator ranges are always inclusive on the starting position, exclusive on the ending position, so the end position of a search range could not possibly be returned by a successful search.

We can provide the equivalent of an [ordered insert](#) for containers that support the `insert` operations using another standard template function:

```

Container<Element> container;
:
cin >> x;
Container::iterator p = lower_bound (container.begin(), container.end(), x);
container.insert (x, p);

```

`lower_bound` returns the first location where `x` could be inserted if the collection is being maintained in sorted order.

What makes `lower_bound` a really nice function is that it uses a binary search when given random-access (e.g., `std::vector`) or trivial (array) iterators, and uses a sequential search when given merely forward or bi-directional (e.g., `std::list`) iterators.

There is a related function, `upper_bound`, which is called the same way, that returns the *last* position where `key` could be inserted. For example, suppose we had a container with 3 copies of `key` already in it. Then `lower_bound` would point to the first of these three, and `upper_bound` would point to the position just after the third copy.

### 3.3 count

```
std::string a(50, ' ');
:
count(a, a+50, "xxx", n);
```

Counts the number of occurrences of "xxx" in the array `a`.

### 3.4 fill

```
std::string a[50];
fill_n(a, 5, "Hello");
```

fills the first 5 positions of `a` with "Hello".

~

```
fill(a+5, a+50, "GoodBye");
```

Fills the remaining positions of `a` with "GoodBye".

## 4 std:: Library Generics That Take Functions

### 4.1 Passing Functions as Parameters

In C++, we can pass functions as parameters to other functions. For example, this is legal (though a bit silly):

```
typedef int *FunctionType (int); // FunctionType is declared as a name
                                // for the set of all functions that
                                // take 1 int and return an int.
int doItTwice(FunctionType f, int i)
{
    return f(f(i));
}
int mult2 (int x) {return 2*x;}
:
int Twelve = doItTwice(mult2, 3);
```

The function `doItTwice` will actually call `mult2` twice, passing the result of the first call as the parameter of the second.

Functions can be useful as parameters in various applications. They are often used in conjunction with templates.

For example, we have already seen the `find` function that performs an `unorderedSearch` for a value in some range of positions:

```
p = find(ws.begin(), ws.end(), "foobar");
```

searches a container for the indicated string.

Although the preceding example may have been a bit silly, there are some very useful generic functions that expect to be given a function as a parameter.

### 4.2 for\_each

Some of the most commonly used generics (in my own coding, only `copy` gets used more often) are for applying a function to every element in a range:

`for_each` applies a unary function (i.e., a function taking a single parameter) to each element in a range.

There's an old saying that "if the only tool a man has is a hammer, then every problem looks like a nail."

```

void printLength(string s)
{
    cout << s << " is of length "
        << s.length() << endl;
}
:
for_each (ws.begin(), ws.end(), printLength);

```

In this case, if ws contains the words [“aardvarks”, “are”, “furry”], then the output would be:

```

aardvarks is of length 9
are is of length 3
furry is of length 5

```

`for_each` is the “hammer” of generic functions. Given a sufficiently complicated function for its third parameter, you can use it to replace any loop and to do anything you might do with almost any of the other generic functions in std.

But that doesn’t mean you *should* wield this hammer on every problem you see. The purpose of using generics is to express the code in a way that makes that “instant recognition” of the purpose of a loop possible. So if the computation being performed is really a “copy”, or a search (“find”), or a transform, or a selective erasure, or any of the other more specialized kinds of iterations provided as std generics, then you should use those more specific functions rather than `for_each`.

When in doubt, choose the most expressive option, the one that gives the most information to the person reading your code.

## 4.3 transform

`for_each` applies a function to every element in a range and disregards the return values, if any.

`transform`, on the other hand, applies a function to every element in a range and collects the returned values by copying them into an output range.

```

#include <iostream>

string convertToString(int i)
{
    char buffer[256];
    ostringstream obuffer(buffer); // whatever we write into
        // obuffer will appear in the buffer character array
    out << i << ends; // ends terminates a string output
    return string(buffer);
}
:
int v[5] = {-1, 5, 5, 5, 8};
string s[5];
transform (v, v+5, s, convertToString);

```

In this example, each of the five v values will be passed, one at a time, to `convertToString` and the five resulting string values, [“-1”, “5”, “5”, “5”, “8”], stored in s.

The first two parameters give the input range, and the third parameter is the beginning of the output range. We don’t have to specify the end of the output range, because there will be just as many outputs as there are input values. The final parameter is, of course, the function we want to apply to each element.

`transform` can be used to replace each element in a range by some function of itself. To do this, we simply make the output range the same as the input range.

For example, suppose we had an array `myNumbers` of N floating point numbers and were really interested in working with the absolute value of all those numbers. We could write

```
transform(myNumbers, myNumbers+N, myNumbers, fabs);
```

and thereby replace every element in `myNumbers` by its absolute value (the `fabs` function).

## 4.4 “\_if” Generics and Predicates

Many of the generic functions that do searching or comparisons of some kind (including some we have already discussed) have an alternate “\_if” version that can take a function that is used in place of the obvious defaults.

For example, we earlier looked at this example of `find`:

```

list<string> ws;
:
p = find(ws.begin(), ws.end(), "foobar");

```

to search a container for a specific string.

Suppose, however, that we were interested in searching for a string that *contained* “foobar”. We could do this by supplying the appropriate test as a function:

```

bool containsFoobar (const string& s)
{
    return s.find("foobar") != string::npos;
}
:
list<string> ws;
:
p = find_if(ws.begin(), ws.end(), containsFoobar);

```

The function passed to `find_if` must return a `bool`, for reasons that should be apparent. This kind of function is sometimes called a [predicate](#).

Or, suppose that we wanted to find out if any of the strings in our container were exactly one character long:

```
bool oneCharLong (const string& s)
{
    return s.size() == 1;
}
:
list<string> ws;
:
p = find_if(ws.begin(), ws.end(), oneCharLong);
```

C++11 also introduced a useful variation `find_if_not`.

Similarly,

```
int k = 0;
count_if (ws.begin(), ws.end(), oneCharLong, k);
```

would count how many strings in the container `ws` are one character long.

And, the rather quixotically named

```
remove_copy_if (ws.begin(), ws.end(),
                ostream_iterator<string>(cout, "\n"),
                oneCharLong);
```

copies all the words in `ws` *except* for the single-character ones, copying them to the output stream (i.e., writing them on `cout`, one per line).

## 4.5 all\_of, any\_of, none\_of

Introduced in C++11, these functions offer the equivalents of the logical quantifiers  $\forall$ ,  $\exists$ , and  $\nexists$ .

These take a range of positions to examine, and a unary predicate (“unary” == one parameter, “predicate” == function with a `bool` return type), e.g.,

## 4.6 all\_of, any\_of, none\_of

Introduced in C++11, these functions offer the equivalents of the logical quantifiers  $\forall$ ,  $\exists$ , and  $\nexists$ .

These take a range of positions to examine, and a unary predicate (“unary” == one parameter, “predicate” == function with a `bool` return type)

```
bool containsFoobar (const string& s)
{
    return s.find("foobar") != string::npos;
}
:
list<string> ws;
:
if (all_of(ws.begin(), ws.end(), containsFoobar))
    cout << "Every element in ws contains 'foobar'." << endl;
if (any_of(ws.begin(), ws.end(), containsFoobar))
    cout << "At least one element in ws contains 'foobar'." << endl;
if (none_of(ws.begin(), ws.end(), containsFoobar))
    cout << "At least one element in ws contains 'foobar'." << endl;
```

Of course, we could do these same tests using `find_if` and checking the position it returned, for example:

```
// These two tests are the same.
bool test1 = any_of(ws.begin(), ws.end(), containsFoobar);
bool test2 = find_if(ws.begin(), ws.end(), containsFoobar) != ws.end();

// These two tests are the same.
bool test3 = none_of(ws.begin(), ws.end(), containsFoobar);
bool test4 = find_if(ws.begin(), ws.end(), containsFoobar) == ws.end();

// These two tests are the same.
bool test5 = all_of(ws.begin(), ws.end(), containsFoobar);
bool test4 = find_if(ws.begin(), ws.end(), not1(containsFoobar)) == ws.end();
```

But the `all_of`, `any_of`, and `none_of` functions are easier to read and convey the programmer’s intent more clearly.

# 5 C++11 Lambda Expressions

One thing that you may have noticed is that using generics that take function parameters works only if we already have a suitable function or are willing to write one.

The need to provide these small functions can result in an explosion of short, used-only-one-time functions in our code. And, because C++ does not allow functions to be nested within other functions, these small functions will be separated from the generic function call that uses them. This can impair the readability of the code.

The new C++11 standard has something of an answer for this problem. It allows you to write an anonymous description of a short function right in the place where you would call it or, more often, pass it to other functions. This description called a [lambda expression](#).

One of the most common mistakes that I see students make when trying to work with generics is to try and take a short-cut by simply writing an expression in place of a “proper” function. For example, instead of

```
bool oneCharLong (const string& s)
{
    return s.size() == 1;
}
:
list<string> ws;
:
p = find_if(ws.begin(), ws.end(), oneCharLong);
```

The name comes from the “lambda calculus”, a mathematical notation for functions, such as

$$\lambda x . x^2 + 2x - 1$$

as a way of writing an unnamed function equivalent to

$$f(x) = x^2 + 2x - 1$$

The idea of embedding lambda expressions in a programming language dates back to LISP, the second-oldest high-level programming language.

I often see students do

```
list<string> ws;
:
p = find_if(ws.begin(), ws.end(), s.size() == 1); // No!
```

which doesn’t work because “s” is undeclared.

In effect, the lambda expression provides a [legal](#) way to do what these students were attempting.

A lambda expression has components:

*capture-description parameter-list function-body*

Of these, the *parameter-list* and the *function-body* are pretty much the same as they would be in an ordinary, non-member function.

Here’s our “find a single-character string” search using a lambda expression:

```
list<string> ws;
:
p = find_if(ws.begin(), ws.end(),
            [] (const string& s) {s.size() == 1});
```

The *capture-description* explains what to do about variables that are “captured” – used in the function body but not passed as parameters. There are several options for what to do here, but the most common are likely to be:

- [] if the function won’t use any variable names that are not declared, as s is above, as a function parameter
- [&] if the function should capture variables as references to identically named variables in the current scope,
- [this] for functions that should capture the this pointer of the current scope (in effect turning the lambda expression into a member function).

What you [don’t](#) see in a lambda expression is a name for the function, because the whole point is to use these for one-shot functions that aren’t going to be referenced anywhere else in the program, nor will you see a description of the function’s return type, because the compiler will deduce this from examining the return statements in the function body.

Here’s some of the examples from the previous section, redone using lambda expressions:

C++

```
void printLength(string s)
{
    cout << s << " is of length "
        << s.length() << endl;
}
:
for_each (ws.begin(), ws.end(), printLength);
```

C++ 11

```
for_each (ws.begin(), ws.end(),
          [] (string s) {
              cout << s << " is of length "
                  << s.length() << endl;
          });
:
```

C++

```
string convertToString(int i)
{
    char buffer[256];
    ostrstream obuffer(buffer);
    out << i << ends;
    return string(buffer);
}
;
int v[5] = {-1, 5, 5, 5, 8};
string s[5];
transform (v, v+5, s, convertToString);
```

C++ 11

```
int v[5] = {-1, 5, 5, 5, 8};
string s[5];
transform (v, v+5, s,
          [] (int i) {
              char buffer[256];
              ostrstream obuffer(buffer);
              out << i << ends;
              return string(buffer);
});
```

C++

```
bool containsFoobar (const string& s)
{
    return s.find("foobar") != string::npos;
}
;
list<string> ws;
;
p = find_if(ws.begin(), ws.end(),
             containsFoobar);
```

C++ 11

```
list<string> ws;
;
p = find_if(ws.begin(), ws.end(),
            [] (const string& s) {
                return s.find("foobar") != string::npos;
});
```

It will be interesting to see just how strongly the C++ programming community embraces this new feature.

## 6 Function

Function types seem a bit awkward, and there are times when we want to pass a “behavior” to a function or to save a “behavior” in a data structure, but a true function just won’t do. That takes us into the strange and peculiar idiom of programming called “functors”

A [functor](#) is an object that is created to simulate a function. Why would we want to do that? Well, objects can do things that functions can’t. Objects can store information, and can be stored in other data structures. Functions can’t do these, at least not with the same ease and flexibility. A functor can often have the best of both worlds.

### 6.1 Example: Function and User Interface Programming

An example of functors can be found in many windowing libraries. Think of the problem of building a menu bar, like the one you see across the top of most windows in user interfaces. A menu bar is probably just an ordered collection of menus:

```
class MenuBar {
    :
    vector<Menu> menus;
    :
};
```

A menu would have a name (e.g., “File”, “Edit”), but would itself contain a number of menu items.

```
class Menu {
    :
    string menuName;
    vector<MenuItem> items;
    :
};
```

And MenuItem? Well, they certainly have names, but they also will typically have a place to store an object or a pointer to an object that actually performs the desired function.

```
class MenuItemAction {
public:
    void perform() /* by default, do nothing */
};

class MenuItem {
    string itemName;
    MenuItemAction action;
public:
    MenuItem (string name, MenuItemAction act)
```

```

        : itemName (name), action(act) {}
    void setAction (MenuItemAction act) {action = act;}
    void itemWasSelected () {action.perform();}
};

```

When a menu is being built, the MenuItem objects are created with appropriate actions:

```

class FileReader: public MenuItemAction
{
    void perform()
    {
        :
        code to read from a file
        :
    }
};

class FileSaver: public MenuItemAction
{
    void perform()
    {
        :
        ...code to write to a file ...
        :
    }
};

class Quitter: public MenuItemAction
{
    void perform()
    {
        :
        ...code to close window and shut down program ...
        :
    }
};

FileReader rdr;
FileSaver svr;
Quitter quit;
// build a typical file menu
fileMenu.items.push_back (MenuItem("load", rdr));
fileMenu.items.push_back (MenuItem("save", svr));
fileMenu.items.push_back (MenuItem("exit", quit));

```

When a user actually selects one of these items from the menu, the windowing system calls the item's `itemWasSelected` function, which in turn calls the `perform()` function of its action.

`rdr`, `svr`, and `quit` are examples of functors; they are objects created for the sole purpose of providing a single function, which in this case is called `action`.

## 6.2 operator()

Now in the previous example, the functors are called by calling their `action` function member. But C++ has special support for functors. We can write functors that are called just like regular functions. We do this by defining an `operator()`.

Now, we've seen that you can define operators like `<`, `==`, `=`, and `*`.

But it is a truly strange feature of C++ that `()` is considered an operator. It is a postfix operator (appearing to the right of the object that it operates on, e.g., `x()`). And, it can be defined to take any number of parameters of any legal C++ type, e.g., `x(23, "abcdef")`.

So when you see something like `w(z)` written in C++, the only way to tell if you are looking at

- a function `w` applied to a parameter `z`, or
- a call to the `operator()` member of an object `w`

is to find the declaration of `w` and see if it really is a function or an object.

## 6.3 A Predicate Functor

Let's see how this works. When we introduced the notion of iterators, we looked briefly at the standard function template `find_if`.

The code shown here, for example, searches a vector for the first string containing no more than 4 characters.

```

vector<string> v;
:
bool isShort(const std::string& s)
{
    return (s.size() <= 4);
}
:
p = find_if(v.begin(), v.end(), isShort);

```

Now, let's write the same thing replacing `isShort` by a functor.

```
vector<string> v;
⋮
class IsShort {
public:
    bool operator() (const std::string& s)
    {
        return (s.size() <= 4);
    }
};
IsShort isShort;
⋮
p = find_if(v.begin(), v.end(), isShort);
```

## 6.4 Why Do Functors Work Where Functions Are Expected?

The code for `find_if` looks like:

```
template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred) {
    while (first != last && !pred(*first)) ++first;
    return first;
}
```

So when we call `find_if( ... , isShort)`, `isShort` is passed to `find_if` as the parameter `pred`, and the body of `find_if` calls `pred(*first)`.

In the original version, `isShort` (and therefore `pred`) was a function, so `pred(*first)` was an ordinary function call.

Now, however, `isShort` is an object that happens to define an `operator()` taking a single `string` parameter, so `pred(*first)` is a call to `pred`'s `operator()`.

## 6.5 Functors Can Have Memory

OK, so what? What does the `isShort` functor do that the `isShort` function did not? Absolutely nothing.

But now, suppose that we're not always interested in strings of length 4 or less. Sometimes we may want strings of length 2 or less, or 8 or less, ...

```
vector<string> v;
⋮
int length;
cout << "What's the longest acceptable string length?" << flush;
cin >> length;
p = find_if(v.begin(), v.end(), ?????);
```

There's no good way to write an ordinary function that we can pass to `find_if` that would search for strings of variable lengths. But a functor can fill the bill very nicely, by storing the critical length inside the object.

```
vector<string> v;
⋮
class IsShort {
    int length;
public:
    IsShort (int len): length(len) {}

    bool operator() (const std::string& s)
    {
        return (s.size() <= length);
    }
};

vector<string> v;
⋮
int length;
cout << "What's the longest acceptable string length?" << flush;
cin >> length;
IsShort isShort (length);
p = find_if(v.begin(), v.end(), isShort);
```

Once we know the length we want to hunt for, we create an object that remembers that length and that, when its `operator()` is called with some string, compares that string's length to the value it has saved.

Finally, we note that we can actually do without the `isShort` variable by using the constructor to create a temporary functor object to pass to `find_if`.

```
vector<string> v;
⋮
class IsShort {
    int length;
```

```

public:
    IsShort (int len): length(len) {}

    bool operator() (const std::string& s)
    {
        return (s.size() <= length);
    }
};

vector<string> v;
:
int length;
cout << "What's the longest acceptable string length?" << flush;
cin >> length;

p = find_if(v.begin(), v.end(), IsShort(length));

```

## 6.6 std Functors for Comparisons

The C++ standard library provides a number of functors. The most commonly used are functors for comparing pairs of objects using the relational operators.

Suppose we have a vector of strings that we want to sort into ascending order. The standard sort function takes three parameters. The first two are iterators denoting the range of items to be sorted. The third is a comparison function or functor that is used to compare two objects and return “true” if the first object should come before the second in the desired sorted order.

We want the strings arranged into ascending order, so we would like to use the ordinary `<` for comparisons. We might be able to do this:

```
sort(v.begin(), v.end(), operator<);
```

taking advantage of the “real” name of the less-than operator. This should work for strings, but won’t work for some other data types for which `operator<` is a member function.

A safer alternative that will work for any data type that provides an `operator<`, member function or standalone function, is provided by the C++ standard library. The standard library provides `less<T>` for this purpose, so we could write:

```
sort (v.begin(), v.end(), less<string>());
```

`less` is not particularly complicated:

```

template <class T>
struct less : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const
    { return x < y; }
};
```

As you can see, the `less` class simply provides an `operator()` that uses the `<` operator on its parameters.

In addition to `less`, the standard library provides `equal_to`, `not_equal_to`, `greater`, `greater_equal`, and `less_equal`, all declared in the header `<functional>`.

---

A bit of a challenge: what does this function do, and can you think of a better name for it?

```

template <typename T>
function mysteryFunction (list<T>& aList, const T& x)
{
    list<T>::iterator pos = find_if (aList.begin(), aList.end(),
                                    bind2nd(x, greater<T>()));
    aList.insert (pos, x);
}
```

[bind2d](#) turns a two-parameter functor into a one-parameter functor by supplying a fixed value for the first parameter.

[Answer](#) +

“addInOrder” or something similar.

The `find_if` call searches for the position of the first element in the list greater than `x`, or the `end()` position if no such element exists. The `insert` call then puts `x` into the list right in front of that position. The net result is an ordered insertion.

If you change “list” to “vector”, the code would still compile and would run correctly. Why would that be a bad idea?

Answer +

It's slower than the ordered insertion algorithms we developed for array-like structures. That's because the `insert` call, although  $O(1)$  for `std::list`, is  $O(\text{size}())$  for a vector.

## 6.7 Substituting Your Own Comparison Functions

Sometimes none of the standard relations will do. In those cases, we just define our own functors.

Suppose you were keeping a vector of `PersonnelRecord`, but there is no `<` for entire `PersonnelRecords`.

- You need to pick some appropriate key, a data member or group of members that uniquely defines each record.

For example, we might use a combination of name and address.

- Provide a functor that compares the keys:

```
class CompareByNameAddress {
public:
    bool operator()
        (const PersonnelRecord& p1,
         const PersonnelRecord& p2)
    {return (p1.name() < p2.name())
     || ((p1.name() == p2.name())
          && (p1.address() < p2.address()));}
};

set<PersonnelRecord, CompareByNameAddress> employees;
```

Then

```
sort (v.begin(), v.end(), CompareByNameAddress());
```

would sort your personnel records into order by name, with any people with the same name being sorted by address.

The `()` following “`CompareByNameAddress`” in the call above are important. `CompareByNameAddress` is a class, but we don't pass classes as parameters to functions, we pass objects. So what is `CompareByNameAddress()`? It's a call to the default constructor for the `CompareByNameAddress` class, which returns an object of that type which, in turn, we pass to the `sort` function.

## 7 References

This has not been an exhaustive list of all the generic functions in the C++ standard library. There are many others, but enough to give you a taste. Others are scattered through your textbook.

For a more compact listing, look at [this summary sheet](#).

# Recursion

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Overview](#)
- [2 You've Gotta Believe!](#)
- [3 Recursion versus Iteration](#)
- [4 Making the Choice](#)

Most of the time, when you want to process a lot of data, you use a loop. Recursion is an alternative that you could employ, but if you're like most programmers, you're a lot more comfortable with looping or iteration than you are with recursion. That can be a problem, in some circumstances, because there are some tasks that really are more easily solved via recursion.

A function is *recursive* if it calls itself, or calls some other function that eventually causes it to be called again. This means that a recursive function may have several different calls to it active at the same time. In fact, we call the collection of information that represents a function call in progress an *activation* of the function.

Recursive functions tend to fall into certain familiar patterns. Every recursive function will test its inputs to see if they represent a special case simple enough to be solved without recursion. These are called *base cases*. When the function's inputs do not constitute a base case, the recursive function must somehow break apart the problem to be solved into one or more smaller sub-problems. These sub-problems are solved via recursive calls, and when the recursive calls are done the function must combine the sub-problem solutions into a solution for the original problem.

## 1 Overview

Here is a simple example of a recursive function.

```
template <class Iterator>
int length (Iterator start, iterator stop)
{
    if (start == stop) // range is empty: base case
        return 0;      // base case solution
    else
    {
        ++start;       // shorten the range
        return 1 + length(start, stop);
    }
}
```

This one uses recursion to compute the length of a range of positions. (We'll ignore, for now, the fact that there are easier ways to do this.) As simple as this example is, it illustrates all the important characteristics of a recursive function.

- The base case occurs when the range is empty. In that case, we can compute the length of the range immediately, without recursion or iteration, because we know that an empty range has length zero.
- If the range is not empty, we need to take the problem apart into a smaller sub-problem. We can do this by computing the length of the range left when we remove the first element of the current range. We use a recursive call to get the length of this short-term range, and then add one to it to get the length of the original range.

## 2 You've Gotta Believe!

In an earlier edition of your textbook, Weiss suggested the that understanding a recursive function is a bit like attending an old time gospel meeting: "you've gotta believe!"

```
template <class Iterator>
int length (Iterator start, iterator stop)
{
    if (start == stop) // range is empty: base case
        return 0;      // base case solution
    else
    {
        ++start;       // shorten the range
        return 1 + length(start, stop);
    }
}
```

**Question:** How do we know next the recursive call on the shorter range really will return the correct value?

Answer +



*"You've gotta believe!"*

If we

- take it for granted that the recursive call really will work, and
- we can convince our self that the additional work we do with the some problem solution to convert it into the solution to the entire problem really would work, and
- we have the correct solution to the base cases, and, finally,
- we convince ourselves that every recursive call really is on a smaller problem than the one before it, so that the chain of recursive calls will eventually reach one of our base cases.

then the entire package should work.

That may seem like a lot of work, but it's not as if iterative functions were a whole lot easier to get right. When we write an algorithm using loops, we need to convince ourselves that the loop processes each individual element correctly, that the processing of the individual elements adds up to a solution for the entire problem, that the loop condition is correct and will cause us to go around the loop the correct number of times, that the entire loop will function correctly even in the case where we go around the loop zero times (if that is possible), and finally we must convince ourselves that the loop will eventually exit, and not go on looping forever.

## 3 Recursion versus Iteration

Recursion and iteration (looping) are equally powerful. We know, for example, that any recursive algorithm can be rewritten to use loops instead. We know this is true because that's how recursion is implemented on the underlying machine. Your text describes how computer systems use a runtime stack (called the [activation stack](#)) to keep track of the return addresses, actual parameters, and local variables associated with function calls. Each function call actually results in pushing an [activation record](#) containing that information onto the stack. Returning from a function is accomplished by getting and saving the return address out of the top record on the stack, popping the stack once, and jumping to the saved address.

In a sense, then, computers really don't do recursion. What we might write as a recursive algorithm really gets translated as a series of stack pushes followed by a jump back to the beginning of the recursive function, all implemented using the underlying CPU whose internal code is, fundamentally, iterative.

We can go the other way was well. Given an algorithm with loops, we could, without too much trouble, replace each loop body with a recursive function that would perform a single iteration of the original loop, check to see if the loop would terminate, and if not call itself recursively to simulate the next time around the loop.

## 4 Making the Choice

If neither recursion nor iteration is fundamentally more powerful than another, how do we decide which to use?

- Some algorithms are just easier to write one way than another. The "length of a range" function would have been easier to write with a loop. Later we will encounter some searching and sorting algorithms that are so naturally recursive that It's hard to imagine doing them iteratively, and any iterative form would be far more difficult to understand.
- Some languages support recursion but not looping. Some languages have loops but not recursion. Although such programming languages are increasingly rare, in those cases, our choice may be made for us.
- Iteration is usually (though not always) faster than an equivalent recursion. For example, binary search can be written recursively, as shown in the loop-free version here:

```
int binSearch (const int arr[], int first, int last, int target)
// search for target in ordered array of data
// return index of target, or index of
// next smaller target if not in collection
{
    int mid;      // index of the midpoint
    int midValue; // object that is assigned arr[mid]
    int origLast = last; // save original value of last

    // Reduce the area of search
    // until it is just one target
    if (first < last) { // test for nonempty sublist
        mid = (first + last) / 2;
        midValue = arr[mid];
        if (target == midValue)
            return mid;
        else if (target < midValue)
            return binSearch (arr, first, mid, target);      // search lower sublist
        else
            return binSearch (arr, mid+1, last, target);    // search upper sublist
    }
    else
        return origLast;
}
```

But that's not any simpler and will run noticeably slower. (Not in a big-O sense. The complexity of the recursive and iterative forms are the same, but the multiplicative constants for recursion will be higher.)

- Other performance/environmental issues may come into play.

In some operating systems, the system's activation stack is very small. You may see this in programming for [\*embedded systems\*](#), small computer systems used to control devices (e.g., microwave ovens, aircraft displays, automotive computers that control engine timing, fuel injection, etc.). Such systems generally have very limited memory overall, including very limited activation stacks.

On such systems, recursion may be out of the question.

# Stacks

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 The Stack ADT](#)
  - [1.1 Overview of Implementation](#)
- [2 std Interface](#)
  - [2.1 Using a Stack](#)
- [3 Applications & Examples](#)
  - [3.1 Expression Evaluation](#)

Like vectors and lists, stack and queues are also ordered collections, but the construction and access rules are purposely limited.

- high recognition factor — people reading your code can more easily recognize what you are doing.
- improved choices for the implementing data structure — as a general rule, the more elaborate the ADT interface, the fewer options you will have for implementation.

## 1 The Stack ADT

Stacks organize the data in a Last-In, First-Out (LIFO) manner. We can think of a stack as a sequence of elements where we always add to and remove from the same end.

- Data is pushed onto a stack or
- popped off of a stack.
- We can only examine data on the top of the stack.

### 1.1 Overview of Implementation

Stacks can be easily implemented using array/vector-like data structures or via linked lists. You can see both possibilities below.

Actually, implementation of a stack is pretty trivial using either the `std::vector<T>` or `std::list<T>` types as an underlying implementation.

- To push onto the stack, do a `push_back` onto the underlying vector or list
- To pop from the stack, do a `pop_back` on the underlying vector or list
- To access the top of the stack, get the `back()` of the vector or list.

## 2 std Interface

The stack and queue abstractions do not allow access to arbitrary contained items.

- Therefore, many std conventions do not apply
  - `[i]`, iterators, `insert`, etc.
- Stacks and queues in std are not “full-fledged” containers

The std library treats stack and queue as special cases by letting you convert

- a `vector`, `list` or `deque` (we’ll introduce deques later in this course Section) into a stack, or
- a `deque` or `list` into a queue

These “converted” ADTs are called “adapters” in std.

### 2.1 Using a Stack

```

stack<string, vector<string> > stk;
stk.push("abc");
stk.push("de");
assert (stk.top() == "de");
stk.pop();
assert (stk.size() == 1);
stk.pop();
assert (stk.empty());

```

Here are some examples of typical stack manipulation.

(All the asserts should be OK.)

Notice that when instantiating the stack, we specify both the type of elements to go on the stack:

```
stack<string, vector<string> >
```

and the sequence to be used as the implementing data structure:

```
stack<string, vector<string> >
```

By the way, be careful with things like `stack<string, vector<string> >`. The blank between the two “>” is important. Without that, C++ assumes that you are writing the `>>` operator (as in `cin >> n;`), which would not be legal here.[^This annoying glitch in the language design is fixed in C++11.]

## 3 Applications & Examples

Stacks are among the most common data structures and crop up in a variety of applications. They are especially useful in parsing and translation of computer languages, mathematics, and other formal notations.

- parentheses matching
- postfix translation and evaluation
- compilers

Your text gives examples of some of these, and you will do one in the next assignment.

### 3.1 Expression Evaluation

Postfix (also known as Reverse Polish Notation or RPN) is a parentheses-free notation for mathematical expressions:

- operators appear after their operands, e.g.:
- `1 2 +` instead of `1+2`
- `1 2 3 * +` instead of `1+2*3`
- `1 2 + 3 * +` instead of `(1+2)*3`

Postfix is easily evaluated using a stack:

- when you see a constant, push it onto the stack
- when you see an operator that needs k operands,
  - pop k numbers from the stack
  - apply the operator to them
  - push the result back onto the stack

You can see how this works by running [this RPN calculator](#).

Now, you might find this example a bit artificial-looking. Most people don't write their expressions out in postfix, preferring the more conventional infix notation. Postfix does have the slight virtue that it can represent any algebraic expression without parentheses.

In fact, back in the not-so-distant past when scientific calculators did not have parentheses keys, anyone using such a calculator was accustomed to entering their intended calculation in postfix form. It's still common for calculators, compilers, and other software that must process expressions to convert those into postfix form and then use the above algorithm to evaluate the resulting postfix expression.

Conversion to postfix is, itself, generally performed via a stack (or via recursion, which we will later see has a close relationship to stack-based algorithms). For example, to convert a “normal” infix algebraic expression without parentheses into postfix, you could do this:

```

S = an empty stack;
while (more input available) {
  read the next token, T;
  if T is a variable name or a number,
    print it;
}

```

```
else { // T is an operator
    while (S is not empty and T has lower precedence than top operator on S) {
        print top operator on S;
        pop S;
    }
    push T onto S
}
while (S is not empty) {
    print top operator on S;
    pop S;
}
```

Try this with an expression like “1 + 2\*3 + 4”. The output will be “1 2 3 \* + 4 +”.

This algorithm can be modified, without too much trouble, to work with parentheses as well.

Another common application of stacks is in converting recursive algorithms to iterative, which we will discuss later.

# Queues

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 std Interface](#)
- [2 Implementing Queues](#)
  - [2.1 Implementing Queues Using Arrays](#)
- [3 Applications & Examples](#)

A *queue* is a sequential container in which

- elements are removed in the same order as their insertion. (*FIFO*: First-In, First-Out)
- Data is added to the end and removed from the front of a queue.
- only the front (least recently inserted) and back (most recently inserted) element may be accessed at any given time.
  - Actually, many authors would limit access to only the front element.
- In general, queues are used when things must be processed “in order”, but can “pile up” before we get to them.

## 1 std Interface

```
queue<string, list<string>> q;
q.push("abc");
q.push("de");
assert (q.front() == "abc");
assert (q.back() == "de");
q.pop();
assert (q.size() == 1);
assert (q.front() == "de");
q.pop();
assert (q.empty());
```

Here are some examples of typical queue manipulation.

(Again, all the assertions should be true.)

## 2 Implementing Queues

Like stacks, queues can be implemented using arrays or lists. A list-based implementation is pretty straightforward:

Implementing queues via a linked list

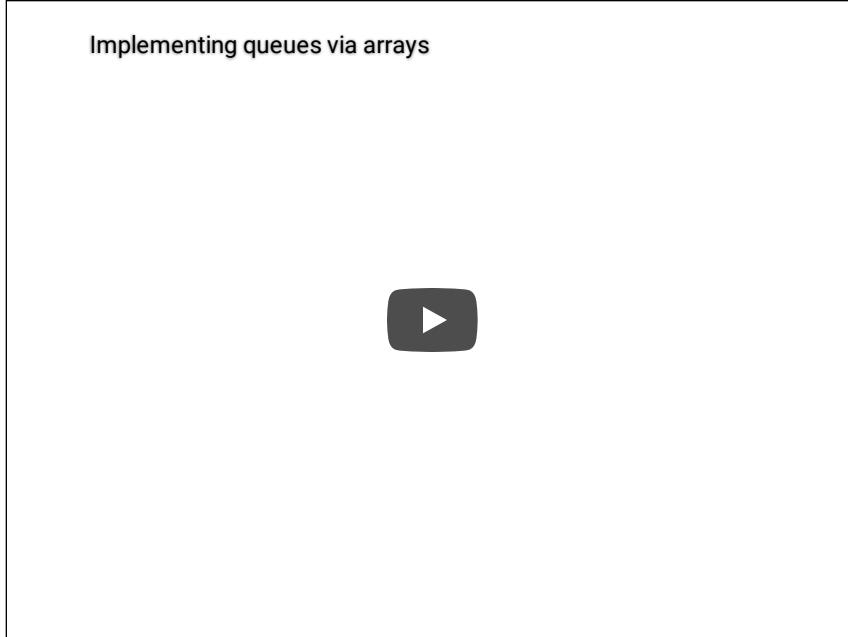


Notice that adding and removing elements from the queue are O(1) operations.

## 2.1 Implementing Queues Using Arrays

Surprisingly, an array-based implementations of queues is far more difficult than it was for stacks, because we must allow the structure to grow at one end and shrink from the other. If we add to the “end” of an array or vector, then we would be removing from the front. But that’s an  $O(\text{size}())$  operation. We’d like to avoid that.

Here’s an array-based implementation.



It repeatedly goes through the sequence: add “Adams” to the end, then add “Baker” to the end, then add “Carter” to the end, then remove the front (“Adams”), add “Davis” to the end, then remove (“Baker”), remove (“Carter”), and remove (“Davis”), leaving us with an empty queue.

Watch carefully as we go through multiple repetitions of this sequence. Notice how, as we do this, the “useful” portion of the array, the part that actually contains the queue’s data, keeps sliding further back in the array?

What do we do when we bump into the end of the array and still want to add something to the queue? There may be lots of unused space at the front, so we should be able to squeeze the new data in *somewhere*!

Now, we could just copy all the remaining data back to the beginning of the array. But doing this would make adding data to the end of the queue an  $O(N)$  operation (where  $N$  is the number of elements still in the queue). For the list we managed this in  $O(1)$  time, so we might ask if we can’t do that well with arrays.

The solution to this problem is to simply let `start` and `stop` “wrap around” to the beginning of the array. For example, when adding to the queue, instead of

`stop++;`

we use

`stop = (stop + 1) % size;`

where `size` is the size of the array being used to hold the queue.

The `%` is the “modulus” operator – it returns the remainder when the first argument is divided by the second. So if `stop==size-1`, then `size+1` equals `stop`, and the remainder when `stop` is divided by itself is 0, so `(stop+1)% size == 0`. Thus after `stop` (or `start`) have made it all the way to the end of the array, the next time we advance them they will “wrap around” to 0.

Here’s the actual implementation of pushing and popping:

`gpush.cpp` +

```
template <class T>
void queue<T>::push (const T& x)
{
    assert (theSize < ArraySize);
    stop = (stop + 1) % ArraySize;
    array[stop] = x;
    theSize++;
}

template <class T>
inline
```

```
void queue<T>::pop()
{
    assert (theSize > 0);
    start = (start + 1) % ArraySize;
    theSize--;
}
```

## 3 Applications & Examples

Common applications of queues include:

- tree & graph processing
- simulations
- I/O & storage buffers

Tree and graph processing will be explored in later sections of the course. Your text describes the use of queues in simulations. We'll explore some I/O uses of queues in the assignments.

# Deques

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 What is a Deque?](#)
- [2 The deque Interface](#)
- [3 Implementing Deques](#)
  - [3.1 Blocks and BlockMaps](#)
  - [3.2 The deque header](#)
  - [3.3 expandMap and indexAt](#)
  - [3.4 begin\(\) & front\(\)](#)
  - [3.5 indexing](#)
  - [3.6 back\(\) and end\(\)](#)
  - [3.7 Inserting Data into a deque](#)

## 1 What is a Deque?

A *queue* is a sequence of data such that

- data can be added only at the end
- data can be removed only from the front
- data can be examined only at the front

A *DQueue* (double-ended queue, pronounced “dee-cue”) is a sequence of data such that

- data can be added only at the front or at the end
- data can be removed only from the front or the end
- data can be examined only at the front and end

Both the queue and the deque can grow to arbitrary size.

The C++ standard deque (pronounced “deck”) is a generalized deque:

- data can be added only at the front or at the end
- data can be removed only from the front or the end
- data can be examined at *any arbitrary* position

## 2 The deque Interface

The deque interface is identical to that of vector, except for

- Two new operations:
- `push_front`
- `pop_front`

which are guaranteed to be O(1) time.

- The operations
- `capacity()`
- `reserve(size_type)`

are not available.

So a deque is a vector that can grow/shrink at *both* ends.

## 3 Implementing Deques

Key items influencing the implementation: According to the C++ standard, we

- need to add/delete from both ends in O(1) time (amortized)
- need a structure that grows automatically
- need O(1) time access to all elements (operator[])

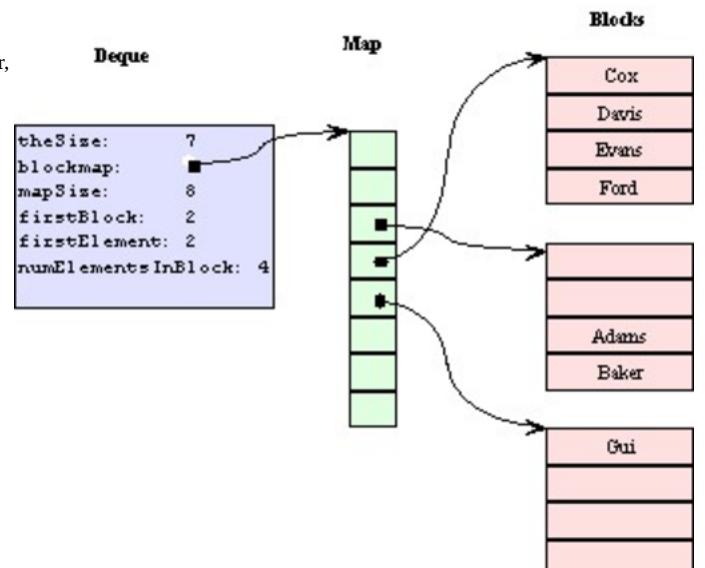
Most implementations of deque are based on a scheme that technically meets the O(1) requirements for the push and pop operations only in an amortized sense, but does guarantee that each push or pop will only do O(1) copies of the actual heap elements.

This implementation is based upon the idea of allocating “blocks” (fixed-sized arrays) of elements and then using a “map” array (an array of pointers to these blocks) to keep track of the blocks.

## 3.1 Blocks and BlockMaps

The diagram shows a typical deque of strings under this scheme. To simplify the presentation, I have placed the strings in the deque in alphabetic order: Adams, Baker, Cox, .... You can clearly see the block and map structure.

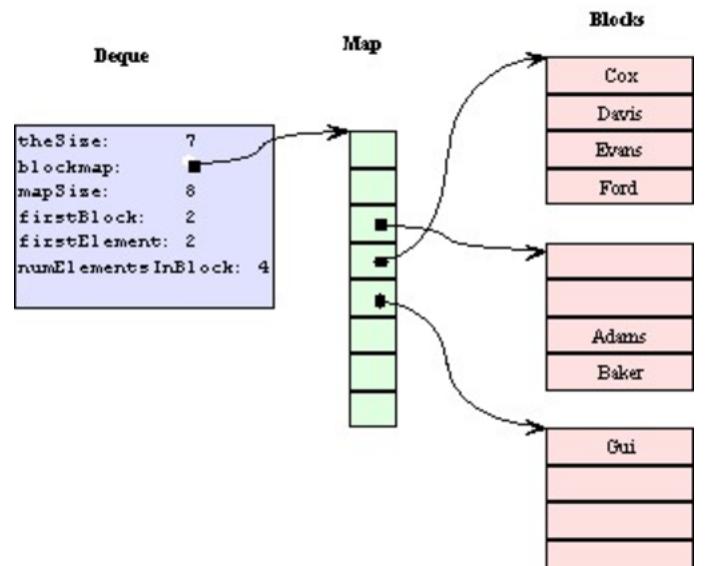
Note that the blocks can occur essentially anywhere on the heap. The order of the blocks within the deque is determined by the map array.



## 3.2 The deque header

The deque itself has several data members.

- `theSize` records the number of elements currently in the deque.
- `blockmap` is a pointer to the map array (on the heap).
- `mapSize` indicates the size of the map array.
- `firstBlock` is the index of the first occupied position in the map array.
- `firstElement` is the index of the first occupied position in the the first block.
- `numElementsInBlock` indicates the size of the blocks.



## 3.3 expandMap and indexAt

Here is the code for the private section of this deque implementation.

```

template <class T>
class deque
{
public:
    ...
private:
    size_type theSize;
    ...
  
```

```

T** blockmap;
size_type mapSize;

size_type firstBlock;
size_type firstElement;

const static size_type BlockSize = 4096;
static size_type numElementsInBlock;

void deque<T>::expandMap () // Double the map size

struct dqPosition {
    size_type blockNum;
    size_type elementNum;
};

dqPosition indexAt (deque<T>::size_type n) const;
// where do we find element # n?
};

```

In addition to the items already described, a few other things are worth taking note of:

- The `expandMap` function will be described later.
- The `indexAt` function will be used to locate elements by number. It returns a structure containing two integers, one being the index of the desired block within the map, the other the index of the desired element within that block.

Both of these functions are “utilities” intended to simplify the implementation of the “real” deque member functions. They are private because they are only supposed to be called by the deque member functions, not by application code that uses the deque.

We can now start to describe how we actually accomplish some of the typical deque operations.

### 3.4 begin() & front()

For example, `begin()` and `front()` are pretty straightforward. To find the first element in the deque, we simply look in `blockmap[firstElement]`. That points us to the block containing the first element. Then within that block, we look at the `firstElement` position.

Here is the code for `front`. As is often the case, this operation comes in a pair: one version for `const` deques and one for non-`const` deques. The non-`const` version returns a reference to the front element, allowing applications to change it:

```
myNonConstDeque.front() = someValue;
```

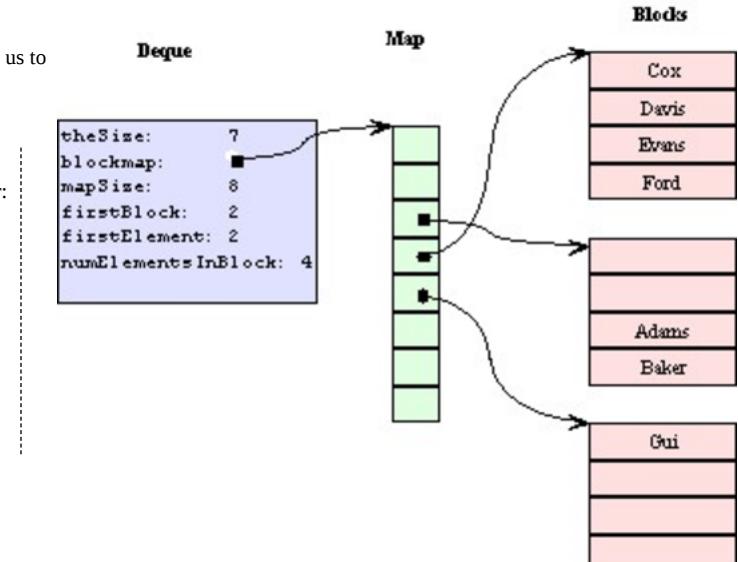
The `const` version returns a `const` reference, and so changes to the front element of a `const` deque would result in a compilation error.

```

template <class T>
T& deque<T>::front ()
{
    return blockmap[firstBlock][firstElement];
}

template <class T>
const T& deque<T>::front () const
{
    return blockmap[firstBlock][firstElement];
}

```



`begin` is similar in theory, but slightly more complicated because we need to use that same reasoning to construct an iterator. We haven't discussed the structure of the deque iterators yet, and I don't intend to do so, because it's not as instructive as the deque itself.

### 3.5 indexing

Next, let's consider the indexing operation, e.g., `myDeque[i]`, which is provided by operator`[ ]`.

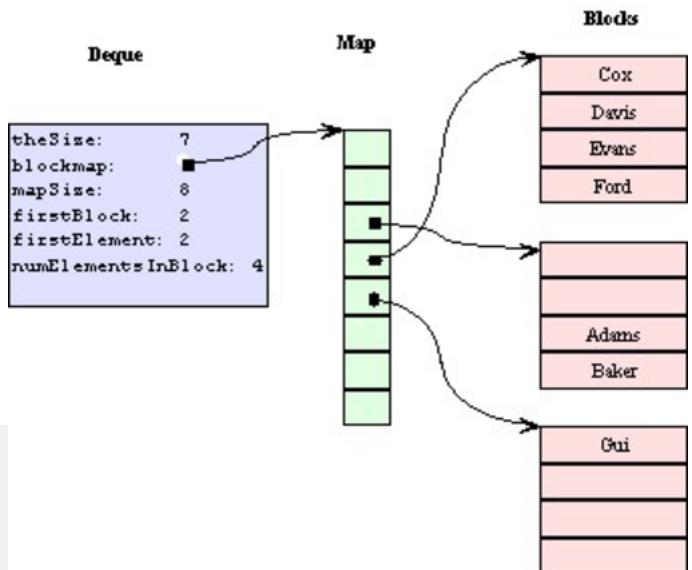
We can split the problem of finding the  $i^{\text{th}}$  element into cases:

- If  $i$  is less than `numElementsInBlock - firstElement`, then the element we are looking for is in the first block.
- If it's not in the first block, then we can find out how many map positions past `firstBlock` by dividing  $i - (\text{numElementsInBlock} - \text{firstElement})$  by `numElementsInBlock`.

Use the resulting quotient to index into the map and find the desired block. Then use the remainder of that division as the index within the block.

This calculation is one that we need to do quite often in this ADT implementation, so it's worthwhile breaking that out into a separate utility function.

```
template <class T>
deque<T>::dqPosition
    deque<T>::indexAt (deque<T>::size_type n) const
{
    dqPosition pos;
    pos.blockNum = firstBlock;
    if (n < numElementsInBlock - firstElement)
    {
        pos.elementNum = n + firstElement;
    }
    else
    {
        n -= numElementsInBlock - firstElement;
        ++pos.blockNum;
        int k = n / numElementsInBlock;
        pos.blockNum += k;
        pos.elementNum = n - k * numElementsInBlock;
    }
    return pos;
}
```



Here you see that utility, `indexAt`. It carries out the calculation we have just described, and stores the result into a simple structure with two fields, `blockNum` and `elementNum`.

And, once we have that utility function, the implementation of the indexing operators becomes trivial.

```
template <class T>
T& deque<T>::operator [ ] (deque<T>::size_type n)
{
    dqPosition<T> pos = indexAt(n);
    return blockmap[pos.blockNum][pos.elementNum];
}

template <class T>
const T& deque<T>::operator [ ] (deque<T>::size_type n) const
{
    dqPosition<T> pos = indexAt(n);
    return blockmap[pos.blockNum][pos.elementNum];
}
```

## 3.6 back() and end()

The same utility makes for a simple implementation of the `back()` function and would also help greatly in implementing `end()`.

```
template <class T>
T& deque<T>::back ()
{
    dqPosition pos = indexAt(theSize-1);
    return blockmap[pos.blockNum][pos.elementNum];
}

template <class T>
const T& deque<T>::back () const
{
    dqPosition pos = indexAt(theSize-1);
    return blockmap[pos.blockNum][pos.elementNum];
}
```

It's worth pointing out that, while there is a fair amount of calculation involved in getting to the  $i^{\text{th}}$  element in a deque, it's all  $O(1)$ . This is rather typical in comparing deques to arrays and vectors. Deques tend to have more overhead, but it's in the constant-multipliers. In a big-O sense, deques are always no slower than arrays and vectors, and may be faster for some operations (e.g., inserting at the front of the deque).

In addition, there are practical benefits to having at least one sequential container that does not keep all of its data in one huge contiguous area. Some operating systems place rather stringent limits on the largest amount of contiguous space you can allocate on the heap. For example, Windows 3.1 would not allow you to allocate more than 64K in a single block of memory. That may sound like a lot, but suppose you had an array or vector of `PersonnelRecords`, and each `PersonnelRecord` occupied 1024 bytes (which is not at all difficult to achieve). Then an array or vector could hold no more than 64 elements, total. dequees, on the other hand, because they break their storage into blocks, could grow to quite large size (up to the total limit of memory) as long as each individual block contained no more than 64 elements.

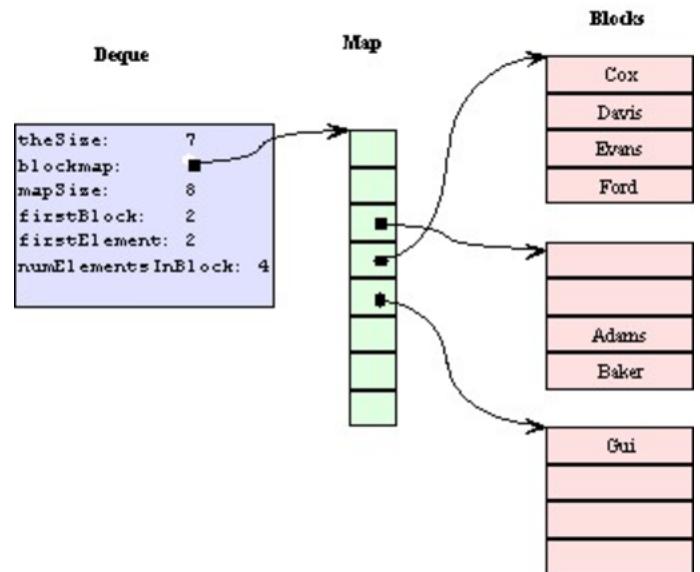
## 3.7 Inserting Data into a deque

That brings us to the question of just how the data gets inserted into the deque in the first place.

Looking at the structure shown here, you can see that pushing a new element onto either the front or the back would be no problem for this particular deque. We have room in both the first and the last block for additional elements.

What if we want to push something onto the end/front of the deque and the last/first block is full? Then we would need to allocate space for a new block, putting the pointer to the new block in the map just after/before the currently occupied slots, and placing the new element at the beginning/end of the newly created block.

Which leaves us with only one problem: what if the `map` is full, and there's no room in the map for any more pointers?



### 3.7.1 Expanding the map

In that case we need to create a new, larger map, as shown here.

```
template <class T>
void deque<T>::expandMap ()
// double the size of the block map, copying the
// current set of pointers into the middle of the
// new map array
{
    T** newMap = new T*[2*mapSize];
    fill_n(newMap, 2*mapSize, (T*)0);
    copy (blockmap+firstBlock, blockmap+mapSize,
          newMap+mapSize/2);
    delete [] blockmap;
    blockmap = newMap;
    firstBlock = mapSize/2;
    mapSize *= 2;
}
```

Note that the fill and copy operations are  $O(\text{mapSize})$ , and `mapSize` could be as large as `theSize/numElementsInBlock`, so this means that a `push_back` or `push_front` on a deque with  $N$  elements is really  $O(N)$ .

This worst case cost does amortize to  $O(1)$  if we start with an empty deque and push  $N$  successive elements into it. Furthermore, and this may be important if we are dealing with elements of a large or complicated data type, `expandMap` only copies pointers, not actual elements. The number of elements copied by a push call remains  $O(1)$ .

The remainder of the details for implementing the push operations is left as an exercise for the reader.

# Average Case Analysis

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Definition](#)
- [2 Probably Not a Problem](#)
- [3 What's an Average?](#)
  - [3.1 The Mean Average](#)
  - [3.2 Expected Value](#)
- [4 Determining the Average Case Complexity](#)
  - [4.1 It Still All Boils Down to Addition](#)
  - [4.2 All of Your Variables Must be Defined](#)
  - [4.3 Complexity is Written in Terms of the Inputs](#)
  - [4.4 The Complexity of Any Block of Code Must be Numeric](#)
  - [4.5 Surprises Demand Explanation](#)
- [5 The Complexity of Expression Evaluation](#)
- [6 The Complexity of Compound Statements](#)
  - [6.1 Sequences of Statements](#)
  - [6.2 Conditional statements](#)
  - [6.3 Loops](#)
- [7 Extended Example: Ordered Insertion, Different Input Distributions](#)
  - [7.1 Input in Sorted Order](#)
  - [7.2 Input in Arbitrary Order:](#)
  - [7.3 Inputs in Almost-Sorted Order](#)
  - [7.4 Almost Sorted - version 2](#)
- [8 The Input Distribution is Key](#)

In the first section of this course, we looked at the process of analyzing the worst-case running time of an algorithm, and the use of worst-case analysis and big-O notation as a way of describing it.

In this section, we will introduce average-case complexity. Just as the worst-case complexity describes an upper bound on the worst-case time we would see when running an algorithm, average case complexity will present an upper bound on the average time we would see when running the program many times on many different inputs.

It is true that worst-case complexity gets used more often than average-case. There are a number of reasons for this.

- The worst-case complexity is often easier to compute than the average case. Just figuring out what an “average” set of inputs will look like is often a challenge. To figure out the worst case complexity, we only need to identify that one single input that results in the slowest running.
- In many cases, it will turn out the worst and average complexity will turn out to be the same.
- Finally, reporting the worst case to your boss or your customers is often “safer” than reporting the average. If you give them the average, then sometimes they will run the program and see slower performance than they had expected. Human nature being what it is, they will probably get rather annoyed. On the other hand, if you go to those same customers with the worst-case figure, most of the time they will observe faster-than-expected behavior, and will be more pleased.

This appears to be particularly true of interactive programs. When people are actually sitting there typing things in or clicking with the mouse and then waiting for a response, if they have to sit for a long time waiting for a response, they’re going to remember that. Even if 99.9% of the time they get instant response (so the average response is still quite good), they will characterize your program as “sluggish”.

In those circumstances, it makes sense to focus on worst-case behavior and to do what we can to improve that worst case.

On the other hand, suppose we’re talking about a batch program that will process thousands of inputs per run, or we’re talking about a critical piece of an interactive program that gets run hundreds or thousands of times in between *each* response from the user. In that situation, adding up hundreds or thousands of worst-cases may be just too pessimistic. In that circumstance, the cumulative time of thousands of different runs should show some averaging out of the worst-case behavior, and an average case analysis may give a more realistic picture of what the user will be seeing.

## 1 Definition

### Definition: Average Case Complexity

We say that an algorithm requires *average time proportional to f(n)* (or that it has average-case complexity  $O(f(N))$ ) if there are constants  $c$  and  $n_0$  such that the average time the algorithm requires to process an input set of size  $n$  is no more than  $c * f(n)$  time units whenever  $n \geq n_0$ .

This definition is very similar to the one for worst case complexity. The difference is that for worst-case complexity, we want  $T_{\max}(n) \leq c * f(n)$  where  $T_{\max}(n)$  is the *maximum* time taken by any input of size  $n$ , but for average case complexity we want  $T_{\text{avg}}(n) \leq c * f(n)$  where  $T_{\text{avg}}(n)$  is the *average* time required by inputs of size  $n$ .

The average case complexity describes how quickly the average time increases when  $n$  increases, just as the worst case complexity describes how quickly the worst case time increases when  $n$  increases.

In both forms of complexity, we are looking for upper bounds, so our big-O notation (and its peculiar algebra) will still apply.

**Question:** Suppose we have an algorithm with worst case complexity  $O(n)$ .

True or false: It is possible for that algorithm to have average case complexity  $O(n^2)$ .

Answer +

**False** (sort of): If it were true, the average case time would be increasing faster than the worst case time. Eventually, that means that the average case time for some large  $n$  would be larger than the worst case among all input sets of size  $n$ . But an average can never be larger than the maximum value in the set being averaged.

So the only way this statement could be true is in the trivial sense that any algorithm in  $O(n)$  is also in  $O(n^2)$ . (Remember, the big-O expressions describe sets of functions. The set of all functions that can be bounded by a linear function is a subset of the set of all functions that can be bounded by a quadratic:  $O(n) \subset O(n^2)$ .)

This is an important idea to keep in mind as we discuss the rules for average case analysis. The worst-case analysis rules all apply, because they do provide an upper bound. But that bound is sometimes not as tight as it could be. One of the things that makes average case analysis tricky is recognizing when we can get by with the worst-case rules and when we can gain by using a more elaborate average-case rule. There's no hard-and-fast way to tell — it requires the same kind of personal judgment that goes on in most mathematical proofs.

## 2 Probably Not a Problem

We're going to need a few basic facts about probabilities for this section.

- Every probability is between 0 and 1, inclusive.
  - For example, the probability of a flipped coin coming up heads is 0.5.
  - The probability of an ordinary six-sided die rolling a '3' is  $1/6$ .
    - The probability of that same die rolling an even number is 0.5.
- The sum of the probabilities of all possible events must be 1.0.
  - For example, the probability of an ordinary six-sided die rolling some number between 1 and 6 is 1.0.
- If  $p$  is the probability that an event will happen, then  $(1 - p)$  is the probability that the event will not happen.
  - The probability of a 6-sided die rolling any number other than '3' is  $1 - 1/6 = 5/6$ .
- If two events with probabilities  $p_1$  and  $p_2$  are *independent* (the success or failure of the first event does not affect the success or failure of the other), then
  1. the probability that both events will occur is  $p_1 * p_2$ .
  2. the probability that neither will occur is  $(1 - p_1)(1 - p_2)$ .
  3. the probability that at least one will occur is  $1 - (1 - p_1)(1 - p_2)$ .

For example, if I roll two 6-sided dice, the probability that both will roll a '3' is  $\frac{1}{6} * \frac{1}{6} = \frac{1}{36}$ .

If I roll two six-sided dice, the probability that at least one will come up '3' is

$$1 - \left(1 - \frac{1}{6}\right) \left(1 - \frac{1}{6}\right) = 1 - \left(\frac{5}{6}\right) \left(\frac{5}{6}\right) = 1 - \frac{25}{36} = \frac{11}{36}$$

- More generally, if I have a series of *independent* events with probabilities  $p_1, p_2, \dots, p_k$ , then

1. the probability that all events will occur is  $\prod_{i=1}^k p_i$ .
2. the probability that neither will occur is  $\prod_{i=1}^k (1 - p_i)$ .
3. the probability that at least one will occur is  $1 - \prod_{i=1}^k (1 - p_i)$ .

If I flip a coin 3 times, the probability that I will see the sequence heads-tails-heads is  $\frac{1}{2} * \frac{1}{2} * \frac{1}{2} = \frac{1}{8}$

- Note that the probability of seeing the sequence heads-heads-heads is exactly the same.

If I flip a coin 3 times, the probability that I will see at least one heads is heads-tails-heads is  $\frac{1}{2} * \frac{1}{2} * \frac{1}{2} = \frac{1}{8}$

- If we make a number of independent attempts at something, and the chance of seeing an event on any single attempt is  $p$ , on average we will need  $1/p$  attempts before seeing that event.
  - For example, if I start rolling a six-sided die until I see a '2', on average I would wait  $\frac{1}{1/6} = 6$  rolls.

This is a great illustration of the difference between average and worst-case times, because in the worst case you would keep rolling forever!

You can find a fuller tutorial [here](#).

## 3 What's an Average?

For some people, average case analysis is difficult because they don't have a very flexible idea of what an "average" is.

**Example:**

Last semester, Professor Cord gave out the following grades in his CS361 class:

A, A, A-, B+, B, B, B-, C+, C, C-, D, F, F, F

Translating these to their numerical equivalent,

4, 4, 3.7, 3.3, 3, 3, 3, 2.7, 2.3, 2, 1.7, 1, 0, 0, 0, 0

what was the average grade in Cord's class?

According to some classic forms of average:

#### Median

the middle value of the sorted list, (the midpoint between the two middle values given an even number of items)

$$\text{avg median} = 2.5$$

#### Mode

the most commonly occurring value

$$\text{avg modal} = 0$$

#### Mean

Computed from the sum of the elements

$$\begin{aligned}\text{avg mean} &= (4 + 4 + 3.7 + 3.3 + 3 + 3 + 3 + 2.7 + 2.3 + 2 + 1.7 + 1 + 0 + 0 + 0 + 0) / 16 \\ &= 2.11\end{aligned}$$

## 3.1 The Mean Average

The mean average is the most commonly used, and corresponds to most people's idea of a "normal" average, but even that comes in many varieties:

#### Simple mean

$$\bar{x} = \frac{\sum_{i=1}^N x_i}{N}$$

#### Weighted mean

$$\bar{x} = \frac{\sum_{i=1}^N w_i * x_i}{\sum_{i=1}^N w_i}$$

The  $w_i$  are the weights that adjust the relative importance of the scores.

#### Example:

Last semester Professor Cord gave the following grades

Grade	# students
4.0	2
3.7	1
3.3	1
3.0	3
2.7	1
2.3	1
2.0	1
1.7	1
1.3	0
1.0	1
0.0	4

$$\text{The weighted average is } \frac{2*4.0+1*3.7+1*3.3+3*3.0+1*2.7+1*2.3+1*2.0+1*1.7+0*1.3+1*1.0+4*0.0}{2+1+1+3+1+1+1+0+1+4} = 2.11$$

Another example of weighted averages:

When one student asked about his overall grade for the semester, Professor Cord pointed out that assignments were worth 50% of the grade, the final exam was worth 30%, and the midterm exam worth 20%. The student has a B, A, and C-, respectively on these.

Category	Score	Weight
Assignments	3.0	50
Final	4.0	30
Midterm	1.7	20

So the student's average grade was

$$\frac{50 * 3.0 + 30 * 4.0 + 20 * 1.7}{50 + 30 + 20} = 3.04$$

## 3.2 Expected Value

The expected value is a special version of the weighted mean in which the weights are the probability of seeing each particular value.

If  $x_1, x_2, \dots$ , are all the possible values of some quantity, and these values occur with probability  $p_1, p_2, \dots$ , then the expected value of that quantity is

$$E(x) = \sum_{i=1}^N p_i * x_i$$

Note that if we have listed all possible values, then

$$\sum_{i=1}^N p_i = 1$$

so you can regard the  $E(x)$  formula above as a special case of the weighted average in which the denominator (the sum of the weights) becomes simply “1”.

### Example:

After long observation, we have determined that Professor Cord tends to give grades with the following distribution:

Grade	probability
4.0	2/16
3.7	1/16
3.3	1/16
3.0	3/16
2.7	1/16
2.3	1/16
2.0	1/16
1.7	1/16
1.3	0/16
1.0	1/16
0.0	4/16

So the expected value of the grade for an average student in his class is

$$\begin{aligned} & ((2/16) * 4.0 + (1/16) * 3.7 + (1/16) * 3.3 + (3/16) * 3.0 \\ & + (1/16) * 2.7 + (1/16) * 2.3 + (1/16) * 2.0 + (1/16) * 1.7 + (0/16) * 1.3 \\ & + (1/16) * 1.0 + (4/16) * 0.0) \\ & = 2.11 \end{aligned}$$

The expected value is the kind of average we will use throughout this course in discussing average case complexity.

## 4 Determining the Average Case Complexity

In many ways, determining average case complexity is similar to determining the worst-case complexity.

### 4.1 It Still All Boils Down to Addition

If you want to know how much time a complicated process takes, you figure that out by *adding up* the times of its various components.

That basic observations is the same for average times as it is for worst-case times.

When in doubt, just add things up.

- Just keep in mind that you want to add up their average times instead of their worst-case times.

### 4.2 All of Your Variables Must be Defined

In math, as in programming, all of Your variables must be declared/defined before you can use them.

### 4.3 Complexity is Written in Terms of the Inputs

The complexity of a block of code must be a function of the inputs (only!) to that block.

## 4.4 The Complexity of Any Block of Code Must be Numeric

No reason for this to change.

## 4.5 Surprises Demand Explanation

The vast majority of algorithms we will look at will have the same average-case complexity as worst-case. So, if you come up with a different value for the average-case, make sure that you understand why.

By the same token, though, if you make it to the end of an average-case analysis and never once took the time to even consider how the “average input” is different from the “worst-case input”, you may need to start over.

# 5 The Complexity of Expression Evaluation

Very little changes for evaluating the cost of C++ expressions:

- Arithmetic & Relational Operations on Primitive Types are O(1)

Still true.

- Assignments of Primitive Types are O(1)

Still true.

- Basic Address Calculations are O(1)

Still true.

- A function calls has the average-case complexity of the function body, with appropriate substitutions of the actual parameters for the formal parameters.

No change from worst-case analysis, except that we must have done an average-case analysis on the function body instead of a worst-case analysis.

With so much the same between average and worst-case analysis, where do the differences come in? The differences come in the treatment of compound statements, especially loops.

# 6 The Complexity of Compound Statements

As in worst-case analysis, we will usually analyze compound statements in an inside-out fashion, starting with the most deeply nested components and working our way outward from there.

We will annotate our code in the same way as before to record our analysis.

## 6.1 Sequences of Statements

The simplest form of compound statement is the sequence or block, usually written in C++ between { } brackets. Examples include function bodies, loop bodies, and the “then” and “else” parts of if statements.

A sequence tells the machine to process statements one after the other. So,

The time for a sequence of statements is the sum of the times of the individual statements.

More formally, for a sequence of statements  $s_1, s_2, \dots, s_k$

$$E(t_{\text{seq}}) = \sum_{i=1}^k E(t_{s_i})$$

When doing worst-case analysis, we added up the worst-case times.

When doing average-case analysis, we add up the average-case (expected value) of the times instead.

## 6.2 Conditional statements

When we have an if statement, we know that we will execute either the then part *or* the else part, but not both. However, when doing an average-case analysis we generally do not know which part we will take. Instead, we have to consider the probability of taking each part.

Let  $p$  be the probability that the if condition is true (i.e., that we will take the “then” part).

$$E(t_{\text{if}}) = E(t_{\text{condition}}) + (p * E(t_{\text{then}}) + (1 - p) * E(t_{\text{else}}))$$

The expression  $(p * E(t_{\text{then}}) + (1 - p) * E(t_{\text{else}}))$  is simply the expected value (average) of the time to take the “then” and “else” parts.

## 6.3 Loops

When we analyze a loop, we need to add up the expected time required for all of its iterations.

The complicating factor here is that, in worst case analysis, we can almost always say “in the worst case, for an input of size  $N$ , this loop will repeat  $k$  times”, and then sum up the times over those  $k$  iterations.

When we are doing an average case analysis, we often have to consider the possibility that the number of iterations may vary, even among inputs of the same size  $N$ . Each possible number of iterations will have a certain probability, and we would need to get the expected value of the time over all of those possible numbers of iterations

In general,

- Let  $t_L(k)$  denote the average time that a loop  $L$  requires to run on those inputs that cause the loop to repeat exactly  $k$  times before exiting.
- $t_L$  can be computed using the [same rules](#) as used for  $t_{\text{while}}$  and  $t_{\text{for}}$  in worst-case analysis, so long as we add up average times instead of worst-case times.
- Let  $p_k$  denote the probability of an arbitrary input causing the loop to repeat exactly  $k$  times before exiting.

Then the average time  $t_{\text{loop}}$  time expected for that loop is

$$t_{\text{loop}} = \sum_{k=0}^{\infty} p_k t_L(k)$$

- The  $\infty$  is not a mistake. It expresses the general case that some loops could execute an arbitrary number of times depending upon the input.

### 6.3.1 Special case: constant or linear loops

Luckily, we often will not need to evaluate this formula in all of its generality.

For example, if  $t_L(k)$  is  $O(1)$  or  $O(k)$  we can instead say

$$t_{\text{loop}} = t_L(E(k))$$

i.e., we simply need to find the average number of iterations and evaluate the loop time on that number.

#### Example 1: Simulating a Rolling Die

The function [rand\(\)](#) returns a pseudo-random non-negative integer each time it is called. [rand\(\)](#) runs in  $O(1)$  time (worst-case and average-case).

We can simulate the roll of a six-sided die by taking  $(\text{rand()} \% 6) + 1$ . The  $\% 6$  takes the remainder of the random number when divided by 6, which we can regard as a uniform random selection in the range 0..5. “Uniform” means here that all six possible values are equally likely on any given call to [rand\(\)](#). Then adding 1 to that gives us a uniform random selection in the range 1..6.

Consider the following code:

```
int roll = (rand() % 6) + 1;
while (roll != 2)
{
    roll = (rand() % 6) + 1;
}
```

What is the worst-case complexity of this code? +

All of the operations in the first line are  $O(1)$

```
int roll = (rand() % 6) + 1; // O(1)
while (roll != 2)
{
    roll = (rand() % 6) + 1;
}
```

All of the operations in the loop body are  $O(1)$ .

```
int roll = (rand() % 6) + 1; // O(1)
while (roll != 2)
{
    roll = (rand() % 6) + 1; // O(1)
}
```

The condition of the while loop is  $O(1)$ :

```
int roll = (rand() % 6) + 1; // O(1)
while (roll != 2)           // cond: O(1)
{
    roll = (rand() % 6) + 1; // O(1)
}
```

Things get tricky when we ask the question “How many times does this loop repeat?”

How many times, in the worst case, can we roll a die until a ‘2’ comes up? In the worst case, there is no limit to the number of times we might roll.

```
int roll = (rand() % 6) + 1; // O(1)
while (roll != 2)           // cond: O(1) #: infinity
{
    roll = (rand() % 6) + 1; // O(1)
}
```

And we conclude that the loop, and the entire block of code, is  $O(\infty)$ .

*What is the average-case complexity of this code?* +

All of the operations in the first line are  $O(1)$

```
int roll = (rand() % 6) + 1; // O(1)
while (roll != 2)
{
    roll = (rand() % 6) + 1;
}
```

All of the operations in the loop body are  $O(1)$ .

```
int roll = (rand() % 6) + 1; // O(1)
while (roll != 2)
{
    roll = (rand() % 6) + 1; // O(1)
}
```

The condition of the while loop is  $O(1)$ :

```
int roll = (rand() % 6) + 1; // O(1)
while (roll != 2)           // cond: O(1)
{
    roll = (rand() % 6) + 1; // O(1)
}
```

How many times does this loop repeat? We don't know, but can compute an expected value. How many times, on average, would we roll a die until a '2' comes up?

- On any given roll, the probability of seeing a '2' is  $1/6$ .
- And if we are looking at a string of independent events with probability  $p$ , on average we wait  $1/p$  times to see that event.

So the average number of times we would repeat this loop is  $\frac{1}{1/6}$  or 6 times.

```
int roll = (rand() % 6) + 1; // O(1)
while (roll != 2)           // cond: O(1) #: 6
{
    roll = (rand() % 6) + 1; // O(1)
}
```

Each iteration of the loop is  $O(1)$  time for both the condition and the body, so we conclude that the loop average  $6 * O(1)$  time, which simplifies to  $O(1)$

```
int roll = (rand() % 6) + 1; // O(1)
while (roll != 2)           // cond: O(1) #: 6 total: O(1)
{
    roll = (rand() % 6) + 1; // O(1)
}
```

And the entire block of code has an average-case complexity of  $O(1)$ .

### 6.3.2 Special case: a fixed number of iterations

Another common simplification occurs when we *can* determine the number of iterations for all inputs of size  $N$ . If we know that the loop executes  $h(N)$  times for all inputs of size  $N$ , then the probabilities  $p_k$  of executing the loop  $k$  times is

$$p_k = \begin{cases} 1.0 & \text{if } k = h(N) \\ 0.0 & \text{if } k \neq h(N) \end{cases}$$

so that all of the terms in

$$t_{\text{loop}} = \sum_{k=0}^{\infty} p_k t_L(k)$$

are zero except the  $k = h(N)$  term:

$$t_{\text{loop}} = t_L(h(N))$$

#### Example 2: Filling an Array

Consider the code:

```
void fill_n (int* array, int n, int value)
{
```

```
for (int i = 0; i < n; ++i)
    array[i] = value;
}
```

where the values in the array are known to be in the range 0..10, with half of them being zero.

*What is the average case complexity of this code?* +

The body is O(1):

```
void fill_n (int* array, int n, int value)
{
    for (int i = 0; i < n; ++i)
        array[i] = value;           // O(1)
}
```

The loop condition is O(1), and the loop repeats  $n$  times.

```
void fill_n (int* array, int n, int value)
{
    for (int i = 0; i < n; ++i) // cont: O(1) #: n
        array[i] = value;      // O(1)
}
```

Now, because the number of loop iterations depends only on the size of the input ( $n$ ) and not upon the variations in input within a given size, we can say that

$$t_{\text{loop}} = t_L(n)$$

and evaluate  $t_L$  according to our conventional rules:

```
void fill_n (int* array, int n, int value)
{
    for (int i = 0; i < n; ++i) // cont: O(1) #: n total: O(n)
        array[i] = value;      // O(1)
}
```

and the entire function has average case complexity  $O(n)$  – the same as it's worst-case complexity. That should not be a surprise - this function behaves exactly (time-wise) the same no matter values we supply in the array or in value.

## 7 Extended Example: Ordered Insertion, Different Input Distributions

We'll illustrate the process of doing average-case analysis by looking at a simple but useful algorithm, exploring how changes in the [input distribution](#) (the probabilities of seeing various possible inputs) affect the average case behavior.

Here is our [ordered insertion algorithm](#).

```
template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop, const Comparable& value)
{
    Iterator preStop = stop;
    --preStop;
    while (stop != start && value < *preStop) {
        *stop = *preStop;
        --stop;
        --preStop;
    }
    // Insert the new value
    *stop = value;
    return stop;
}
```

We will, as always, assume that the basic iterator operations are  $O(1)$ . For the sake of this example, we will also assume that the operations on the Comparable type are  $O(1)$ . (We'll discuss the practical implications of this at the end.)

We start, as usual, by marking the simple bits O(1).

```
template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop, const Comparable& value)
{
    Iterator preStop = stop;           // O(1)
    --preStop;                        // O(1)
    while (stop != start && value < *preStop) {
        *stop = *preStop;             // O(1)
        --stop;                      // O(1)
        --preStop;                   // O(1)
    }
    // Insert the new value
    *stop = value;                  // O(1)
    return stop;                    // O(1)
}
```

Next we note that the loop body can be collapsed to O(1).

```

template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop, const Comparable& value)
{
    Iterator preStop = stop;      // O(1)
    --preStop;                  // O(1)
    while (stop != start && value < *preStop) {
        // O(1)
    }
    // Insert the new value
    *stop = value;              // O(1)
    return stop;                // O(1)
}

```

The loop condition is O(1):

```

template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop, const Comparable& value)
{
    Iterator preStop = stop;      // O(1)
    --preStop;                  // O(1)
    while (stop != start && value < *preStop) { //cond: O(1)
        // O(1)
    }
    // Insert the new value
    *stop = value;              // O(1)
    return stop;                // O(1)
}

```

Because the loop condition and body are  $O(1)$ , we can use the shortcut of simply analyzing this loop on the expected number of iterations.

That, however, depends on the values already in the container, and how the new value compares to them.

The loop might execute

- 0 times (if value is larger than anything already in the container),
- 1 time (if value is larger than all but one element already in the container),
- and so on up to a maximum of  $\text{distance}(\text{start}, \text{stop})$  times (if value is smaller than everything already in the container).

What we *don't* know are the probabilities to associate with these different numbers of iterations.

- These depend upon the way the successive inputs to this function are distributed.

Consider using this algorithm as part of a spell checking program. We can envision two very different input patterns:

- If we are reading words from the spell check dictionary into an array for later search purposes, those words are most likely already sorted. There may be a few exceptions, e.g., words that the user has added to their personal dictionary. For example, I often find it useful to add my last name to my spellcheck dictionaries.
- If we are building a concordance, a list of words actually appearing in the document, then we would receive the words in whatever order they appear in the document text, essentially a random order.

Let's analyze each of these cases in turn and see how they might differ in performance.

## 7.1 Input in Sorted Order

If the input arrives in sorted order, then each call to the function will execute the loop zero times, because the word being inserted will always be alphabetically greater than all the words already in the container.

So if  $p_k$  denotes the probability of executing the loop  $k$  times, then  $p_0 = 1, p_1 = 0, p_2 = 0, \dots$ .

So the time is

$$t_{\text{loop}} = t_L(0) \\ = O(1)$$

For this input pattern, the entire algorithm has an average-case complexity of  $O(1)$ .

```

template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop,
                const Comparable& value)
{
    Iterator preStop = stop;
    --preStop;
    while (stop != start && value < *preStop) {
        *stop = *preStop;
        --stop;
        --preStop;
    }
    // Insert the new value
    *stop = value;
    return stop;
}

```

## 7.2 Input in Arbitrary Order:

In this case, we are equally likely to need 0 iterations, 1 iteration, 2 iterations, ...,  $n$  iterations, where  $n$  is  $\text{distance}(\text{start}, \text{stop})$ . So the possible numbers of iterations from 0 to  $n$  are all equally likely:

$$p_i = \begin{cases} \frac{1}{n+1} & \text{if } 0 \leq k \leq n \\ 0 & \text{otherwise} \end{cases}$$

The cost of the loop condition and of the body is constant for each iteration, however, so we can use the special case

$$t_{\text{loop}} = t_L(E(k))$$

where  $E(k)$  is the expected number of iterations of the loop.

What is  $E(k)$ ?

Intuitively, if we are equally likely to repeat the loop 0 times, 1 time, 2 times, ...,  $n$  times, the average number of iterations would seem to be  $n/2$ .

Formally,

$$\begin{aligned} E(k) &= \sum_{k=0}^{\infty} p_k k \\ &= \sum_{k=0}^n p_k k \quad (\text{because } p_k = 0 \text{ when } k > n) \\ &= \sum_{k=0}^n \frac{1}{n+1} k \\ &= \frac{1}{n+1} \sum_{k=0}^n k \\ &= \frac{1}{n+1} \frac{n(n+1)}{2} \\ &= \frac{n}{2} \end{aligned}$$

Chalk one up for intuition!

So the loop is  $\frac{n}{2}O(1) = O(n)$

```
template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop, const Comparable& value)
{
    Iterator preStop = stop;      // O(1)
    --preStop;                  // O(1)
    while (stop != start && value < *preStop) { // cond: O(1) #: n/2 total: O(n)
        // O(1)
    }
    // Insert the new value
    *stop = value;              // O(1)
    return stop;                // O(1)
}
```

And we can then replace the entire loop by  $O(n)$ .

```
template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop, const Comparable& value)
{
    Iterator preStop = stop;      // O(1)
    --preStop;                  // O(1)
    // O(n)
    // Insert the new value
    *stop = value;              // O(1)
    return stop;                // O(1)
}
```

And now, we add up the complexities in the remaining straight-line sequence, and conclude that the entire algorithm has an average case complexity of  $O(n)$ , where  $n$  is the distance from start to stop, when presented with randomly arranged inputs.

This is the same result we had for the worst case analysis. Does this mean that it runs in the same time on average as it does in the worst case? No, on average, it runs in half the time of its worst case, but that's only a constant multiplier, so it disappears when we simplify.

Under similar randomly arranged inputs, the average case complexity of ordered search is  $O(n)$  and the average case complexity of binary search is  $O(\log n)$ . Again, these are the same as their worst-case complexities.

## 7.3 Inputs in Almost-Sorted Order

We've already considered the case where the inputs to this function were already arranged into ascending order. What would happen if the inputs were almost, but not exactly, already sorted into ascending order?

```
template <typename Iterator, typename Comparable>
int addInOrder (Iterator start, Iterator stop,
                const Comparable& value)
{
    Iterator preStop = stop;
    --preStop;
    while (stop != start && value < *preStop) {
        *stop = *preStop;
        --stop;
        --preStop;
    }
    // Insert the new value
    *stop = value;
    return stop;
}
```

For example, suppose that, on average, one out of  $n$  items is out of order. Then the probability of a given input repeating the loop zero times would be  $p_0 = \frac{n-1}{n}$ , and some single  $p_i$  would have probability  $1/n$ , with all the other probabilities being zero.

Assuming the worst (because we want to find an upper bound), let's assume that the one out-of-order element is the very last one added, and that it actually gets inserted into position 0. Then we have  $p_0 = (n-1)/2, p_1 = 0, p_2 = 0, \dots, p_{n-1} = 0, p_n = 1/n$

So the average number of iterations would be given by

$$\begin{aligned} E(k) &= \sum_{k=0}^n kp_k \\ &= 0 * n - 1/n + n * 1/n \\ &= n/n \\ &= 1 \end{aligned}$$

and the function is  $O(E(k)) = O(1)$

## 7.4 Almost Sorted - version 2

Now, that's only one possible scenario in which the inputs are almost sorted. Let's look at another. Suppose that we knew that, for each successive input, the probability of it appearing in the input  $m$  steps out of its correct position is proportional to  $1/(m+1)$  (i.e., each additional step out of its correct position is progressively more unlikely). Then we have  $p_0 = c, p_1 = c/2, p_2 = c/3, \dots, p_{n-1} = c/n, p_n = c/(n+1)$ .

The constant  $c$  is necessary because the sum of all the probabilities must be exactly 1. We can compute the value of  $c$  by using that fact:

$$\begin{aligned} \sum_{i=0}^n p_i &= 1 \\ \sum_{i=0}^n \frac{c}{i+1} &= 1 \\ c \sum_{i=0}^n \frac{1}{i+1} &= 1 \end{aligned}$$

This sum, for reasonably large  $n$ , is approximately  $\log n$ .

So we conclude that  $c$  is approximately  $1/\log(n)$ .

So the function, for this input distribution, is

$$\begin{aligned} \text{\begin{array}{l} t_{\text{\{loop\}}} = O(E(k)) \\ = O\left(\sum_{i=0}^n (i+1)p_i\right) \\ = O\left(\sum_{i=0}^n (i+1)\frac{c}{i+1}\right) \\ = O\left(\sum_{i=0}^n c\right) \end{array}} \end{aligned}$$

So the average case is slightly smaller than the worst case, though not by much (remember that  $\log n$  is nearly constant over large ranges of  $n$ , so  $n/(\log(n))$  grows only slightly slower than  $n$ ).

## 8 The Input Distribution is Key

You can see, then, that average case complexity can vary considerably depending upon just what constitutes an “average” set of inputs.

Utility functions that get used in many different programs may see different input distributions in each program, and so their average performances in different programs will vary accordingly.

# Case Study: Schemes to Improve the Average Case of Sequential Search

Contents:

- [1 A Rough Mock-up of a Spell Checker](#)
- [2 Trying to Improve the Sequential Search](#)
  - [2.1 Self-Organizing Lists](#)
  - [2.2 Pre-Organized Lists](#)
  - [3 Better than Binary?](#)

## 1 A Rough Mock-up of a Spell Checker

We have previously looked at a sequential search operation, specifically the `std::find` operation.

```
template <typename Iterator, typename T>
Iterator find (Iterator start, Iterator stop, T key)
{
    while (start != stop && (*start == key))
        ++start;
    return start;
}
```

This function performs  $O(\text{distance}(\text{start}, \text{stop}))$  comparisons in the worst case.

Let's consider this function within a scenario of spell checking a book by loading  $N$  words from a dictionary file and then performing  $M$  searches of words obtained from the book.:

```
void checkWords (int N, int M)
{
    list<string> dictionary;
    ifstream dictIn ("dictionary.txt");
    for (int i = 0; i < N; ++i)
    {
        string word;
        dictIn >> word;
        dictionary.push_back(word);
    }
    dictIn.close();

    ifstream bookIn ("book.txt");
    for (int i = 0; i < M; ++i)
    {
        string word;
        bookIn >> word;
        auto pos = find(dictionary.begin(), dictionary.end(), word);
        if (pos == dictionary.end())
            cout << " appears to be misspelled." << endl;
    }
    bookIn.close();
}
```

In the worst case, that `find` call in the second loop is  $O(N)$ , making the second loop, and the entire function,  $O(N * M)$ .

**Question:** Let's assume that all of the words in the book are correctly spelled. They occur *somewhere* in the dictionary. Assume that the words of the dictionary occur in random order.

What is the average case complexity of this function?

Answer +

Well, the visible loops still go around N and M times, respectively. The only thing that can vary based upon the input is how long, on average, `find` takes to find a word from the book.

Given the assumptions, `find` is equally likely to locate the desired word in any position of the list. It is just as likely to find it in position 0 as position N-1, just as likely to find it in position 1 and N-2, in 2 as in N-3, ... in N/2-1 as in N/2. On average, `find` travels halfway through the list.

So `find` contributes an average complexity of  $O(N/2) = O(N)$ , and the overall function remains  $O(M*N)$ .

## 2 Trying to Improve the Sequential Search

### 2.1 Self-Organizing Lists

Now, in the real world, none of us are likely to know all of the words in a good spellcheck dictionary. And even if we knew them, in the sense of being able to know what they meant if we read them, a typical person uses far fewer words in their speech and writing than they actually understand. For example, I know the words “prolixity” and “Satyricon”, but I doubt that I have ever used either in a sentence. And, then, of course, there’s subject matter. Words like “flour” and “sugar” may occur commonly if we are spell checking a cookbook, but rarely if we are spell checking a programming textbook.

So one strategy that algorithm designers have played with is to try to move the words most likely to occur in the book closer to the beginning of the list. One way to do this is called a [self-organizing list](#): when we search for and find a word, we move it to the front of the list, so that if we see it again, we find it very quickly:

```
void checkWords (int N, int M)
{
    list<string> dictionary;
    ifstream dictIn ("dictionary.txt");
    for (int i = 0; i < N; ++i)
    {
        string word;
        dictIn >> word;
        dictionary.push_back(word);
    }
    dictIn.close();

    ifstream bookIn ("book.txt");
    for (int i = 0; i < M; ++i)
    {
        string word;
        bookIn >> word;
        auto pos = find(dictionary.begin(), dictionary.end(), word);
        if (pos == dictionary.end())
            cout << word << " appears to be misspelled." << endl;
        else
        {
            dictionary.erase(pos); // O(1);
            dictionary.push_front(word); // O(1)
        }
    }
    bookIn.close();
}
```

Of course, that word won’t stay right at the front as we continue. How far it moves back as we encounter other words depends on the vocabulary of the book’s author.

### 2.2 Pre-Organized Lists

Self-organizing lists are tricky to analyze, so we’re going to look at an even simpler scheme as an approximation to what they might do for us. It’s not hard to find various lists of the [most common words in the English language](#). Suppose that we put the 100 most commonly used words at the front of our dictionary:

```
void checkWords (int N, int M)
{
    list<string> dictionary;
    ifstream commonIn ("common100.txt");
    for (int i = 0; i < 100; ++i)
    {
        string word;
        commonIn >> word;
        dictionary.push_back(word);
    }
    commonIn.close();
    ifstream dictIn ("dictionary.txt");
    for (int i = 0; i < N; ++i)
    {
        string word;
        dictIn >> word;
        dictionary.push_back(word);
    }
    dictIn.close();

    ifstream bookIn ("book.txt");
    for (int i = 0; i < M; ++i)
    {
        string word;
        bookIn >> word;
```

```
    auto pos = find(dictionary.begin(), dictionary.end(), word);
    if (pos == dictionary.end())
        cout << word << " appears to be misspelled." << endl;
}
bookIn.close();
}
```

**Question:** What is the average case complexity of this function if *all* of the words in the book are among the 100 most common?

Answer +

Again, the loops within the function are fixed to repeat  $N$  and  $M$  times, respectively. But now `find` will, on average, do 50 comparisons per word searched. So `find` is  $O(50) = O(1)$ .

The first loop in `checkwords` remains  $O(N)$ . The second loop is now  $M * O(1) = (M)$ . The total is  $O(M + N)$ .

Much better.

OK, that wasn't very realistic.

**Question:** Suppose that we have those 100 most common words in English at the start of the dictionary list, and that 90% of the words in the book are one of those common words. What is the average complexity of `checkwords`?

Answer +

Again, the loops within the `checkwords` function are fixed to repeat  $N$  and  $M$  times, respectively. But now `find` will, on average, do 50 comparisons per word searched on 90% of the  $M$  searches. On the remaining 10% of the  $M$  searches, `find` will search past the 100 common words, and then, on average, search halfway through the the  $N$  words from the dictionary. So the average number of iterations within `find` is  $0.9 * 50 + 0.1 * (50 + N/2)$ , making the second loop of `checkwords`  $O(M * (50 + 0.1 * N/2)) = O(M * N)$

The first loop in `checkwords` remains  $O(N)$ .

So the total is  $O(M * N)$ .

We may have sped up things by a significant constant factor, but the complexity is the same, so that speedup will get lost quickly as  $M$  and  $N$  grow.

Now, what we have done here is actually closely related to the practice of *caching*, in which recently accessed data is kept in a handy, fast store in order the speed up anticipated repeat requests for the same data. Your web browser caches pages that you have visited recently. Your operating system caches pages of virtual memory. Your CPU caches blocks of memory values that have been fetched from RAM.

One thing that cache designers have learned is that the ideal size of a cache (not always obtainable) is relative to the size of the entire data store. Let's try something along those lines.

**Question:** Suppose that we have those 100 most common words in English at the start of the dictionary list, and that  $\frac{N-1}{N}$  of the words in the book are one of those common words. What is the average complexity of `checkwords`?

Answer +

Again, the loops within the `checkwords` function are fixed to repeat  $N$  and  $M$  times, respectively.

But now `find` will, on average, do 50 comparisons per word searched on  $M \frac{N-1}{N}$  searches. On the remaining  $M \frac{1}{N}$  of the searches, `find` will search past the 100 common words, and then, on average, search halfway through the the  $N$  words from the dictionary. So the average number of iterations within `find` is  $\frac{N-1}{N} * 50 + \frac{1}{N} * (50 + N/2)$ , making the second loop of `checkwords`

$$\begin{aligned} & M * O\left(\frac{N-1}{N} * 50 + \frac{1}{N} * (50 + N/2)\right) \\ &= M * O(50 - 50/N + 50/N + 50/N * 1/2) \\ &= M * O(1) \\ &= O(M) \end{aligned}$$

The first loop in `checkwords` remains  $O(N)$ .

So the total is  $O(M + N)$ .

So it is plausible to reduce the average complexity, but only if the input satisfies some pretty stringent restrictions.

## 3 Better than Binary?

Of course, all of the above has been predicated upon the idea that we wanted to work with sequential search in the first place. We know that a binary search would be faster than sequential search in most cases. However to use binary search, we would need to know that the input is already in sorted order, or we would have to sort it.

Is the extra cost of sorting the data worth it?

Suppose we change our code to

```
void checkwords (int N, int M)
{
    vector<string> dictionary;
    ifstream dictIn ("dictionary.txt");
    for (int i = 0; i < N; ++i)
    {
        string word;
        dictIn >> word;
        dictionary.push_back(word);
    }
    dictIn.close();
    sort(dictionary.begin(), dictionary.end());

    ifstream bookIn ("book.txt");
    for (int i = 0; i < M; ++i)
    {
        string word;
        bookIn >> word;
        auto pos = lower_bound(dictionary.begin(), dictionary.end(), word);
        if (pos == dictionary.end() || *pos != word)
            cout << word << " appears to be misspelled."
    }
    bookIn.close();
}
```

**Question:** What is the worst-case (and average case) complexity of `checkwords`? (The `std::sort` function is  $O(n \log(n))$  where  $n = \text{distance}(\text{start}, \text{stop})$ ).

Answer +

The first loop remains  $O(N)$ .

The sort call is  $O(N \log(N))$  because there are  $N$  items in the dictionary.

The second loop body uses a `lower_bound` call that does a binary search in  $O(\log(N))$  time. That makes the second loop a total of  $O(M \log(N))$ .

So the total complexity of the function is  $O((M + N) \log(N))$ .

So it is possible that a pre-organized or self organizing list would be faster than a combination of sorting and binary search, but only for very selective input distributions, and it won't be a whole lot faster (only a factor of  $\log N$ ) even then.

# Sorting --- Insertion Sort

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 The Algorithm](#)
- [2 Insertion Sort: Worst Case Analysis](#)
- [3 Insertion Sort: special case](#)
- [4 Average-Case Analysis for Insertion Sort](#)
  - [4.1 Inversions](#)
  - [4.2 A Speed Limit on Adjacent-Swap Sorting](#)

Sorting: given a sequence of data items in an unknown order, re-arrange the items to put them into ascending (descending) order by key.

Sorting algorithms have been studied extensively. There is no one best algorithm for all circumstances, but the big-O behavior is a key to understanding where and when to use different algorithms.

The [insertion sort](#) divides the list of items into a sorted and an unsorted region, with the sorted items in the first part of the list.

Idea: Repeatedly take the first item from the unsorted region and insert it into the proper position in the sorted portion of the list.

## 1 The Algorithm

This is the insertion sort:

```
// Ford & Topp, Chapter 4
//
template <typename T>
void insertionSort(vector<T>& v)
{
    int i, j, n = v.size();
    T target;

    // place v[i] into the sublist
    // v[0] ... v[i-1], 1 <= i < n,
    // so it is in the correct position
    for (i = 1; i < n; i++)
    {
        // index j scans down list from v[i] looking for
        // correct position to locate target. assigns it to
        // v[j]
        j = i;
        target = v[i];
        // locate insertion point by scanning downward as long
        // as target < v[j-1] and we have not encountered the
        // beginning of the list
        while (j > 0 && target < v[j-1])
        {
            // shift elements up list to make room for insertion
            v[j] = v[j-1];
            j--;
        }
        // the location is found; insert target
        v[j] = target;
    }
}
```

- At the beginning of each outer iteration, items 0 ... i-1 are properly ordered.
- Each outer iteration seeks to insert item v[i] into the appropriate position within 0 ... i.

[Run this algorithm](#) until you are comfortable with your understanding of how it works.

## 2 Insertion Sort: Worst Case Analysis

- Assume comparisons & copying are  $O(1)$ .

```
// Ford & Topp, Chapter 4
//
template <typename T>
void insertionSort(vector<T>& v)
{
    int i, j, n = v.size();           // O(1)
    T target;                      // O(1)

    // place v[i] into the sublist
```

```

// v[0] ... v[i-1], 1 <= i < n,
// so it is in the correct position
for (i = 1; i < n; i++)
{
    // index j scans down list from v[i] looking for
    // correct position to locate target. assigns it to
    // v[j]
    j = i;                                // O(1)
    target = v[i];                         // O(1)
    // locate insertion point by scanning downward as long
    // as target < v[j-1] and we have not encountered the
    // beginning of the list
    while (j > 0 && target < v[j-1])
    {
        // shift elements up list to make room for insertion
        v[j] = v[j-1];                      // O(1)
        j--;                               // O(1)
    }
    // the location is found; insert target
    v[j] = target;                        // O(1)
}

```

Looking at the inner loop,

```

// Ford & Topp, Chapter 4
//
template <typename T>
void insertionSort(vector<T>& v)
{
    int i, j, n = v.size();                // O(1)
    T target;                            // O(1)

    // place v[i] into the sublist
    // v[0] ... v[i-1], 1 <= i < n,
    // so it is in the correct position
    for (i = 1; i < n; i++)
    {
        // index j scans down list from v[i] looking for
        // correct position to locate target. assigns it to
        // v[j]
        j = i;                                // O(1)
        target = v[i];                         // O(1)
        // locate insertion point by scanning downward as long
        // as target < v[j-1] and we have not encountered the
        // beginning of the list
        while (j > 0 && target < v[j-1])
        {
            // shift elements up list to make room for insertion
            v[j] = v[j-1];                      // O(1)
            j--;                               // O(1)
        }
        // the location is found; insert target
        v[j] = target;                        // O(1)
    }
}

```

**Question:** In the worst case, how many times do we go around the inner loop (to within plus or minus 1)?

- 0 times
- 1 time
- $i$  times
- $j$  times
- $n$  times

\*\*Answer:\*\*

“i times” is the best answer.

“n times” is correct only on the final time that we enter the loop. On earlier visits to this loop,  $i < n$ , so “i times” is a tighter bound and therefore more correct.

“j times” is also a plausible answer, but because j changes each time around the loop, we would really have to say that we go around  $j_0$  times, where  $j_0$  is defined as the value of j upon entry to the loop. As it happens, we already have a variable that can stand in for  $j_0$ , namely, “i”, so that’s a better answer to this question.

With that determined,

```
// Ford & Topp, Chapter 4
//
template <typename T>
void insertionSort(vector<T>& v)
{
    int i, j, n = v.size();           // O(1)
    T target;                      // O(1)

    // place v[i] into the sublist
    //   v[0] ... v[i-1], 1 <= i < n,
    // so it is in the correct position
    for (i = 1; i < n; i++)
    {
        // index j scans down list from v[i] looking for
        // correct position to locate target. assigns it to
        // v[j]
        j = i;                      // O(1)
        target = v[i];              // O(1)
        // locate insertion point by scanning downward as long
        // as target < v[j-1] and we have not encountered the
        // beginning of the list
        while (j > 0 && target < v[j-1]) //cond: O(1) #: i
        { //total: O(1)
            // shift elements up list to make room for insertion
            v[j] = v[j-1];          // O(1)
            j--;                   // O(1)
        }
        // the location is found; insert target
        v[j] = target;             // O(1)
    }
}
```

Moving on...

**Question:** So what is the complexity of the inner loop?

- O(1)
- O(i)
- O(j)
- O(n)
- None of the above

\*\*Answer:\*\*

The body and condition are  $O(1)$ , and the loop executes  $i^*$ , so the entire loop is  $O(i)$ .

```
// Ford & Topp, Chapter 4
//
template <typename T>
void insertionSort(vector<T>& v)
{
    int i, j, n = v.size();           // O(1)
    T target;                      // O(1)

    // place v[i] into the sublist
    // v[0] ... v[i-1], 1 <= i < n,
    // so it is in the correct position
    for (i = 1; i < n; i++)
    {
        // index j scans down list from v[i] looking for
        // correct position to locate target. assigns it to
        // v[j]
        j = i;                      // O(1)
        target = v[i];              // O(1)
        // locate insertion point by scanning downward as long
        // as target < v[j-1] and we have not encountered the
        // beginning of the list
        while (j > 0 && target < v[j-1]) // cond: O(1) #: i total: O(i)
        {
            // shift elements up list to make room for insertion
            v[j] = v[j-1];          // O(1)
            j--;                   // O(1)
        }
        // the location is found; insert target
        v[j] = target;             // O(1)
    }
}
```

Don't be fooled by the fact that  $i$  is always less than  $n$  into jumping immediately to  $O(n)$ . We always analyze loops from the inside out, and the relationship between  $i$  and  $n$  is outside the current portion of the algorithm you were asked about.

Although,  $O(n)$  is, technically correct, it's a looser bound than  $O(i)$ , and moving prematurely to unnecessarily loose bounds can, in some cases, cause us to miss possible simplifications and wind up with a final bound that will be much larger than necessary. (After all, not only is  $O(n)$  technically correct, but so is  $O(n^{100})$ , and clearly if we adopted *that* as our bound, the resulting answer would be too large to be meaningful.)

Now, looking at the outer loop body,

```
// Ford & Topp, Chapter 4
//
template <typename T>
void insertionSort(vector<T>& v)
{
    int i, j, n = v.size();           // O(1)
    T target;                      // O(1)

    // place v[i] into the sublist
    // v[0] ... v[i-1], 1 <= i < n,
    // so it is in the correct position
    for (i = 1; i < n; i++)
    {
        // index j scans down list from v[i] looking for
        // correct position to locate target. assigns it to
        // v[j]
        j = i;                      // O(1)
        target = v[i];              // O(1)
        // locate insertion point by scanning downward as long
        // as target < v[j-1] and we have not encountered the
        // beginning of the list
        // O(i)
        // the location is found; insert target
        v[j] = target;             // O(1)
    }
}
```

So the entire outer loop body is  $O(i)$ .

```
// Ford & Topp, Chapter 4
//
template <typename T>
void insertionSort(vector<T>& v)
{
    int i, j, n = v.size();           // O(1)
    T target;                      // O(1)

    // place v[i] into the sublist
    // v[0] ... v[i-1], 1 <= i < n,
    // so it is in the correct position
    for (i = 1; i < n; i++)         // cond: O(1) #: (n-1)
    {
        O(i)
```

```
}
```

The outer loop executes  $n - 1$  times.

Question: What, then is the complexity of the entire outer loop?

- $O(i)$
- $O(n)$
- $O(i * (n - 1))$
- $O(i * n)$
- $O(i^2)$
- $O(n^2)$
- $O((n * (n - 1))/2)$

\*\*Answer:\*\*  +

$O(n^2)$  is correct.

Technically, the time for the loop is

$$\sum_{i=1}^{n-1} O(i) = O\left(\frac{((n-1)n)}{2}\right) = O(n^2)$$

```
// Ford & Topp, Chapter 4
//
template <typename T>
void insertionSort(vector<T>& v)
{
    int i, j, n = v.size();           // O(1)
    T target;                      // O(1)

    // place v[i] into the sublist
    // v[0] ... v[i-1], 1 <= i < n,
    // so it is in the correct position
    for (i = 1; i < n; i++)        // cond: O(1) #: (n-1) total: O(n^2)
    {
        0(i)
    }
}
```

If you gave any answer involving  $i$ , you should have known better from the start. Complexity of a block of code must always be described in terms of the *inputs* to that code.  $i$  is not an input to the loop - any value it might have held prior to the start of the loop is ignored and overwritten. In fact, if this were my code I would dropped the earlier declaration of  $i$  and rewritten the loop header as

```
for (int i = 1; i < n; i++)
```

just to emphasize the fact that  $i$  exists purely to function as the loop counter and, just as importantly, to avoid leaving  $i$  uninitialized after it is originally declared. I would do a similar rewrite to  $j$  and the inner loop.

So how do we get  $O(n^2)$ ? There are two ways to get there.

```
// Ford & Topp, Chapter 4
//
template <typename T>
void insertionSort(vector<T>& v)
{
    int i, j, n = v.size();           // O(1)
    T target;                      // O(1)

    // place v[i] into the sublist
    // v[0] ... v[i-1], 1 <= i < n,
    // so it is in the correct position
    0(n^2)
}
```

**Question:** What, then is the complexity of the entire function?

- $O(n)$
- $O(n^2)$
- $O((n * (n - 1))/2)$
- None of the above

\*\*Answer:\*\*

- None of the above

```
// Ford & Topp, Chapter 4
//
template <typename T>
void insertionSort(vector<T>& v)
{
    int i, j, n = v.size();           // O(1)
    T target;                      // O(1)

    // place v[i] into the sublist
    // v[0] ... v[i-1], 1 <= i < n,
    // so it is in the correct position
    O(n^2)
}
```

We can't say the complexity of the function is  $O(n^2)$  because  $n$  is not an input to this function. It's a local variable that is initialized based upon the actual inputs to the function and that disappears when we return from a call to this function.

A proper answer would be that this function is  $O(v.size()^2)$  or, we could say that

Insertion sort has a worst case of  $O(N^2)$  where  $N$  is the size of the input vector.

(Note that we use "N" and not "n" to avoid confusion between the mathematical construct "N" that we introduce in the above statement and with the variable "n" that appears in the code.)

## 3 Insertion Sort: special case

As a special case, consider the behavior of this algorithm when applied to an array that is already sorted.

```
//
// From Budd, "Data Structures in C++ using
// the Standard Template Library"
// section 4.2.4
//
void insertionSort (double v[], unsigned int n)
// exchange the values in the vector v
// so they appear in ascending order
{
    for (unsigned int i = 1; i < n; i++) {
        // move element v[i] into place
        double element = v[i];           // O(1)
        int j = i - 1;                  // O(1)
        while (j >= 0 && element < v[j]) { // cond: O(1)
            // slide old value up
            v[j+1] = v[j];              // O(1)
            j = j - 1;                  // O(1)
        }
        // place element into position
        v[j+1] = element;             // O(1)
    }
}
```

- Note that if the array is already sorted, then we never enter the body of the inner loop. The inner loop is then  $O(1)$  and `insertionSort` is  $O(v.size())$ .
- This makes insertion sort a reasonable choice when adding a few items to a large, already sorted array.

## 4 Average-Case Analysis for Insertion Sort

Instead of doing the average case analysis by the copy-and-paste technique, we'll produce a result that works for all algorithms that behave like it.

Define an inversion of an array  $a$  as any pair  $(i, j)$  such that  $i < j$  but  $a[i] > a[j]$ .

**Question:** How many inversions in this array?

[ [29 \; 10 \; 14 \; 37 \; 13] ]

\*\*Answer\*\* +

There are 5 inversions in [ 29 ; 10 ; 14 ; 37 ; 13 ]

The inversions are: (1,2), (1,3), (1,5), (3,5), (4,5)

## 4.1 Inversions

In an array of  $n$  elements, the most inversions occur when the array is in exactly reversed order. Inversions then are

inversions	count
(1,2), (1,3), (1,4), ..., (1,n),	$n-1$
(2,3), (2,4), ..., (2,n),	$n-2$
(3,4), ..., (3,n),	$n-3$
:	:
(n-1,n)	1

Counting these we have (starting from the bottom):  $\sum_{i=1}^{n-1} i$  inversions. So the total # of inversions is  $\frac{n*(n-1)}{2}$ .

We'll state this formally:

**Theorem:** The maximum number of inversions in an array of  $n$  elements is  $(n * (n - 1)) / 2$ .

We have just proven this.

**Theorem:** The average number of inversions in an array of  $n$  randomly selected elements is  $(n * (n - 1)) / 4$ .

We won't prove this, but note that it makes sense, since the minimum number of inversions is 0, and the maximum is  $(n * (n - 1)) / 2$ , so it makes intuitive sense that the average would be the midpoint of these two values.

## 4.2 A Speed Limit on Adjacent-Swap Sorting

Now, the result we have been working toward:

**Theorem:** Any sorting algorithm that only swaps adjacent elements has average time no faster than  $O(n^2)$ .

### Proof

Swapping 2 adjacent elements in an array removes at most 1 inversion.

But on average, there are  $O(n^2)$  inversions, so the total number of swaps required is at least  $O(n^2)$ .

Hence the algorithm as a whole can be no faster than  $O(n^2)$ .

QED

And,

**Corollary:** Insertion sort has average case complexity  $O(n^2)$ .

### Proof

Insertion sort only exchanges adjacent elements. By the theorem just given, the best average case complexity we could therefore get is  $O(n^2)$ .

The theorem does not preclude an average case complexity even slower than that, but we know that the worst case complexity is also  $O(n^2)$ , and the average case can't be any slower than the worst case.

So we conclude that the average case complexity is, indeed,  $O(n^2)$ .

# Sorting Speed Limits

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

[1 Counting Comparisons](#)

[2 Speed Limit](#)

[3 But Can We Even Go That Fast?](#)

OK, we've seen an algorithm that sorts in  $O(n^2)$  time.

There is actually an algorithm called Shell's sort that uses insertion sorting as an internal component and that yields a family of sorting algorithms that run in some rather odd times:  $O(n^{5/4})$ ,  $O(n^{3/2})$ , etc.

Those are faster than an ordinary insertion sort, so we might ask...

Just how fast *can* a sorting algorithm get?

## 1 Counting Comparisons

Consider how many comparisons are needed to determine the proper order for a set of elements:

- $n = 2$  needs 1 comparison
- $n = 3$  needs between 2 and 3 comparisons
  - E.g., if  $a < b$  then is  $c < a$ ? If so, done. If not, is  $c > b$ ?
- $n = k + 1$  needs  $\log k$  comparisons

Figure out the order for the first  $k$  elements, then use  $\log k$  steps to figure out where the  $k + 1$ st element goes.

Why  $\log k$ ?

Because we can use *binary search* to find the place where a given element should be inserted.

## 2 Speed Limit

So, it looks like we need  $\sum_{i=1}^n \log i$  comparisons.

- We could prove this formally, with more work — more than is really justified for this course.

What does this mean about performance?

$$\begin{aligned}\sum_{i=1}^n \log(i) &\geq \sum_{i=n/2}^n \log(i) \\ &\geq \sum_{i=n/2}^n \log(n/2) \\ &= n/2 \log(n/2) \\ &= n/2(\log(n) - \log(2)) \\ &= O(n \log(n))\end{aligned}$$

So no sorting algorithm that works by pair-wise comparison (i.e., comparing elements 2 at a time) can be faster than  $O(n * \log(n))$  (worst or average case).

## 3 But Can We Even Go That Fast?

$O(n \log(n))$  is a pretty good speed in practice, often not noticeably slower than  $O(n)$ . After all, we usually need  $O(n)$  time just to read or fetch the array of data that we want to sort.

But can we actually find algorithms that achieve this  $O(n \log(n))$  optimum speed?

Our next lessons will show that we can.

# Sorting --- Merge Sort

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Merging Sorted Data](#)
  - [1.1 Understanding the Merge Algorithm](#)
  - [1.2 Merge Analysis](#)
- [2 Merge Sort](#)
  - [2.1 The Algorithm](#)
  - [2.2 MergeSort Analysis](#)

Our next algorithm actually achieves the optimal big-O behavior for a sorting algorithm. The merge sort has  $O(n \log n)$  time for both its worst and average case.

This doesn't necessarily make it the ideal choice, however, in all sorting applications. The constant multiplier on the timing is somewhat high, and merge sort may require an unusually high amount of memory.

Variants of the basic merge sort algorithm are, however, often used with linked lists (which can't be sorted by most other  $O(n \log n)$  algorithms) and are used to sort data residing on disk or magnetic tape.

## 1 Merging Sorted Data

Before tackling the merge sort itself, we start with a simpler function that is used by merge sort.

Suppose that our sequence of data can be divided into two parts, such that  $a[first..mid-1]$  is already sorted and  $a[mid..last-1]$  is already sorted. Then we could merge the two parts into a combined sorted sequence using the code shown here.

```
template <typename T>
void merge(vector<T>& v, int first, int mid, int last)
{
    // temporary vector to merge the sorted sublists
    vector<T> tempVector;
    int indexA, indexB, indexV;

    // set indexA to scan sublistA (index range [first,mid)
    // and indexB to scan sublistB (index range [mid, last)
    indexA = first;
    indexB = mid;

    // while both sublists are not exhausted, compare v[indexA] and
    // v[indexB]copy the smaller to vector temp using push_back()
    while (indexA < mid && indexB < last) <:<span>&#x2780;</span>:>
        if (v[indexA] < v[indexB])
        {
            tempVector.push_back(v[indexA]); // copy element to temp
            indexA++; // increment indexA
        }
        else
        {
            tempVector.push_back(v[indexB]); // copy element to temp
            indexB++; // increment indexB
        }

    // copy the tail of the sublist that is not exhausted
    while (indexA < mid) <:<span>&#x2781;</span>:>
    {
        tempVector.push_back(v[indexA]);
        indexA++;
    }

    while (indexB < last)
    {
        tempVector.push_back(v[indexB]);
        indexB++;
    }

    // copy vector tempVector using indexV to vector v using indexA
    // which is initially set to first
    indexA = first;

    // copy elements from temporary vector to original list
    for (indexV = 0; indexV < tempVector.size(); indexV++) <:<span>&#x2782;</span>:>
    {
        v[indexA] = tempVector [indexV];
        indexA++;
    }
}
```

## 1.1 Understanding the Merge Algorithm

The heart of the merge algorithm is the first loop (①).

The variables *first*, *mid*, and *last* mark off two subsequences that we want to merge. We can think of  $a[\text{first} \dots \text{mid}-1]$  and  $a[\text{mid} \dots \text{last}-1]$  as two separate, sorted sequences. We want to combine them into a single sorted sequence, *tempVector*.

The way to do this is quite simple. Just compare the first element in each of the two input (sub)sequences and copy the smaller one.

For example, if we were merging subsequences

2 4 5 6

and

1 3

we would compare the first element in each one (2 and 1) and decide to copy 1.

Then we continue with the remainder, merging

2 4 5 6

and

3

On the next step we would copy 2, and be left with the merge of

4 5 6

and

3

We would then copy 3.

At this point, our temporary vector contains

1 2 3

We would now exit from this main loop, because one of the arrays has been completely emptied out.

The rest of the algorithm is “cleanup”. We exit the main loop when we have emptied one of the two subsequences, so there is a possibility that the other subsequence still has data. The next two loops (②) copy that data from the remainder of the two subsequences. (Because one of those subsequences has been emptied, one of these loops will execute zero times.)

Finally (③), we copy the entire merged data set back out of the temporary vector into the original vector.

[Run this algorithm](#) until you are comfortable with your understanding of how it works.

## 1.2 Merge Analysis

```
template <typename T>
void merge(vector<T>& v, int first, int mid, int last)
{
    // temporary vector to merge the sorted sublists
    vector<T> tempVector;
    int indexA, indexB, indexV;

    // set indexA to scan sublistA (index range [first,mid)
    // and indexB to scan sublistB (index range [mid, last)
    indexA = first;
    indexB = mid;

    // while both sublists are not exhausted, compare v[indexA] and
    // v[indexB]copy the smaller to vector temp using push_back()
    while (indexA < mid && indexB < last)
        if (v[indexA] < v[indexB])
        {
            tempVector.push_back(v[indexA]);    // O(tempVector.size())
            indexA++;
        }
        else
        {
            tempVector.push_back(v[indexB]);    // O(tempVector.size())
            indexB++;
        }

    // copy the tail of the sublist that is not exhausted
    while (indexA < mid)
    {
```

```

    tempVector.push_back(v[indexA]);      // O(tempVector.size())
    indexA++;                          // O(1)
}

while (indexB < last)
{
    tempVector.push_back(v[indexB]);      // O(tempVector.size())
    indexB++;                          // O(1)
}

// copy vector tempVector using indexV to vector v using indexA
// which is initially set to first
indexA = first;

// copy elements from temporary vector to original list
for (indexV = 0; indexV < tempVector.size(); indexV++)
{
    v[indexA] = tempVector [indexV];    // O(1)
    indexA++;                          // O(1)
}
}

```

There are several vector `push_back` calls, which have a worst-case behavior proportional to the size of the vector. However, a little time looking at them shows that they are all on `tempVector`, which is initially empty. So this falls into the special case pattern of filling an initially empty vector with repeated `push_backs`, in which case those pushes will amortize to  $O(1)$ .

This means that all the loop bodies amortize to  $O(1)$ .

Looking at the code for the first 3 loops, note that

- each one adds one element into `tempVector`.
- no element is copied multiple times. If we copy the element at `indexA`, we also increment `indexA`, so we will not copy that element again. Similarly, if we copy the element at `indexB`, we also increment `indexB`, so we will not copy that element again.

Since there are a total of `last - first` elements, each loop can repeat no more than `last - first` times.

In fact, the `sum` of the number of iterations of *all three loops* is `last - first`.

So all three loops are  $O(\text{last} - \text{first})$ .

[merge2.cpp](#) +

```

template <typename T>
void merge(vector<T>& v, int first, int mid, int last)
{
    // temporary vector to merge the sorted sublists
    vector<T> tempVector;
    int indexA, indexB, indexV;

    // set indexA to scan sublistA (index range [first, mid])
    // and indexB to scan sublistB (index range [mid, last])
    indexA = first;
    indexB = mid;

    // while both sublists are not exhausted, compare v[indexA] and
    // v[indexB] copy the smaller to vector temp using push_back()
    while (indexA < mid && indexB < last) // total: O(last-first)
        // O(1)

    // copy the tail of the sublist that is not exhausted
    while (indexA < mid) // total: O(last-first)
    {
        // O(1)
    }

    while (indexB < last) // total: O(last-first)
    {
        // O(1)
    }

    // copy vector tempVector using indexV to vector v using indexA
    // which is initially set to first
    indexA = first;

    // copy elements from temporary vector to original list
    for (indexV = 0; indexV < tempVector.size(); indexV++) // cond: O(1) #: (last-first) total: O(last-first)
    {
        // O(1)
    }
}

```

The last loop clearly repeats once for each element in `tempVector`. But we have just determined that the total number of elements in `tempVector` will be `last - first`.

[merge3.cpp](#) +

```

template <typename T>
void merge(vector<T>& v, int first, int mid, int last)
{
    // temporary vector to merge the sorted sublists
    vector<T> tempVector;
    int indexA, indexB, indexV;

    // set indexA to scan sublistA (index range [first,mid)
    // and indexB to scan sublistB (index range [mid, last)
    indexA = first;
    indexB = mid;

    // O(last-first)
    // O(last-first)
    // O(last-first)

    // copy vector tempVector using indexV to vector v using indexA
    // which is initially set to first
    indexA = first;

    // copy elements from temporary vector to original list
    // O(last-first)
}

```

That leaves only a handful of  $O(1)$  statements that will all be dominated by the complexity of the loops, so

- merge is  $O(\text{last} - \text{first})$ .

## 2 Merge Sort

The merge function lets us combine two sorted sequences of data into a single sorted sequence. But how do we get the two sorted sequences in the first place? By merge'ing two even smaller sorted sequences!

### 2.1 The Algorithm

```

template <typename T>
void mergeSort(vector<T>& v, int first, int last)
{
    // if the sublist has more than 1 element continue
    if (first + 1 < last)
    {
        // for sublists of size 2 or more, call mergeSort()
        // for the left and right sublists and then
        // merge the sorted sublists using merge()
        int midpt = (last + first) / 2;

        mergeSort(v, first, midpt);
        mergeSort(v, midpt, last);
        merge(v, first, midpt, last);
    }
}

```

The algorithm shown here is the actual sorting algorithm. It is almost amazingly simple, consisting simply of two recursive calls to itself, each attempting to sort half the vector, followed by a call to merge to combine the two sorted halves into a single sorted sequence.

For many people, the very simplicity of this algorithm makes it hard to believe that it can work. I therefore recommend strongly that you [run this algorithm](#) until you are comfortable with your understanding of it.

### 2.2 MergeSort Analysis

```

template <typename T>
void mergeSort(vector<T>& v, int first, int last)
{
    // if the sublist has more than 1 element continue
    if (first + 1 < last)
    {
        // for sublists of size 2 or more, call mergeSort()
        // for the left and right sublists and then
        // merge the sorted sublists using merge()
        int midpt = (last + first) / 2;

        mergeSort(v, first, midpt);
        mergeSort(v, midpt, last);
        merge(v, first, midpt, last);
    }
}

```

- Each call to mergeSort is either done in  $O(1)$  time (if  $\text{first}+1 \geq \text{last}$ ) or splits the array into two equal ( $\pm 1$ ) pieces. We can do this split up to log  $N$  times.

We can envision the recursive `mergeSort` calls (in blue) and the subsequent calls to merge (in yellow) as a tree-like structure.

Let  $N$  denote the total number of elements being sorted (the value of `last-first` on the very first call to `mergeSort`).

- Each level in the tree involves no more than  $N$  objects, split in various ways and needing to be merged.
- merge is  $O(k)$ , where  $k$  is the number of elements to be merged. The sum of all the  $k$  values at any level of the yellow tree is  $N$ . Consequently the combined set of merges at each level of the tree is  $O(N)$ .
- The blue tree represents all the non-merge work in `mergeSort`. But there's only  $O(1)$  work in each of those blue nodes. Since the most blue nodes we could have at one level is  $N$ , each blue level is, at most,  $O(N)$  total work.
- Because we have  $\log N$  levels, each level taking  $O(N)$  work, the overall merge sort code is (worst & average case)  $O(N \log N)$ .

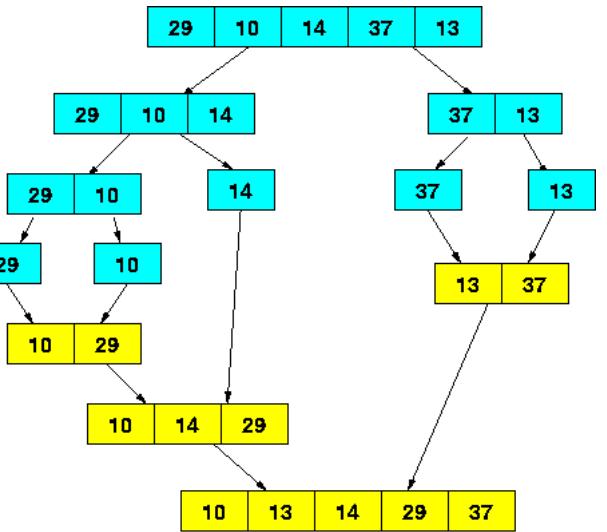
So merge sort is as fast as any pairwise-comparison sort can be. Still, merge sort is not considered to be the “ideal” sorting algorithm. Its primary drawbacks are

- It requires  $O(N)$  extra storage (for the `tempVector`)
- It does the full set of comparisons and copies even when applied to arrays that are already sorted.

On the other hand, merge sort has an advantage that may, at first glance, not have seemed very important. The merge routine itself moves sequentially through its working arrays, not jumping from place to place. This behavior would be absolutely wonderful if we were storing our arrays in some strange kind of memory where moving forward one place is cheap, but jumping to an arbitrary position is expensive.

In fact, that “strange kind of memory” does exist:

- Linked lists support this movement pattern and can be sorted quickly using merge sort. Many fast sorting algorithms are limited to array-like structures.
- Disk drives and magnetic tape both meet that movement pattern as well. Hence variations of merge sort have long been the algorithm of choice in [external sorting](#), sorting sets of material stored in disk/tape files that are too large to load into memory.



# Sorting --- Quick Sort

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Partitioning the Array](#)
  - [1.1 Analysis of pivotIndex](#)
- [2 Quick Sort](#)
- [3 Quick Sort Analysis](#)
  - [3.1 Best Case](#)
  - [3.2 Average Case](#)
  - [3.3 Worst Case](#)
  - [3.4 Great Average, Dreadful Worst](#)
- [4 Optimizing the Algorithm](#)
- [5 Average Case Analysis of Quick Sort](#)

Quick sort is a sorting algorithm with the optimal  $O(n \log n)$  average case, but a  $O(n^2)$  worst case.

Despite its slower worst case behavior, the Quick sort is generally preferred over Merge sort when sorting array-like structures (including vectors and deques). This is because the constant multiplier on quick sort's average case is much smaller, and, if we are careful, we can make the worst case input of quick sort a relatively obscure case that would seldom be seen in practice.

Like Merge sort, Quick sort also works by dividing the array into progressively smaller parts, but instead of chopping at an arbitrary point (e.g., halfway), it divides into pieces in which

- all items in the left section are less than or equal to a *pivot* value,
- all items on the right are greater than the pivot value.

This is often called *partitioning the array*.

For example, given the array

1 4 3 9 5 8 7 6

it's not hard to see that we can split the array this way

1 4 3 | 9 5 8 7 6

so that all items to the left are less than or equal to a pivot value 4, and all items on the right are greater than 4.

On the other hand, for some arrays:

9 1 8 7 2 6 4 0

there's really no place where we can divide the array "as is" to meet these requirements.

So an algorithm to partition the array must, in some cases, move elements around to make the partition possible. The *pivotIndex* algorithm from your text does just this.

## 1 Partitioning the Array

[pivot1.cpp](#) +

```
template <typename T>
int pivotIndex(vector<T>& v, int first, int last)
{
    // index for the midpoint of [first,last) and the
    // indices that scan the index range in tandem
    int mid, scanUp, scanDown;
    // pivot value and object used for exchanges
    T pivot, temp;

    if (first == last)
        return last;
    else if (first == last-1)
        return first;
    else
    {
        mid = (last + first)/2;
        pivot = v[mid];
```

```

// exchange the pivot and the low end of the range
// and initialize the indices scanUp and scanDown.
v[mid] = v[first];
v[first] = pivot;

scanUp = first + 1;
scanDown = last - 1;

// manage the indices to locate elements that are in
// the wrong sublist; stop when scanDown <= scanUp
for(;;)
{
    // move up lower sublist; stop when scanUp enters
    // upper sublist or identifies an element >= pivot
    while (scanUp <= scanDown && v[scanUp] < pivot)
        scanUp++;

    // scan down upper sublist; stop when scanDown locates
    // an element <= pivot; we guarantee we stop at arr[first]
    while (pivot < v[scanDown])
        scanDown--;

    // if indices are not in their sublists, partition complete
    if (scanUp >= scanDown)
        break;

    // indices are still in their sublists and identify
    // two elements in wrong sublists. exchange
    swap(v[scanUp], v[scanDown]);

    scanUp++;
    scanDown--;
}

// copy pivot to index (scanDown) that partitions sublists
// and return scanDown
v[first] = v[scanDown];
v[scanDown] = pivot;
return scanDown;
}
}

```

First, we choose a value to serve as the pivot element. In this version, we are using the midpoint of the data sequence. There are other possibilities, some of which will be discussed later.

The pivotIndex algorithm basically starts with the *scanUp* and *scanDown* indices at opposite ends of the array. These are moved towards each other until they meet.

*scanUp* is moved up first, until it either meets *scanDown* or hits an element greater than or equal to the pivot.

Then *scanDown* is moved down until it hits an element less than or equal to the pivot.

If *scanUp* is pointing to an element greater than the pivot, and *scanDown* is pointing to an element less than the pivot, those two elements are clearly out of order with respect to each other. Clearly, we will never find a place to partition the array as long as we have these two elements, the smaller one coming after the larger one. So, we swap the two, and then resume moving *scanUp* and *scanDown* towards each other again.

[Run this algorithm](#) until you are comfortable with your understanding of how it works.

## 1.1 Analysis of pivotIndex

Let's do the analysis of this routine.

The loop bodies of the two while loops are clearly O(1).

```

pivot2.cpp +

template <typename T>
int pivotIndex(vector<T>& v, int first, int last)
{
    // index for the midpoint of [first,last) and the
    // indices that scan the index range in tandem
    int mid, scanUp, scanDown;
    // pivot value and object used for exchanges
    T pivot, temp;

    if (first == last)
        return last;
    else if (first == last-1)
        return first;
    else
    {
        mid = (last + first)/2;
        pivot = v[mid];

        // exchange the pivot and the low end of the range
        // and initialize the indices scanUp and scanDown.
        v[mid] = v[first];
        v[first] = pivot;

        scanUp = first + 1;
        scanDown = last - 1;

        // manage the indices to locate elements that are in
        // the wrong sublist; stop when scanDown <= scanUp
        for(;;)
        {
            // move up lower sublist; stop when scanUp enters
            // upper sublist or identifies an element >= pivot
            while (scanUp <= scanDown && v[scanUp] < pivot)
                scanUp++;

            // scan down upper sublist; stop when scanDown locates
            // an element <= pivot; we guarantee we stop at arr[first]
            while (pivot < v[scanDown])
                scanDown--;

            // if indices are not in their sublists, partition complete
            if (scanUp >= scanDown)
                break;

            // indices are still in their sublists and identify
            // two elements in wrong sublists. exchange
            swap(v[scanUp], v[scanDown]);

            scanUp++;
            scanDown--;
        }

        // copy pivot to index (scanDown) that partitions sublists
        // and return scanDown
        v[first] = v[scanDown];
        v[scanDown] = pivot;
        return scanDown;
    }
}

```

```

v[first] = pivot;
scanUp = first + 1;
scanDown = last - 1;

// manage the indices to locate elements that are in
// the wrong sublist; stop when scanDown <= scanUp
for(;;)
{
    // move up lower sublist; stop when scanUp enters
    // upper sublist or identifies an element >= pivot
    while (scanUp <= scanDown && v[scanUp] < pivot)
        scanUp++; // O(1)

    // scan down upper sublist; stop when scanDown locates
    // an element <= pivot; we guarantee we stop at arr[first]
    while (pivot < v[scanDown])
        scanDown--; // O(1)

    // if indices are not in their sublists, partition complete
    if (scanUp >= scanDown)
        break;

    // indices are still in their sublists and identify
    // two elements in wrong sublists. exchange
    swap(v[scanUp], v[scanDown]);

    scanUp++;
    scanDown--;
}

// copy pivot to index (scanDown) that partitions sublists
// and return scanDown
v[first] = v[scanDown];
v[scanDown] = pivot;
return scanDown;
}
}

```

So are the while loop conditions.

[pivot3.cpp](#) +

```

template <typename T>
int pivotIndex(vector<T>& v, int first, int last)
{
    // index for the midpoint of [first,last) and the
    // indices that scan the index range in tandem
    int mid, scanUp, scanDown;
    // pivot value and object used for exchanges
    T pivot, temp;

    if (first == last)
        return last;
    else if (first == last-1)
        return first;
    else
    {
        mid = (last + first)/2;
        pivot = v[mid];

        // exchange the pivot and the low end of the range
        // and initialize the indices scanUp and scanDown.
        v[mid] = v[first];
        v[first] = pivot;

        scanUp = first + 1;
        scanDown = last - 1;

        // manage the indices to locate elements that are in
        // the wrong sublist; stop when scanDown <= scanUp
        for(;;)
        {
            // move up lower sublist; stop when scanUp enters
            // upper sublist or identifies an element >= pivot
            while (scanUp <= scanDown && v[scanUp] < pivot) // O(1)
                scanUp++; // O(1)

            // scan down upper sublist; stop when scanDown locates
            // an element <= pivot; we guarantee we stop at arr[first]
            while (pivot < v[scanDown]) // O(1)
                scanDown--; // O(1)

            // if indices are not in their sublists, partition complete
            if (scanUp >= scanDown)
                break;

            // indices are still in their sublists and identify
            // two elements in wrong sublists. exchange
            swap(v[scanUp], v[scanDown]);

            scanUp++;
        }
    }
}

```

```
        scanDown--;
    }

    // copy pivot to index (scanDown) that partitions sublists
    // and return scanDown
    v[first] = v[scanDown];
    v[scanDown] = pivot;
    return scanDown;
}
}
```

**Question:** How many times does each while loop execute (in the worst case)?

- 1
- n
- scanDown
- scanUp
- scanDown-scanUp
- scanDown+scanUp

**\*\*Answer:\*\***

Each time around the loop, we either increment  $scanUp$  or decrement  $scanDown$ . So the loop can only repeat  $scanDown_0 - scanUp_0$  times, where  $scanDown_0$  and  $scanUp_0$  denote the starting values of  $scanDown$  and  $scanUp$ .

[pivot4.cpp](#) +

```
template <typename T>
int pivotIndex(vector<T>& v, int first, int last)
{
    // index for the midpoint of [first,last) and the
    // indices that scan the index range in tandem
    int mid, scanUp, scanDown;
    // pivot value and object used for exchanges
    T pivot, temp;

    if (first == last)
        return last;
    else if (first == last-1)
        return first;
    else
    {
        mid = (last + first)/2;
        pivot = v[mid];

        // exchange the pivot and the low end of the range
        // and initialize the indices scanUp and scanDown.
        v[mid] = v[first];
        v[first] = pivot;

        scanUp = first + 1;
        scanDown = last - 1;

        // manage the indices to locate elements that are in
        // the wrong sublist; stop when scanDown <= scanUp
        for(;;)
        {
            // move up lower sublist; stop when scanUp enters
            // upper sublist or identifies an element >= pivot
            while (scanUp <= scanDown && v[scanUp] < pivot) // cond: O(1) #: (scanDown-scanUp)
                scanUp++; // O(1)

            // scan down upper sublist; stop when scanDown locates
            // an element <= pivot; we guarantee we stop at arr[first]
            while (pivot < v[scanDown]) // cond: O(1) #: (scanDown-scanUp)
                scanDown--; // O(1)

            // if indices are not in their sublists, partition complete
            if (scanUp >= scanDown)
                break;

            // indices are still in their sublists and identify
            // two elements in wrong sublists. exchange
            swap(v[scanUp], v[scanDown]);

            scanUp++;
            scanDown--;
        }

        // copy pivot to index (scanDown) that partitions sublists
        // and return scanDown
        v[first] = v[scanDown];
        v[scanDown] = pivot;
        return scanDown;
    }
}
```

So the while loops are each  $O(scanDown-scanUp)$ .

[pivot5.cpp](#) +

```
template <typename T>
int pivotIndex(vector<T>& v, int first, int last)
{
    // index for the midpoint of [first,last) and the
    // indices that scan the index range in tandem
    int mid, scanUp, scanDown;
    // pivot value and object used for exchanges
    T pivot, temp;

    if (first == last)
        return last;
    else if (first == last-1)
        return first;
    else
    {
        mid = (last + first)/2;
        pivot = v[mid];

        // exchange the pivot and the low end of the range
        // and initialize the indices scanUp and scanDown.
```

```

v[mid] = v[first];
v[first] = pivot;

scanUp = first + 1;
scanDown = last - 1;

// manage the indices to locate elements that are in
// the wrong sublist; stop when scanDown <= scanUp
for(;;)
{
    0(scanDown-scanUp)
    0(scanDown-scanUp)

    // if indices are not in their sublists, partition complete
    if (scanUp >= scanDown)
        break;

    // indices are still in their sublists and identify
    // two elements in wrong sublists. exchange
    swap(v[scanUp], v[scanDown]);

    scanUp++;
    scanDown--;
}

// copy pivot to index (scanDown) that partitions sublists
// and return scanDown
v[first] = v[scanDown];
v[scanDown] = pivot;
return scanDown;
}
}

```

So the body of the for loop is clearly  $O(\text{scanDown} - \text{scanUp})$ . How many times does this loop repeat? It's hard to say, but we don't actually need an exact count. Each time the two inner loops were executed, they may move `scanDown` and `scanUp` closer to one another. In addition, the final increments in the for loop move those variables still closer to one another. The *total* number of times the for loop body gets executed can be no more than `scanDown - scanUp` times, and the total number of times both while loop bodies can be executed is also `scanDown - scanUp`.

Of course, those loops also change the values of `scanDown` and `scanUp`, so it is more accurate to say that the total number of executions of those loops is the initial value of `scanDown - scanUp`.

Looking at the initialization before the loop, we see that these initial values are `last - 1` and `first + 1`, respectively. So the loop (and the `pivotIndex` algorithm) are  $O(\text{last} - \text{first})$ . In other words, this algorithm is linear in the number of elements to be partitioned.

## 2 Quick Sort

To actually sort data, we use the `pivotIndex` function within the `quicksort` routine shown here.

```

template <typename T>
void quicksort(vector<T>& v, int first, int last)
{
    // index of the pivot
    int pivotLoc;
    // temp used for an exchange when [first,last) has
    // two elements
    T temp;

    // if the range is not at least two elements, return
    if (last - first <= 1)
        return;

    // if sublist has two elements, compare v[first] and
    // v[last-1] and exchange if necessary
    else if (last - first == 2)
    {
        if (v[last-1] < v[first])
        {
            temp = v[last-1];
            v[last-1] = v[first];
            v[first] = temp;
        }
        return;
    }
    else
    {
        pivotLoc = pivotIndex(v, first, last);

        // make the recursive call
        quicksort(v, first, pivotLoc);

        // make the recursive call
        quicksort(v, pivotLoc + 1, last);
    }
}

```

The recursive routine is designed to sort all elements of an array from position *low* to *high*, inclusively.

- **First**, we check to be sure this is not a “degenerate” case of 0, 1, or 2 elements. We can resolve those cases immediately.
- **Second**, we use the `pivotIndex` function to partition the array.
- At the end of the previous step, we know that all of the elements on the left are less than any element on the right. Consequently, *if we could sort the left part and the right part separately, the array as a whole would be sorted.*

And so, we accomplish that separate sorting of the left and right parts by `recursively calling quicksort` on each of the two parts.

[Run this algorithm](#) until you are comfortable with your understanding of how it works.

## 3 Quick Sort Analysis

### 3.1 Best Case

```
qsl.cpp +
```

```
template <typename T>
void quicksort(vector<T>& v, int first, int last)
{
    // index of the pivot
    int pivotLoc;
    // temp used for an exchange when [first, last) has
    // two elements
    T temp;

    // if the range is not at least two elements, return
    if (last - first <= 1)
        return;

    // if sublist has two elements, compare v[first] and
    // v[last-1] and exchange if necessary
    else if (last - first == 2)
    {
        if (v[last-1] < v[first])
        {
            temp = v[last-1];
            v[last-1] = v[first];
            v[first] = temp;
        }
        return;
    }
    else
    {
        pivotLoc = pivotIndex(v, first, last);

        // make the recursive call
        quicksort(v, first, pivotLoc);

        // make the recursive call
        quicksort(v, pivotLoc +1, last);
    }
}
```

We don't often do a “best case” analysis, but it might help to understand the behavior of this particular algorithm.

Ideally, Each call to `pivotIndex` will divide the array exactly in half. Thus each recursive call will work on exactly half of the array.

`pivotIndex`, we have determined, is  $O(\text{last} - \text{first})$ . Suppose we start with  $N$  items. Partitioning this via `pivotIndex` takes  $O(N)$  time.

Then, in this best case, we would have two sub-arrays, each with  $N/2$  elements. Eventually, each of these will be partitioned at a cost of  $O(N/2) + O(N/2) = O(N)$  effort.

Each of those sub-arrays will, upon partitioning, yield a pair of sub-arrays of size  $N/4$ . So, we will, over the course of the entire algorithm, be asked 4 times to partition sub-arrays of size  $N/4$ , for a total cost of  $4 * O(N/4) = O(N)$ .

You can see what's happening. In general, at the  $k^{\text{th}}$  level of recursion, there will be  $2^k$  sub-arrays of size  $N/(2^k)$  to be pivoted, at a total cost of  $\sum_{i=1}^{2^k} O(N/(2^k)) = O(N)$ . This continues until  $N/(2^k)$  has been reduced to 1.

So all calls at on sub-arrays of the same size add up to  $O(N)$ . There are  $\log(N)$  levels, so the best case is  $O(N \log(N))$ .

### 3.2 Average Case

The average case for quick sort is  $O(N \log(N))$ . The proof of is rather heavy going, but is presented in the final section for those who are interested.

### 3.3 Worst Case

In the worst case, each call to `pivotIndex` would divide the array with one element on one side and all of the other elements on the other side. Then, when we do the recursive calls, one call returns immediately but the other has almost as much work left to do as before we did the pivot. The recursive calls go  $N - 1$  levels deep,

each applying a linear time pivot to its part of the array. Hence we get a total effort from these calls of

$$O(N-1) + O(N-2) + \dots + O(1) = O\left(\sum_{i=1}^{n-1} i\right) = O(N^2)$$

## 3.4 Great Average, Dreadful Worst

Quick sort therefore poses an interesting dilemma. Its average case is very fast. In addition, its multiplicative constant is also fairly low, so it tends to be the fastest, on average, of the known  $O(N \log N)$  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 near-worst-case behavior. That, in turn, depends upon the choice of pivot.

### 3.4.1 Choosing the Pivot

The code shown here selects the midpoint of the range as the pivot location.

```
pivot6.cpp +
```

```
template <typename T>
int pivotIndex(vector<T>& v, int first, int last)
{
    // index for the midpoint of [first, last) and the
    // indices that scan the index range in tandem
    int mid, scanUp, scanDown;
    // pivot value and object used for exchanges
    T pivot, temp;

    if (first == last)
        return last;
    else if (first == last-1)
        return first;
    else
    {
        mid = (last + first)/2;
        pivot = v[mid];

        // exchange the pivot and the low end of the range
        // and initialize the indices scanUp and scanDown.
        v[mid] = v[first];
        v[first] = pivot;

        scanUp = first + 1;
        scanDown = last - 1;

        // manage the indices to locate elements that are in
        // the wrong sublist; stop when scanDown <= scanUp
        for(;)
        {
            // move up lower sublist; stop when scanUp enters
            // upper sublist or identifies an element >= pivot
            while (scanUp <= scanDown && v[scanUp] < pivot)
                scanUp++;

            // scan down upper sublist; stop when scanDown locates
            // an element <= pivot; we guarantee we stop at arr[first]
            while (pivot < v[scanDown])
                scanDown--;

            // if indices are not in their sublists, partition complete
            if (scanUp >= scanDown)
                break;

            // indices are still in their sublists and identify
            // two elements in wrong sublists. exchange
            swap(v[scanUp], v[scanDown]);

            scanUp++;
            scanDown--;
        }

        // copy pivot to index (scanDown) that partitions sublists
        // and return scanDown
        v[first] = v[scanDown];
        v[scanDown] = pivot;
        return scanDown;
    }
}
```

Other choices that were once popular included choosing the first element or the last element of the range to serve as the pivot. You can see that, once we have chosen a pivot, the first thing we do is to move it “out of the way” by swapping it with the first element. So, if we simply chose the first [or last] element as the pivot, it would already be out of the way, and we could save that step.

These turned out to be poor choices, however, as the worst case would then occur whenever the array was already sorted or was in exactly reverse order. Those are, in practice, very common situations.

Choosing the midpoint is somewhat better. The worst case is somewhat more obscure and far less likely in practice. It's not entirely unlikely however – some tree traversal algorithms that we will cover in later lessons could give rise to listings of elements that would have the largest/smallest elements in each range at the midpoint, causing the  $O(N^2)$  behavior.

In practice then, the most commonly recommended choices seem to be:

- Select a position in the range to be sorted at random.

This doesn't change the fact that the worst case exists and is  $O(N^2)$ , but it reduces the probability of actually seeing the worst case and, because of the randomness, dramatically reduces the chances that any given application would generate patterns of data that would trigger the worst-case many times.

- Examine the first, last, and midpoint values in the range to be sorted. Choose for the pivot whichever of these three values lies between the other two (in value, not position). This is called the [median-of-three](#) rule.

The idea here is that we know that the pivot is neither the smallest nor largest value (though it could be the next-to-largest or next-to-smallest). Sorted arrays and reverse-sorted arrays will actually yield best-case behavior (because the mid-point value will be used in these cases). The  $O(N^2)$  worst case still exists, but it's not a common input pattern.

## 4 Optimizing the Algorithm

If you have [run the quick sort](#), you may have noticed that a lot of its work is spent on very small sub-arrays. These small arrays are also responsible for a good portion of the storage overhead of Quick sort — space on the run-time stack.

A significant speedup comes from only applying quick sort to fairly large arrays:

```
qs2.cpp +  
  
template <typename T>  
void quicksort(vector<T>& v, int first, int last)  
{  
    // index of the pivot  
    int pivotLoc;  
    // temp used for an exchange when [first,last) has  
    // two elements  
    T temp;  
  
    // if the range is not at least ten elements, return  
    if (last - first <= 10)  
        return;  
  
    // if sublist has two elements, compare v[first] and  
    // v[last-1] and exchange if necessary  
    else if (last - first == 2)  
    {  
        if (v[last-1] < v[first])  
        {  
            temp = v[last-1];  
            v[last-1] = v[first];  
            v[first] = temp;  
        }  
        return;  
    }  
    else  
    {  
        pivotLoc = pivotIndex(v, first, last);  
  
        // make the recursive call  
        quicksort(v, first, pivotLoc);  
  
        // make the recursive call  
        quicksort(v, pivotLoc +1, last);  
    }  
}  
template <typename T>  
void quicksortR(vector<T>& v, int first, int last)  
{  
    // index of the pivot  
    int pivotLoc;  
    // temp used for an exchange when [first,last) has  
    // two elements  
    T temp;  
  
    // if the range is not at least two elements, return  
    if (last - first <= 10)  
        return;  
  
    // if sublist has two elements, compare v[first] and  
    // v[last-1] and exchange if necessary  
    else if (last - first == 2)  
    {  
        if (v[last-1] < v[first])  
        {  
            temp = v[last-1];  
            v[last-1] = v[first];  
            v[first] = temp;  
        }  
    }
```

```

        return;
    }
else
{
    pivotLoc = pivotIndex(v, first, last);

    // make the recursive call
    quicksort(v, first, pivotLoc);

    // make the recursive call
    quicksort(v, pivotLoc +1, last);
}

void quicksort(vector<T>& v, int first, int last)
{
    quicksortR (v, first, last);
    insertionSort (v, first, last);
}

```

- This test used to be if `(last - first >= 1)`, so that we would quick sort any array of 2 or more elements. Now we change it to only quick sort arrays of more than 10 elements.

With this change, our recursive quick sort only guarantees that each element will be within 10 places of where it should be.

- Then we do a final “cleanup” by applying `insertionSort` to the array after all the recursion is done.

Although `insertionSort` is  $O(N^2)$  in its worst case, in the special case where all elements are no more than  $k$  places away from their final position, `insertionSort` is  $O(k * N)$ .

In this situation, we know that the recursive quick sort has brought each element to within 10 positions of its sorted position, so this `insertionSort` pass is  $O(10 * N) = O(N)$ . Since we’ve already expended an average of  $O(N \log(N))$  work before that pass, this cleanup phase is still efficient by comparison.

## 5 Average Case Analysis of Quick Sort

The average case run time of Quick sort is  $O(n \log n)$ . Proving this is really a bit much for this course, but we present the proof here for the curious:

- Assume a strategy for choosing pivots such that, after partitioning  $A$  into  $A1$  and  $A2$ , all lengths of  $A1$  from 0 to  $|A| - 1$  are equally likely.
- The running time of `quickSort` equals the time of its two recursive calls plus time to do the partition.
  - `pivotIndex` is  $O(\text{last} - \text{first})$ .

Suppose that we partition  $n$  elements into sub-arrays of length  $i$  and  $(n - i)$ .

Time  $T$  to sort the  $n$  elements is then:

$$T(n) = T(i) + T(n - i) + c * n$$

- This kind of formula is called a recurrence relation. They are very common in describing the performance of recursive routines.

Because  $i$  is equally likely to be any value from 0 to  $n - 1$ , the average (expected) value of  $T(i)$  is

$$E(T(i)) = 1/n \sum_{j=0}^{n-1} T(j)$$

Since  $n - i$  can take on the same values as  $i$ , and all such values are equally likely,

$$E(T(n - i)) = E(T(i))$$

On average, then

$$T(n) = 2/n \left( \sum_{j=0}^{n-1} T(j) \right) + c * n$$

Multiply through by  $n$ :

$$n * T(n) = 2 * \left( \sum_{j=0}^{n-1} T(j) \right) + c * n^2$$

$n$  is just a variable, so this must be true no matter what value we plug in for it. Try replacing  $n$  by  $n - 1$ .

$$(n - 1) * T(n - 1) = 2 * \left( \sum_{j=0}^{n-2} T(j) \right) + c * (n - 1)^2$$

So now both of these are true:

$$n * T(n) = 2 * \left( \sum_{j=0}^{n-1} T(j) \right) + c * n^2$$

$$(n - 1) * T(n - 1) = 2 * \left( \sum_{j=0}^{n-2} T(j) \right) + c * (n - 1)^2$$

Subtract the 2nd equation from the first, obtaining:

$$nT(n) - (n - 1)T(n - 1) = 2T(n - 1) + 2cn - c$$

Collect the  $T(n - 1)$  terms together and drop the  $-c$  term:

$$nT(n) = (n + 1)T(n - 1) + 2cn$$

Next we apply a standard technique for solving recurrence relations. Divide by  $n(n + 1)$  and “telescope”:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$

$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1}$$

$$\vdots$$

Note that most of the terms on the left will have appeared on the right in the previous equation, so if we were to add up all these equations, these terms would appear on both sides and could be dropped:

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c \sum_{j=3}^{n+1} 1/j$$

As  $n$  gets very large,  $\sum_{j=3}^{n+1} 1/j$  approaches  $\ln(n) + \gamma$ , where  $\gamma$  is Euler's constant, 0.577 . . .

Hence

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c * \ln(n) + 2c\gamma$$

$$= \ln(n) + c_2$$

$$= O(\ln(n))$$

and so  $T(n)$  is  $O(n \log(n))$ .

# Trees

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Tree Terminology](#)
  - [1.1 All in the Family](#)
  - [1.2 Binary Trees](#)
  - [1.3 Paths](#)
  - [1.4 Depth & Height](#)
- [2 Tree Traversal](#)
  - [2.1 Kinds of Traversals](#)
  - [2.2 Recursive Traversals](#)
  - [2.3 Level-Order Traversal](#)
- [3 Example: Processing XML](#)

Most of the data structures we have looked at so far have been devoted to keeping a collection of elements in some linear order.

Trees are the most common non-linear data structure in computer science. Trees are useful in representing things that naturally occur in hierarchies (e.g., many company organization charts are trees) and for things that are related in a “is-composed-of” or “contains” manner (e.g., this country is composed of states, each state is composed of counties, each county contains cities, each city contains streets, etc.)

Trees also turn out to be exceedingly useful in implementing associative containers like `std::set`. Properly implemented, a tree can lead to an implementation that can be both searched and inserted into in  $O(\log N)$  time. Compare this to the data structures we’ve seen so far, which may allow us to search in  $O(\log N)$  time but insert in  $O(N)$ , or insert in  $O(1)$  but search in  $O(N)$ .

## 1 Tree Terminology

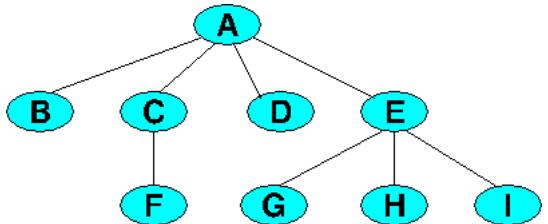
A [tree](#) is a collection of nodes.

If nonempty, the collection includes a designated node  $r$ , the [root](#), and zero or more (sub)trees  $T_1, T_2, \dots, T_k$ , each of whose roots are connected by an [edge](#) to  $r$ .

The collection of nodes shown here is a tree. We can designate A as the root, and we note that the collections of nodes  $\{B\}$ ,  $\{C, F\}$ ,  $\{D\}$ , and  $\{E, G, H, I\}$ , together with their edges, are all trees whose own roots are connected to A.

It’s a subtle, but important point to note that in discussing trees, we sometimes focus on the things connected to the root as individual nodes, and other times as entire trees.

The definition above focuses on a tree as a root plus a collection of (sub)trees. (The “sub” is just a convenience – there’s no formal definition of a “subtree” as opposed to a “tree”.)



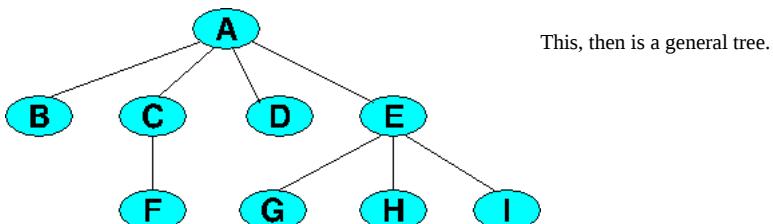
### 1.1 All in the Family

Focusing on a tree as a collection of nodes leads to some other terminology:

- Each node except the root has a [parent](#).
- Parent nodes have [children](#). Nodes without children are [leaves](#).
- Nodes with the same parent are [siblings](#).

### 1.2 Binary Trees

- A tree in which every parent has at most 2 children is a [binary tree](#).
- Trees in which parents [may](#) have more than 2 children are [general trees](#).



## 1.3 Paths

- A path from  $n_1$  to  $n_k$  is a sequence  $n_1, n_2, \dots, n_k$  such that

$\forall i, 1 \leq i < k, n_i$  is the parent of  $n_{i+1}$ .

◦ The length of a path is the number of edges in it.

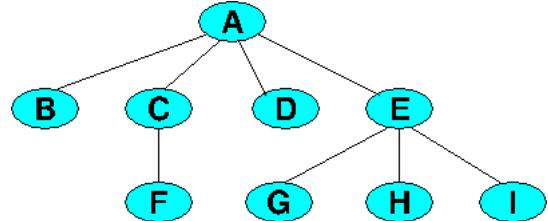
◦  $n_1$  is an ancestor of  $n_k$ .

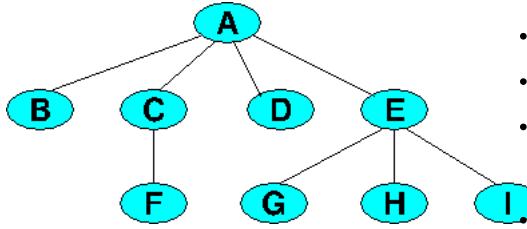
◦  $n_k$  is a descendant of  $n_1$ .

**Question:** Which of the following sequences of nodes are paths? (May be more than one)

- [C, A, E, G]
- [A, E, G]
- [E]
- []

\*\*Answer:\*\* +





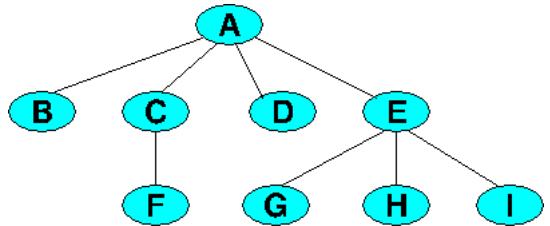
- [C, A, E, G] is not a path, because C is not a parent of A.
- [A, E, G] is a path. Each node in the sequence is a parent of the one that follows it.
- [E] is a path. The definition says a path is a sequence of nodes such that for each successive pair of nodes, the first is the parent of the other. Since there are no such pairs, the definition reduces to “a sequence of nodes” in this case.
- Empty sequences are paths.

## 1.4 Depth & Height

- The *depth* of a node is the length of the path from the root to that node.
- The *height* of a node is the length of the longest path from it to any leaf.
- The height of an empty tree is -1.

**Question:** What is the height of E?

\*\*Answer:\*\*



The height of E is 1, because the longest path from E to any leaf is of length 1. (If you answered “2”, remember that the length of a path is computed by counting the edges, not the nodes.)

## 2 Tree Traversals

Many algorithms for manipulating trees need to “traverse” the tree, to visit each node in the tree and process the data in that node. In this section, we’ll look at some prototype algorithms for traversing trees, mainly using recursion.

Later, we’ll look at how to devise iterators for tree traversal. But iterators are primarily for application code. The underlying implementation of a tree-based ADT will still need to employ the kinds of algorithms we are about to discuss.

### 2.1 Kinds of Traversals

- A pre-order traversal is one in which the data of each node is processed before visiting any of its children.
- A post-order traversal is one in which the data of each node is processed after visiting all of its children.
- An in-order traversal is one in which the data of each node is processed after visiting its left child but before visiting its right child.
  - This traversal is specific to binary trees.
- A level-order traversal is one in which all nodes of the same height are visited before any lower nodes.

Compilers, interpreters, spreadsheets, and other programs that read and evaluate arithmetic expressions often represent those expressions as trees. Constants and variables go in the leaves, and each non-leaf node represents the application of some operator to the subtrees representing its operands. The tree here, for example, shows the product of a sum and of a subtraction.

If we were to traverse this tree, printing each node as we visit it, we would get:

#### Pre-order

$* + 13 a - 1$

#### In-order

$13 + a * x - 1$

Compare this to  $((13+a)*(x-1))$ , the “natural” way to write this expression. You can see that in-order traversal yields the normal way to write this expression except for parentheses. If we made our output routine put parentheses around everything, we would actually get an algebraically correct expression.

When applied to a “binary search tree”, which we will introduce in a later lesson, in-order traversal processes the nodes in sorted order.

#### Post-order

$13 a + x 1 - *$

Post-order traversal yields post-fix notation, which in turn is related to stack-based algorithms for expression evaluation.

#### Level-order

$* + - 13 a x 1$

## 2.2 Recursive Traversals

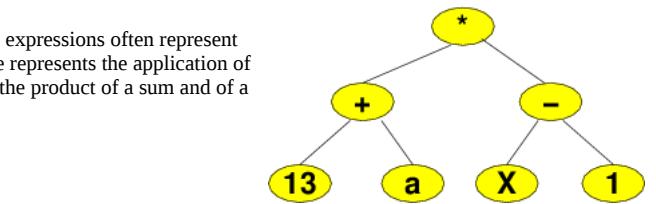
Let’s suppose that we have a binary tree whose nodes are declared as shown here.

This is a fairly typical binary tree structure, with a field for data and pointers for up to two children. If the node is a leaf, both the `left` and `right` pointers will be null. If the node has only one child, either the `left` or `right` will be null.

It’s fairly easy to write pre-, in-, and post-order traversal algorithms using recursion.

```
template <typename T>
void basicTraverse (tnode<T>* t)
{
  if (t != 0)
  {
    basicTraverse(t->left);
    basicTraverse(t->right);
  }
}
```

This is the basic structure for a recursive traversal. If this function is called with a null pointer, we do nothing. But if we have a real pointer to some node, we invoke the function recursively on the left and right subtrees. In this manner, we will eventually visit every node in the tree.



```
// represents a node in a binary tree
template <typename T>
class tnode
{
public:
  // tnode is a class implementation structure. making the
  // data public simplifies building class functions
  T nodeValue;
  tnode<T> *left, *right;

  // default constructor. data not initialized
  tnode()
  {}

  // initialize the data members
  tnode (const T& item, tnode<T> *lptr = nullptr,
         tnode<T> *rptr = nullptr):
    nodeValue(item), left(lptr), right(rptr)
  {};
};
```

We can generalize this to trees with any number of children by moving the child pointers into an array, list, or vector. (We could use this for binary trees as well, treating `children[0]` as the left child and `children[1]` as the right child.)

Now the basic traversal looks like:

```
template <typename T>
void basicTraverse (treenode<T>* t)
{
    if (t != 0)
    {
        for (treenode<T>* child: t->children)
            basicTraverse(child);
    }
}
```

```
// represents a node in a general tree
template <typename T>
class treenode
{
public:
    T nodeValue;
    std::vector<treenode<T>> children;

    treenode (const T& item = T()):
        nodeValue(item)
    {};
};
```

The problem with these basic traversal algorithms is that they don't *do* anything with the data in the trees.

## 2.2.1 Pre-Order Traversals

But we can convert the basic traversal into a pre-order traversal by applying the rule:

process the node *before* visiting its children

```
template <typename T>
void preorder(tnode<T> *t)
{
    // the recursive scan terminates on an empty subtree
    if (t != nullptr)
    {
        doSomethingWith (t->nodeValue);
        preorder(t->left);    // descend left
        preorder(t->right);   // descend right
    }
}
```

## 2.2.2 Post-Order Traversals

We get a post-order traversal by applying the rule:

process the node *after* visiting its children

```
template <typename T>
void postorder(tnode<T> *t)
{
    // the recursive scan terminates on an empty subtree
    if (t != nullptr)
    {
        postorder(t->left);    // descend left
        postorder(t->right);   // descend right
        doSomethingWith (t->nodeValue);
    }
}
```

## 2.2.3 In-Order Traversals

And we get an in-order traversal by applying the rule:

process the node *after* visiting its left descendants and *before* visiting its right descendants.

```
template <typename T>
void inorder(tnode<T> *t)
{
    // the recursive scan terminates on an empty subtree
    if (t != nullptr)
    {
        inorder(t->left);    // descend left
        doSomethingWith (t->nodeValue);
        inorder(t->right);   // descend right
    }
}
```

Note that, while pre- and post- order traversals can be applied to trees with any number of children, in-order really only makes sense when applied to binary trees.

Try running these traversals on some trees until you are comfortable with these.

## 2.3 Level-Order Traversal

This form of traversal is different from the others. In a level-order traversal, we visit the root, then all elements 1 level below the root, then all elements two levels below the root, and so on. Unlike the other traversals, elements visited successively may not be related in the parent-child sense except for having the root as a common (and possibly distant) ancestor.

To program a level-order traversal, we use a queue to keep track of nodes at the next lower level that need to be visited.

```
ordertraverse.cpp +  
  
template <typename T>  
void levelOrder (tnode<T>* t)  
{  
    // store siblings of each node in a queue so that they are  
    // visited in order at the next level of the tree  
    queue<tnode<T>> q;  
    tnode<T> *p;  
  
    // initialize the queue by inserting the root in the queue  
    q.push(t);  
  
    // continue the iterative process until the queue is empty  
    while(!q.empty())  
    {  
        // delete front node from queue and output the node value  
        p = q.front();  
        q.pop();  
        doSomethingWith (t->nodeValue);  
  
        // if a left child exists, insert it in the queue  
        if(p->left != nullptr)  
            q.push(p->left);  
        // if a right child exists, insert next to its sibling  
        if(p->right != nullptr)  
            q.push(p->right);  
    }  
}
```

Try [running these traversals](#) on some trees to get an appreciation of how they work.

## 3 Example: Processing XML

XML is a markup language used for exchanging data among a wide variety of programs on the web. It is flexible enough to represent almost any kind of data.

In XML, data is described by structures consisting of nested “elements” and text. An element is described as an opening “tag”, the contents of the element, and a closing tag.

Tags are written inside < > brackets. Inside the brackets, an opening tag has a tag name followed by zero or more “attributes”. An attribute is a name="value" pair, with the value inside quotes. Closing tags have no attributes, and are indicated by placing the tag name immediately after a '/' character.

Finally, if an element has no text and no internal elements, it can be written as

```
<tag attributes></tag>
```

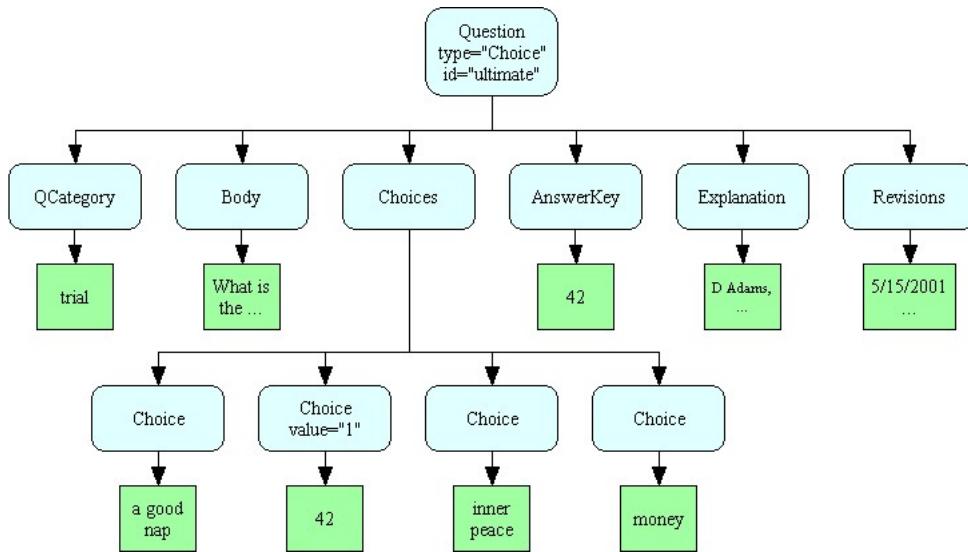
or abbreviated as

```
<tag attributes />
```

As an example of XML, here is a question from a quiz, in the XML form that serves as input to an on-line quiz generator:

```
<Question type="Choice" id="ultimate">  
    <QCategory>trial</QCategory>  
    <Body>What is the answer to the ultimate question of life,  
        the universe, and everything?  
    </Body>  
    <Choices>  
        <Choice>a good nap</Choice>  
        <Choice value="1">42</Choice>  
        <Choice>inner peace</Choice>  
        <Choice>money</Choice>  
    </Choices>  
    <AnswerKey>42</AnswerKey>  
    <Explanation>  
        D Adams, The Hitchhiker's Guide to the Galaxy  
    </Explanation>  
    <Revisions>5/15/2001 1:35:29 PM</Revisions>  
</Question>
```

Although it may not be obvious, XML actually describes a tree structure.



For example, the structure above shows that all the elements are inside a “Question”. One of those elements inside the Question is a “Choices” element, and each individual “Choice” occurs inside there. We can diagram this tree structure as shown here.

If this description of XML sounds familiar, you may be noticing it’s close relationship to HTML, the web page markup language. HTML is almost a kind of XML in which all the elements are chosen as descriptions of different text formatting properties and different portions of a web page. Actually, HTML does not follow all the rules of XML (attributes can often be written without placing the value in quotes, empty elements can be written as <tagname> instead of <tagname/>, and some non-empty elements can be written without a closing </tagname>). HTML and XML are closely related enough that there are a number of programs, such as the Unix `tidy` command, that will rewrite an HTML page into “proper” XML.

For example, the text shown here can be produced by the following XML-legal HTML (a.k.a., [xhtml](#)):

<b>Just a test</b>	<pre>&lt;html&gt;   &lt;head&gt;     &lt;title&gt;Just a Test&lt;/title&gt;   &lt;/head&gt;    &lt;body&gt;     &lt;h1&gt;Just a test&lt;/h1&gt;     &lt;p&gt;Nothing much to       see &lt;a href="test.html"&gt;here&lt;/a&gt;.     &lt;/p&gt;      &lt;p&gt;       Move &lt;a href="nextpage.html"&gt;along&lt;/a&gt;.     &lt;/p&gt;   &lt;/body&gt; &lt;/html&gt;</pre>
--------------------	--

The tree structure for that HTML page is shown here.

Now suppose that we wished to write a program that would read a web page and print a list of all links (<a>elements with `href=` attributes). Although we could certainly write such a program entirely from scratch, XML is popular enough that we should expect there to be at least a few libraries that will include code to read web pages and to prepare the tree structure (called the [DOM](#) for “Domain Object Model”) for us.

```
/** Example of tree manipulation using XML documents */

#include <iostream>

using namespace std;

#include <xercesc/parsers/XercesDOMParser.hpp>
#include <xercesc/dom/DOM.hpp>
#include <xercesc/sax/HandlerBase.hpp>
#include <xercesc/util/XMLString.hpp>
#include <xercesc/util/PlatformUtils.hpp>

using namespace XERCES_CPP_NAMESPACE;

DOMDocument* readXML (const char *xmlFile)
{
  :
```

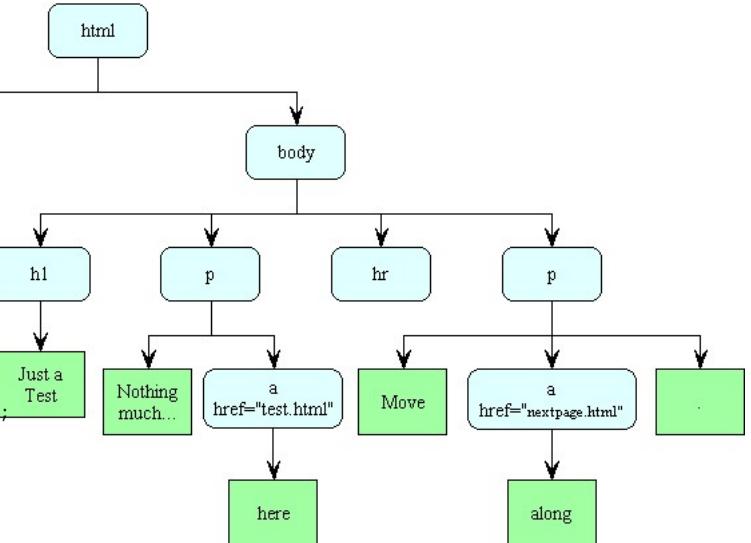
```

}

string getHrefAttribute (DOMNode* linkNode)
{
    DOMElement* linkNodeE = (DOMElement*)linkNode;
    const XMLCh* href = XMLString::transcode("href");
    const XMLCh* attributeValue = linkNodeE->getAttribute(href);
    return string(XMLString::transcode(attributeValue))>>read
}

void processTree (DOMNode *tree)
{
    if (tree != nullptr)
    {
        if (tree->getNodeType() == DOMNode::ELEMENT_NODE)
        {
            const XMLCh* elName = tree->getNodeName();
            const XMLCh* aName = XMLString::transcode("a");
            if (XMLString::equals(elName, aName))
                cout << "Link to " << getHrefAttribute(tree) << endl;
        }
        for (DOMNode* child = tree->getFirstChild();
             child != nullptr; child = child->getNextSibling())
            processTree(child);
    }
}

```



```

int main(int argc, char **argv)
{
    if (argc != 2)
    {
        cerr << "usage: " << argv[0] << " xmlfile" << endl;
        return 1;
    }

    // Initialize the Xerces XML C++ library
    try {
        XMLPlatformUtils::Initialize();
    }
    catch (const XMLException& toCatch) {
        char* message = XMLString::transcode(toCatch.getMessage());
        cout << "Error during initialization! :\n"
            << message << "\n";
        XMLString::release(&message);
        return 1;
    }

    DOMDocument* doc = readXML(argv[1]);
    if (doc == 0)
    {
        cerr << "Could not read " << argv[1] << endl;
        return 2;
    }
    processTree (doc->getDocumentElement());

    // Cleanup
    doc->release();
    return 0;
}

```

Here is my code for listing the links in an HTML file, based upon the [Xerces-C++](#) library.

```

void processTree (DOMNode *tree)
{
    if (tree != nullptr)
    {
        if (tree->getNodeType() == DOMNode::ELEMENT_NODE)
        {
            const XMLCh* elName = tree->getNodeName();
            const XMLCh* aName = XMLString::transcode("a");
            if (XMLString::equals(elName, aName))
                cout << "Link to " << getHrefAttribute(tree) << endl;
        }
        for (DOMNode* child = tree->getFirstChild();
             child != nullptr; child = child->getNextSibling())
            processTree(child);
    }
}

```

If this code is compiled and run on the HTML page we have just seen, it would print

```

Link to test.html
Link to nextpage.html

```

**Question:** Now, this code features an interface that you have never seen before, and a lot of the details are bound to look mysterious. Nonetheless, if you look at the `processTree` function, you can readily tell that this function works by

- pre-order traversal
- in-order traversal
- post-order traversal
- level-order traversal

**\*\*Answer:\*\***

The code for this function has the general form

```
void processTree (DOMNode *tree)
{
    if (tree != nullptr)
    {
        process-this-node;

        for (DOMNode* child = tree->getFirstChild();
            child != nullptr; child = child->getNextSibling())
            processTree(child);
    }
}
```

and the loop is clearly designed to visit each child in turn, applying the same processing step to it, but *after* having processed the current node.

That makes it a pre-order traversal.

# Binary Search Trees

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Definition: Binary Search Trees](#)
  - [1.1 The Binary Search Tree ADT](#)
- [2 Implementing Binary Search Trees](#)
  - [2.1 Searching a Binary Tree](#)
  - [2.2 Inserting into Binary Search Trees](#)
  - [2.3 Deletion](#)
- [3 How Fast Are Binary Search Trees?](#)
  - [3.1 Balancing](#)
  - [3.2 Performance](#)
  - [3.3 Average-Case](#)
  - [3.4 Can We Avoid the Worst Case?](#)

A tree in which every parent has at most 2 children is a [binary tree](#).

The most common use of binary trees is for ADTs that require frequent searches for arbitrary keys.

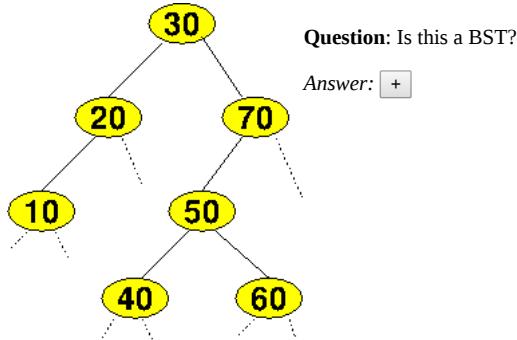
- E.g., sets, maps

For this we use a special form of binary tree, the binary search tree.

## 1 Definition: Binary Search Trees

A binary tree  $T$  is a [binary search tree](#) if, for each node  $n$  with children  $T_L$  and  $T_R$ :

- The value in  $n$  is greater than the values in every node in  $T_L$ .
- The value in  $n$  is less than the values in every node in  $T_R$ .
- Both  $T_L$  and  $T_R$  are binary search trees.



Yes, this is a binary search tree. 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.

## 1.1 The Binary Search Tree ADT

Let's look at the basic interface for a binary search tree, from your textbook:

```
BinarySearchTree.h +
```

```
#ifndef BINARY_SEARCH_TREE_H
#define BINARY_SEARCH_TREE_H

#include "dsexceptions.h"
#include <algorithm>
using namespace std;

// BinarySearchTree class
//
// CONSTRUCTION: zero parameter
//
// ***** PUBLIC OPERATIONS *****
// void insert( x )      --> Insert x
// void remove( x )      --> Remove x
// bool contains( x )    --> Return true if x is present
// Comparable findMin( ) --> Return smallest item
// Comparable findMax( ) --> Return largest item
// boolean isEmpty( )    --> Return true if empty; else false
// void makeEmpty( )     --> Remove all items
// void printTree( )     --> Print tree in sorted order
// ***** ERRORS *****
// Throws UnderflowException as warranted

template <typename Comparable>
class BinarySearchTree            ③
{
public:
    BinarySearchTree( ) : root{ nullptr }
    {
    }

    /**
     * Copy constructor
     */
    BinarySearchTree( const BinarySearchTree & rhs ) : root{ nullptr }
    {
        root = clone( rhs.root );
    }

    /**
     * Move constructor
     */
    BinarySearchTree( BinarySearchTree && rhs ) : root{ rhs.root }
    {
        rhs.root = nullptr;
    }

    /**
     * Destructor for the tree
     */
    ~BinarySearchTree( )
    {
        makeEmpty( );
    }

    /**
     * Copy assignment
     */
    BinarySearchTree & operator=( const BinarySearchTree & rhs )
    {
        BinarySearchTree copy = rhs;
        std::swap( *this, copy );
        return *this;
    }

    /**
     * Move assignment
     */
    BinarySearchTree & operator=( BinarySearchTree && rhs )
    {
        std::swap( root, rhs.root );
        return *this;
    }

    /**
     * Find the smallest item in the tree.
     * Throw UnderflowException if empty.
     */
    const Comparable & findMin( ) const
    {
```

```

    if( isEmpty( ) )
        throw UnderflowException{ };
    return findMin( root )->element;
}

< /**
 * Find the largest item in the tree.
 * Throw UnderflowException if empty.
 */
const Comparable & findMax( ) const
{
    if( isEmpty( ) )
        throw UnderflowException{ };
    return findMax( root )->element;
}

< /**
 * Returns true if x is found in the tree.
 */
bool contains( const Comparable & x ) const ④
{
    return contains( x, root );
}

< /**
 * Test if the tree is logically empty.
 * Return true if empty, false otherwise.
 */
bool isEmpty( ) const
{
    return root == nullptr;
}

< /**
 * Print the tree contents in sorted order.
 */
void printTree( ostream & out = cout ) const
{
    if( isEmpty( ) )
        out << "Empty tree" << endl;
    else
        printTree( root, out );
}

< /**
 * Make the tree logically empty.
 */
void makeEmpty( )
{
    makeEmpty( root );
}

< /**
 * Insert x into the tree; duplicates are ignored.
 */
void insert( const Comparable & x ) ⑤
{
    insert( x, root );
}

< /**
 * Insert x into the tree; duplicates are ignored.
 */
void insert( Comparable && x )
{
    insert( std::move( x ), root );
}

< /**
 * Remove x from the tree. Nothing is done if x is not found.
 */
void remove( const Comparable & x ) ⑥
{
    remove( x, root );
}

private:
struct BinaryNode ①
{
    Comparable element;
    BinaryNode *left;
    BinaryNode *right;

    BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt )
        : element{ theElement }, left{ lt }, right{ rt } { }

    BinaryNode( Comparable && theElement, BinaryNode *lt, BinaryNode *rt )
        : element{ std::move( theElement ) }, left{ lt }, right{ rt } { }
};

BinaryNode *root;

```

```


/*
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( Comparable & x, BinaryNode * & t )    ⑦
{
    if( t == nullptr )
        t = new BinaryNode{ x, nullptr, nullptr };
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );
    else
        ; // Duplicate; do nothing
}

/*
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( Comparable && x, BinaryNode * & t )
{
    if( t == nullptr )
        t = new BinaryNode{ std::move( x ), nullptr, nullptr };
    else if( x < t->element )
        insert( std::move( x ), t->left );
    else if( t->element < x )
        insert( std::move( x ), t->right );
    else
        ; // Duplicate; do nothing
}

/*
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void remove( Comparable & x, BinaryNode * & t )
{
    if( t == nullptr )
        return; // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != nullptr & t->right != nullptr ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;
        t = ( t->left != nullptr ) ? t->left : t->right;
        delete oldNode;
    }
}

/*
 * Internal method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */
BinaryNode * findMin( BinaryNode *t ) const
{
    if( t == nullptr )
        return nullptr;
    if( t->left == nullptr )
        return t;
    return findMin( t->left );
}

/*
 * Internal method to find the largest item in a subtree t.
 * Return node containing the largest item.
 */
BinaryNode * findMax( BinaryNode *t ) const
{
    if( t != nullptr )
        while( t->right != nullptr )
            t = t->right;
    return t;
}

/*
 * Internal method to test if an item is in a subtree.
 * x is item to search for.


```

```

    * t is the node that roots the subtree.
    */
bool contains( const Comparable & x, BinaryNode *t ) const
{
    if( t == nullptr )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true;      // Match
}
***** NONRECURSIVE VERSION*****
bool contains( const Comparable & x, BinaryNode *t ) const
{
    while( t != nullptr )
        if( x < t->element )
            t = t->left;
        else if( t->element < x )
            t = t->right;
        else
            return true;      // Match

    return false;      // No match
}
*****
/** Internal method to make subtree empty.
 */
void makeEmpty( BinaryNode * &t )
{
    if( t != nullptr )
    {
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
    }
    t = nullptr;
}

/** Internal method to print a subtree rooted at t in sorted order.
 */
void printTree( BinaryNode *t, ostream &out ) const
{
    if( t != nullptr )
    {
        printTree( t->left, out );
        out << t->element << endl;
        printTree( t->right, out );
    }
}

/** Internal method to clone subtree.
 */
BinaryNode * clone( BinaryNode *t ) const
{
    if( t == nullptr )
        return nullptr;
    else
        return new BinaryNode{ t->element, clone( t->left ), clone( t->right ) };
};

#endif

```

Some points of note:

- ① : The nested `BinaryNode` struct implements individual tree nodes. It shows the very characteristic structure of binary tree nodes: a data member to hold the “real” data value, and a pair of pointers to other tree nodes, one for the left subtree and one for the right subtree.

It is private here so that applications cannot get access to the internals of this tree.

- That may be overkill: making the pointer to the root ② private would probably have sufficed. It would mean that application programmers could create and access their own tree node objects, but would still have no access to the specific nodes that make up this particular tree.
- ③ : The `BinarySearchTree` template represents the entire tree (the whole collection of related nodes), with functions for searching, insertion, iteration, etc..
- Our primary focus in this lecture will be on the `contains` ④ , `insert` ⑤ , and `remove` ⑥ functions.
- Many of the functions making up this tree are implemented as a simple public function (e.g., ⑦) that passes the tree root to a similarly named, private, “internal” function (e.g., ⑧) that uses recursion (starting from that root) to perform to do the actual work.

## 2 Implementing Binary Search Trees

Since you have, presumably, read your text's discussion of how to implement BSTs, I'm mainly going to hit the high points.

## 2.1 Searching a Binary Tree

We'll start by reviewing the basic searching algorithm.

```
/**  
 * Returns true if x is found in the tree.  
 */  
bool contains( const Comparable & x ) const ④  
{  
    return contains( x, root );  
}
```

The tree's contains operation works by using a private utility function, also named contains, to find the node containing the desired data by starting a search from the root.

We search a tree by comparing the value we're searching for to the "current" node,  $t$ . If the value we want is smaller, we look in the left subtree. If the value we want is larger, we look in the right subtree.

You may note that this algorithm bears a certain resemblance to the [binary search](#) algorithm we studied earlier in the semester. We shall see shortly that the performance of both search algorithms on a collection of  $N$  items is  $O(\log N)$ , but that binary trees support faster insertion operations, allowing us to build the searchable collection in less time than when using binary search over sorted arrays.

You can [run this algorithm](#) to see how it works.

```
/**  
 * Internal method to test if an item is in a subtree.  
 * x is item to search for.  
 * t is the node that roots the subtree.  
 */  
bool contains( const Comparable & x, BinaryNode *t ) const  
{  
    if( t == nullptr )  
        return false;  
    else if( x < t->element )  
        return contains( x, t->left );  
    else if( t->element < x )  
        return contains( x, t->right );  
    else  
        return true; // Match  
}
```

## 2.2 Inserting into Binary Search Trees

[bstInsert.cpp](#) +

```
/**  
 * Insert x into the tree; duplicates are ignored.  
 */  
void insert( const Comparable & x ) ①  
{  
    insert( x, root );  
}  
  
:  
  
/**  
 * Internal method to insert into a subtree.  
 * x is the item to insert.  
 * t is the node that roots the subtree.  
 * Set the new root of the subtree.  
 */  
void insert( const Comparable & x, BinaryNode * & t )  
{  
    if( t == nullptr )  
        t = new BinaryNode{ x, nullptr, nullptr }; ③  
    else if( x < t->element )  
        insert( x, t->left );  
    else if( t->element < x )  
        insert( x, t->right );  
    else  
        ; // Duplicate; do nothing  
}
```

We start, again, with a public function ① that simply passes the buck to a recursive version, telling it to start from the root.

Note that the recursive function receives this pointer as a reference to a pointer ②, meaning that it can change the value of the pointer that it was given. It does this specifically when our traversal brings us to a null pointer in the tree ③, indicating that this is the place where we want to insert a new tree node with a copy of the data that we are trying to insert.

But how do we *find* the place at which to insert that new node? Basically, we ask "where would we go if we were searching for this data in the tree?" The remaining code in this function is almost exactly copied from the search code in the earlier contains function.

You might want to [run this algorithm](#) and 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 "randomly" ordered data.

## 2.3 Deletion

Removing a value starts, again, with a public function that simply passes the job to a private recursive one, telling it to start from the root.

```
/**  
 * Remove x from the tree. Nothing is done if x is not found.  
 */  
void remove( const Comparable & x ) ⑥  
{  
    remove( x, root );  
}
```

Here is the recursive part of the remove algorithm.

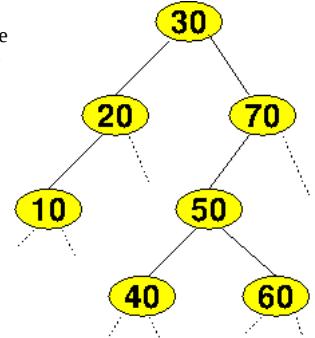
Looking first at the beginning of the function, we see the by-now-familiar search for the desired value. So eventually, if the data we said to remove is really in the tree, we should find it.

```
/**  
 * Internal method to remove from a subtree.  
 * x is the item to remove.  
 * t is the node that roots the subtree.  
 * Set the new root of the subtree.  
 */  
void remove( const Comparable & x, BinaryNode * & t )  
{  
    if( t == nullptr )  
        return; // Item not found; do nothing  
    if( x < t->element )  
        remove( x, t->left );  
    else if( t->element < x )  
        remove( x, t->right );  
    else if( t->left != nullptr && t->right != nullptr ) // Two children  
    {  
        t->element = findMin( t->right )->element;  
        remove( t->element, t->right );  
    }  
    else  
    {  
        BinaryNode *oldNode = t;  
        t = ( t->left != nullptr ) ? t->left : t->right;  
        delete oldNode;  
    }  
}
```

Well, what do we do with it when we find it? Well, we *can't* just delete the tree node. Take a look at this tree. If we were to remove 10, 40, or 60 by simply deleting the tree node, that might work. But deleting any other node would break the tree into two or three pieces, rendering it useless.

So, we'll need to be careful here. Let's break this problem down into cases:

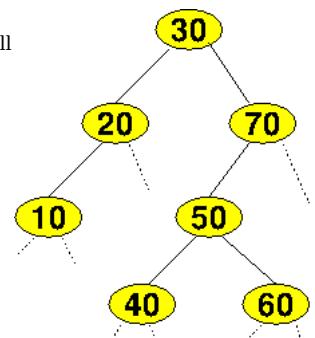
- Removing a leaf
- Removing a node that has only one child
  - only a left child
  - only a right child
- Removing a node that has two children



### 2.3.1 Removing a Leaf

**Question** Suppose we wanted to remove the “40” from this tree. What would we have to do so that the remaining nodes would still be a valid BST?

Answer +



*Nothing at all!*

If we simply delete this node (setting the pointer to it from its parent to `nullptr`), what's left would still be a perfectly good binary search tree — it would satisfy all the BST ordering requirements.

Now, take a look at the `remove` function.

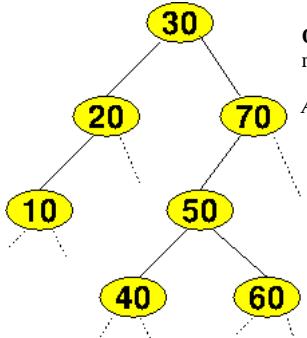
[bstErase3.cpp](#) +

```
/*
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void remove( const Comparable & x, BinaryNode * & t )
{
    if( t == nullptr )
        return; // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != nullptr && t->right != nullptr ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;
        t = ( t->left != nullptr ) ? t->left : t->right;
        delete oldNode;
    }
}
```

Look at the “leaf” case code, and you can see that all we do is to delete the node. We reach this code when `t` points to a leaf that contains the data we want to remove. In that case, we replace the address in `t` by `t->right`. If `t` is pointing to a leaf, then `t->right` is null, so we wind up writing a null pointer into the parent node, replacing whichever of its two children pointers was the one that we followed to get to `t`.

So if we are removing a tree leaf, we “replace” it by a null pointer.

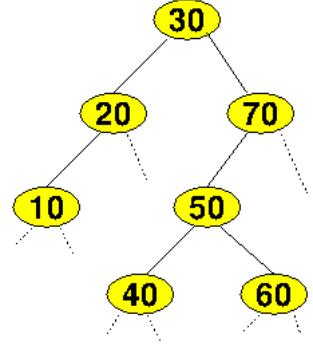
### 2.3.2 Removing A Non-Leaf Node with a Null Right Child



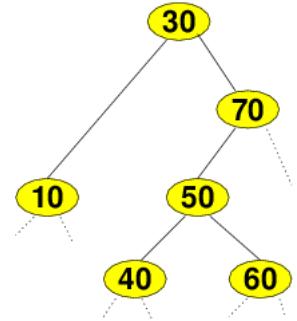
**Question** Suppose we wanted to remove the “20” or the “70” from this tree. What would we have to do so that the remaining nodes would still be a valid BST?

**Answer** +

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 reroute the pointer from the parent ("30") to the node we want to remove, making that pointer point directly to the child of the node we are going to remove.



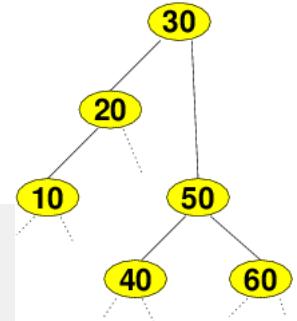
For example, starting from the tree shown here, verify for yourself that, if we remove 20:



or 70:

in this manner, that the results are still valid BSTs.

Looking again at the `remove` function,



```

bstErase3.cpp +
```

```

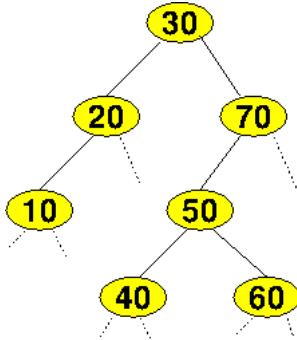

/*
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void remove( const Comparable & x, BinaryNode * & t )
{
    if( t == nullptr )
        return; // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != nullptr && t->right != nullptr ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;
        t = ( t->left != nullptr ) ? t->left : t->right;
        delete oldNode;
    }
}


```

we talked about [this code](#) in for the leaf case, but we also come to the same code when the node with the desired data has one null pointer.

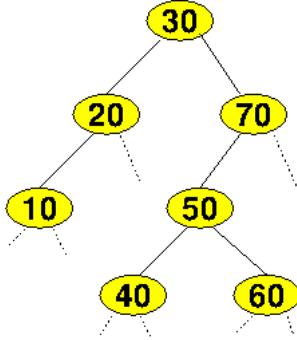
If, in this case, the right child is null but the left child is not, then we replace the parent's pointer to this node, `t`, by `t->left`, so it winds up pointing directly to the only child of the node holding the data we want to remove.

### 2.3.3 Removing A Non-Leaf Node with a Null Left Child



This tree does not feature any non-leaf nodes with null left children, but examination of that [same code](#) will show that there is a symmetry with the prior case. If  $t \rightarrow \text{left}$  is null but  $t \rightarrow \text{right}$  is not, then we force the parent's pointer  $t$  to change to point to  $t \rightarrow \text{right}$ .

## 2.3.4 Removing a Node with Two Non-Null Children



Suppose we wanted to remove the “50” or the “30” from this tree. What would we have to do so that the remaining nodes would still be a valid BST?

This is a hard case. Clearly, if we remove either the “50” or “30” nodes, we break the tree into pieces, with no obvious place to put the now-detached subtrees.

So let’s take a different tack. Instead of deleting this node, is there some other data value that we could put into that node that would preserve the BST ordering (all nodes to the left must be less, all nodes to the right must be greater or equal)?

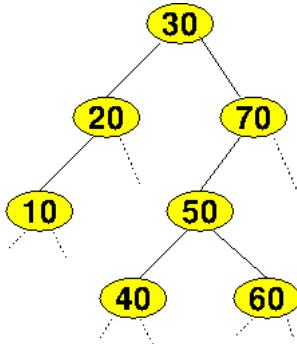
There are, in fact, two values that we could safely put in there: the smallest value from the right subtree, or the largest value from the left subtree.

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

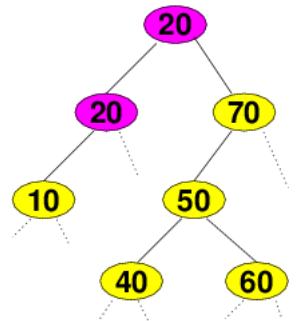
We can find the smallest value on the right by

- taking one step to the right
- then running as far down to the left as we can go



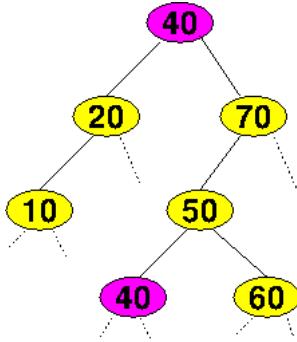
Now, if we replace “30” by ...

... the largest value from the left:



or by the smallest value from the right,

the results are properly ordered for a BST, except, arguably, for the node we just copied the value from. But since that node is now redundant, we can delete it from the tree.



And here's the best part. Since we find the node to copy from by running as far as we can go in one direction or the other, we know that the node we copied from has at least 1 null child pointer (otherwise we would have kept running past it). So removing it from the tree will always fall into one of the earlier, simpler cases (leaf or only one child).

Again, take a look at the code for removing a node.

[bstErase4.cpp](#) +

```
/*
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void remove( Comparable & x, BinaryNode * & t )
{
    if( t == nullptr ) // Item not found; do nothing
        return;
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != nullptr && t->right != nullptr ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;
        t = ( t->left != nullptr ) ? t->left : t->right;
        delete oldNode;
    }
}
```

- This is the test to see if we are actually trying to remove from a node with two non-null children.
- This does the “step to the right, then run to the left” behavior we have just described in order to find the replacement value. That value is then used to replace the value in this node, the one we want to remove.
- The remaining code then removes the replacement value from the right subtree.

Try [running this algorithm](#) on a variety of trees and nodes. Try to observe each of the major cases, as outlined here, in action.

## 3 How Fast Are Binary Search Trees?

Each step in the BST insert and contains algorithms move one level deeper in the tree. Similarly, in remove, the only part that is not constant time is the “running down the tree” to find the smallest value to the right.

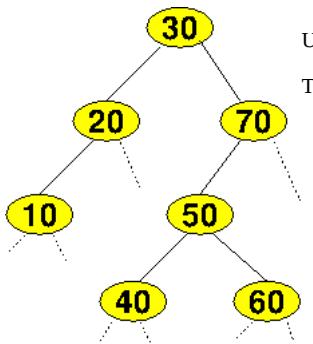
The number of recursive calls/loop iterations in all these algorithms is therefore no greater than the height of the tree.

But how high can a BST be?

That depends on how well the tree is “balanced”.

### 3.1 Balancing

A binary tree is [balanced](#) if for every interior node, the height of its two children differ by at most 1.

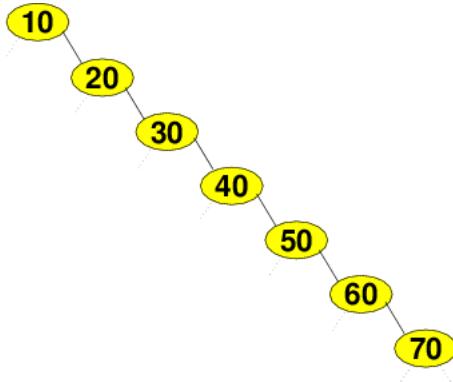


Unbalanced trees are easy to obtain.

This is a BST.

But, so is this!

The shape of the tree depends upon the order of insertions. Try [running the BST demo again](#). This time, clear the tree and then insert the values 1, 2, 3, 4, 5. Then clear the tree again and insert the values 8, 6, 4, 2.



The worst case behavior for binary search trees is when the data being inserted is already in order (or in reverse order). In that case, the tree *degenerates* into a sorted linked list.

The best case is when the inserted data yields a tree that is *balanced*, meaning that, for each node, the heights of the node's children are nearly the same.

## 3.2 Performance

Consider the contains operation on a nearly balanced tree with N nodes.

**Question:** What is the complexity of the best case?

- $O(1)$
- $O(\log N)$
- $O(N)$
- $O(N \log N)$
- $O(N^2)$

\*\*Answer:\*\* +

```
bool contains( const Comparable & x, BinaryNode *t ) const
{
    if( t == nullptr )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true;      // Match
}
```

In the best case, we find what we're looking for in the root of the tree. That's  $O(1)$  time.

**Question:** Consider the `contains` operation on a nearly balanced tree with  $N$  nodes.

What is the complexity of the worst case?

- $O(1)$
- $O(\log N)$
- $O(N)$
- $O(N \log N)$
- $O(N^2)$

**\*\*Answer:\*\***

The contains operation starts at the root and moves down one level each recursion. So it is, in the worst case,  $O(h)$  where  $h$  is the height of the tree.

But how high is a balanced tree?

A nearly balanced tree will be height  $\log N$ .

Consider a tree that is completely balanced and has its lowest level full. Since every node on the lowest level shares a parent with one other, there will be exactly half as many nodes on the next-to-lowest level as on the lowest. And, by the same reasoning, each level will have half as many nodes as the one below it, until we finally get to the single root at the top of the tree.

So a balanced tree has height  $\log N$ , and searching a balanced binary tree would be  $O(\log N)$ .

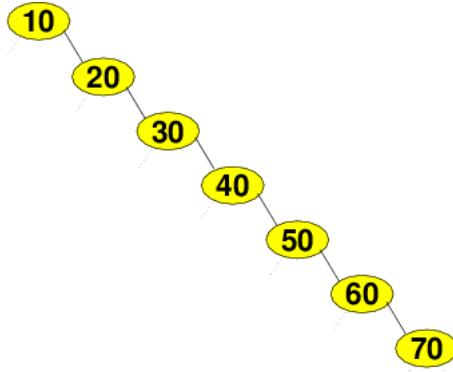
**Question:** Consider the contains operation on a degenerate tree with  $N$  nodes.

What is the complexity of the worst case?

- $O(1)$
- $O(\log N)$
- $O(N)$
- $O(N \log N)$
- $O(N^2)$

**\*\*Answer:\*\***

```
bool contains( const Comparable & x, BinaryNode *t ) const
{
    if( t == nullptr )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true;      // Match
}
```



A degenerate tree looks like a linked list. In the worst case, the value we're looking for is at the end of the list, so we have to search through all  $N$  nodes to get there. Thus the worst case is  $O(N)$ .

There's quite a difference, then, between the worst case behavior of trees, depending upon the tree's "shape".

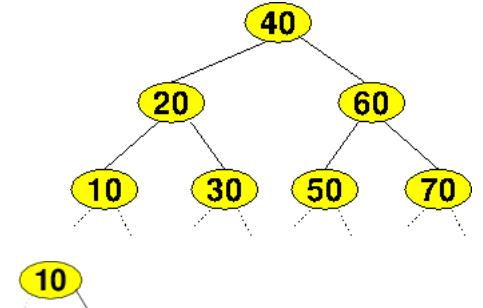
### 3.3 Average-Case

So we might wonder, then, does the "average" binary tree look more like the balanced or the degenerate case?

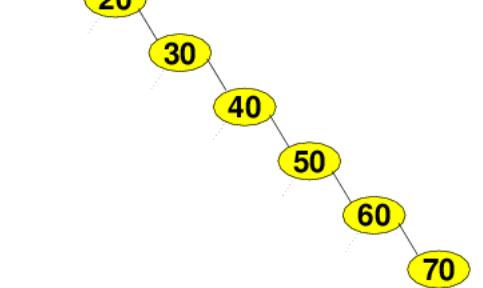
An intuitive argument is:

- No tree with  $n$  nodes has height  $< \log n$ .
- No tree with  $n$  nodes has height  $> n$
- The average depth of all nodes is therefore somewhere between  $n/2$  and  $(\log n)/2$ .
- The more unbalanced a tree is, the less likely that a random insertion would increase the tree height.

For example, if we are inserting into this tree, then any insertion will increase the tree's height.



But if we were inserting a randomly selected value into this one, then there is only a  $2/8$  chance that we will increase the height of the tree.



For trees that are somewhere between those two extremes, the chances of a random insertion actually increasing the height of the tree will fall somewhere between those two probability extremes.

- Insertions that don't increase the tree height make the tree more balanced.

So, the more unbalanced a tree is, the more likely that a *random* insertion will actually tend to increase the balance of the tree. This suggests (but does not prove) that randomly constructed binary search trees tend to be reasonably balanced.

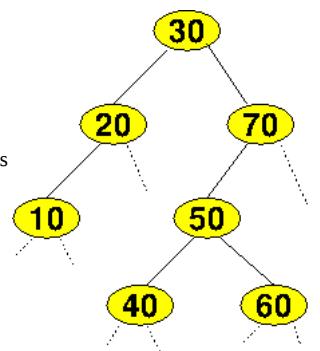
It is possible to prove this claim, but the proof is beyond the scope of this class.

But, yes, we expect randomly created binary search trees will be reasonably balanced.

But, it's not safe to be too confident about the height of binary search trees. Although random construction tends to yield reasonable balance, in real applications we often do not get random values.

**Question:** Which of the following data would, if inserted into an initially empty binary search tree, yield a degenerate tree?

- data that is in ascending order



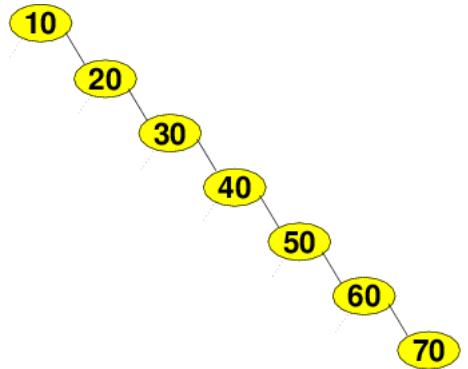
- data that is in descending order
- both of the above
- none of the above

*Answer* +

Both of the above.

If we insert data that is in ascending order, we get this degenerate BST.

If we insert data that is in descending order, we get the mirror image of this tree.



### 3.4 Can We Avoid the Worst Case?

Both data in ascending and descending order results in degenerate trees.

It's very common to get data that is in sorted or almost sorted order, so degenerate behavior turns out to be more common than we might expect.

Also, the arguments made so far don't take deletions into account, which tend to unbalance trees.

Later, we'll look at variants of the binary search tree that use more elaborate insertion and deletion algorithms to maintain tree balance.

# Traversing Trees with Iterators

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

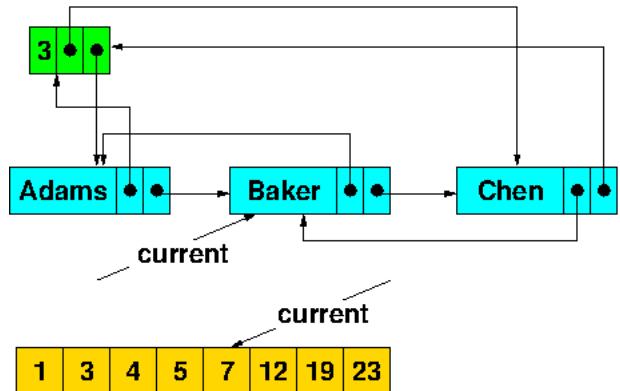
- [1 Iterating over Trees](#)
  - [1.1 begin\(\) and end\(\)](#)
  - [1.2 operator++](#)
- [2 Iterators using Parent Pointers](#)
  - [2.1 Basic operations](#)
  - [2.2 begin\(\) and end\(\)](#)
  - [2.3 operator++](#)
  - [2.4 Working with Parents](#)
- [3 Threads](#)

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 (e.g., using binary search trees to implement `std::set`), we may need to provide iterators for walking through 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.

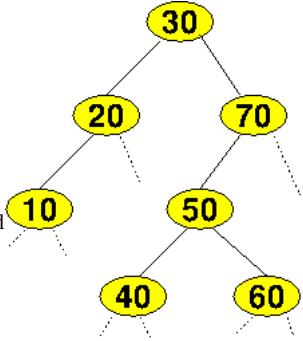


## 1 Iterating over Trees

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, which, for a BST, means that we will visit the data in ascending order.

It's not immediately obvious what our data structure for storing our "current position" (i.e., an iterator) will be. We might suspect that a pointer to a tree node will be part or whole of that data structure, in only because that worked for us with iterators over linked lists. With that in mind, ...



### 1.1 begin() and end()

**Question:** How would you implement `begin()`?

(Hint: which node is the first in an in-order traversal?)

Answer +

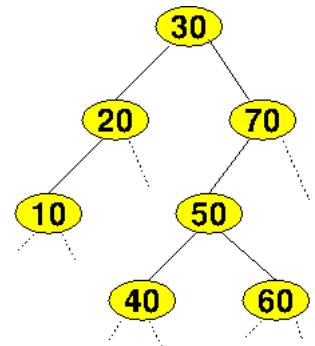
We find the `begin()` position by starting from the root and working our way down, always taking left children, until we come to a node with no left child.

That doesn't sound so hard.

Careful, now.

**Question:** How would you implement `end()`?

*Answer* +



Probably by returning a null pointer.

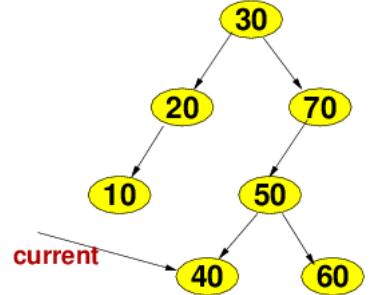
It's tempting to guess that you could do much the same as for `begin()`, this time seeking out the rightmost node. But that would leave you pointing to the *last* node in the tree, and `end()`, for any container, is supposed to denote the position *after* the last element in the container.

## 1.2 operator++

Now it gets trickier. Suppose you are still trying to implement iterators using a single pointer, you have one such pointer named `current` as shown in the figure.

**Question:** How would you implement `++current`?

Answer +



Well, we know that we should wind up at node 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 do operator++.

- 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. In essence, we would use the stack to simulate the activation stack during a recursive traversal.

But that's pretty clumsy. Iterators tend to get assigned (copied) a lot, and we'd really like that to be an  $O(1)$  operation. Having to copy an entire stack of pointers just isn't very attractive.

## 2 Iterators 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.

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.

tree.h +

```
template <typename Comparable>
class BinarySearchTree
{
public:
    class BstIterator {
        ...
    };
    ...

    typedef BstIterator const_iterator;
    typedef const_iterator iterator;

    BinarySearchTree( );

    /**
     * Copy constructor
     */
    BinarySearchTree( const BinarySearchTree & rhs );

    /**
     * Move constructor
     */
    BinarySearchTree( BinarySearchTree && rhs );

    /**
     * Destructor for the tree
     */
    ~BinarySearchTree( );

    /**
     * Copy assignment
     */
    BinarySearchTree & operator=( const BinarySearchTree & rhs );

    /**
     * Move assignment
     */
    BinarySearchTree & operator=( BinarySearchTree && rhs );

    /**
     * search for item. if found, return an iterator pointing
     * at it in the tree; otherwise, return end()
     */
    const_iterator find(const Comparable& item) const;

    /**
     * return an iterator pointing to the first item (inorder)
     */
    const_iterator begin() const;

    /**
     * return an iterator pointing just past the end of
     * the tree data
     */
}
```

```
template <typename Comparable>
struct BinaryNode
{
    Comparable element;
    BinaryNode *left;
    BinaryNode *right;
    BinaryNode *parent;

    BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt,
               BinaryNode *par )
        : element{ theElement }, left{ lt }, right{ rt }, parent { par } { }
};
```

```

    const_iterator end() const;

    /**
     * Find the smallest item in the tree.
     * Throw UnderflowException if empty.
     */
    Comparable & findMin( ) const;

    /**
     * Find the largest item in the tree.
     * Throw UnderflowException if empty.
     */
    Comparable & findMax( ) const;

    /**
     * Returns true if x is found in the tree.
     */
    bool contains( Comparable & x ) const;

    /**
     * Test if the tree is logically empty.
     * Return true if empty, false otherwise.
     */
    bool isEmpty( ) const { return root == nullptr; }

    /**
     * Print the tree contents in sorted order.
     */
    void printTree( ostream & out = cout ) const;

    /**
     * Make the tree logically empty.
     */
    void makeEmpty( );

    /**
     * Insert x into the tree; duplicates are ignored.
     */
    void insert( Comparable & x );

    /**
     * Remove x from the tree. Nothing is done if x is not found.
     */
    void remove( Comparable & x );

private:
    BinaryNode<Comparable> *root;
    :
};

```

A slightly subtle point here. The typedefs use the same data type for both the `iterator` and `const_iterator` types. That's because we really only want `const`-like behavior for this ADT. If we provided a "true" non-const iterator, it would let reassign data in the tree:

```
BinarySearchTree<int>::iterator it = myTree.find(50);
*it = 10000;
```

which would very likely upset the internal ordering of data in the tree, making it useless for any future searches. So we are only going to provide a `const`-style iterator that allows us to look at data in the container but not change that data. That means we only need one data type to implement both the `iterator` and `const_iterator`.

Later, when we use this tree to implement `std::set` and `std::map`, we'll see that this is precisely the behavior that they expect for their iterators.

## 2.1 Basic operations

Here's the basic declaration for an iterator to do in-order traversals.

```

class BstIterator
    : public std::iterator<std::bidirectional_iterator_tag, Comparable> {
public:
    BstIterator();

    // comparison operators. just compare node pointers
    bool operator== (const BstIterator& rhs) const;
    bool operator!= (const BstIterator& rhs) const;

    // dereference operator. return a reference to
    // the value pointed to by nodePtr
    Comparable& operator* () const;

    // preincrement. move forward to next larger value
    BstIterator& operator++ ();

    // postincrement

```

```

BstIterator operator++ (int);
// predecrement. move backward to largest value < current value
BstIterator operator-- ();

// postdecrement
BstIterator operator-- (int);

private:
    friend class BinarySearchTree<Comparable>;

    // nodePtr is the current location in the tree. we can move
    // freely about the tree using left, right, and parent.
    // tree is the address of the tree object associated
    // with this iterator. it is used only to access the
    // root pointer, which is needed for ++ and --
    // when the iterator value is end()
    const Comparable* nodePtr;
    const BinarySearchTree<Comparable> *tree;

    // used to construct an iterator return value from
    // a node pointer
    BstIterator (const Comparable* p,
                const BinarySearchTree<Comparable> *t);
};

}

```

You will note that the public interface is pretty much a standard iterator. The odd bit with `std::iterator` near the top of the class adds a number of internal type declarations that enhance the ability of [generic function templates](#) to work with this iterator.

The private section declares a pair of pointers. One points to the tree that we are walking through. The other points to the node denoting our current position within that tree.

## 2.2 begin() and end()

As discussed earlier, `begin()` works by finding the leftmost node in the tree:

```

/**
 * return an iterator pointing to the first item (inorder)
 */
template <class Comparable>
typename BinarySearchTree<Comparable>::const_iterator
inline
BinarySearchTree<Comparable>::begin() const
{
    return const_iterator(findMin(root), this);
}

```

And `end()` uses a null pointer.

```

/**
 * return an iterator pointing just past the end of
 * the tree data
 */
template <class Comparable>
typename BinarySearchTree<Comparable>::const_iterator
inline
BinarySearchTree<Comparable>::end() const
{
    return BstIterator(nullptr, this);
}

```

Each of these functions calls upon a [constructor for BstIterator](#):

```

class BstIterator {
public:
    BstIterator();

    // comparison operators. just compare node pointers
    bool operator== (const BstIterator& rhs) const;
    bool operator!= (const BstIterator& rhs) const;

    // dereference operator. return a reference to
    // the value pointed to by nodePtr
    const Comparable& operator* () const;

    // preincrement. move forward to next larger value
    BstIterator& operator++ ();

    // postincrement
    BstIterator operator++ (int);

    // predecrement. move backward to largest value < current value
    BstIterator operator-- ();

    // postdecrement
    BstIterator operator-- (int);
};

```

```

private:
    friend class BinarySearchTree<Comparable>;
    const BinaryNode<Comparable> *nodePtr;
    const BinarySearchTree<Comparable> *tree;

    // used to construct an iterator return value from
    // a node pointer
    BstIterator (const BinaryNode<Comparable> *p,
                 const BinarySearchTree<Comparable> *t);
};

:

template <class Comparable>
inline
BinarySearchTree<Comparable>::BstIterator::BstIterator
(const BinaryNode<Comparable> *p, const BinarySearchTree<Comparable> *t)
    : nodePtr(p), tree(t)
{
}

```

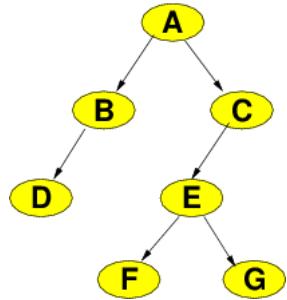
That constructor is actually private within `BstIterator`, and so is not available to programmers to call directly. However, because the `BstIterator` class names `BinarySearchTree` as a `friend`, the `BinarySearchTree` code is allowed access to that private data and so can call that constructor from within its begin and end functions.

## 2.3 operator++

Before trying to write the code for this iterator's `operator++`, let's try to figure out just what it should *do*.

**Question:** Suppose that we are currently at node E. What is the in-order successor (the node that comes next during an in-order traversal) of E?

\*\*Answer\*\* +



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 descendants and before visiting any of its right descendants. 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.

Let's explore that idea further.

**Question:** Suppose that we are currently at node A. What is the in-order successor (the node that comes next during an in-order traversal) of A?

**\*\*Answer\*\***

F is the in-order successor of A.

If we are at A during an in-order traversal, we have already visited all of A's left descendants. So the answer has to be C or one of its descendants. 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 descendants and before visiting any of its right descendants. 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

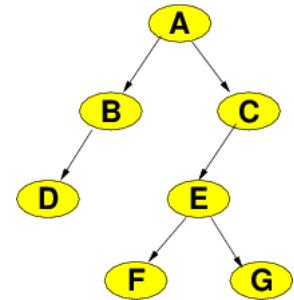
- Take a step down to the right, then
- Run as far down to the left as we can.

You can see how this would take us from A to F. And, for that matter, it 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?

\*\*Answer\*\* +



C does not *have* an in-order successor. C is actually the final node in an in-order traversal. After C is only `end()`.

OK, that's an interesting special case, but 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?

**\*\*Answer\*\***



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?

\*\*Answer\*\*



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, 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.

Notice that, applying this procedure to C, we would 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.

### 2.3.1 Implementing operator++

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

With that in mind, the operator++ code should be easily :-) understood.

```
// preincrement. move forward to next larger value
template <class Comparable>
typename BinarySearchTree<Comparable>::BstIterator&
BinarySearchTree<Comparable>::BstIterator::operator++ ()
{
    BinaryNode<Comparable> *p;

    if (nodePtr == nullptr)
    {
        // ++ from end(). get the root of the tree
        nodePtr = tree->root;

        // error! ++ requested for an empty tree
        if (nodePtr == nullptr)
            throw UnderflowException { };

        // move to the smallest value in the tree,
        // which is the first node inorder
        while (nodePtr->left != nullptr) {
            nodePtr = nodePtr->left;
        }
    }
    else
    {
        if (nodePtr->right != nullptr)
        {
            // successor is the farthest left node of
            // right subtree
            nodePtr = nodePtr->right;

            while (nodePtr->left != nullptr) {
                nodePtr = nodePtr->left;
            }
        }
        else
        {
            // have already processed the left subtree, and
            // there is no right subtree. move up the tree,
            // looking for a parent for which nodePtr is a left child,
            // stopping if the parent becomes NULL. a non-NUL parent
            // is the successor. if parent is NULL, the original node
            // was the last node inorder, and its successor
            // is the end of the list
            p = nodePtr->parent;
            while (p != nullptr && nodePtr == p->right)
            {
                nodePtr = p;
                p = p->parent;
            }

            // if we were previously at the right-most node in
            // the tree, nodePtr = nullptr, and the iterator specifies
            // the end of the list
            nodePtr = p;
        }
    }
}
```

```

    return *this;
}

```

You can run these iterator algorithms [here](#).

A similar process of analysis would eventually lead us to an implementation of operator--.

## 2.4 Working with Parents

There is, of course, a cost associated with this approach to iteration. We needed to add parent pointers to each node. That increases the storage overhead of the trees somewhat. It also means some modification to the code for building the trees. For example, here is our old code for inserting a data value into a BST:

```

/*
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * par is the parent node of t (null if t is the tree root)
 * Set the new root of the subtree.
 */
template <typename Comparable>
void BinarySearchTree::insert( const Comparable & x,
                                BinaryNode<Comparable> * & t,
                                BinaryNode<Comparable> * par )
{
    if( t == nullptr )
        t = new BinaryNode{ x, nullptr, nullptr };
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );
    else
        ; // Duplicate; do nothing
}

```

Here is the revised code, incorporating the parent pointers:

```

/*
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * par is the parent node of t (null if t is the tree root)
 * Set the new root of the subtree.
 */
template <typename Comparable>
void BinarySearchTree::insert( const Comparable & x,
                                BinaryNode<Comparable> * & t,
                                BinaryNode<Comparable> * par )
{
    if( t == nullptr )
        t = new BinaryNode<Comparable>{ x, nullptr, nullptr, par };
    else if( x < t->element )
        insert( x, t->left, t );
    else if( t->element < x )
        insert( x, t->right, t );
    else
        ; // Duplicate; do nothing
}

```

It's not terribly more complicated. Basically, we just have to remember that, if we are about to recursively visit a child of *t*, then we pass *t* as the parent pointer.

The full implementation of the binary search tree with iterators is [here](#).

## 3 Threads

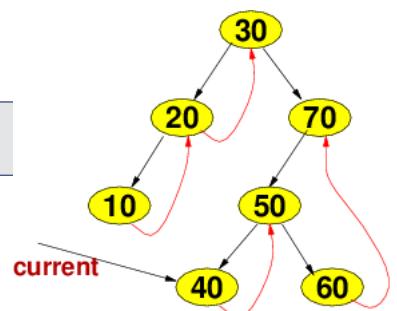
Another approach to supporting iteration is threading. Threading uses more complicated insert and remove algorithms to avoid the storage cost of adding the parent pointers.

**Threaded trees** replace all null right pointers by a *thread* (pointer) to that node's in-order successor.

- need to add a boolean flag to each node to tell if the right pointer is a child or a thread
- Can also thread the null left pointers to allow operator--

With threads in place the operator++ can be implemented as follows:

- Follow the “right child” pointer.
- If that pointer was not flagged as a thread, then descend as far to the left as possible.



`operator++` never needs to move “up” in the tree (which, lacking a parent pointer, it can’t do anyway).

The cost of this much simpler implementation of `operator++` is a correspondingly more complicated implementation of the `insert` and `remove` functions, as these need to create and maintain the threads.

# Balanced Search Trees

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Thinking about tree heights](#)
- [2 AVL Trees](#)
  - [2.1 Example](#)
  - [2.2 Single Rotations](#)
  - [2.3 Double Rotations](#)
  - [2.4 Inserting into AVL Trees](#)
  - [2.5 Complexity](#)
- [3 B Trees](#)
  - [3.1 Properties of B-Trees](#)
  - [3.2 Complexity of BTREE operations](#)
- [4 Red-Black Trees](#)

We've seen that the performance of the main BST operations is bounded by the height of the tree, which can range from an ideal of  $O(\log N)$  for balanced trees to an all-too-common  $O(N)$  for degenerate trees.

Various algorithms have been developed for building search trees that remain balanced. We'll look at 2:

- AVL trees
- B trees

## 1 Thinking about tree heights

Clearly, our tree-based algorithms will run faster if our trees are as short as possible. But how short a tree can get depends upon how many nodes we have.

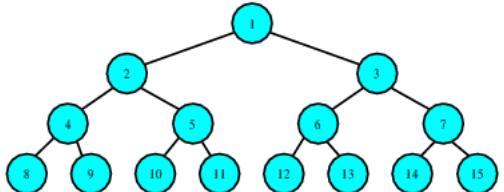
We can imagine trying to fill in a binary tree starting at the root, then filling in all of the depth 1 children, then all of the depth 2 children, and so on.

A binary tree is said to be *full* when, for some value  $k$ , every possible position at a depth  $\leq k$  has a node and no nodes occur at depths  $> k$ . Alternatively, we can say that a binary tree is full when all nodes have either zero or two children and all the leaves are at the same depth.

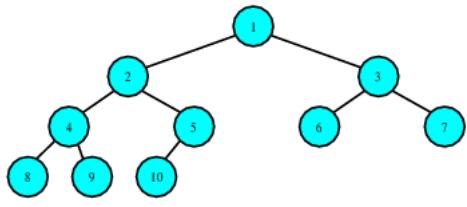
Here is an example of a full binary tree.

Not all sets of data that we might want to put into a binary search tree could be arranged into a full tree.  
Only certain numbers of nodes can be formed into a full tree. (Question to you: what numbers are these?).

Answer +



The number of nodes must be a number of the form  $2^k - 1$ .



An intermediate stage that, like a full tree, packs nodes into the shortest possible tree is the complete tree. A binary tree of height  $k$  is *complete* when all positions at depths  $< k$  are filled and the nodes at depth  $k$  are arranged from left to right.

The full tree shown earlier is also a complete tree. The tree shown here is complete, but is not full.

In practice, most approaches to generating well-balanced binary search trees do not yield complete trees, either. But complete trees do arise in some later data structures, and they are useful as an ideal against which tree balancing algorithms may be compared.

## 2 AVL Trees

An *AVL tree* (Adelson-Velskii and Landis) is a binary search tree for which each node's children differ in height by at most 1.

- Guarantees that the height of the tree is  $O(\log N)$ .
- Need to maintain height info in each node.

AVL insertion starts out identical to normal binary search tree insertion. But after the new node has been created and put in place, each of its ancestors must check to see if still balanced.

If any are unbalanced, the balance is restored by a process called *rotation*.

```
template <class T>
class avlNode
{
public:
    ;
    T value;
    avlNode<T> * parent;
    avlNode<T> * left;
    avlNode<T> * right;
    short balanceFactor;
};
```

Conceptually, an AVL tree node looks like an ordinary BST node except for the addition of a new integer data member to hold the height of the node.

As it turns out, however, it's not really necessary to record the exact height. Because we will never allow our AVL tree to ever get out of balance by more than 1, we can simply get by with an integer value that records the difference in heights between the right and left subtrees.

In a balanced tree, this difference must be -1, 0, or 1. 0 means that both subtrees have the same height. -1 means that the left tree is higher (by 1), and 1 means that the right tree is higher.

Suppose that we started with an AVL tree, made an insertion, and then discovered that a node is no longer balanced:

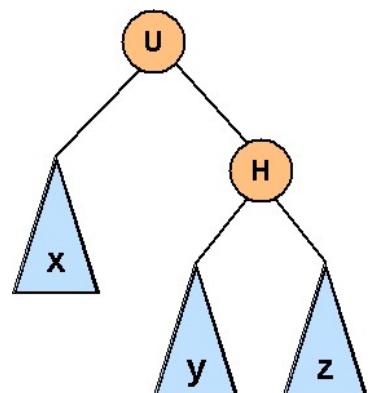
- Call that node  $U$  (for unbalanced)
- Difference in height between  $U$ 's children must be 2
- Designate the “heavier” or “higher” of the two children as  $H$ .

The diagram here shows one possible arrangement (the mirror image is also possible). The large triangles represent entire subtrees, possibly empty, possibly containing an arbitrarily large number of nodes.

**Question:** Now, all these nodes must still satisfy the BST ordering properties. For example, what can we say about the nodes in the subtree “ $y$ ”?

- They must have values less than that of  $U$ .
- They must have values greater than that of  $U$  but less than that of  $H$ .
- They must have values greater than that of  $H$ .
- None of the above

Answer:



Because the nodes that make up the subtree “y” are right descendants of U, they must have greater values than U. Because they are left descendants of H, they must have values smaller than H.

In fact, we can state that all the tree components, arranged into ascending order, would be:

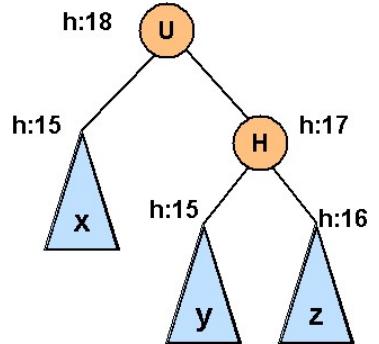
x u y h z

Keep that in mind — it will be important later.

## 2.1 Example

For the sake of example, let’s say that U has height 18. Now, because we are assuming that U is unbalanced, we know that the height of its children must differ by 2, and we have already said that H is the higher child. So H must have height 17, and x must have height 15.

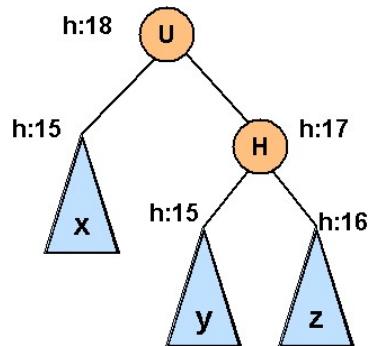
There are two possibilities for the heights of H’s children. They could both be height 16, or one could be 16 and the other 15. We’ll use the values shown in the diagram for example’s sake.



## 2.2 Single Rotations

Most texts seem to have a lot of trouble explaining the rotation process and wind up giving the code with little intuitive justification for how it works.

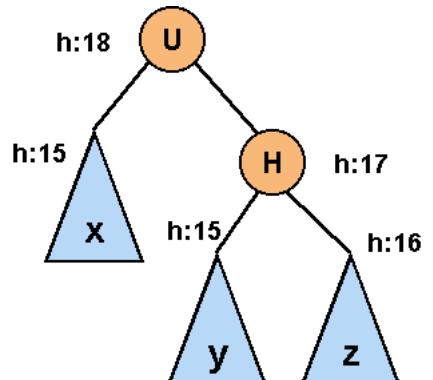
For me, the rotation process become easy to understand when I started to think of the tree nodes as beads connected by strings. When we start out, the whole assembly is “hanging” from its root, U.



But now, picture what would happen if we hoisted the entire assembly of beads by node H instead ...

You can see that the entire assembly is shorter and somewhat more balanced. It is not, however, a binary tree any longer, as H now has 3 children.

In fact, if you look closely, you’ll see that U is still unbalanced, because it is left with only a single child.



We can solve both of these problems by shifting the “y” subtree over to become a child of U. The resulting tree is balanced, and is shorter than it had been. But is it still a BST?

We said that, before the rotation, the elements of the tree arranged into ascending order would be: x u y h z

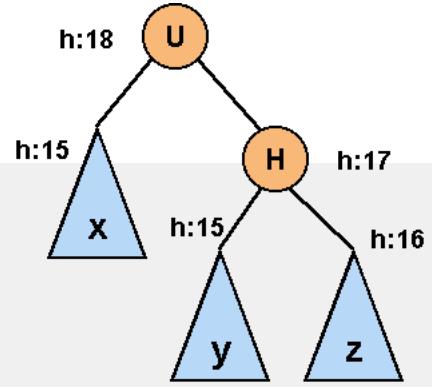
Look at the rotated tree. U is a left child of H and so must have a lower value. x is the set of U’s left descendants, all of which must be less than U. y is the set of right descendants of U but are also left descendants of H, so everything in y needs to be greater than U and less than H. z is the set of right descendants of H and so must be greater. In other words, the BST ordering rules suggest that the elements of the rotated tree, in ascending order, would be: x u y h z.

So the implied ordering is the same as before, and this is still a BST after the rotation.

This transformation is called a [single left rotation](#). If H had originally been a right child of U, we could perform the mirror-image transformation, a single right rotation.

Here you see the code to effect a single left rotation of a node.

```
template <class T>
avlNode<T>* avlNode<T>::singleRotateLeft ()  
    // perform single rotation rooted at current node  
{  
    avlNode<T>* U = this;          ①  
    avlNode<T>* H = U->right;  
    avlNode<T>* I = H->left;     ②  
  
    U->right = I;                ③  
    H->left = U;                 ④  
    if (I != 0)  
        I->parent = U;  
    H->parent = U->parent;       ⑤  
    U->parent = H;  
  
    // now update the balance factors  
    int Ubf = U->balanceFactor;  
    int Hbf = H->balanceFactor;  
    if (Hbf <= 0) {  
        if (Ubf >= 1)  
            H->balanceFactor = Hbf - 1;  
        else  
            H->balanceFactor = Ubf + Hbf - 2;  
  
        U->balanceFactor = Ubf - 1;  
    }  
    else {  
        if (Ubf <= Hbf)  
            H->balanceFactor = Ubf - 2;  
        else  
            H->balanceFactor = Hbf - 1;  
        U->balanceFactor = (Ubf - Hbf) - 1;  
    }  
    return H;  
}
```



The basic steps are

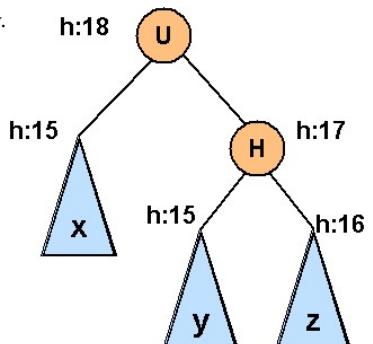
- ① Let U be the unbalanced node and H the higher of U's two children.
- ② Let I be the “interior” child of H, the child reached by stepping in the opposite direction used in going from U to H. For example, if H is a left child of U, then I would be the right child of H
- ③ In U, replace the pointer to H by I.
- ④ In H, replace the pointer to I by U.
- ⑤ Treat H as the root of the resulting structure.

In fact, these same steps work for right rotations as well, it's just that for left rotations, H is a right child and for right rotations, H is a left child.

Once the nodes have been rearranged, we have to recompute the heights of the affected nodes. Because we are using balance factors (+1, 0, -1) instead of heights, this gets a trifle messier, but it still not too bad.

## 2.3 Double Rotations

A single rotation doesn't always do the job. In setting up our rotation example, we assumed that subtree z was higher than y.



What would happen if y were the higher of the two?

In this case, the rotation does not produce a balanced tree.

A single rotation produces a balanced tree only if the interior subtree of H is no higher than the other subtree of H.

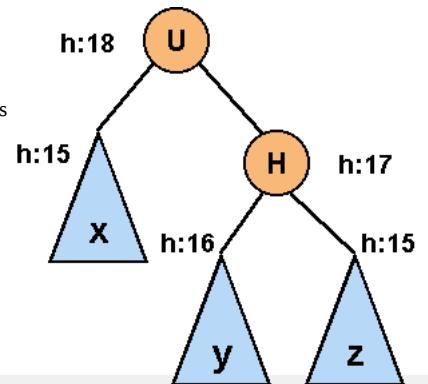
But, we can note that a left rotation shifts height from the right of the root to the left. Similarly, a right rotation shifts height from left to right.

So in this case, we are faced with a problem in that "y" is too high compared to "z". The solution is to do a single right rotation of H to shift height to the right, making "z" higher, then do the single left rotation of U.

This combination is called a double left rotation. (There is, of course, a mirror image "double right rotation" as well.)

So the process of rebalancing a node consists mainly of determining whether we need a single or double rotation, then applying the appropriate rotation routines.

```
template <class T>
avlNode<T>* avlNode<T>::balance ()
{ // balance tree rooted at node
  // using single or double rotations as appropriate
  if (balanceFactor < 0) {
    if (left->balanceFactor <= 0)
      // perform single rotation
      return singleRotateRight();
    else {
      // perform double rotation
      left = left->singleRotateLeft();
      return singleRotateRight();
    }
  }
  else {
    if (right->balanceFactor >= 0)
      return singleRotateLeft();
    else {
      // perform double rotation
      right = right->singleRotateRight();
      return singleRotateLeft();
    }
  }
}
```



## 2.4 Inserting into AVL Trees

We bring this all together in the AVL insert routine shown here.

```
template <class T>
avlNode<T>* avlNode<T>::insert (const T& val)
  // insert a new element into balanced AVL tree
{
  if (val < value) { // insert into left subtree ①
    if (left != 0) {
      int oldbf = left->balanceFactor;
      left = left->insert (val);
      // check to see if tree grew          ②
      if ((left->balanceFactor != oldbf) &&
          left->balanceFactor)
        balanceFactor--;
    }
    else {
      left = new avlNode(val, this);
      balanceFactor--;
    }
  }
  else { // insert into right subtree
    if (right != 0) {
      int oldbf = right->balanceFactor;
      right = right->insert (val);
      // check to see if tree grew          ②
      if ((right->balanceFactor != oldbf) &&
          right->balanceFactor)
        balanceFactor++;
    }
    else {
      right = new avlNode(val, this);
      balanceFactor++;
    }
  }

  // check if we are now out of balance, if so balance
  if ((balanceFactor < -1) || (balanceFactor > 1))      ③
    return balance();
  else
    return this;
}
```

- ① This routine starts as a conventional BST insert.
- ② There's a little bit of extra code to update the balance factors as we insert.
- ③ Then at the end, we check to see if this node is out of balance because of the insert. If so, we invoke the balance routine.

## 2.5 Complexity

An AVL tree is balanced, so its height is  $O(\log N)$  where  $N$  is the number of nodes.

The rotation routines are all themselves  $O(1)$  (messy as they are, notice that they have no loops or recursion), so they don't significantly impact the insert operation complexity, which is still  $O(k)$  where  $k$  is the height of the tree. But as noted before, this height is  $O(\log N)$ , so insertion into an AVL tree has a worst case  $O(\log N)$ .

Searching an AVL tree is completely unchanged from BST's, and so also takes time proportional to the height of the tree, making  $O(\log N)$ .

Removing nodes from a binary tree also requires rotations, but remains  $O(\log N)$  as well.

## 3 B Trees

B-trees are a form of balanced search tree based upon general trees (trees that are not restricted to two children).

A B-tree node can contain several data elements, rather than just one as in binary search trees.

They are especially useful for search structures stored on disk. Disks have different retrieval characteristics than internal memory (RAM).

- Obviously, disk access is much, much slower.
- Furthermore, data is arranged in concentric circles (called *tracks*) on each side of a disk "platter". (Most disks these days have a single platter, but some disks are a stack of platters.) A disk is read by read/write heads mounted on an arm that is moved in and out from track to track. Moving that arm takes time, so there is a real timing benefit to grouping data so that it can be read without moving the arm. The amount of data that can be read without moving the arm (from both sides of all platters) is called a *cylinder*. It's much faster to read an entire cylinder than to read a little, move the arm, read a little more, move the arm, etc., even if the total amount of data in a cylinder is much more than we need.

B-trees are a good match for on-disk storage and searching because we can choose the node size to match the cylinder size. In doing so, we will store many data members in each node, making the tree flatter, so fewer node-to-node transitions will be needed.

### 3.1 Properties of B-Trees

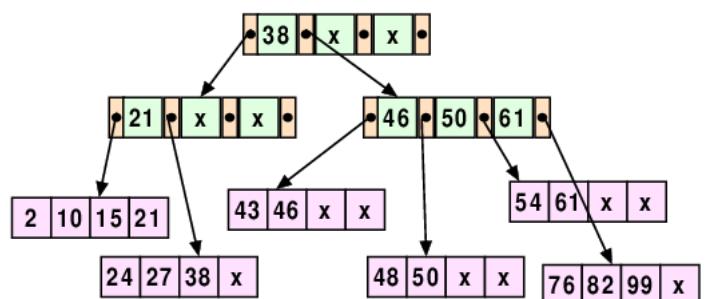
For a B-tree of order  $m$ :

- All data is in leaves. Keys (only) can be replicated in interior nodes.[This assumes that the tree is implementing a map, where we have distinct data types for keys and the associated data. For sets, the two types are the same.](#ret-This assumes that the tree is implementing a map, where we have distinct data types for keys and the associated data. For sets, the two types are the same.):
- The root is either
- a leaf, or
- an interior node with  $2 \dots m$  children
- All interior nodes other than the root have  $\lfloor m/2 \rfloor \dots m$  children
- All leaves are at the same depth.

Here is an example of a B-tree of order 4.

The *find* operation for B-trees is similar to that of binary search trees.

```
BTree find (const Etype & x, BTree t)
{
    if (t is a leaf)
        return t;
    else
    {
        i = 1;
        while ((i < m) && (x >= t->key[i]))
            ++i;
        return find(x, t->child[i]);
    }
}
```



Inserting into a B-tree starts out by "find"ing the leaf in which to insert.

- If there is room in the leaf for another data item, then we're done.

- If the leaf already has  $m$  items, then there's no room.
- Split the overfull node in half and pass the middle value up to the parent for insertion there.
- If the value passed up to the parent causes the parent to be over-full, then it too splits and passes the middle value up to its parent.

Deletion is usually lazy or semi-lazy (delete from leaf but do not remove keys within the interior nodes).

## 3.2 Complexity of BTree operations

- The maximum depth of an order  $m$  BTree is  $\lfloor \log_{\lfloor m/2 \rfloor}(n) \rfloor$
- At each node, we do  $O(\log m)$  work to choose branch
- An insert or delete may need  $O(m)$  work to fix up info in a node

Worst cases are:

- find:  $O(\log(m) * \log_m(n))$

But, since  $\log_m(n) = \frac{\log(n)}{\log(m)}$ , this simplifies to  $O(\log(n))$ .

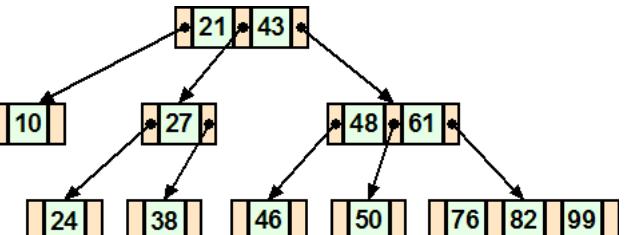
- insert/delete:  $O(m \log_m(n)) = O\left(\frac{m}{\log(m)} \log(n)\right)$

## 4 Red-Black Trees

B-trees are generally used with a fairly high width (order). That's because the most common application of B-trees is for search trees stored on disks, and the physical and electronic properties of a disk generally give the best performance to programs that read and process an entire sector or cylinder of the disk at a time. An on-disk B-tree is therefore usually configured to fill an entire sector or cylinder of the disk.

A closely related data structure arises when we take a B-tree of order 4 and relax just a few rules, including not storing all the data in the leaves but allowing some data to reside in the internal tree nodes.

The result is called a [2-3-4 tree](#) because each non-leaf node will, depending upon how full it is, have either 2, 3, or 4 children.



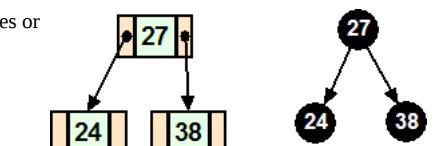
2-3-4 trees are, like all B-trees, a balanced tree whose height grows no faster than the log of the number of elements in tree.

Unlike B-trees, 2-3-4 trees are commonly used for in-memory data structures. But programmers seldom implement 2-3-4 trees directly. Instead, there is a fairly simple way to map 2-3-4 trees onto binary trees to which a "color" has been added.

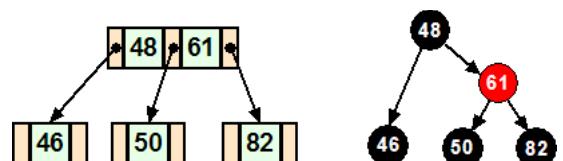
```

class RedBlackNode<T>
{
public:
    :
    T value;
    RedBlackNode<T> * parent;
    RedBlackNode<T> * left;
    RedBlackNode<T> * right;
    bool color; // true=red, false=black
};
  
```

A 2-3-4 node with 2 children (1 data value) is represented by a black binary tree node whose children are either leaves or black nodes.



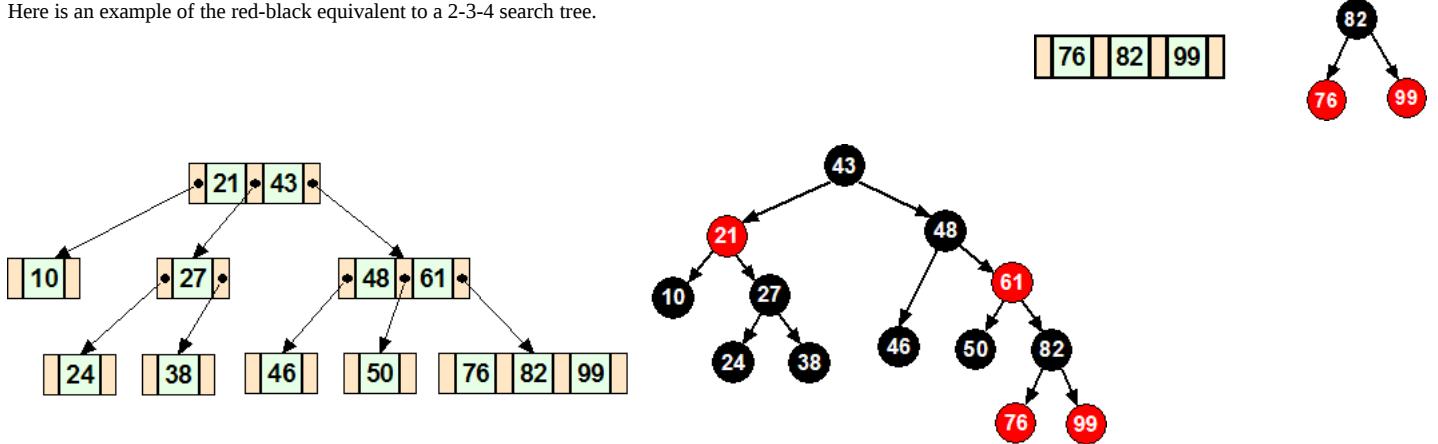
A 2-3-4 node with 3 children (2 data values) is represented by a black binary tree node with one red child, the other child being a leaf or a black node. (Either child could be the red one, so the mirror image of the binary tree in this diagram is also legal.)



A 2-3-4 node with 4 children (3 data values) is represented by a black binary tree node with two red children.

In essence, the red nodes define a kind of extension of their parent node. Each red node can be thought of as adding one extra data field and child pointer to its parent.

Here is an example of the red-black equivalent to a 2-3-4 search tree.



Some things to note:

- The root of a red-black tree is always black.
- No red node will ever have a red child.
- The red-black tree is a binary search tree and can be searched using the conventional binary search tree “find” algorithm.
- The height of a red-black tree is no more than twice the height of the equivalent 2-3-4 tree.
  - And we have already noted that the height of B-trees, including 2-3-4 trees, is  $O(\log N)$  where  $N$  is the number of data items in the tree.
  - We therefore know that the height of a red-black tree is also  $O(\log N)$ .
  - And that searches on a red-black tree have a  $O(\log N)$  worst case.

The algorithms to insert nodes into a red-black tree add no more than a constant time for each node in the path from the root to the newly added leaf. Consequently, insertions into a red-black tree are worst case  $O(\log N)$ . In fact the code for red-black trees are based on rotations very similar to those of AVL trees.

Red-black trees are used in most implementations of `set`, `mset`, `map`, and `mmap` in the C++ `std` library.

# Sets and MultiSets

Steven J. Zeil

Last modified: Jul 17, 2018

## Contents:

- [1 Overview of Sets and Maps](#)
- [2 The Set ADT](#)
  - [2.1 The template header](#)
  - [2.2 Internal type names](#)
  - [2.3 Constructors & Assignment](#)
  - [2.4 Status](#)
  - [2.5 Insert & Erase](#)
  - [2.6 Access](#)
- [3 A Simple Example of Using Set](#)
- [4 Implementing std::set with Binary Search Trees](#)
  - [4.1 Inserting Data](#)
  - [4.2 Searching and Iterating](#)
  - [4.3 Copying](#)

The `std::` ADTs we have looked at so far in this course have been what are sometimes called *sequential* containers — they maintain elements in a known *sequence* and we access the elements by indicating a position in the sequence from which we wish to obtain an element. Sometimes we indicate this position numerically (e.g., `a[23]`), sometimes symbolically (e.g., `myVector.front()`), but always, in order to get an element, we have to know where it is in relation to the other elements.

Now we turn our attention to *associative* containers, ones in which the container's implementation maintains the elements in a sequence that is intended to allow rapid access to elements based upon their value.

## 1 Overview of Sets and Maps

The major associative classes in the standard library are

- `set`
- `map`
- `multiset`
- `multimap`

Sets are containers to which we can add elements (called “*keys*”) and later check to see if certain key values are present in the set.

Maps, also known in other contexts as “lookup tables” or “dictionaries”, allow us to store pairs consisting of a *key* value and associated *data* value and to later look up the data value, if any, associated with a given key. (In some contexts, especially more mathematical ones, the set of keys is called the *domain* of the map and the set of associated data values is called the *range* of the map.)

In a set or a map, a given key value may appear only once. Attempting to add a key  $K$  to a set replaces any existing key equal to  $K$ . Adding a key-data pair  $(K, D_1)$  to a map that already has  $(K, D_2)$  replaces  $(K, D_2)$  by  $(K, D_1)$ .

But in a multiset or multimap, the same key can occur any number of times. For a multiset, this means that instead of just asking “is  $K$  in this set?” we can now ask “how many  $K$ 's are in this set?”. For a multimap, adding a key-data pair  $(K, D_1)$  to a multimap that already has  $(K, D_2)$  results in multimap that has both  $(K, D_1)$  and  $(K, D_2)$ .

We'll look at sets and multisets first, and then turn to maps and multimaps in a later lecture.

## 2 The Set ADT

A set collects elements and lets you check to see if an element is already in the collection.

- A given element can occur at most once in a set.
- Provides iterators that allow you to list the elements in sorted order.

```
#ifndef SET_H
#define SET_H

#include <cstddef>

template <class Key, class Compare=less<Key>>
class set {
private:
    Compare comparator;
```

```

public:
// typedefs:

    typedef Key key_type;
    typedef Key value_type;
    typedef Compare key_compare;
    typedef Compare value_compare;

    typedef const Key& reference;
    typedef const Key& const_reference;

    typedef ... const_iterator;
    typedef const_iterator iterator;

    typedef ... const_reverse_iterator;
    typedef const_reverse_iterator reverse_iterator;

    typedef ... size_type;
    typedef ... difference_type;

// allocation/deallocation

    explicit set(const Compare& comp = Compare());
    set(const set<Key, Compare>& x);

    template <class InputIterator>
    set(InputIterator first, InputIterator last,
        const Compare& comp = Compare());
    set<Key, Compare>& operator=(const set<Key, Compare>& x);

// accessors:

    key_compare key_comp() const { return comparator; }
    value_compare value_comp() const { return comparator; }

    iterator begin() const;
    iterator end() const;

    reverse_iterator rbegin() const { return t.rbegin(); }
    reverse_iterator rend() const { return t.rend(); }

    bool empty() const { return (size() == 0); }
    size_type size() const;
    size_type max_size() const;
    void swap(set<Key, Compare>& x);

// insert/erase
    pair<iterator, boolconst value_type& x);

    iterator insert(iterator position, const value_type& x);
    void clear();
    void erase(iterator position);
    size_type erase(const key_type& x);
    void erase(iterator first, iterator last);

// set operations:

    iterator find(const key_type& x) const;
    size_type count(const key_type& x) const;

    iterator lower_bound(const key_type& x) const;
    iterator upper_bound(const key_type& x) const;
    pair<iterator, iterator> equal_range(const key_type& x) const;

private:
    : // declaration of implementing data structure
};

template <class Key, class Compare>
bool operator==(const set<Key, Compare>& x,
                  const set<Key, Compare>& y);

template <class Key, class Compare>
bool operator< (const set<Key, Compare>& x,
                  const set<Key, Compare>& y);

#endif

```

The interface to multiset is identical to that of the set (aside from replacing the name “set” by “multiset”) - it’s only the behavior of a few of the operations that differ. So, as we look at the set interface, keep in mind that the same things apply to multisets as well.

## 2.1 The template header

```
template <class Key, class Compare=less<Key>>
class set {
```

Typically, you would declare a set by instantiating the template on the date type that you want to use for the key:

```
set<string> myStringSet;
```

But as you can see from the template header, this is an oversimplification. There is a second parameter, `Compare`, that supplies the comparison function to be used in comparing elements of the `set`. This parameter is given a default value, `less<Key>`, so the above instantiation is actually equivalent to

```
set<string, less<string>> myStringSet;
```

What does `less< ... >` do? It simply uses the Key’s own `operator<` to do the comparisons, and that’s good enough 90% of the time.[^fnless]

Actually, even this template header is somewhat oversimplified. The “real” header has yet another parameter:

[^fnless]: `less` is a rather odd creature, however. It’s actually a template for a class of [functors](#), which we first encountered some time ago.

```
template <class Key, class Compare=less<Key>,
          class Allocator = allocator<Key> >
class set {
```

but the `Allocator` parameter is only used in very rare situations where the application needs specialized control over how memory for the `set` will be allocated. We won’t worry about that in this course.

## 2.2 Internal type names

```
// typedefs:
typedef Key key_type;
typedef Key value_type;
typedef Compare key_compare;
typedef Compare value_compare;

typedef const Key& reference;
typedef const Key& const_reference;

typedef ... const_iterator;
typedef const_iterator iterator;
```

Many standard containers provide a set of type names for programming convenience and to provide a “standard” way of declaring things without making direct reference to the data structures used to implement them. We’ve seen this with type names like `vector::const_iterator` and `list::size_type`.

The associative containers add a couple of useful type names. The name `key_type` gives the data type of the keys in the container. The name `value_type` gives the data type that describes what we insert into the container and what is returned whenever we dereference (apply `operator*` to) an iterator. For `Set` and `multiset`, the `value_type` is the same as the `key_type`, but that won’t be true when we get to `map` and `mymap`.

## 2.3 Constructors & Assignment

```
explicit set(const Compare& comp = Compare());
set(const set<Key, Compare>& x);

template <class InputIterator>
set(InputIterator first, InputIterator last,
    const Compare& comp = Compare());

set<Key, Compare>& operator=(const set<Key, Compare>& x);
```

Looking at the constructors for `set`, the most interesting things are

- The first constructor takes a single parameter, a comparator. But we already supply a comparator when we instantiate the class! It turns out that when we instantiate the class, e.g.,

```
typedef set<string, less<string>> MySetType;
```

we’re only telling the compiler what the default comparator should be.

We can use that default:

```
MySetType ascendingOrder; // uses < to compare strings
```

or we can create objects that use a different comparator:

```
MySetType descendingOrder(greater<string>()); // uses > to compare strings
```

Note that, because this constructor has a default value for its only parameter, we can use it, as we did above, as the “default constructor” for the class. (If you don’t remember what a default constructor is, go back to the ADTs lectures.)

- The third constructor lets us build a new set from any range specified by a pair of iterators. For example,

```
vector<string> v;
⋮
multiset<string> ms (v.begin(), v.end());
```

## 2.4 Status

```
bool empty() const { return (size() == 0); }
size_type size() const;
size_type max_size() const;
```

No surprises here ...

## 2.5 Insert & Erase

```
// insert/erase
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator position, const value_type& x);
void clear();
void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);
```

As described in your text,

```
s.insert("def");
```

adds a new element, “def”, into *s*.

- If something equal to “def” is already in the set, nothing happens.
  - But in a multiset, we would go ahead and add a second copy of “def”.

The return type from this operation is a bit interesting. There’s two things you might want to know after trying to add something to a set:

- Did it go in? (i.e., was this key already in the set), or
- Where is it in the set?

Rather than choose between these two equally valuable pieces of information, the `insert` operation actually returns both of them in a pair. (The `std::pair` template was one of our first [examples of a class template](#).) It returns a pair (*b*, *p*) where *b* is a bool indicating whether an insertion occurred, and *p* is the position where the key was inserted. You are supposed to ignore the value of *p* if *b* is false.

The second of the two `insert` functions is a bit odd:

```
s.insert(position, "def");
```

Since this is an associative container, it’s supposed to decide where to put things. Why then might we supply a position?

- The *position* is taken as a “hint” of where to begin searching for position to insert.
  - The set implementation may ignore the hint if a quick check shows that the new key really does not belong at that position.
  - But the C++ standard says that we are guaranteed that this hint version of `insert` will have an amortized O(1) worst case if we have data that is already in order and we use the position where one key was inserted as the hint for the next larger key, e.g.:

```
vector<int> v;
⋮
sort (v.begin(), v.end(), less<int>());
// we know the elements of v are in order
set<int> mySet;
set<int>::iterator hint = mySet.begin();
for (int i = 0; i < v.size(); ++i)
{
    hint = mySet.insert (hint, v[i]); // amortized O(1) inserts
    ++hint;
}
```

- Many generic algorithms expect an `insert` operation of this form. In particular, suppose we wanted to copy a list into a set. You may remember that we can’t do this:

```
list<Foo> fooList;  
:  
set<Foo> fooSet;  
copy (fooList.begin(), fooList.end(), fooSet.begin());
```

**Question:** Why doesn't this copy work?

\*\*Answer:\*\*

```

list<Foo> fooList;
⋮
set<Foo> fooSet;
copy (fooList.begin(), fooList.end(), fooSet.begin());

```

The copy does not work here because copy simply writes into its destination positions – it doesn't allocate new space for stuff, it assumes that the space already exists.

When we faced this problem copying into vectors and lists, we got around it by using `back_inserter`:

```

list<Foo> fooList;
⋮
vector<Foo> fooVect;
copy (fooList.begin(), fooList.end(), back_inserter(fooVect));

```

`back_inserter` is a function that returns a special iterator that uses `push_back` whenever we try to store something at the iterator.

Now that won't help for copying into sets, because sets don't have a `push_back` function. But another iterator-returning function is `inserter`, that returns an iterator that uses `insert(position,value)` whenever we try to store to it. For example, we can copy into the middle of a vector this way:

```

list<Foo> fooList;
vector<Foo> fooVect;
⋮
copy (fooList.begin(), fooList.end(),
      inserter(fooVect, fooVect.begin() + fooVect.size() / 2));

```

And, to finally get to the point of all this, because `set` and `multiset` have a member function that looks like `insert(position,value)`, we can use `inserter` with them as well:

```

list<Foo> fooList;
⋮
set<Foo> fooSet;
copy (fooList.begin(), fooList.end(),
      inserter(fooSet, fooSet.end()));

```

If the list is unsorted, then the position supplied to `inserter` is only a hint. But if the list were sorted already, then this copy would get the amortized O(1) hint benefit, and the entire copy would be accomplished in O(`fooList.size()`) time.

## 2.6 Access

```

iterator find(const key_type& x) const;
size_type count(const key_type& x) const;

```

There are two ways to see if an element is present in a set:

```

set<string, less<string> >::iterator i;
i = s.find("abc");

```

We can search the set for "abc". If the key is found, `find` returns the position where that element resides. If not found, it returns `s.end()`. So it's not unusual to see code that looks like:

```

if (mySet.find(yourName) != mySet.end())
    cout << "I found " << yourName << " in my set." << endl;

```

Alternatively, we can count the number of times we find an element.

```

if (mySet.count(yourName) > 0)
{
    cout << "I found " << yourName << " in my set." << endl;
    cout << "I found there " << mySet.count(yourName) << " times." << endl;
}

```

Of course, for sets, `count(...)` will always return 0 or 1, because either the key isn't in there at all, or there's only one copy of it in the set. But for multiset, `count` may return any non-negative number.

### 2.6.1 Searching for Ranges of Equal Items

```

iterator lower_bound(const key_type& x) const;
iterator upper_bound(const key_type& x) const;
pair<iterator, iterator> equal_range(const key_type& x) const;

```

Suppose that we're interested, not so much in whether or not an element is present, but in where it might be.

We can, as we have seen, use `find` to get the position of an element in a set. But if we have a multiset, there may be many instances of the same key. How do we know which instance will be pointed out by `find`?

The functions shown here allow us to get the positions of all keys equal to some specified value. `lower_bound` returns the position of the first element in the set/multiset with the indicated value. `upper_bound` returns the position just after the last element equal to a given value. `equal_range` returns a pair consisting of both the `lower_bound` and the `upper_bound`.

For example, a library system, having many copies of the same book, might keep track of its books by ISBN number:

```

class Book {
public:
    ...
    string author() const;
    string title() const;
    string isbn() const;
    int copyNumber() const;
    bool isCheckedOut() const;
};

bool operator < (const Book& left, const Book& right)
{
    return left.isbn() < right.isbn();
}

```

Then, given a multiset of books:

```

typedef multiset<Book, less<Book>> Holdings;
Holdings library;

```

we could list all the copies of a given book that are checked out this way:

```

const string textISBN = "0-201-30879-7";
pair<Holdings::iterator, Holdings::iterator> rng
    = library.equal_range (textISBN);

for (Holdings::iterator i = rng.first; i != rng.second; ++i)
{
    if (i->isCheckedOut())
        cout << "Copy " << i->copyNumber()
            << " is checked out." << endl;
}

```

Both set and multiset support these operations, though they aren't really very useful for sets.

By the way, this is a good example of a place where the new C++11 auto type declaration and range-based for loop can simplify the code:

```

const string textISBN = "0-201-30879-7";
auto rng = library.equal_range (textISBN);
for (auto& book: rng)
{
    if (book.isCheckedOut())
        cout << "Copy " << book.copyNumber()
            << " is checked out." << endl;
}

```

The first `auto` matches the `pair` type and the second matches the `Book` type.

## 3 A Simple Example of Using Set

As part of a program to be presented in a later example, we need to read an English language document in plain text form and collect all the words that are used to begin sentences. For example, if we read

```
Hello! How are you? I haven't seen you in a long time. How is
your family?
```

we would want a set with the words “Hello!”, “How”, and “I”. (This particular application will need us to preserve all upper/lower case distinctions and to treat punctuation as part of the preceding word.)

This is a pretty straightforward application for set, and here is the code to collect the sentence-starting words:

```

void readDocument (const char* docFileName,
                  set<string>& startingWords,

```

```

        ...
    }

    ifstream docIn (docFileName);
    char lastChar = '.';
    :
    string word;

    while (docIn >> word)
    {
        if (lastChar == '.' || lastChar == '?'
            || lastChar == '!') {
            startingWords.insert(word);
        }
        lastChar = word[word.length()-1];
        :
    }
}

```

## 4 Implementing std::set with Binary Search Trees

We can use our earlier [binary search tree with iterators](#) to implement a set. We'll need **two main data structures**: the search tree and an integer counter to count the number of elements in tree. For the **iterator types**, we will use the same iterators already provided by the binary search tree.

```

template <typename Key, typename Compare=less<Key> >
class Set {
private:
    Compare comparator;

public:
// typeDefs:

    typedef Key key_type;
    typedef Key value_type;
    typedef Compare key_compare;
    typedef Compare value_compare;

    typedef const Key& reference;
    typedef const Key& const_reference;

    typedef typename BinarySearchTree<Key>::const_iterator const_iterator;
    typedef const_iterator iterator;

    typedef unsigned int size_type;
    typedef ptrdiff_t difference_type;

    :

private:
    size_type treeSize;
    BinarySearchTree<key_type> bst;
};

```

### 4.1 Inserting Data

When inserting into a set, we basically insert into the tree, but increment our counter if the data gets inserted:

```

// insert/erase
template <typename Key, typename Compare>
std::pair<typename Set<Key,Compare>::iterator, bool>
Set<Key,Compare>::insert(const Set<Key,Compare>::value_type& x)
{
    const_iterator it = bst.insert(x);
    if (it == end())
        return std::make_pair(it, false);
    else
    {
        ++treeSize;
        return std::make_pair(it, true);
    }
}

```

This takes advantage of a search tree `insert` function that returns the location where a data value was inserted, or `end()` if the data was not inserted (because it is a duplicate of a data value already in the tree, and we don't store duplicates in a set).

Erasing data is similar, but we would decrement the `treeSize` counter if we successfully remove anything.

### 4.2 Searching and Iterating

Searching the tree is even simpler:

```

template <typename Key, typename Compare>
inline

```

```
typename Set<Key, Compare>::iterator Set<Key, Compare>::find(const Set<Key, Compare>::key_type& x) const
{
    return bst.find(x);
}
```

as are the functions to provide the beginning and ending iterators:

```
template <typename Key, typename Compare>
inline
typename Set<Key, Compare>::iterator Set<Key, Compare>::begin() const
{
    return bst.begin();
}

template <typename Key, typename Compare>
inline
typename Set<Key, Compare>::iterator Set<Key, Compare>::end() const
{
    return bst.end();
}
```

## 4.3 Copying

Finally, we note that our binary search tree already implemented its own versions of the Big 3, and none of the remaining data members in our `set` class are pointers, so we can rely on the compiler-generated versions of the Big 3 for our `set`.

The full version of the `set` implementation is available [here](#).

# Maps and MultiMaps

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

### [1 Interface](#)

#### [1.1 The template header](#)

#### [1.2 Internal type names](#)

#### [1.3 Insert & Erase](#)

#### [1.4 Access](#)

### [2 Example: Literary Style Emulator](#)

#### [2.1 The Program](#)

#### [2.2 Once More, with MultiMaps](#)

### [3 Sets, MultiSets, Maps, and MultiMaps: all in the family](#)

#### [3.1 From Set to Map](#)

#### [3.2 From Map to MultiSet](#)

#### [3.3 From Map to MultiMap](#)

A map can be thought of as

- a lookup table, or as
- a generalization of an array/vector in which the index need not be numeric.

```
typedef map<string, Zipcodes, less<string>> ZipTable;
ZipTable zips;

zips["Jones"] = 23529;
zips["Zeil"] = 23452;
cout << zips["Zeil"];
```

- A map associates a key type Key with an associated data type T. In the example above, the key type is string. The associated data is int.
- The map allows you to store (key,data) pairs (e.g., the two assignment statements in the example above) and to retrieve the data previously stored with some key (e.g., the cout output statement).
- Like sets, a map may only contain a single copy of any given key value
  - But a multimap can contain multiple copies of the same key.
  - Like sets, you must supply a comparison functor class (for the Key type) or accept the less<Key> default.

## 1 Interface

The code shown here presents a version of map that is pretty close to the standard.

```
#ifndef MAP_H
#define MAP_H

#include <utility>

template <class Key, class T, class Compare = less<Key>>
class map
public:

// typedefs:

typedef Key key_type;
typedef T data_type;
typedef pair<const Key, T> value_type;
typedef Compare key_compare;

typedef ... pointer;
typedef ... const_pointer;
typedef ... reference;
typedef ... const_reference;
typedef ... iterator;
typedef ... const_iterator;
typedef ... reverse_iterator;
typedef ... const_reverse_iterator;
typedef ... size_type;
typedef ... difference_type;

// allocation/deallocation

map();
```

```

explicit map(const Compare& comp);

template <class InputIterator>
map(InputIterator first, InputIterator last);

template <class InputIterator>
map(InputIterator first, InputIterator last, const Compare& comp);

map(const map<Key, T, Compare>& x);
map<Key, T, Compare>& operator=(const map<Key, T, Compare>& x);

// accessors:

key_compare key_comp() const;
value_compare value_comp() const;

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend();

bool empty() const;
size_type size() const;
size_type max_size() const;

T& operator[](const key_type& k);

void swap(map<Key, T, Compare>& x);

// insert/erase

pair<iterator,bool> insert(const value_type& x);
iterator insert(iterator position, const value_type& x);

template <class InputIterator>
void insert(InputIterator first, InputIterator last);

void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);
void clear();

// map operations:

iterator find(const key_type& x);
const_iterator find(const key_type& x) const;

size_type count(const key_type& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;

pair<iterator,iterator> equal_range(const key_type& x);
pair<const_iterator,const_iterator> equal_range(const key_type& x) const;

};

template <class Key, class T, class Compare>
bool operator==(const map<Key, T, Compare, Alloc>& x,
                  const map<Key, T, Compare, Alloc>& y);

template <class Key, class T, class Compare>
inline bool operator<(const map<Key, T, Compare, Alloc>& x,
                      const map<Key, T, Compare, Alloc>& y);

#endif

```

The `map` interface is styled very similarly to that of `set`, so we'll concentrate on the "surprises" that `map` has in store. Keep in mind that everything we say for `maps` applies to `mymaps` as well, except for the limitation to having only a single copy of any given key value.

## 1.1 The template header

You will most often create a `map` by instantiating the template on the two data types that you wish to use for the key and for the associated data:

```
map<string, int> myStringToIntMap;
```

```
template <class Key, class T, class Compare = less<Key>>
class map
```

But, as with `Set`, there is another parameter, `Compare`, used to compare key values. The `Compare` parameter defaults to `less<Key>`.

## 1.2 Internal type names

Let's look at the type names declared inside each map. As usual, we see `const_iterator`, `iterator` and `size_type`.

Like `set`, the `map` template defines type names for `key_type`, giving the data type of the keys in the container, and `value_type`, giving the data type that describes what we insert into the container and what is returned whenever we dereference (apply `operator*` to) an iterator.

For `set` and `multiset`, the `value_type` is the same as the `key_type`, but we can see that this is not true here. Instead the `value_type` is a pair. The first element of the pair is a key value, the second element a data value.

Notice also that the first element of the `value_type` pair is marked `const`. What this tells us is that, if we have a map iterator:

```
map<string, int>::iterator p = myStringToIntMap.find("foo");
```

we can use that iterator to change the data associated with this key:

```
(*p).second = 42;
```

but *not* to change the key at that location:

```
(*p).first = "bar"; // compiler flags this as an error
```

The exact form of `value_type` seems to vary among compilers, but it is always a key-data [std::pair](#) and it should always prohibit changing the key but allow changing the data.

So I would recommend `_not_writing` code like this:

```
pair<const Key, T> currentData = *mapIterator;
```

but suggest instead that you take advantage of the more convenient name that every `map` provides:

```
map<Key, T>::value_type currentData = *mapIterator;
```

or, in C++11,

```
auto currentData = *mapIterator;
```

## 1.3 Insert & Erase

Maps support the same kind of `insert` operations that we saw for `sets`.

You need to keep in mind, however, that you are inserting a key-data pair:

```
typedef map<PersonnelRecord,
           string,
           CompareByNameAddress>
    Departments;
Departments depts;
PersonnelRecord keyes, maly, zeil;
const string cs = "Computer Science";
const string math = "Mathematics";
:
depts.insert (Departments::value_type(keyes, math));
depts.insert (Departments::value_type(maly, cs));
```

```
// insert/erase
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator position, const value_type& x);
template <class InputIterator>
void insert(InputIterator first, InputIterator last);

void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);
```

`value_type` is a type name declared inside every `std::` container and describes the data type that is inserted into the container and that is retrieved when we dereference an iterator. For the containers we have seen earlier, `value_type` was not all that useful, because we pretty much knew what kind of data we were inserting.

For maps and multimaps, however, `value_type` is a [std::pair](#). Pairs have a constructor that takes the two elements to be composed into the pair, which is what you are seeing in the expressions `Departments::value_type(keyes, math)` and `Departments::value_type(maly, cs)` above.

Another way to do much the same thing is with the `std::` template `make_pair`:

```
depts.insert (make_pair(keyes, math));
depts.insert (make_pair(maly, cs));
```

As with sets, we can supply a position as a hint of where to store things. If we're consistently right, we get an amortized  $O(1)$  insertion time.

```
depts.insert (depts.end(), Departments::value_type(zeil, cs));
```

A simpler way to insert is to use the indexing notation (available for maps, but not for multimaps). We could write the same set of insertions this way:

```
depts[keyes] = math;
depts[maly] = cs;
depts[zeil] = cs;
```

which certainly looks a lot more attractive.

This shorter form can be a bit inefficient however. Here's how it works:

- The expression on the left of the assignment is evaluated first.
  - The dept map's operator[] is invoked.
  - This searches through the dept map for any key-data pair having, in the first assignment, keyes as its key part.
    - If keyes is not in the map yet, a new pair is added to the map with keyes as its key and the default constructor for the data type (`string()`) used to create the initial value of the data part.
  - A reference (`String&`) to the data part of the key-data pair is returned from operator[].
- The expression on the right of the assignment is evaluated, yielding a string.
- The string from the right side is assigned to the `string&` returned on the left, overwriting the value already there.

Now, only a single search through the table is done, but two different strings get put in there, the second replacing the first. So if our data type was one in which the default constructor was particularly expensive (the default constructor for `String` simply produces an empty string "", so that's no big deal), then we might want to avoid the extra data value creation and use the `insert` function instead.

## 1.4 Access

### 1.4.1 find

We can search maps for key values in much the same way that we do sets:

```
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
size_type count(const key_type& x) const;
```

```
typedef map<PersonnelRecord, string, CompareByNameAddress>
    Departments;
Departments depts;
:
Departments::iterator i;
i = depts.find(maly);
if (i != depts.end())
    cout << maly.name() << " is in the "
    << i->second << " department."
    << endl;
```

### 1.4.2 operator[]

Again, for maps (but not multimaps), we can use the more convenient indexing notation:

```
typedef map<PersonnelRecord, string, CompareByNameAddress>
    Departments;
Departments depts;
:
cout << maly.name() << " is in the "
    << depts[maly] << " department."
    << endl;
```

But this code isn't really the same. Remember, the `operator[]` for a map searches the map for the given key, and, *if it doesn't find it, adds it to the map*. So if maly wasn't already in the map, we would see the following output:

Maly is in the department.

(The new entry for `maly` would be created using the `string` default constructor for the data.)

The following hybrid code, which I often see in student programs, is particularly inefficient:

```
typedef map<PersonnelRecord, string, CompareByNameAddress>
    Departments;
Departments depts;
:
Departments::iterator i = depts.find(maly);
if (i != depts.end())
    cout << maly.name() << " is in the "
    << depts[maly] << " department."
    << endl;
```

because it actually searches the map twice for the same value. Even though searching a map is only  $O(\log(\text{size}()))$ , there's no point to doubling the execution time like this.

In a similar vein, the fact that `map` access looks so much like accessing an array often leads people to write code like:

```
map<string, int> wordCounts;
:
wordCounts[word] = wordCounts[word] + 1;
```

If `wordCounts` were an array, most compilers would, at least on their highest optimization settings, recognize that the two indexing expressions access the same address and would avoid doing the calculation twice. But no compiler will perform that optimization when `wordCounts` is a map, so the above code does two searches through the map. Better would be:

```
+&wordCounts[word];
```

or

```
int& count = wordCounts[word];
++count;
```

or

```
map<string,int>::iterator p = wordCounts.find(word);
++(p->second);
```

or

```
auto p = wordCounts.find(word);
++(p->second);
```

### 1.4.3 lower\_bound, upper\_bound, & equal\_range

Searching for a range of positions – most useful with multimaps.

If we are using a multimap, we can have multiple data values associated with the same key. So a `find(...)` operation that returns only a single position is probably not what we need.

For example, suppose we were implementing an appointment calendar and had:

```
iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;

pair<iterator, iterator> equal_range(const key_type& x);
pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
```

```
multimap<Date, string> events;
:
events.insert(make_pair(Date("9/21/2016"), "project meeting"));
events.insert(make_pair(Date("9/21/2016"), "status report due"));
events.insert(make_pair(Date("9/21/2016"), "seminar"));
events.insert(make_pair(Date("9/22/2016"), "guest lecture"));
events.insert(make_pair(Date("9/24/2016"), "assignment due"));
```

We can envision this map as a sequence of pairs (even though we expect that it's *really* stored in a binary search tree or similar data structure).

```
events: [(9/21/2016, "project meeting"), 9/21/2016, "status report due"),
          (9/21/2016, "seminar"), (9/22/2016, "guest lecture"),
          (9/24/2016, "assignment due")]
```

Now, if we do

```
auto pos = events.find(Date("9/21/2017"));
```

then we will get an iterator pos that points to one of the three events for 9/21, but we have no guarantee which of the three we will see. This would not be a problem for an ordinary map, because there could only be one pair with that key in an ordinary map, but you can see this might be less than ideal for a multimap.

`lower_bound` returns the position of the first pair with a given key. `upper_bound` gives the position just *after* the last pair with that key. So we can list all of the events for 9/21 with the following code:

```
Date d921 = Date("9/21/2017");
for (auto pos = events.lower_bound(d921);
     pos != events.upper_bound(d921); ++pos)
{
    cout << pos->second << endl;
}
```

Note that `upper_bound` is not the same as `end()`. `end()` gives us the end of the *entire container*: `upper_bound` gives us the end of a range of positions for a specific key. In the example we have been working with, `upper_bound(Date("9/21/2016"))` is probably the same position as `lower_bound(Date("9/22/2016"))`.

This code has one flaw, however. It actually searches the multimap twice, once to get the lower bound and once to get the upper bound. `equal_range` lets us get both in a single search, returnign a pair of iterators:

```
auto events921 = events.equal_range(Date("9/21/2017"));
for (auto pos = events921->first;
     pos != events921->second; ++pos)
{
    cout << pos->second << endl;
}
```

## 2 Example: Literary Style Emulator

In those cases, we can copy work? The standard says that the functor can occur at all, or group of am associative containers, ones that is a different from the same key value and the course have data value (In some ordering on its destination positions of the map - The name and trust the object that we are called just ask if is somewhat oversimplified. The text's discussion, especially more mathematical one, the key values are checked out by "multiset") - The set of the of keys in a multiset of that default: or - Now, let's write functors for later.

No, the above paragraph was not supposed to make sense. In fact, I did not write it. It was written by feeding the text of the previous lecture notes on sets into a simple "style emulator" program that attempts to generate text in the same "style" as a sample document it is given. It does this by two simple rules:

- It begins its output with a word that was used to begin a sentence in the sample document.
- After producing any word w, it chooses its next word from among those that occurred after the same word w in the sample document.

The program produces gibberish, but gibberish that is based on the same vocabulary and phrasing of the sample document's author.

The differences in output, given different sample documents, can be quite striking. Here is an output from the program after feeding it a copy of Shakespeare's *Hamlet*:

Pooh! you hear me, he so; At his meed he's the earth which dearest father grossly, full of majesty should murder sanctuarize; Revenge should do a piece of this.— There is this should be as level to tell us so and less than another? Why, look like a more willingly part them, sir? They are embark'd: farewell: And, as grace, As e'er my blood, And he beseech'd me packing: I'll teach me. The tedious old man. My fault is to be at thy distemper Sprinkle cool patience! Whereon the carriages, sir, but reserve thy brain,

and after feeding it the U.S. Constitution:

But in the United States; To promote the Constitution of Marque and punish its own Members, and of all Crimes, except as under this Union, according to the several States, and establish. The President of the Appointment of the United States; and Water; To promote the Contrary notwithstanding. The Privilege of three Years after it shall be as to which shall be eligible to pay Duties of Impeachment shall be divided as may, on the Legislature, which shall be President of Commerce or being disapproved by the United States. A Person holding any State, the Senators from any State, the

and, finally, after giving it a copy of Lewis Carroll's *Alice Through the Looking Glass*:

'Mind the word. 'It's very ignorant of you,' the bow, not being punished: and all what I go on, turning from day was going on: 'for I think about. They looked round them into this time that the silence, with a pause in fact, the Knight said "if".' But I've had eyes and one of puzzling, was, before she has the house! Well then, when he's happy. His name them, 'just in bright thought that needed any lions or other, rolled down there, they've each other White King was a little further, 'to be a battle?' Tweedledum spied a Queen,

### 2.1 The Program

Now let's look at how this program can be written.

Here is the high-level pseudocode for the main routine.

```

int main() {
    get sample doc filename and N, the # of
    words to generate, from command-line args;
    read the sample document, collecting all
        sentence-starting words and all consecutive
        word pairs;
    generate N words of text from the collected
        sentence-starting words and consecutive pairs;
}

```

We'll tackle this as a top-down design. Starting with this pseudocode, we could choose to expand any of these statements. For now, let's just assume that the last two steps will be handled by separate functions.

### 2.1.1 main() calls functions for bulk of processing

```

int main() {
    get sample doc filename and N, the # of
    words to generate, from command-line args;

    // A set of words that have been used to start new sentences.
    ??? startingWords;

    // For each word appearing in the document, a vector of all the
    // words that have immediately followed it.
    ??? consecutiveWords;

    readDocument (filename of sample doc,
                  startingWords, consecutiveWords);
    generateEmulatedText (startingWords, consecutiveWords, N);

    return 0;
}

```

Getting the command line arguments is one of those things that may appear mysterious at first, but because we do it for every almost program we write, it can eventually be done almost by rote.

### 2.1.2 main(): Command-Line Arguments

```

int main() {
    if (argc != 3)
    {
        cerr << "Usage: " << argv[0] << "document N\n"
            << " where document is a plain-text document and\n"
            << " N is the number of words of output desired." << endl;
        return -1;
    }

    srand (time(0));

    int N;
    { // convert parameter string to an int
        istrstream Nin (argv[2]);
        Nin >> N;
    }

    get sample doc filename and N, the # of
    words to generate, from command-line args;

    // A set of words that have been used to start new sentences.
    ??? startingWords;

    // For each word appearing in the document, a vector of all the
    // words that have immediately followed it.
    ??? consecutiveWords;

    readDocument (argv[1], startingWords, consecutiveWords);
    generateEmulatedText (startingWords, consecutiveWords, N);

    return 0;
}

```

We also throw in a use of `srand` to initialize the random number generator, as we will clearly be making random selections within this program.

Let's look next at the process of generating the emulated text. (As a personal preference, I usually work on designing the “heart” of a program before worrying about its input/output routines, because I believe that you often discover important I/O requirements when you get into the “core” algorithms.)

### 2.1.3 generateEmulatedText()

Here's a reasonable start, but it has some problems.

```

void generateEmulatedText
    (const ???& startingWords,
     const ???& consecutiveWords,

```

```

    int N)
{
    randomly select and print one word
        from startingWords;
    do this N-1 times {
        Look up the last-printed word in
            consecutiveWords, getting all the
            words that followed that one in the
            sample document.
        Randomly select one word from among those
            followers and print it;
    }
}

```

- Suppose that the sample document ended with some word that doesn't appear anywhere else in the document. If we ever select that word for printing, we will then be stuck because its list of followers would be empty.
- We really don't want all the output appearing on a single line, so we need to insert line breaks at appropriate places.

## 2.1.4 generateEmulatedText(): expand the loop body

Fixing those problems:

```

: generateEmulatedText0.listing [+]

void generateEmulatedText
    (const ???& startingWords,
     const ???& consecutiveWords,
     int N)
{
    randomly select and print one word
        from startingWords;
    do this N-1 times {
        Look up the last-printed word in
            consecutiveWords, getting all the
            words that followed that one in the
            sample document.
        if at least one such word exists
            randomly select one word from among those
                followers;
        else
            randomly select one word
                from startingWords;

        if selected word would make the
            current output line 80
            or more characters wide
                print a line break;
            print the selected word;
    }
}

```

## 2.1.5 generateEmulatedText(): simplified loop

We can simplify this a bit with a little transformation of the loop.

```

void generateEmulatedText
    (const ???& startingWords,
     const ???& consecutiveWords,
     int N)
{
    word = "";
    do this N times {
        Look up word in
            consecutiveWords, getting all the
            words that followed that one in the
            sample document.
        if at least one such follower exists
            word = random selection from among those
                followers;
        else
            word = random selection
                from startingWords;

        if word would make the
            current output line 80
            or more characters wide
                print a line break;
            print word;
    }
}

```

We've got enough detail here to start filling in some real C++ now.

## 2.1.6 generateEmulatedText(): line filling

```
void generateEmulatedText
  (const ???& startingWords,
   const ???& consecutiveWords,
   int N)
{
  const int MAXLINELENGTH = 80;

  string word = "";
  int lineLength = 0;

  for (int wordCount = 0; wordCount < N; ++wordCount)
  {
    Look up word in
      consecutiveWords, getting all the
      words that followed that one in the
      sample document.
    if at least one such follower exists
      word = random selection from among those
        followers;
    else
      word = random selection
        from startingWords;

    if (word.length() + lineLength > MAXLINELENGTH)
    {
      cout << endl;
      lineLength = 0;
    }
    if (lineLength > 0)
      cout << ' ';
    cout << word;
    lineLength += word.length();
  }
}
```

Now, let's consider the issue of making a random selection from a collection of words. If we have a sequence of K words, we could generate a random integer in the range 0...K-1 and use that to index into the sequence and access the selected word.

The data structures we have that allow retrieval via numeric indices are arrays, vectors, and dequeus. A vector would seem the most likely choice.

## 2.1.7 randomChoice()

A function like this should then do nicely for making a random selection.

```
string randomChoice (const vector<string>& choices)
{
  int k = rand() % choices.size();
  return choices[k];
}
```

Returning to our generateEmulatedText function, then, it would seem that the startingWords collection should be a `vector<string>`, so that we can apply randomChoice to it.

## 2.1.8 generateEmulatedText(): data structure for parameters

[generateEmulatedText1.listing](#) +

```
void generateEmulatedText
  (const vector<string>& startingWords,
   const ???& consecutiveWords,
   int N)
{
  const int MAXLINELENGTH = 80;

  string word = "";
  int lineLength = 0;

  for (int wordCount = 0; wordCount < N; ++wordCount)
  {
    Look up word in
      consecutiveWords, getting all the
      words that followed that one in the
      sample document.
    if at least one such follower exists
      word = random selection from among those
        followers;
    else
      word = randomChoice(startingWords);

    if (word.length() + lineLength > MAXLINELENGTH)
    {
      cout << endl;
      lineLength = 0;
    }
  }
}
```

```

        if (lineLength > 0)
            cout << ' ';
        cout << word;
        lineLength += word.length();
    }
}

```

Now, what about consecutiveWords? The “look up” suggests a map or multimap. Because the thing we are looking up is a string, we know that the key type must be string. But what should the data type be?

### 2.1.9 This time, let's try a map

[generateEmulatedText2.listing](#) +

```

void generateEmulatedText
    (const vector<string>& startingWords,
     const map<string, vector<string>>& consecutiveWords,
     int N)
{
    const int MAXLINELENGTH = 80;

    string word = "";
    int lineLength = 0;

    for (int wordCount = 0; wordCount < N; ++wordCount)
    {
        vector<string> followers = consecutiveWords[word];
        if (followers.size() > 0)
            word = randomChoice(followers);

        else
            word = randomChoice(startingWords);

        if (word.length() + lineLength > MAXLINELENGTH)
        {
            cout << endl;
            lineLength = 0;
        }
        if (lineLength > 0)
            cout << ' ';
        cout << word;
        lineLength += word.length();
    }
}

```

This is shaping up pretty nicely. But if we try to compile this code, we will quickly find that the compiler complains about our trying to use operator[] on a const map.

Remember that, when you do something like myMap[key], the map is searched for the key value. If that key isn't in the map yet, a new (key,value) pair is created and placed into the map.

That means that the [ ] operator is not const and so can only be used with maps that you are allowed to change.

### 2.1.10 Improved map access

[generateEmulatedText3.listing](#) +

```

void generateEmulatedText
    (const vector<string>& startingWords,
     const map<string, vector<string>>& consecutiveWords,
     int N)
{
    const int MAXLINELENGTH = 80;

    string word = "";
    int lineLength = 0;

    for (int wordCount = 0; wordCount < N; ++wordCount)
    {
        map<string, vector<string>>::const_iterator followers
            = consecutiveWords.find(word);
        if (followers != consecutiveWords.end()
            && followers->second.size() > 0)
            word = randomChoice(followers->second);

        if (word.length() + lineLength > MAXLINELENGTH)
        {
            cout << endl;
            lineLength = 0;
        }
        if (lineLength > 0)
            cout << ' ';
        cout << word;
        lineLength += word.length();
    }
}

```

So operator[] can only be applied to non-const maps, and we are receiving consecutiveWords as a const reference. That means that we must fall back on the clumsier find approach.

That finishes the generateEmulatedText function. Next we can turn our attention to the problem of reading the sample document.

### 2.1.11 readDocument()

This appears to be a good starting point.

```
void readDocument (const char* docFileName,
                  vector<string>& startingWords,
                  map<string, vector<string> >& wordPairs)
{
    open a stream to read the file named docFileName;

    while (we can read another word)
    {
        if the word before this one had ended in
        a sentence-ending punctuation mark
        add word to startingWords;

        Add (previous word, word) to wordPairs
    }
}
```

Something worth noting is that we don't want to filter these words the way we might for a spell checker, reducing everything to lowercase and discarding all punctuation. In this application, punctuation is our friend! It makes the generated text more natural-looking. We retain upper/lower-case distinctions as well. In most sample documents, every sentence will begin with a capitalized word. That means that, in our wordPairs map, words ending with '.', '?', etc., will only have capitalized words among their followers. Therefore whenever we happen to print a word that looks like it ends a sentence, the next word selected will wind up being capitalized. Again, this adds to the appearance of the generated text.

Similarly, we aren't concerned about weeding out duplicate occurrences of the same word. If an author tends to use certain words over and over again, that is part of the author's "style" and we want to emulate that pattern of word selection.

With this in mind, we start coding this function by noting that we will need variables to track the current word, the previous word, and the last character in the previous word.

### 2.1.12 readDocument(): setting up the variables

Next, we can take care of the file handling ...

```
readDocument1.listing [+]

void readDocument (const char* docFileName,
                  vector<string>& startingWords,
                  map<string, vector<string> >& wordPairs)
{
    open a stream to read the file named docFileName;

    char lastChar = '.';
    string lastWord;
    string word;

    while (we can read another word)
    {
        if the word before this one had ended in
        a sentence-ending punctuation mark
        add word to startingWords;

        Add (previous word, word) to wordPairs
        lastWord = word;
        lastChar = word[word.length()-1];
    }
}
```

### 2.1.13 readDocument(): file I/O

Adding the word to startingWords isn't too much of a challenge ...

```
readDocument2.listing [+]

void readDocument (const char* docFileName,
                  vector<string>& startingWords,
                  map<string, vector<string> >& wordPairs)
{
    ifstream docIn (docFileName);
    char lastChar = '.';
    string lastWord;
    string word;

    while (docIn >> word)
    {
```

```

        if the word before this one had ended in
            a sentence-ending punctuation mark
            add word to startingWords;

        if (lastWord != "") {
            Add (previous word, word) to wordPairs
            lastWord = word;
            lastChar = word[word.length()-1];
        }
    }
}

```

## 2.1.14 readDocument(): adding a starting word

[readDocument3.listing](#) +

```

void readDocument (const char* docFileName,
                  vector<string>& startingWords,
                  map<string, vector<string> >& wordPairs)
{
    ifstream docIn (docFileName);
    char lastChar = '.';
    string lastWord;
    string word;

    while (docIn >> word)
    {
        if (lastChar == '.' || lastChar == '?' || lastChar == '!')
            startingWords.push_back(word);

        if (lastWord != "") {
            Add (previous word, word) to wordPairs
            lastWord = word;
            lastChar = word[word.length()-1];
        }
    }
}

```

This may look familiar. It's almost identical to [the example](#) we looked at in the sets lecture notes, but I have opted for a vector instead of a set because I don't want to lose duplicate values.

At this point, we are left only with the problem of adding a pair of words to our map.

More precisely, our map should give us access to a vector of following words for `lastWord`. We then want to add `word` to the end of that vector.

## 2.1.15 adding a word pair

This is one way to do it.

[readDocument4.listing](#) +

```

void readDocument (const char* docFileName,
                  vector<string>& startingWords,
                  map<string, vector<string> >& wordPairs)
{
    ifstream docIn (docFileName);
    char lastChar = '.';
    string lastWord;
    string word;

    while (docIn >> word)
    {
        if (lastChar == '.' || lastChar == '?' || lastChar == '!')
            startingWords.push_back(word);

        if (lastWord != "") {
            vector<string> wordFollowers = wordPairs[lastWord];
            wordFollowers.push_back(word);
            wordPairs[lastWord] = wordFollowers;
        }
        lastWord = word;
        lastChar = word[word.length()-1];
    }
}

```

- We start by getting the current vector of followers out of the map. (If we have never seen `lastWord` before, then this will put an empty vector into the map and return a copy of that empty vector.)

Notice that what we are getting is a copy of the vector stores in the map.

- We then add our new `word` onto the end of the vector.

Because this is a copy of the vector actually in the map, this `push_back` changes only our local copy, not the one that's actually in the map.

- Therefore we update the map by inserting the modified vector.

This code isn't exactly thrilling. We are copying an entire vector of words out of the map, then copying an entire vector back into the map.

## 2.1.16 Using a reference variable to avoid copying

It is possible to do this without actually copying the vectors in and out of the map. The change required to accomplish this is subtle, but important.

```
readDocument5.listing +
```

```
void readDocument (const char* docFileName,
                  vector<string>& startingWords,
                  map<string, vector<string> >& wordPairs)
{
    ifstream docIn (docFileName);
    char lastChar = '.';
    string lastWord;
    string word;

    while (docIn >> word)
    {
        if (lastChar == '.' || lastChar == '?' || lastChar == '!')
            startingWords.push_back(word);

        if (lastWord != "")
        {
            vector<string>& wordFollowers = wordPairs[lastWord];
            wordFollowers.push_back(word);
        }
        lastWord = word;
        lastChar = word[word.length()-1];
    }
}
```

By making `wordFollowers` not a vector but a *reference to a* vector, we actually extract from the map the “address of” its own copy of the vector. The `push_back` in the next line is therefore directly modifying the vector *inside the map*.

## 2.1.17 Summary

At this point, changes are pretty much finished. Put together our `readDocument`, `generateEmulatedText` and `randomChoice` functions, together with the `main` function we started with, slap a few `#includes` to cover the map and vector data structures we've employed, and we're ready to go.

# 2.2 Once More, with MultiMaps

The use of a `map< ... , vector< ... >` structure may strike you as being a bit convoluted. After all, aren't multimaps expressly designed for situations where a given key value might map onto multiple associated data values?

That's certainly true. In fact, to my mind, a `multimap<T, U>` is pretty much equivalent to `map<T, list<U>>` or `map<T, vector<U>>`. Whether you prefer a multimap or a map that returns a container is a judgment call, depending on what you're comfortable with and kind of manipulation you might need to make on the whole collection of values mapped by a given key.

Let's look at what would happen to our program if we opted for a multimap instead of a map onto vectors.

## 2.2.1 Tracking word pairs

Inserting a word pair is both simpler and messier.

```
readDocument6.listing +
```

```
typedef multimap<string, string> FollowersMap;

void readDocument (const char* docFileName,
                  vector<string>& startingWords,
                  FollowersMap& wordPairs)
{
    ifstream docIn (docFileName);
    char lastChar = '.';
    string lastWord;
    string word;

    while (docIn >> word)
    {
        if (lastChar == '.' || lastChar == '?' || lastChar == '!')
            startingWords.push_back(word);

        if (lastWord != "")
        {
            wordPairs.insert(FollowersMap::value_type(lastWord, word));
        }
        lastWord = word;
        lastChar = word[word.length()-1];
    }
}
```

We don't have to worry about extracting out a vector and performing vector operations. We just insert the pair into the multimap. But we do have to deal with the fact that the multimap `insert` function takes a pair of values (key and data). The exact data type for that pair is given to us by the multimap as its `value_type`.

## 2.2.2 generateEmulatedText - retrieving from a multimap

The changes to `generateEmulatedText` (shown here in the previous, map-of-vector version) will be fairly extensive, because we can no longer extract a vector of following words directly.

[generateEmulatedText4.listing](#) +

```
void generateEmulatedText
  (const vector<string>& startingWords,
   const FollowersMap& consecutiveWords,
   int N)
{
  const int MAXLINELENGTH = 80;

  string word = "";
  int lineLength = 0;

  for (int wordCount = 0; wordCount < N; ++wordCount)
  {
    map<string, vector<string> >::const_iterator followers
      = consecutiveWords.find(word);
    if (followers != consecutiveWords.end()
        && followers->second.size() > 0)
      word = randomChoice(followers->second);
    vector<string> followers = consecutiveWords[word];
    if (followers.size() > 0)
      word = randomChoice(followers);
    else
      word = randomChoice(startingWords);

    if (word.length() + lineLength > MAXLINELENGTH)
    {
      cout << endl;
      lineLength = 0;
    }
    if (lineLength > 0)
      cout << ' ';
    cout << word;
    lineLength += word.length();
  }
}
```

## 2.2.3 generateEmulatedText - alternate retrieval from a multimap

We could accomplish the switch-over to a multimap most easily by getting the range of positions in the multimap corresponding to the key `lastWord` (via the `equal_range`) function.

[generateEmulatedText5.listing](#) +

```
void generateEmulatedText
  (const vector<string>& startingWords,
   const FollowersMap& consecutiveWords,
   int N)
{
  const int MAXLINELENGTH = 80;

  string word = "";
  int lineLength = 0;

  for (int wordCount = 0; wordCount < N; ++wordCount)
  {
    pair<FollowerMap::const_iterator,
         FollowerMap::const_iterator> followersp =
      = consecutiveWords.equal_range(word);
    if (followersp.first != followersp.second)
    {
      vector<string> followers;
      for (FollowerMap::const_iterator p = followersp.first;
           p != followersp.second; ++p)
        followers.push_back ((*p).second);
      word = randomChoice(followers);
    }
    else
      word = randomChoice(startingWords);

    if (word.length() + lineLength > MAXLINELENGTH)
    {
      cout << endl;
      lineLength = 0;
    }
    if (lineLength > 0)
      cout << ' ';
    cout << word;
    lineLength += word.length();
  }
}
```

```
:    }
```

Then we can loop through those positions, copying the data values (each position `p` points to a `(key,data)` pair, which is why we actually copy `(*p).second`).

Not exactly pretty. I think I liked the map-onto-vector version better.

This version *looks* inefficient because of the loop used for copying. But it's worth noting that we're not actually copying any more strings here than we did with the map-onto-vector version. In each case, we were making copying out the entire vector just so we could do a random selection from it.

Could we eliminate this copy by randomly selecting a position from the range inside the map, and then only extracting the single data element we really want?

## 2.2.4 randomChoice

We can do that by generalizing our `randomChoice` function to handle any kind of position (iterator).

```
template <class Iterator> Iterator randomChoice (Iterator start, Iterator finish) { int nChoices = finish - start; return start + rand() % choices.size(); }
```

This template would work just fine when used with our `startingWords` vector:

```
word = randomChoice (startingWords.begin(), startingWords.end());
```

but it can't be used, as it is, with the positions we get from our multimap.

The reason for the difference is that operations like subtracting one iterator from another (to determine the number of positions between them) or adding an integer to an iterator are valid only for *random access* iterators. `vector` provides random access iterators. `multimap` does not.

## 2.2.5 generic randomChoice

This sort of problem comes up often enough that the standard library provides some functions for dealing with it.

```
template <class Iterator>
Iterator randomChoice (Iterator start, Iterator finish)
{
    int nChoices = distance(start, finish);
    return advance(start, rand() % choices.size());
}
```

`distance` computes the number of positions between two iterators. When `distance(first, last)` is called with a pair of random access iterators, `distance` does its work in  $O(1)$  time by simply computing `last - first`. But if `first` and `last` are not random access, then `distance(first, last)` simply repeatedly applies `++` to `first` until it becomes equal to `last`, counting the number of steps required to do this. So, for non-random access iterators, `distance(first, last)` is  $O(\text{distance(first, last)})$  - it runs in time proportional to the size of its answer.

Similarly, `advance(start, k)` returns the iterator denoting the position  $k$  steps past `start`. If `start` is random access, this is done in  $O(1)$  time. If not, the operation is  $O(k)$ .

With this version, the multimap solution may be easier to read than the map-onto-vector, and should run somewhat faster because we have eliminated a lot of pointless copying of vectors and strings.

## 2.2.6 Summary

[generateEmulatedText6.listing](#) +

```
void generateEmulatedText
    (const vector<string>& startingWords,
     const FollowersMap& consecutiveWords,
     int N)
{
    const int MAXLINELENGTH = 80;

    string word = "";
    int lineLength = 0;

    for (int wordCount = 0; wordCount < N; ++wordCount)
    {
        vector<string> followers;
        pair<FollowerMap::const_iterator,
              FollowerMap::const_iterator> followersp =
            = consecutiveWords.equal_range(word);

        if (followersp.first != followersp.second)
            word = randomChoice
                (followersp.first,
                 followersp.second).second;

        else
            word = randomChoice(startingWords.begin(),
                                startingWords.end());
```

```

        if (word.length() + lineLength > MAXLINELENGTH)
    {
        cout << endl;
        lineLength = 0;
    }
    if (lineLength > 0)
        cout << ' ';
    cout << word;
    lineLength += word.length();
}

```

## 3 Sets, MultiSets, Maps, and MultiMaps: all in the family

We've already looked at how to implement sets. Given an efficient implementation of `set`, the others could be built on top of that with little trouble.

### 3.1 From Set to Map

To implement a map using a set, we start by remembering what the `value_type` of a map looks like:

```

template <class Key, class T, class Compare=less<Key> >
class map
{
:
typedef pair<const Key&, T> value_type;

```

Since the `operator*` of our map iterators will need to return references to `value_type`s, we know we need to actually use these pairs to store the keys and data.

It won't do us any good though, to store the map data in a `set<value_type>`, because we can't change the elements in a set, and we do need to be able to change the data portion of a map. So, instead, we will use a set of *pointers to* `value_type`s.

```

template <class Key, class T, class Compare=less<Key> >
class map {
public:
    typedef pair<const Key, T> value_type;
private:
    Compare comp;

    struct VTComparison
    {
        bool operator() (const value_type* left,
                          const value_type* right)
        {return comp(left->first, right->first);}
    };

    typedef set<value_type*, VTComparison> reptype;
    reptype data;
:

```

We'll illustrate how this works by looking at the `operator[]` for our new map.

```

template <class Key, class T, class Compare=less<Key> >
T& map<Key,T,Compare>::operator[] (const Key& key)
{
    value_type temp (key, T());
    reptype::iterator p = data.find(temp);
    if (p == data.end())
        p = data.insert (temp);
    return (*p).second;
}

```

All we do is to search the underlying set. If we don't find a match, we create a new key-data pair. Finally, we return a reference to the data portion of the appropriate key-data pair.

(It is possible to do this without creating the dummy `T()` value unless we need to insert it, but we don't really need to see the complications.)

### 3.2 From Map to MultiSet

```

template <class Key, class Compare=less<Key> >
class multiset {
public:
private:
    map <Key, int> data;

```

To implement a `multiset<T>`, we could use a `map<T, int>`, where the `int` indicates how many copies of an element are in the multiset.

```

template <class Key, class Compare=less<Key> >
multiset<Key,Compare>::iterator
    multiset<Key,Compare>::insert (const Key& key)
{
    auto pos = data.find(key);
    if (pos == data.end())
        // This key is not in the multiset yet
        return data.insert (map <Key, int>(key, 1));
    else
        { // This key is already in the multiset.
            ++(pos->second);
            return pos;
        }
}

```

If we are adding a new key to the multiset, we add it with count 1. If we are adding a key that is already in the multiset, we simply increment the exiting key's count.

### 3.3 From Map to MultiMap

A `multimap<Key, T>` can be formed from a `map<Key, list<T> >` so that each key can be mapped onto an entire list of related data values. The only tricky part in doing this is implementing the multimap iterators, as we need to record both a position within the map and within the most recently-accessed list.

The point of this exercise has been to show that there are no new concepts, just “grunt work” programming involved in implementing multiset, map, and multimap once we have a suitable set type.

# Hashing

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Hashing 101: the Fundamentals](#)
  - [1.1 The Ideal: Perfect Hash Functions](#)
  - [1.2 The Reality: Collisions](#)
- [2 Hash Functions](#)
- [3 Hash Functions: Examples](#)
  - [3.1 Hashing Integers](#)
  - [3.2 Hashing Character Strings](#)
  - [3.3 Hashing Compound Structures](#)
  - [3.4 Hashing and Equality](#)

Hashing is an important approach to set/map construction.

We've seen sets and maps with  $O(N)$  and  $O(\log N)$  search and insert operations.

Hash tables trade off space for speed, sometimes achieving an average case of  $O(1)$  search and insert times.

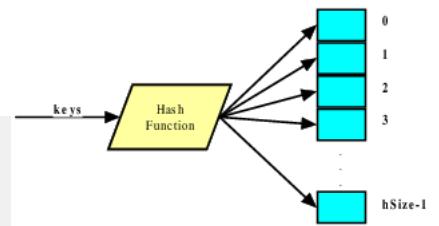
## 1 Hashing 101: the Fundamentals

Hash tables use a [hashing function](#) to compute an element's position within the array that holds the table.

If we had a *really* good hashing function, we could implement set insertion this way:

```
template <class T>
class set {
    ...
private:
    const unsigned hSize = ...;
    T table[hSize];
};

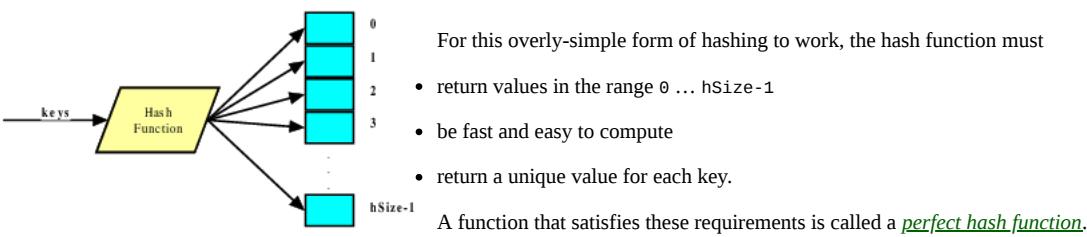
template <class T>
void set<T>::insert (const T& key)
{
    unsigned h = hash(key);
    table[h] = key;
}
```



and searching through the table would not be much harder:

```
template <class T>
size_type set<T>::count (const T& key) const
{
    int h = hash(key);
    if (table[h] == key)
        return 1;
    else
        return 0;
}
```

### 1.1 The Ideal: Perfect Hash Functions



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 week into numbers indicating how far into the week the day is:

Key	Value
Sunday	1
Monday	2
Tuesday	3
Wednesday	4
Thursday	5
Friday	6
Saturday	7

If we are willing to use a table with a little bit (or a lot) of extra space, we could use a function

```
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, 7> days = {"Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday", "Friday", "Saturday"};
int table[96];
for (int i = 0, i < 7; ++i)
    table[hash(days[i])] = i+1;
```

and then afterwards, we can look up those day names in  $O(1)$  time:

```
int dayOfWeek (const string& dayName)
{
    return table[hash(dayName)];
}
```

Perfect hash functions are usually only possible if we know all the keys in advance. That 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 large 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’s keywords to quickly recognize when a word read from the source code file is really a reserved word.

## 1.2 The Reality: Collisions

For the most part, though, we can’t really 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. Since collisions are, in most cases, unavoidable, we say that a good hash function will

- return values in the range  $0 \dots \text{hSize}-1$
- be fast and easy to compute
- minimize the number of collisions.

## 2 Hash Functions

A good hash function will

- return values in the range  $0 \dots \text{hSize}-1$
- be fast and easy to compute
- minimize the number of collisions.

Actually, the first of these three requirements is usually enforced inside the hash table code by the simple technique of taking `hash()` modulo `hSize`:

```
template <class T>
void set<T>::insert (const T& key)
{
    unsigned h = hash(key) % hSize;
    table[h] = key;
}

template <class T>
size_type set<T>::count (const T& key) const
{
    int h = hash(key) % hSize;
    if (table[h] == key)
```

```

    return 1;
else
    return 0;
}

```

And, 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, i.e., if I am drawing keys at random, the probability of the next key’s going into any particular position in the hash table should be the same as for any other position.

So the characteristics that we’ll look for in a good hash function

- Fast and easy to compute
- Distributes the keys uniformly across the table.

The possibility of collisions also forces us to revise those simple algorithms to include [collision handling](#), which we will discuss a little later..

## 3 Hash Functions: Examples

The proper choice of hash functions depends upon the structure and distribution of the keys.

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 that they be fast and distribute the keys uniformly over the range  $0 \dots hSize - 1$ .

### 3.1 Hashing Integers

This is the easiest possible case.

If we have a set of integer keys that are already in the range  $0 \dots hSize - 1$ , we don’t need to do anything:

```
int hash(int i) {return i;}
```

If the keys are in a wider range, we could employ the modulus trick:

```
int hash(int i) {return i % hSize;}
```

but usually we don’t bother because, as discussed earlier, this modulus transformation is usually done inside the hash table code anyway.

Later, when we look at variable hashing and at the `std::` containers that use hashing, we won’t even know the value of `hSize`, so we will *have* to trust that the table takes care of that modulus internally.

So integer keys are easy, but you can’t always take them for granted. I once worked for a company that assigned an integer ID number to every employee. When the computerized system for the company payroll was first instituted, the way the IDs were assigned went 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 in subsequent years 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 whose names came just before and after the new person’s.

Because of this scheme, more than 3/4 of the ID numbers in the company were evenly divisible by 5.

Now, suppose we took those numbers and hashed them into a table of size `hSize==100`.

```
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, as it happens, a very easy fix for this: add one more element to the table. The same set of IDs will do very well with an `hSize` of 101:

```
int hash(int i) {return i % 101;}
```

Check it out:

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

keys	hash to
:	:

The lesson here: the distribution of the original key values is important.

### 3.1.1 The Curious Power of the Prime Modulus

The fact that 101 worked so well for `hSize` is no accident. The trick of taking the integer key modulo `hSize`

- preserves uniform distribution if already present in a set of keys, and
- tends to increase the uniformity in a set of keys *if* `hSize` is a prime number.

Consequently, it's a standard part of hashing "lore" to try and use prime (or nearly prime) numbers for the hash table size. Shortly, we'll see that, for some collision handling schemes, the use of prime table sizes is particularly important.

## 3.2 Hashing Character Strings

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 bit careful to get an appropriate distribution. For one thing, although a `char` could be any of 255 different values, most strings actually contain only the 96 "printable" characters starting at 32 (blank).

In addition, we often want to make sure that similar strings, likely to occur together, don't hash to the same location. For example, many words differ from one another only in having two adjacent characters transposed (and, if we were programming a spelling checker, you might want to consider that character transposition is a very common spelling error). So a simple hash function like this:

```
unsigned hash (const string& s)
{
    unsigned h = 0;
    for (int i = 0; i < s.length(); i++)
        h += s[i];
    return h;
}
```

doesn't work very well. Words that differ only by transposition of characters would have the same hash value.

A better approach is to use multipliers to make every character position "count" differently in the final sum.

```
int hash (const string& s)
{
    int h = 0;
    for (int i = 0; i < s.length(); i++)
        h = (C*h + s[i]) % M;
    return h;
}
```

where `C` is an integer multiplier and `M` is a modulus, used to keep the whole sum from overflowing. These are usually chosen as prime numbers. `C` can be a small prime, but `M` needs to be large since this function will return hash values in the range `0 ... M-1`, which has to be at least as large as the hash table size.

- Actually a great deal depends on the programming language. In some languages, if an overflow occurs on an integer multiplication, the program terminates with a runtime error. The modulus `M` is necessary in those languages to prevent that kind of overflow.
  - In those languages, we would probably choose `M` as the largest prime less than `IntMax/C`, where `IntMax` is the largest possible integer value.
- In C++, however, integer overflow does not result in a runtime error (although it may be possible to trigger checks for this as compiler options in some compilers). The overflowed value is stored, dropping the high-order bits that don't fit. With signed integers, this can often have the perplexing result that multiplying two large positive integers can yield a negative answer.

For example, this code:

```
#include <limits>
:
int intMax = std::numeric_limits<int>::max();
int k = 2*intMax;
cout << intMax << ' ' << k << endl;
```

prints: 2147483647 -2

For this reason, C++ programmers are likely to

1. Drop the modulus from the calculation.
2. Compute the hash as an unsigned integer:

```
unsigned hash (const string& s) { unsigned h = 0; for (int i = 0; i < s.length(); i++) h = C*h + s[i]; return h; }
```

### 3.3 Hashing Compound Structures

When you need a hash function for a more elaborate data type, you generally try to

- figure out which components of the compound type are critical to identifying the object
- then compute hash functions on those components and combine those hash values into an overall hash function.

```
class Book {  
public:  
    Book (const Author& author,  
          const string& title,  
          const string& isbn,  
          const Publisher& publisher,  
          int editionNumber,  
          int yearPublished);  
    ;  
    int hash() const;  
private:  
    Author theAuthor;  
    string theTitle;  
    string theIsbn;  
    Publisher thePublisher;  
    int theEdition;  
    int theYear;  
};
```

For example, if we wanted a hash table of books, we would probably take advantage of the fact that each book has a unique ISBN number:

```
unsigned hash (const string& s)  
{  
    unsigned h = 0;  
    for (int i = 0; i < s.length(); i++)  
        h = (7*h + s[i]);  
    return h;  
}  
  
int Book::hash() const  
{  
    return hash(theIsbn);  
}
```

so that hashing a book turns into a simple problem of hashing a single string.

But what if we didn't have that nice convenient ISBN field? Then we would need to use a combination of the other fields that, combined, would uniquely identify the book:

```
int Book::hash() const  
{  
    return theAuthor.hash() + 73*hash(theTitle)  
        + 557*thePublisher.hash() + 677*theEdition;  
}
```

(Why not theYear? Because once we have determined the publisher and the edition, the year is already fixed, so adding that in as well won't help distinguish one book from another.) Notice how this hash function breaks down into a series of hashes on other data types, including strings and integers.

Notice, also, the recurring theme of using prime numbers as multipliers. Prime numbers as multipliers are useful in minimizing collisions when the hash values of different components come out as equal values or as simple multiples of one another.

### 3.4 Hashing and Equality

A key requirement if we are going to use hashing is that comparing hash codes is treated as a way to see if two values *might* be equal to one another.

For a good hash function,

- If  $x == y$ , then  $\text{hash}(x) == \text{hash}(y)$ .
- If  $\text{hash}(x) == \text{hash}(y)$ , then there is a good chance that  $x == y$ .
- If  $\text{hash}(x) != \text{hash}(y)$ , then  $x != y$ .

As we have noted some time ago when discussing [relational operators](#), we often have choices as to what we want `operator==` to mean for a newly created ADT.

If we have a Book ADT that implements its operator== by comparing ISBNs, then we whould hash on the ISBN. But if that Book ADT tests for equality by comparing titles and author names, the we should hash books on their titles and author names.

# Resolving Collisions

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Structural Indexing](#)
  - [1.1 Implementation](#)
  - [1.2 Complexity of Separate Chaining](#)
- [2 Open Addressing](#)
  - [2.1 Implementation](#)
  - [2.2 Choosing f\(i\)](#)
  - [2.3 Analysis of Open Addressing](#)

OK, we've got our hash functions. They're not perfect, so we can expect collisions. How do we resolve these collisions?

There are two general approaches:

- [Structural Indexing](#): The hash table performs as an “index” to a set of structures that hold multiple items.
- [Open Addressing](#): Search for an open slot within the table.

## 1 Structural Indexing

Historically, one of the most common approaches to dealing with collisions has been to use fixed size [buckets](#), e.g., an array that can hold up to k (some small constant) elements.

The problem with this is that if we get more than k collisions at the same location, we still need to fall back to some other scheme.

So instead, we'll look at [separate chaining](#), in which the hash table is implemented as an array of variable sized containers that can hold however many elements that have actually collided at that location. Typical choices for this container would be a linked list (which is where the term “chaining” actually comes from) or a tree-based set. Although these containers are variable size, some people still call them “buckets”.

### 1.1 Implementation

```
template <class T,
          int hSize,
          class HashFun,
          class CompareEQ=equal_to<T> >
class hash_set
{
    typedef list<T> Container;

public:
    hash_set (): buckets(hSize), theSize(0)
    {
        :
    }

private:
    vector<Container> buckets;
    HashFun hash;
    CompareEQ compare;
    int theSize;
};
```

We'll illustrate separate chaining as a vector of linked lists of elements.

This template takes more parameters than usual. The purpose of the parameters T and hSize should be self-evident.

HashFun is a functor class used to provide the hash function hash.

CompareEQ is another functor class, used to provide an equality comparison (like the less-than comparators used in std::set and std::map). This defaults to the std functor class equal\_to, which uses the T class's == operator.

#### 1.1.1 Finding the Bucket

```
private:
    Container& bucket (const T& element)
    {
        return buckets[hash(element) % Size];
    }
```

This utility function locates the list that will contain a given element, if that element really is somewhere in the table.

#### 1.1.2 Searching the Set

Now we use the `bucket()` function to implement the set `count()` function.

```
int count (const T& element) const
{
    const Container& theBucket = bucket(element);
    return (find_if(theBucket.begin(),
                    theBucket.end(),
                    bind1st(compare, element))
            == theBucket.end()) ? 0 : 1;
}
```

First we use `bucket()` to find the list where this element would be. Then we search the list for an item equal to `element`.

The `bind1st` function is rather interesting. Given a functor `foo` that takes two parameters, e.g., `foo(x,y)`, `bind1st(foo,w)` creates a single-parameter functor equivalent to calling `foo` with `w` for the first parameter. In this case, I have used `bind1st` because `find_if` uses a single-parameter functor or function as the test to determine when an appropriate item has been found. The functor `compare` is the comparison functor supplied to the `hash_set` template (defaults to `equal_to`, so `bind1st(compare,element)` is a functor that takes a single parameter and returns true if that parameter is equal to `element`.

There is a similar `bind2nd` function that works on the second parameter of a two-parameter function.

### 1.1.3 Inserting into the Set

The `insert` code is quite similar.

```
void insert (const T& element)
{
    Container& theBucket = bucket(element);
    Container::iterator pos =
        find_if(theBucket.begin(), theBucket.end(),
                bind1st(compare, element));
    if (pos == theBucket.end())
    {
        theBucket.push_back(element);
        ++theSize;
    }
    else
        *pos = element;
}
```

We use `bucket()` to find the list, then search the list for the element we want to insert. If it's already in there, we replace it. If not, we add it to the list.

### 1.1.4 Removing from the Set

And `erase` follows pretty much the same pattern.

```
void erase (const T& element)
{
    Container& theBucket = bucket(element);
    Container::iterator pos =
        find_if(theBucket.begin(), theBucket.end(),
                bind1st(compare, element));
    if (pos != theBucket.end())
    {
        theBucket.erase(pos);
        --theSize;
    }
}
```

You can [run these algorithms](#) to see them in action. You might take note that, as the chains grow longer, the search times become more “linear search” like. Of course, if we increased the table size, we would hope that the data elements would be dispersed across the larger table, making the average list length shorter. So there is a very direct tradeoff here between memory use (table size) and speed.

## 1.2 Complexity of Separate Chaining

Suppose we have inserted  $N$  items into a table of size `hSize`.

- In the worst case, all  $N$  items will hash to the same list, and we will be reduced to doing a linear search of that list:  $O(N)$ . (Your text uses sets instead of lists for the buckets, which would reduce this cost to  $O(\log N)$ .) On the other hand, the use of sets requires that the data elements support a `<` comparison instead of (or in addition to) the `==` comparison required with list buckets.
- In the average case, we assume that the  $N$  items are distributed evenly among the lists. Since we have  $N$  items distributed among `hSize` lists, we are looking at  $O\left(\frac{N}{hSize}\right)$ .

If `hsize` is much larger than  $N$ , and if our hash function uniformly distributes our keys, then most lists will have 0 or 1 item, an the average case would be approximately  $O(1)$ . But if  $N$  is much larger than `hSize`, we are looking at an  $O(N)$  linear search sped up by a constant factor (`hSize`), but still  $O(N)$ . So hash tables let us trade space for speed.

## 2 Open Addressing

In open addressing, 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.

```
enum HashStatus { Occupied, Empty, Deleted };

template <class T>
struct HashEntry
{
    T data;
    HashStatus info;

    HashEntry(): info(Empty) {}
    HashEntry(const T& v, HashStatus status)
        : data(v), info(status) {}

};
```

Each slot in the hash table contains one data element and a status field indicating whether that slot is *occupied*, *empty*, or *deleted*.

## 2.1 Implementation

```
template <class T, int hSize, class HashFun,
         class CompareEQ=equal_to<T> >
class hash_set
{
public:
    hash_set(): table(hSize), theSize(0)
    {}
    ...
private:
    int find (const T& element, int h0) const
    :
    vector<HashEntry<T> > table;
    HashFun hash;
    CompareEQ compare;
    int theSize;
};
```

The hash table itself consists of a vector/array of these HashEntry elements.

Collisions are resolved by trying a series of locations,  $h_0, h_1, h_{hSize-1}$ , until we find what we are looking for. These locations are given by

$$h_i(\text{key}) = (\text{hash}(\text{key}) + f(i))$$

where  $f$  is some integer function, with  $f(0) = 0$ . We'll look at what makes a good  $f$  in a little bit.

With these locations, the basic idea is

- Searching: try cells  $h_i(\text{key}), i = 0, 1, \dots$  until we find the key we want or an empty slot.
- Inserting: try cells  $h_i(\text{key}), i = 0, 1, \dots$  until we find the same key, an empty slot, or a deleted slot. Put the new key there, and mark the slot “occupied”.
- Erasing: try cells  $h_i(\text{key}), i = 0, 1, \dots$  until we find the key we want or an empty slot. If we find the key, mark that slot as “deleted”.

### 2.1.1 find()

Here's a “utility” search function for use with open addressing. It takes as parameters the element to search for and the hash value of that element.

```
int find (const T& element, int h0) const
{
    unsigned h = h0 % hSize;
    unsigned count = 0;
    while ((table[h].info == Deleted ||
           (table[h].info == Occupied
             && (!compare(table[h].data, element)))
           && count < hSize))
    {
        ++count;
        h = (h0 + f(count)) % hSize;
    }
    if (count >= hSize
        || table[h].info == Empty)
        return hSize;
    else
        return h;
}
```

The **loop condition** is fairly complicated and bears discussion. There are three ways to exit this loop:

- If we hit an “Empty” space (i.e., not “Deleted”, and not “Occupied”)
- If we hit an “Occupied” space that has the key we’re looking for
- If we have tried `hSize` different positions. (There’s no place else to look!)

## 2.1.2 Searching a Set

With that utility, search operations like the set `count()` are easy. We simply compute the hash value and then call `find`. Then we check to see if the element was found or not.

```
int count (const T& element)
{
    unsigned h0 = hash(element);
    unsigned h = find(element, h0);
    return (h != hSize) ? 1 : 0;
}
```

## 2.1.3 Removing from a Set

```
void erase (const T& element)
{
    unsigned h0 = hash(element);
    unsigned h = find(element, h0);
    if (h != hSize)
        table[h].info = Deleted;
}
```

The code to remove elements is just as simple. We compute the hash function and try to find that element. If it’s found, we mark that slot “Deleted”.

## 2.1.4 Adding to a Set

Inserting is a bit more complicated.

```
bool insert (const T& element)
{
    unsigned h0 = hash(element);
    unsigned h = find(element, h0);      ①
    if (h == hSize) {
        unsigned count = 0;
        h = h0;
        while (table[h].info == Occupied    ②
                && count < hSize)
        {
            ++count;
            h = (h0 + f(count)) % hSize;
        }
        if (count >= hSize)
            return false; // could not add
        else
        {
            table[h].info = Occupied;
            table[h].data = element;
            return true;
        }
    }
    else { // replace
        table[h].data = element;
        return true;
    }
}
```

- ① Because this is a set (and not a multiset) we first do an ordinary search to see if the element is already there.
- ② If not, we need to find a place to put it. The loop that does this looks a lot like 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 in which 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”) will suffice.

## 2.2 Choosing $f(i)$

The function `f(i)` in the `find` and `insert` functions controls the sequence of positions that will be checked. On our  $i^{\text{th}}$  try, we examine position

$$h_i(\text{key}) = (\text{hash}(\text{key}) + f(i))$$

```
int find (const T& element, int h0) const
{
    unsigned h = h0 % hSize;
    unsigned count = 0;
    while ((table[h].info == Deleted ||
            (table[h].info == Occupied
             && (!compare(table[h].data, element))))
            && count < hSize)
    {
        ++count;
        h = (h0 + f(count)) % hSize;
    }
    if (count >= hSize
        || table[h].info == Empty)
        return hSize;
    else
        return h;
}
```

The most common schemes for choosing  $f(i)$  are

- linear probing
- quadratic probing
- double hashing

### 2.2.1 Linear Probing

$$f(i) = i$$

If a collision occurs at location  $h$ , we next check location  $(h+1) \% \text{hSize}$ , then  $(h+2) \% \text{hSize}$ , then then  $(h+3) \% \text{hSize}$ , and so on.

You should [run these algorithms](#). Some things to look for:

- Note how easy it is for keys to “clump up” in contiguous blocks.
- From the initial setup, try removing Baker, then searching for Davis, then adding Smith. Can you see why “Deleted” slots must be treated differently depending upon whether we’re looking for an existing element or looking for a place to insert a new one?

### 2.2.2 Quadratic Probing

$$f(i) = i^2$$

If a collision occurs at location  $h$ , we next check location  $(h+1) \% \text{hSize}$ , then  $(h+4) \% \text{hSize}$ , then then  $(h+9) \% \text{hSize}$ , and so on.

This function tends to reduce clumping (and therefore 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  $\text{hSize}$  is not a prime number.

Again, try [running these algorithms](#). Try adding and removing several items with the same hash code so that you can see the difference in how collisions are handled in the linear and quadratic cases.

Again, prime numbers!

Remember the earlier discussion about how  $\% \text{hSize}$  tends to improve the key distribution when  $\text{hSize}$  is prime? You can see why it’s part of programming “folklore” that hash tables should be prime-number sized, even if most programmers couldn’t say just *why* that’s supposed to be good.

### 2.2.3 Double Hashing

$$f(i) = i * h_2(\text{key})$$

where  $h_2$  is a second hash function.

If a collision occurs at location  $h$ , and  $h2 = h_2(\text{key})$ , we next check location  $(h+h2) \% \text{hSize}$ , then  $(h+2*h2) \% \text{hSize}$ , then then  $(h+3*h2) \% \text{hSize}$ , 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.

## 2.3 Analysis of Open Addressing

Define  $\lambda$ , 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  $\lambda$ .
- Consequently, the probability of an arbitrary cell being empty is  $1 - \lambda$

- The average number of cells we would expect to examine before finding an open cell is therefore  $\frac{1}{1-\lambda}$ .

Now, we never look at more than `hSize` spaces, so, for an ideal collision strategy, finds and inserts are, on average,

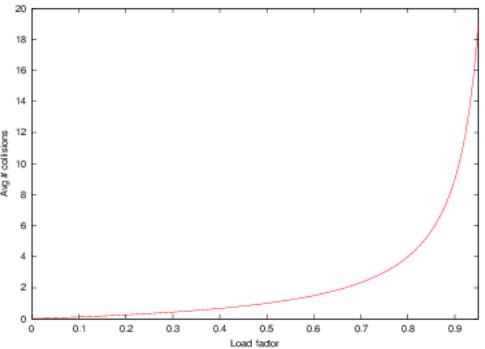
$$O(\min(1/(1-\lambda), hSize))$$

Here you can see the behavior of the function  $1/(1 - \lambda)$  as the load factor,  $\lambda$  increases.

If the table is less than half full ( $\lambda < 0.5$ ), then we are talking about looking at, on average, no more than 2 slots during a search or insert. That's not bad at all. But as  $\lambda$  gets larger, the average number of slots examined grows toward `hSize` (and, if the table is getting full, then  $N$  is approaching `hSize`, so we are once again degenerating toward  $O(N)$  behavior).

So, the 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. But if we let the load factor get much higher, we start seeing  $O(N)$  performance.

Of course, none of the collision resolution schemes we've suggested is truly ideal, so keeping the load factor down to a reasonable level is probably even more important in practice than this idealized analysis would indicate.



# Rehashing (Variable Hashing)

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

[1 Expanding the hash Table](#)

[2 Saving the Hash Values](#)

Hash tables offer exceptional performance when not overly full.

This is the traditional dilemma of all array-based data structures:

- Make the table too small, performance degrades and the table may overflow
- Make the table too big, and memory gets wasted.

*Rehashing* or *variable hashing* attempts to circumvent this dilemma by expanding the hash table size whenever it gets too full.

Conceptually, it's similar to what we do with a vector that has filled up.

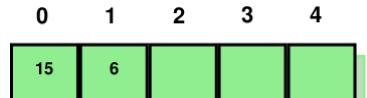
## 1 Expanding the hash Table

For example, using open addressing (linear probing) on a table of integers with  $\text{hash}(k)=k$  (assume the table does an internal `% hSize`):

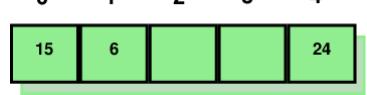
We know that performance degrades when  $\lambda > 0.5$

Solution: rehash when more than half full

So if we have this table, everything is fine.



But if we try to add another element (24), then more than half the slots are occupied ...



So we expand the table, and use the hash function to relocate the elements within the larger table.



The actual expansion mechanism is similar to what we do for vectors. In fact, if we stored the table in a vector, we could use the vector `resize()` function to force the expansion of the table.

However it's important to remember that the value of  $\text{hash}(x) \% \text{hSize}$  changes if `hSize` changes. So the elements need to be repositioned within the new larger hash table.

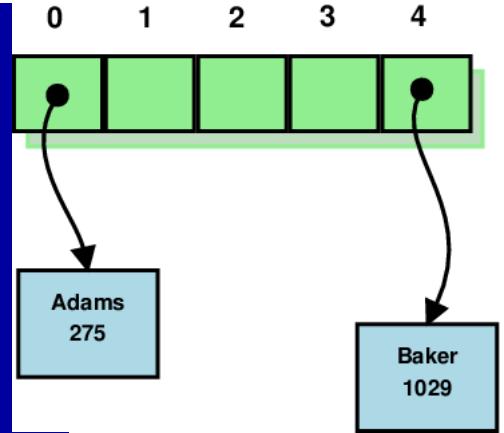
In this case, I've shown the hash table size doubling, because that's easy to do, despite the fact that it doesn't lead to prime-number sized tables.

If we were going to use quadratic probing, we would probably keep a table of prime numbers on hand for expansion sizes, and we would probably choose a set of primes such that each successive prime number was about twice the prior one.

## 2 Saving the Hash Values

The rehashing operation can be quite lengthy. Luckily, it doesn't need to be done very often.

We can speed things up somewhat by storing the hash values in the table elements along with the data so that we don't need to recompute the hash values. Also, if we structure the table as a vector of *pointers to* the hash elements, then during the rehashing we will only be copying pointers, not the entire (potentially large) data elements.



# Hash-Based Sets and Maps

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Examples: The Unordered Set](#)
  - [1.1 An Unordered Set of Strings](#)
- [2 Supplying a Hash Function](#)
  - [2.1 std::hash](#)
  - [2.2 Programmer-defined types](#)
  - [2.3 Passing A Hash Function to an Unordered Container](#)

## C++ 11

The original associative containers (sets and maps) in the C++ std library are based, as we have seen, on balanced binary trees. There are times when even the  $O(\log \text{size}())$  performance of these containers is considered too slow.

The C++11 standard added hashing-based versions of these containers to serve in such circumstances.

The hash-based versions of set and map (and of their “multi-” cousins) will offer an average of nearly  $O(1)$  time for insertion and searching. As always, with hashing, we pay for this increase in speed with an increase in memory required. To guarantee the  $O(1)$  time, these classes will use rehashing when the tables get full enough to degrade the performance.

The tree-based set and map containers have the property that they keep their keys in order. When we use iterators to look at the contents of a `std::set`, for example, we get the data in ascending order.

Hash tables, on the other hand, by their very nature try to distribute their keys as randomly as possible. So one of the things that we give up when using hash-based storage is that ordering. We can still use iterators to get at all the keys, but there’s no telling what in order we will see those data values appear. Because of this, the new hash-based containers have been dubbed *unordered associative containers*.

## 1 Examples: The Unordered Set

When we use ordered sets, based on binary search trees, we are required to supply an ordering relational function such as a less-than operator. With hash-based containers, we have to supply instead an equality comparison function and a hashing function. The first will usually default to `operator==`. The hashing function has default values for strings and other “built-in” types, but we will usually have to supply our own when building unordered sets or maps on our own ADTs.

### 1.1 An Unordered Set of Strings

Here is an example of the use of an unordered set to remove all duplicate strings in a vector.

```
#include <unordered_set>
#include <string>
#include <vector>

void discardDuplicates (vector<string>& v)
{
    unordered_set<string> unique;
    for (int i = 0; i < v.size(); ++i)
    {
        unique.insert (v[i]);
    }
    v.clear();
    for (unordered_set<string>::iterator p = unique.begin();
         p != unique.end(); ++p)
        v.push_back(*p);
}
```

```
}
```

The algorithm takes advantage of the fact that sets ignore attempts to add duplicate elements.

So the first loop simply adds every string in the vector into a set, which has the side effect of only keeping a single copy of any duplicated values. Then the second loop copies the resulting collection of unique strings from the set back into the vector.

There's not a whole lot special about the unordered set we are using. In fact, this code would work, though possibly a bit more slowly, with an ordinary tree-based `std::set`. That's not an accident. The unordered associative containers have an interface essentially identical to that of their tree-based ordered counterparts. The only differences that you will find are some new operations to force rehashing, and some changes to the constructors to allow explicit passing of hash functions and equality tests. (We did not need to do so for this example, because the default values for strings would work perfectly well for us.)

This code, by the way, could easily be simplified, but I used a slightly wordy approach so that you could see how the insertion operation looks just like its ordered counterpart. Here's another possibility:

```
#include <unordered_set>
#include <string>
#include <vector>
#include <algorithm>

void discardDuplicates (vector<string>& v)
{
    unordered_set<string> unique (v.begin(), v.end());
    v.assign(unique.begin(), unique.end());
}
```

What's the complexity of this? In the worst case, there are no actual duplicates, and the first line adds `v.size()` strings at a worst-case cost of  $O(\text{unique.size}())$  for each addition:

```
void discardDuplicates (vector<string>& v)
{
    unordered_set<string> unique (v.begin(), v.end());      // O(v.size()^2)
    v.clear();                                                 // O(1)
    v.reserve (unique.size());                                // O(1)
    copy (unique.begin(), unique.end(), back_inserter(v)); // O(v.size())
}
```

for an overall  $O(v.size()^2)$ .

But, on average, those insertions into `unique` will be  $O(1)$ , so we get an average complexity:

```
void discardDuplicates (vector<string>& v)
{
    unordered_set<string> unique (v.begin(), v.end());      // O(v.size())
    v.clear();                                                 // O(1)
    v.reserve (unique.size());                                // O(1)
    copy (unique.begin(), unique.end(), back_inserter(v)); // O(v.size())
}
```

for an overall  $O(v.size())$ .

It's worth noting that the algorithm would work with the tree based ordered set as well, but then each insertion is  $O(\log \text{unique.size}())$  on average, which leads to

```
void discardDuplicates (vector<string>& v)
{
    set<string> unique (v.begin(), v.end());           // O(v.size() log(v.size()))
    v.clear();                                         // O(1)
    v.reserve (unique.size());                         // O(1)
    copy (unique.begin(), unique.end(), back_inserter(v)); // O(v.size())
}
```

for an overall average  $O(v.size() \log(v.size()))$ .

Neither the `Set` and `unordered_set` versions of this algorithm will preserve the original ordering of the strings in `v`. However, the `Set` version will result in the output `v` containing these strings in sorted order, while the `unordered_set` version leaves them in an arbitrary, largely unpredictable order.

## 2 Supplying a Hash Function

### 2.1 std::hash

The C++ `std` library provides hash functions for most of the types you would expect. Most of these hash functions are in the `<functional>` header. Others are in the same headers where you find the types in the first place. For example, the `string` hash function is provided in `<string>`.

The hash functions are provided by the template `std::hash`, but `std::hash` is not a function - it's a template class that can be used to "generate" hash functions and save them in variables.

```
#include <functional>
#include <string>
```

```

    :
std::hash<int> int_hash; // int_hash(...) is a hash function for integers
std::hash<double> dbl_hash; // dbl_hash(...) is a hash function for doubles
std::hash<string> str_hash; //str1_hash(...) is a hash function for strings

```

## 2.2 Programmer-defined types

For more complex, programmer-defined types, we build our own hash functions by combining hash values for the type's data members.

For example, given

```

class Book {
public:
    Book (const Author& author,
          const string& title,
          const Publisher& publisher,
          int editionNumber,
          int yearPublished);
    :
    std::size_t hash() const;
private:
    Author theAuthor;
    string theTitle;
    Publisher thePublisher;
    int theEdition;
    int theYear;
};

bool operator==(const Book& left, const Book& right);

```

We might use this as a possible hash function:

```

std::hash<string> str_hash;

std::size_t Book::hash() const
{
    return theAuthor.hash() + 31*str_hash(theTitle)
        + 57*thePublisher.hash() + 701*theEdition + 131*theYear;
}

```

This assumes that our Author and Publisher classes provide their own hash() functions. So we combine those values with hashes for integers and the title string. (Note again the pervasive use of prime numbers in hash-related calculations.)

For consistency, we will assume that the equality operator works on the same fields (it's crucial that any two objects that are "equal" must have the same hash value):

```

bool operator==(const Book& left, const Book& right);
{
    return left.theAuthor == right.theAuthor &&
        left.theTitle == right.theTitle &&
        left.thePublisher == right.thePublisher &&
        left.theEdition == right.theEdition &&
    } left.theYear == right.theYear;
}

```

## 2.3 Passing A Hash Function to an Unordered Container

There are three ways to tell an unordered container what hash function you want to use.

- One way is to pass it as a function/functor when constructing a new set or map variable:

```

std::size_t bookHash (const Book& b)
{
    return (std::size_t)b.hash();
}
:
const std::size_t initialSize = 2048;
unordered_map<Book, Price> recommendedPrices (initial_size, bookHash);
:
void purchase(Book b, Price p)
{
    if (p < recommendedPrices[b])
        cout << "You got a bargain!" << endl;
}

```

- If you want to make the hash function a part of the data type (so that multiple variables will all use the same hash function), you can do so by declaring a functor class to provide the hash function, and pass that as a template parameter:

```

class BookHash {
public:
    std::size_t operator() (const Book& b) const

```

```

    {
        return (std::size_t)b.hash();
    }
}
:
typedef unordered_map<Book, Price, BookHash> HashTable;

HashTable recommendedPrices;
:
void purchase(Book b, Price p)
{
    if (p < recommendedPrices[b])
        cout << "You got a bargain!" << endl;
}

```

- Finally, sometimes you would like to specify a “default” hash function for a type such as Book so that anyone can create unordered sets or maps without creating their own hash functions. (You might recall that, in the prior section, we created `unordered_set<string>` without supplying a string hash function.)

This gets a little bit messier. By default, an unordered container will use the functor `std::hash<Key>` as the (type of the) hash function of type Key. We looked at `std::hash` at the start of this section. In `<functional>`, `std::hash` provides hash functions for the C++ primitive types. Other headers, such as `<string>` extend that same template to provide hash functions on non-primitive types.

```

std::hash<string> hash_s;
std::hash<double> hash_d;

string s = "abc";
double d = 3.14;
cout "s hashes to " << hash_s(s) << " and d hashes to " << hash_d(d) << endl;

```

The problem is, `hash<Key>` does not exist for general types. It is already defined for the C++ basic types such as `float` and `double` and for selected standard classes such as `string`. But if you want to use it with your own types, you have to [specialize](#) the template `std::hash` for your type:

```

namespace std {

    template <>
    struct hash<Book> // denotes a specialization of hash<...>
    {
        std::size_t operator() (const Book& book) const
        {
            return (std::size_t)book.hash();
        }
    };

unordered_map<Book, Price> recommendedPrices; // uses hash<Book>
:
void purchase(Book b, Price p)
{
    if (p < recommendedPrices[b])
        cout << "You got a bargain!" << endl;
}

```

Specialization is one of those powerful but ugly features of C++ that most programmers rarely dabble in. It’s a way of giving an existing template a specific instantiation for selected data types.

# Exams

## Steven J. Zeil

Last modified: Jul 17, 2018

### Contents:

All exams are on Blackboard. Look for the “Exams” section in the navigation board on the left of the course area. You can visit the Exams area early

- The dates of the exam are posted in the course [outline](#). All exams are available from midnight (Eastern time) on the announced starting date until 11:59:59PM (Eastern time) of the announced closing date of the exam.
- You do not need to acquire a proctor for the exam.
- Each exam is timed. The time limit may vary by exam, but will be indicated in the instructions in the Exams area on Blackboard.
  - The time limit is consecutive time – you cannot work for an hour, take a break, then come back and work for another hour. Once you start the exam, the clock is running.
  - If circumstances beyond your control (e.g., a dropped network connection) make it impossible to finish, you must document the circumstance, including the amount of time it cost you, continue as best you can, and then send me your documentation via email.

Do this in as timely a manner as possible.

Note that I consider the time limit to be more than enough time to complete the exam, and so am likely to approve corrective action only for circumstances that cost you a significant fraction of that time.

- Exams are open book, open notes, open internet. You must not, however, consult with any other human except the course instructor regarding the content and/or solutions to this exam.
  - Any material found outside the course website ([https://www.cs.odu.edu/...](https://www.cs.odu.edu/)) for your section of the course in the current semester is subject to the quotation and citation requirements explained in the [syllabus](#).
  - If I have reason to believe that you did not write the answer you have supplied to a question, and you do not provide an appropriate citation for the course, the least that you can expect is a zero for that question.

# Converting Recursion to Iteration

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 A Diversion - Function Calls at the Machine Level](#)
- [2 Converting Recursive Algorithms to Iteration](#)
  - [2.1 Tail Recursion](#)
  - [2.2 Conversion Using Stacks](#)
  - [2.3 Conversion Example: searching a graph, depth-first](#)
- [3 Last Thoughts](#)

A common application of stacks is in converting [recursive algorithms](#) to iterative forms.

Recursion and iteration (looping) are equally powerful. Any recursive algorithm can be rewritten to use loops instead. The opposite is also true. Any iterative algorithm can be written in terms of recursion only.

Given a choice of which to use, our decision is likely to come down to issues of

- expressiveness: some algorithms are just naturally simpler or easier to write with loops. Some are simpler with recursion.

Given the fact that testing and debugging of code usually takes much more effort than writing it in the first case, anything we can do to make code simpler (thus reducing the opportunities for making mistakes) is worth giving serious consideration.

- performance: Iteration is usually (though not always) faster than an equivalent recursion.

## 1 A Diversion - Function Calls at the Machine Level

In a sense, all recursion is an illusion. At the machine level, you have an iterative process for fetching and executing instructions. All function calls (including the recursive ones) are implemented via a runtime stack (called the [activation stack](#)) to keep track of the return addresses, actual parameters, and local variables associated with function calls. Each function call actually results in pushing an [activation record](#) containing that information onto the stack. Returning from a function is accomplished by getting and saving the return address out of the top record on the stack, popping the stack once, and jumping to the saved address.

The actual structure of activation stacks is machine and compiler dependent. A typical one is shown here.

Typical contents of the saved state area would include

- the return address - where to go when returning from the call, and
- contents of critical machine registers at the start of the call

Contents of the params area would be

- the values for the actual parameters being passed to the function
- space for the return value that the call will eventually send back to the caller

The locals area holds the space for any local variables declared within the called function's body.

The collection of all this information for a single call is called an [activation record](#) or, sometimes, a [frame](#). An activation stack is typically managed via two pointers. One is the true top of the stack. The other is a pointer to the top completed frame or activation record - the one describing the function that is currently being executed. These two pointers are often slightly different. That's because, if the currently executing function is getting ready to call some other function, it will get ready by pushing its saved state info and any parameters it wants to pass onto the top of the stack, forming an "incomplete" activation record as shown in the diagram.

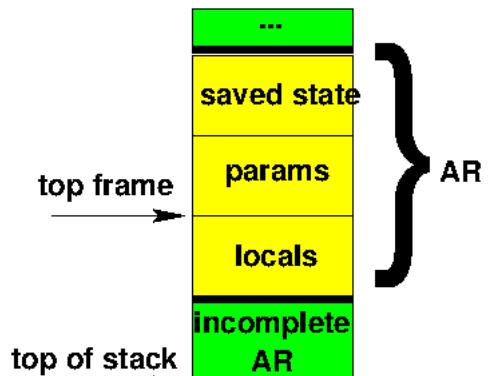
For a function call `foo(a, b+c, d);`, the caller would

- Push state information, including a return address and the current values of the top-of-stack and top-frame pointers.
- Evaluate each of the three parameter expressions, pushing their values onto the stack. If `foo` has a non-void return type, the caller would also push enough space for the return value.
- The caller would then jump to the starting address of `foo`'s function body.

The code in the body of `foo` would typically start by pushing enough additional bytes onto the top of the stack to allow room for all of the local variables declared in that function body. This completes the activation record for the call to `foo`.

The execution of the function body of `foo` then proceeds. When a return statement (or end of the body) is encountered,

- the return value (if any) is written into the space reserved for it in the params area



- a jump is made to the return address recorded in the saved state area

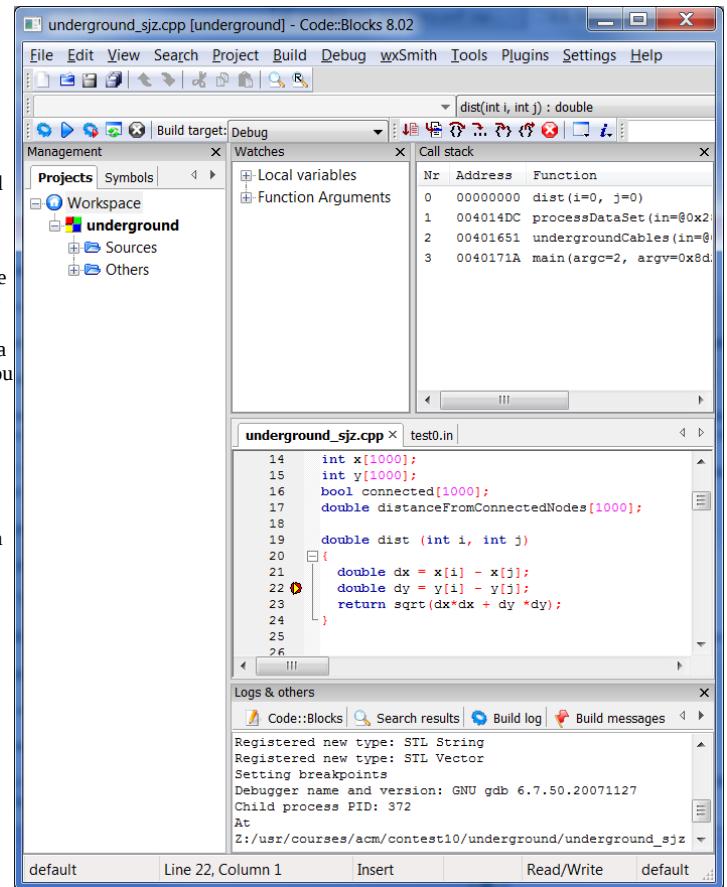
Upon return to the original caller, the caller then

- copies the return value (if any) to some appropriate location
- restores to top frame and top-of-stack pointers to their saved values. In effect, this pops the entire activation record that had been built for foo off of the stack, leaving the caller's activation record back on top.

You can observe activation stacks in almost any debugger. Set a breakpoint and execute the code until that breakpoint is reached. When execution pauses, most debuggers will show not only where you stopped, but how you got there (in the form of the activation stack). In the picture shown here, for example, you can see that we have reached a breakpoint inside a function called dist. In the upper right corner, however you can see the entire activation (call) stack at this point in time. Like all C++ programs, we started by running main. In this program, main called undergroundCables, which called processDataSet, which called dist.

This stack-based approach to execution handles recursive calls naturally. A recursive call is just one that happens to put another activation record for the same function body onto the stack. But that really makes no difference at all in how the machine-level execution treats that call. Arguably, the only difference is what happens when things go wrong. If you have an iterative algorithm and you make a mistake that leads to an infinite loop, the program simply runs forever (or until you kill the program). With a recursive algorithm, if you mess up and get into an infinite recursion, the program runs for a while and then crashes due to a lack of memory. An infinite recursion causes the activation stack to grow longer and longer until it occupies all available memory.

In a sense, then, computers really don't do recursion. What we might write as a recursive algorithm really gets translated as a series of stack pushes followed by a jump back to the beginning of the recursive function, all implemented using the underlying CPU whose internal code is, fundamentally, iterative. With that in mind, you can see why, if we wanted to rewrite our own code to eliminate recursion, stacks are likely to prove useful.



## 2 Converting Recursive Algorithms to Iteration

How, then do we go about converting recursive functions to an iterative form?

### 2.1 Tail Recursion

A function is called `\first-term{tail-recursive}` if each activation of the function will make at most a single recursive call, and will return immediately after that call (with no further calculation performed upon the return value from the call).

Tail recursive routines have an immediate, and simple pattern of conversion.

```
T tailRecursiveFoo(U x, V y)
{
    if (bar(x, y))
        return baz(x,y);
    else
    {
        : // block 1
        return tailRecursiveFoo(w, z);
    }
}
```

becomes

```
T Foo(U x, V y)
{
    while (! bar(x, y))
    {
        : // block 1
        x = w;
```

```

        y = z;
    }
    return baz(x,y);
}

```

### 2.1.1 Tail Recursion Example: binarySearch

An example of tail-recursion is the binary search, which converts by the pattern just shown from the recursive form shown here:

```

unsigned int binarySearch
(T v[], unsigned int n, const T& value)
// search for value in ordered array of data
// return index of value, or index of
// next smaller value if not in collection
{
    binarySearch (v, 0, n, value);
}

unsigned int binarySearch
(T v[], unsigned int low, int high, const T& value)
{
    // repeatedly reduce the area of search
    // until it is just one value
    if (low < high) {
        int mid = (low + high) / 2;
        if (v[mid] < value)
            {
                return binarySearch (v, mid + 1, high, value);
            }
        else
            {
                return binarySearch (v, low, mid, value);
            }
    }
    else
        // return the lower value
        return low;
}

```

into its more familiar iterative form:

```

unsigned int binarySearch
(T v[], unsigned int n, const T& value)
// search for value in ordered array of data
// return index of value, or index of
// next smaller value if not in collection
{

int low = 0;
int high = n;

// repeatedly reduce the area of search
// until it is just one value
while (low < high) {
    int mid = (low + high) / 2;
    if (v[mid] < value)
        {
            low = mid + 1;
        }
    else
        {
            high = mid;
        }
}
// return the lower value
return low;
}

```

## 2.2 Conversion Using Stacks

As noted earlier, CPU's execute recursive code by storing information about each recursive call in an “activation stack”.

If we wanted to convert an algorithm from a recursive form to an iterative form, we could simulate this process with our own stacks.

*Any* recursive algorithm can be converted to an iterative form by using a stack to capture the “history” of

- actual parameters
- local variables

that would have been placed on the activation stack.

The general idea is:

- recursive calls get replaced by push
  - depending on details, may push new values, old values, or both
- returns from recursive calls get replaced by pop
- main calculation of recursive routine gets put inside a loop
  - at start of loop, set variables from stack top and pop the stack

## 2.2.1 Looking at the Pieces

```
T recursiveFoo (U param1, V param2)
{
    U local1;
    V local2 = bar(U,V);
    : // code block 1
    recursiveFoo (local1, local2);
    : // code block 2
    recursiveFoo (param1, local2);
    : // code block 3
    recursiveFoo (local1, param2);
    : // code block 4
}
```

We can think of a recursive function as being divided into several pieces, separated by the recursive calls. (This is a bit of an oversimplification, since we aren't considering what happens if the recursive calls are inside `if` or `loop` statements, but those can be dealt with once you get the basic idea.)

We can simulate calls to the recursive routine by saving, on a stack, all parameters and local variables. In addition, just as a “real” function call needs to know its return address, we may need to save a “location” indicator to let us know which block of code we’re supposed to execute upon a simulated return from a simulated recursive call.

It helps, then, to have a convenient structure to hold each set of information to go on the stack:

```
struct FooStateInfo {
    U param1;
    V param2;
    U local1;
    V local2;
    int location;
};

typedef stack<list<FooStateInfo>> FooStack;
```

## 2.2.2 Getting “parameters” from the stack

Starting with this recursive code, ...

```
T recursiveFoo (U param1, V param2)
{
    U local1;
    V local2;
    : // code block 1
    recursiveFoo (local1, local2);
    : // code block 2
    recursiveFoo (param1, local2);
    : // code block 3
    recursiveFoo (local1, param2);
    : // code block 4
}
```

... we create one large control loop. Inside this loop, we have the blocks of code from the original recursive routine, but the recursive calls at the end of each block is replaced by a push of the information that we want restored upon “return” from a simulated recursive call and another push that sets up a simulated recursive call.

```
T iterativeFoo (U param1, V param2)
{
    U local1;
    V local2;
    FooStack stk;
    stk.push ({param1, param2, local1, local2, 1});
    while (!stk.empty())
    {
        // get parameters from stack
        const FooStackInfo& stkTop = stk.top();
        param1 = stkTop.param1;
        param2 = stkTop.param2;
        local1 = stkTop.local1;
        local2 = stkTop.local2;
        stk.pop();

        switch (stkTop.location) {
```

```

        case 1:
            : // code block 1
            stk.push ({param1, param2, local1, local2, 2});
            stk.push ({local1, local2, local1, local2, 1});
            break;
        case 2:
            : // code block 2
            stk.push ({param1, param2, local1, local2, 3});
            stk.push ({param1, local2, local1, local2, 1});
            break;
        case 3:
            : // code block 3
            stk.push ({param1, param2, local1, local2, 4});
            stk.push ({local1, param2, local1, local2, 1});
            break;
        case 4:
            : // code block 4
            break;
    }
}
}

```

The final block (and any block that, in the original routine, does not do further recursive calls) does not get these pushes. If the original routine only makes recursive calls under certain conditions:

```

if (bar(local1, local2))
{
    recursiveFoo(local1, local2);
}

```

then the same thing happens in the iterative form with the pushes that simulate the recursive call:

```

if (bar(local1, local2))
{
    stk.push ({param1, param2, local1, local2, <...>});
    stk.push ({local1, local2, local1, local2, 1});
}

```

### 2.2.3 Simplifying

Now, the general approach to conversion outlined here is almost always overkill.

- We seldom need to save *every* parameter and local variable.
  - For example, pure inputs whose values are never changed won't need to be put onto the stack.
- If a recursive call occurs at the very end of the routine, we might not need to set up the simulated return.
- If two or more recursive calls immediately follow one another, we can put them both on the stack immediately, rather than simulate a return in between the two.

You need to look carefully at the algorithm you are converting to see if these or other simplifications are possible.

## 2.3 Conversion Example: searching a graph, depth-first

In an [earlier lesson](#), we saw the following as the prototype for depth-first traversal of a graph:

```

void depthFirst (Digraph& dg, Vertex v)
{
    set<Vertex> visited;
    depthFirst (dg, v, visited);
}

void depthFirst (Digraph& dg, Vertex v, set<Vertex>& visited)
{
    visited.insert (v);
    for (AllOutgoingEdges e = dg.outbegin(v);
         e != dg.outend(v); ++e)
    {
        Vertex w = (*e).dest();
        if (visited.count(w) == 0)
            depthFirst (dg, w, visited);
    }
}

```

Let's modify this slightly to turn it into a search to see if a vertex target is in the graph and can be reached by starting from v:

```

bool depthFirstSearch (Digraph& dg, Vertex startingFrom, Vertex target)
{
    set<Vertex> visited;
    return depthFirst (dg, startingFrom, target, visited);
}

```

```

bool depthFirstSearch (Digraph& dg, Vertex v, Vertex target,
                      set<Vertex>& visited)
{
    if (v == target)
        return true;
    visited.insert (v);
    for (AllOutgoingEdges e = dg.outbegin(v);
         e != dg.outend(v); ++e)
    {
        Vertex w = (*e).dest();
        if (visited.count(w) == 0 &&
            depthFirstSearch (dg, w, target, visited))
            return true;
    }
    return false;
}

```

Can we convert this to an iterative form?

There's only a single recursive call in this routine. It passes parameters dg, v, target, and visited. But neither dg nor target change from one recursive call to another, so we won't need to push them onto the stack. There's only one local variable that we would need to worry about restoring, the iterator e. The local variable w is only used to set up the new value of v for the next call. So our stack elements will probably look something like:

```

struct dfsStackElement {
    Vertex v;
    set<Vertex> visited;
    AllOutgoingEdges e;
    int location;
};

```

But the whole point of putting these variables onto the stack is so that, when we simulate a return, we can restore the old values of those variables. If you think a bit about the role of the visited set, you will realize that we never want to restore an old value of visited; we never want to pretend that we have not visited a vertex just because we returned from the call where we did so. (That's actually why the function passes visited as a non-const reference, so that changes made anywhere in the recursion will be seen when we return). So we actually don't need to push visited:

```

struct dfsStackElement {
    Vertex v;
    AllOutgoingEdges e;
    int location;
};

```

So our overall skeleton for the iterative algorithm will be

```

bool depthFirstSearch (Digraph& dg, Vertex v, Vertex target,
                      set<Vertex>& visited)
{
    Vertex w;
    AllOutgoingEdges e;
    stack<dfsStackElement> stk;
    stk.push ({v, e, 1});
    while (!stk.empty())
    {
        const dfsStackElement& stkTop = stk.top();
        v = stkTop.v;
        e = stkTop.e;
        int location = stkTop.location;
        stk.pop();
        switch (location) {
            :
        }
    }
    return false;
}

```

Now, let's think about the cases that we would face each time around the loop and how those match up against the recursive form.

For this, I find it useful to imagine that the loop is “unrolled”:

```

bool depthFirstSearch (Digraph& dg, Vertex v, Vertex target,
                      set<Vertex>& visited)
{
    if (v == target)
        return true;
    visited.insert (v);

    AllOutgoingEdges e = dg.outbegin(v);

    if (e == dg.outend(v)) return false;
    : do the top part of the loop body
    depthFirstSearch (dg, w, target, visited);
    : do the top part of the loop body
    ++e;

    if (e == dg.outend(v)) return false;
    : do the top part of the loop body
    depthFirstSearch (dg, w, target, visited);
}

```

```

    : do the top part of the loop body
++e;

if (e == dg.outend(v)) return false;
: do the top part of the loop body
depthFirstSearch (dg, w, target, visited);
: do the top part of the loop body
++e;
:
}

```

1. We may have just entered a recursive call. v is a vertex that we are visiting for the first time, and e tells us only how we got to v.

In the recursive version we would check to see if v is equal to target, meaning that we found what we were looking for. If not, we mark v as having been visited and then we'll start an iteration over v's outgoing edges.

1. We may have just come to the top of the loop, either because it's a new iteration or because we are repeating an existing iteration.

In the recursive version we get the destination vertex, check to see if we have already visited it and, if not, do a recursive call.

1. We may have just returned from a recursive call (or decided to bypass making one because the vertex has already been visited).

In the recursive version we increment e and then return to the top of the loop.

So, the iterative form cases probably look something like:

```

bool depthFirstSearch (Digraph& dg, Vertex v, Vertex target,
                      set<Vertex>& visited)
{
    Vertex w;
    AllOutgoingEdges e;
    stack<dfsStackElement> stk;
    stk.push ({v, e, 1});
    while (!stk.empty())
    {
        const dfsStackElement& stkTop = stk.top();
        v = stkTop.v;
        e = stkTop.e;
        int location = stkTop.location;
        stk.pop();
        switch (location)
        {
            case 1:
                if (v == target)
                    return true;
                visited.insert(v);
                e = dg.outbegin(v);
                if (e != dg.outend(v))
                {
                    stk.push ({v, e, 2});
                }
                break;
            case 2:
                Vertex w = e->dest();
                if (visited->count(w) == 0)
                {
                    stk.push ({w, e, 3});
                    stk.push ({w, e, 1});
                }
                break;
            case 3:
                ++e;
                if (e != dg.outend(v))
                {
                    stk.push ({e->dest(), e, 2});
                }
                break;
        }
    }
    return false;
}

```

That should be pretty much correct. Can we simplify it? Not a whole lot. We could replace the switch by a nested if:

```

bool depthFirstSearch (Digraph& dg, Vertex startingFrom, Vertex target)
{
    set<Vertex> visited;
    return depthFirst (dg, startingFrom, target, visited);
}

bool depthFirstSearch (Digraph& dg, Vertex v, Vertex target,
                      set<Vertex>& visited)
{
    Vertex w;
    AllOutgoingEdges e;
    stack<dfsStackElement> stk;
    stk.push ({v, e, 1});
    while (!stk.empty())

```

```

{
    const dfsStackElement& stkTop = stk.top;
    v = stkTop.v;
    e = stkTop.e;
    int location = stkTop.location;
    stk.pop();
    if (location == 1)
    {
        if (v == target)
            return true;
        visited.insert(v);
        e = dg.outbegin(v);
        if (e != dg.outend(v))
        {
            stk.push ({v, e, 2});
        }
    }
    else if (location == 2)
    {
        Vertex w = e->dest();
        if (visited->count(w) == 0)
        {
            stk.push ({w, e, 3});
            stk.push ({w, e, 1});
        }
    }
    else
    {
        ++e;
        if (e != dg.outend(v))
        {
            stk.push ({e->dest(), e, 2});
        }
    }
}
return false;
}

```

Whether that's an improvement is something of a matter of taste.

We could combine the two overloaded forms of the function now:

```

bool depthFirstSearch (Digraph& dg, Vertex startingFrom, Vertex target)
{
    set<Vertex> visited;

    Vertex v;
    AllOutgoingEdges e;
    stack<dfsStackElement> stk;
    stk.push ({v,e, 1});
    while (!stk.empty())
    {
        const dfsStackElement& stkTop = stk.top;
        v = stkTop.v;
        e = stkTop.e;
        int location = stkTop.location;
        stk.pop();
        if (location == 1)
        {
            if (v == target)
                return true;
            visited.insert(v);
            e = dg.outbegin(v);
            if (e != dg.outend(v))
            {
                stk.push ({v, e, 2});
            }
        }
        else if (location == 2)
        {
            Vertex w = e->dest();
            if (visited->count(w) == 0)
            {
                stk.push ({w, e, 3});
                stk.push ({w, e, 1});
            }
        }
        else
        {
            ++e;
            if (e != dg.outend(v))
            {
                stk.push ({e->dest(), e, 2});
            }
        }
    }
    return false;
}

```

Now, personally, I would prefer to work with and debug the recursive form. Some people are so uncomfortable with recursion, however, that they would prefer this iterative form.

## 3 Last Thoughts

Sometimes it's really not worth converting algorithms from recursive to iterative. Some elegant, simple recursive algorithms become horrendously complicated in iterative form. On the other hand, as noted earlier, there are times when we have little choice (e.g., embedded systems).

In these kinds of situations, conversion from recursion to iteration may be the only way to get a system running.

We'll look at another example of this process in a later lesson, when we'll convert a recursive sorting algorithm to an iterative form that is actually generally preferred for performance reasons.

# A Gallery of Algorithmic Styles

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Divide and conquer](#)
- [2 Generic](#)
- [3 Generate and test](#)
- [4 Backtracking](#)
  - [4.1 Example: The 3-Houses Puzzle](#)
  - [4.2 A Tabular Approach to Backtracking](#)
  - [4.3 A Recursive Approach to Backtracking](#)
  - [4.4 An Iterative Framework for Backtracking](#)
  - [4.5 Pruning](#)
  - [4.6 Example: Pruning in 3-Houses](#)
  - [4.7 Simpler Pruning](#)
- [5 Convergent](#)
- [6 Dynamic programming](#)
  - [6.1 Example: combinations](#)
  - [6.2 Example: Edit Distance](#)

Suppose you have to design a brand-new algorithm to do something never done before. You could start from scratch, given the problem statement and a programming language textbook, and you might come up with something useful.

But certain distinctive styles of algorithms have arisen over the years, and being familiar with these may give you an edge in coming up with a well-designed algorithm for a new task.

We've actually seen a number of these already, but it's time to put some names to them and take stock of what we've got:

A first question we might ask about any algorithm is how it manages to process multiple "pieces" of data:

- Iterative: algorithms that work by looping
- Recursive: algorithms that call themselves (directly or indirectly)

Among both iterative and recursive algorithms, we recognize some algorithms by an underlying "idea" of how they work:

- Generic
- Divide and conquer
- Generate and test
- Backtracking
- Convergent
- Dynamic programming

Now let's look at some common forms of both iterative and recursive algorithms.

## 1 Divide and conquer

Perhaps the most widely recognized algorithmic style is "divide and conquer". In this approach to design, a problem is broken into two or more smaller sub-problems, the sub-problems are solved, and the resulting sub-problem solutions are recombined into a whole solution.

If this sounds suspiciously similar to my description of recursion, that's no accident. Divide and conquer algorithms frequently are easily expressed recursively, though iterative forms are not unknown.

Examples of divide and conquer that we have seen include the binary search, merge sort, and quick sort.

## 2 Generic

One of the most recent categories of algorithms to be recognized, *generic* algorithms are those that use a general-purpose "iterator" interface to process a range of data that could, in fact, be residing in almost any kind of container.

The copy function shown here is a good example of a generic algorithm, but so are any algorithms that make liberal use of copy and its “cousins” among the standard library templates.

```
template <class InputIterator,
          class OutputIterator>
OutputIterator copy(InputIterator first,
                    InputIterator last,
                    OutputIterator result)
{
    while (first != last)
    {
        *result = *first;
        result++; first++;
    }
    return result;
}
```

In fact, the STL (standard template library), that eventually became part of the C++ standard library, was originally constructed as a showcase for the idea of generic programming.

## 3 Generate and test

A “generate and test” algorithm uses some relatively quick process to produce a series of “guesses” as to the appropriate solution, then tests each guess in turn to see if it is, in fact, a solution.

You can see generate and test in action in this code for producing random permutations.

```
//  
// Generate a random permutation of the integers from  
// 0 .. n-1, storing the results in array a.  
//  
void permute1 (int a[], int n)  
{  
    for (int i = 0; i < n; i++)  
    {  
        a[i] = rnd(n);  
        while (find(a,a+i,a[i]) != a+i)  
            a[i] = rnd(n);  
    }  
}
```

- The random numbers are used to **generate** a possible value for the  $i^{\text{th}}$  number in the permutation.
- Then a search is used to **test** and see if that number has already been used.

Now, this is actually a pretty terrible way to generate random permutations, and you have seen a better approach in one of the earlier assignments. Generate-and-test is often used as a fall-back when we can't come up with a better algorithm.

## 4 Backtracking

A variation of generate-and-test is **backtracking**. Backtracking is a technique that can be applied to problems where you have a large, but finite number of variables, each of which may take on a number of discrete values, and there is some overall test to decide if the entire set of assignments represents an acceptable solution.

### 4.1 Example: The 3-Houses Puzzle

An example of backtracking can be seen in the solution technique many people take to problems like this:

There are 3 adjacent houses, one red, one blue, and one green on Elm St. Each is occupied by a single person, and each has a garden with one kind of flower.

- Bob does not live in the green house.
- Pat lives between Bob and Sue.
- Sue planted daisies.
- Lilies are planted at the blue house.
- The leftmost house has roses.

List the colors of the houses, the occupants, and their flowers, from left to right.

One way to approach this is to make a grid:

	left	middle	right
House			

	left	middle	right
Flowers			
Person			

and list the possible values for each row:

	left	middle	right
House (blue, green, red)			
Flowers (daisies, lilies, roses)			
Person (Bob, Pat, Sue)			

Now, try assigning values in each slot, until a contradiction is reached with one of the rules.

Let's guess that the leftmost house is Blue:

	left	middle	right
House (blue, green, red)	blue		
Flowers (daisies, lilies, roses)			
Person (Bob, Pat, Sue)			

and the middle one Green

	left	middle	right
House (blue, green, red)	blue	green	
Flowers (daisies, lilies, roses)			
Person (Bob, Pat, Sue)			

and the right one Red:

	left	middle	right
House (blue, green, red)	blue	green	red
Flowers (daisies, lilies, roses)			
Person (Bob, Pat, Sue)			

None of these lead to any contradictions with the rules. So let's move on to the next row.

Let's put daisies in the leftmost house.

	left	middle	right
House (blue, green, red)	blue	green	red
Flowers (daisies, lilies, roses)	daisies		
Person (Bob, Pat, Sue)			

Now we have our first contradiction. Rule 4 says that lilies are planted at the blue house and rule 5 says that the leftmost house has roses. So what do we do? We retract our most recent guess, and try a different value for that same slot. Sticking with alphabetic order, we try lilies:

	left	middle	right
House (blue, green, red)	blue	green	red
Flowers (daisies, lilies, roses)	lilies		
Person (Bob, Pat, Sue)			

Again, a contradiction (rule 5). So we retract that guess and try roses.

	left	middle	right
House (blue, green, red)	blue	green	red
Flowers (daisies, lilies, roses)	roses		
Person (Bob, Pat, Sue)			

But that contradicts rule 4 (Lilies are planted at the blue house.). Now we have tried all possible kinds of flowers at this position. Since none of them work, we know that one of our earlier decisions has already precluded any possible solution. Which one? We're probably not sure. So we retract our guess and [backtrack](#) to the slot before this one and try something else.

	left	middle	right
House (blue, green, red)	blue	green	red
Flowers (daisies, lilies, roses)			
Person (Bob, Pat, Sue)			

But we have no open alternatives for the color of the rightmost house, either. So we retract that guess and backtrack again:

	left	middle	right
House (blue, green, red)	blue	green	

	left	middle	right
Flowers (daisies, lilies, roses)			
Person (Bob, Pat, Sue)			

Here we can try a different possibility: maybe the middle house is red.

	left	middle	right
House (blue, green, red)	blue	red	
Flowers (daisies, lilies, roses)			
Person (Bob, Pat, Sue)			

and the rightmost house would have to be green.

	left	middle	right
House (blue, green, red)	blue	red	green
Flowers (daisies, lilies, roses)			
Person (Bob, Pat, Sue)			

You can probably see that this won't work out either. Eventually we will have to back all the way up to our first decision, and try a different color for the leftmost house.

Hopefully, you can see that, given enough time, this systematic approach would eventually arrive at a solution if one is possible, or would conclusively prove that no solution is possible.

Now, this isn't the fastest or smartest way to go about solving this problem. Humans can do much better, relying on a little foresight and intuition. But computers are notoriously lacking in both foresight and intuition.

We use backtracking largely in cases where we can't find a more sophisticated way to arrive at a solution.

## 4.2 A Tabular Approach to Backtracking

The essence of backtracking is:

1. Number the solution variables  $[v_0, v_1, \dots, v_{n-1}]$ .
2. Number the possible values for each variable  $[c_0, c_1, \dots, c_{k-1}]$ .
3. Start by assigning  $c_0$  to each  $v_i$ .
4. If we have an acceptable solution, stop.
5. If the current solution is not acceptable, let  $i = n-1$ .
6. If  $i < 0$ , stop and signal that no solution is possible.
7. Let  $j$  be the index of the value assigned to variable  $v_i$  (i.e.,  $v_i = c_j$ ). If  $j < k - 1$ , assign  $c_{j+1}$  to  $v_i$  and go back to step 4.
8. But if  $j \geq k - 1$ , assign  $c_0$  to  $v_i$ , decrement  $i$ , and go back to step 6.

There are lots of variations possible. It's fairly easy to modify this scheme to deal with situations where different variables have different sets of possible values.

Many game-playing programs use a form of this kind of backtracking to select the computer's move.

### 4.2.1 Example: The 3-House Puzzle

There are 9 variables in this problem:  $\text{color}_L$ ,  $\text{color}_M$ ,  $\text{color}_R$ ,  $\text{flowers}_L$ ,  $\text{flowers}_M$ ,  $\text{flowers}_R$ ,  $\text{person}_L$ ,  $\text{person}_M$ , and  $\text{person}_R$ , where the "L", "M", and "R" subscripts denote "Left", "Middle", and "Right", respectively.

Each can take on 3 possible values. Letting blue=daisies=Bob=0, green=lilies=Pat=1, and red=roses=Sue=2, we can represent our progress through the problem like this:

Start with none of the variables assigned.

$\text{color}_L$	$\text{color}_M$	$\text{color}_R$	$\text{flowers}_L$	$\text{flowers}_M$	$\text{flowers}_R$	$\text{person}_L$	$\text{person}_M$	$\text{person}_R$
0								

Then assign red to the left house:

$\text{color}_L$	$\text{color}_M$	$\text{color}_R$	$\text{flowers}_L$	$\text{flowers}_M$	$\text{flowers}_R$	$\text{person}_L$	$\text{person}_M$	$\text{person}_R$
0								

and green to the middle house:

$\text{color}_L$	$\text{color}_M$	$\text{color}_R$	$\text{flowers}_L$	$\text{flowers}_M$	$\text{flowers}_R$	$\text{person}_L$	$\text{person}_M$	$\text{person}_R$
0	1							

Continuing on, we can show the whole progress (up to the point where we stopped earlier), showing each step as a separate line:

color <sub>L</sub>	color <sub>M</sub>	color <sub>R</sub>	flowers <sub>L</sub>	flowers <sub>M</sub>	flowers <sub>R</sub>	person <sub>L</sub>	person <sub>M</sub>	person <sub>R</sub>
0								
0	1							
0	1	2						
0	1	2	0					
0	1	2	1					
0	1	2	2					
0	2							
0	2	1						

and so on.

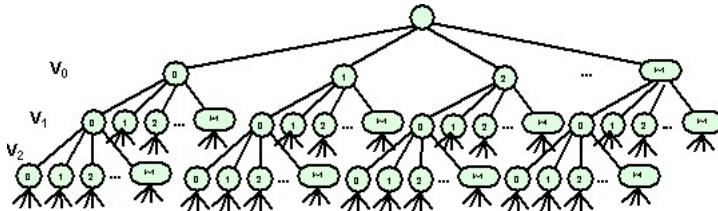
Actually, even here we may have played fast and loose a little with the rules. How did we know that the middle house should start with color 1 instead of 0? Only because there is an implicit rule saying that there can only be a single house of each color. A violation of that rule is just another contradiction, so we could actually argue that the steps we went through were:

color <sub>L</sub>	color <sub>M</sub>	color <sub>R</sub>	flowers <sub>L</sub>	flowers <sub>M</sub>	flowers <sub>R</sub>	person <sub>L</sub>	person <sub>M</sub>	person <sub>R</sub>
0								
0	0							
0	1							
0	1	0						
0	1	1						
0	1	2						
0	1	2	0					
0	1	2	1					
0	1	2	2					
0	2							
0	2	0						
0	2	1						

and so on. This is more typical of how we would be likely to program a solution.

## 4.3 A Recursive Approach to Backtracking

Here's another way to think about this process.



This tree shows the “solution space” for a backtracking problem. At one level is the first problem variable with all of its possible solution values. Each node at the first level represents a possible value for the variable  $v_0$ .

Now, for each possible value of  $v_0$ , we have a number of possible values for the next problem variable,  $v_1$ . These are shown at the next level of the tree. And for each pair of possible values for  $v_0$  and  $v_1$ , we have a number of possibilities for  $v_2$ , and so on. Each path from the top of this tree to any of the bottom nodes represents a distinct potential solution to the problem.

This tree isn't infinitely large - we only have a finite number of variables in the problem. But the total number of possibilities can be mind-boggling.

The traditional way to program backtracking has been to write a recursive function to search this solution space. The first call deals with possible values for  $v_0$ , and loops through each one in turn. For each possible value of  $v_0$ , we recursively call the same routine, but instruct it to work with  $v_1$  instead of  $v_0$ . That recursive call will loop through all possible values of  $v_1$ , trying each one and then recursively calling itself with instructions to work on  $v_2$ , and so on.

### 4.3.1 Example: Scheduling

The CS Dept. has a list of courses to offer next semester, and a list of available time slots in which to place them.

Some courses “conflict” – they cannot be scheduled at the same time because

- students may want to take both courses in the same semester
- the same faculty member is teaching both.

Suppose that the classes are numbered  $0 \dots \text{NumClasses}-1$ , and that the time slots are numbered  $0 \dots \text{NumTimes}-1$ .

We will assume that someone has already written a function

```
bool conflicts(int class1, int class2);
```

to determine if two classes conflict.

Our job is to write the function that actually does the scheduling. We will store the schedule in

```
vector<int> timeOfClass (numClasses, -1);
```

so that `timeOfClass[c]` is the time slot in which class `c` has been scheduled (-1 indicates that we have not yet assigned a time to `c`).

We start by defining a utility function, `noConflicts`, to determine if it's OK to assign a given class to a given time slot.

```
bool noConflicts (int class, int time)
// Would assigning the given class to the given time
// cause a conflict with any already scheduled classes?
// Return true if no conflicts would be caused.
{
    for (int c = 0; c < class; ++c)
    {
        if (timeOfClass[c] = time // if c is already schedule at this time
            && conflicts(c, class)) // and c conflicts with class
            return false;
    }
    return true;
}
```

`noConflicts` simply examines every other class already in this time slot to see if it conflicts with this one.

Here, then, is the main scheduling algorithm. `schedule(i)` returns true if it is able to schedule all the classes from `i` ... `numClasses-1`.

```
bool schedule(int class)
{
    if (class == NumClasses)
        return true;
    else
        for (int t = 1; t <= NumTimes; ++t)
        {
            if (noConflicts(class, t))
            {
                timeOfClass[class] = t;
                if (schedule(i+1)) return true;
            }
        }
    timeOfClass[class] = -1;
    return false; // all choices failed
}
```

It does this by trying every plausible time slot for class `i` until it finds one that allows all the remaining classes to be (recursively) scheduled.

If it can't find a slot in which to place class `i`, it returns false. Whenever a recursive call from `schedule` returns `false` back to an earlier call of itself, the earlier call is forced to try a different time slot for a previously scheduled class.

#### 4.3.2 Worst-Case Analysis

**Analysis:**

`noConflicts()` is  $O(\text{class})$ .

```
bool noConflicts (int class, int time)
// Would assigning the given class to the given time
// cause a conflict with any already scheduled classes?
// Return true if no conflicts would be caused.
{
    for (int c = 0; c < class; ++c)
    {
        if (timeOfClass[c] = time
            && conflicts(c, class))
            return false;
    }
    return true;
}
```

```

bool schedule(int class)
{
    if (class == NumClasses)
        return true;
    else
        for (int t = 1; t <= NumTimes; ++t)
            if (noConflicts(class, t))
            {
                timeOfClass[class] = t;
                if (schedule(i+1)) return true;
            }
    timeOfClass[class] = -1;
    return false; // all choices failed
}

```

The worst case of `schedule()` occurs when no schedule is possible. Then each class will be tried in each time slot. So for the first class, there will be  $\text{NumTimes}$  possibilities. For each of those possibilities, the second class will have a further  $\text{NumTimes}$  possibilities of its own, for a total of  $\text{NumTimes}^2$  possibilities for the two classes. Continuing on, the total number of possibilities for all classes is  $\text{NumTimes}^{\text{NumClasses}}$ .

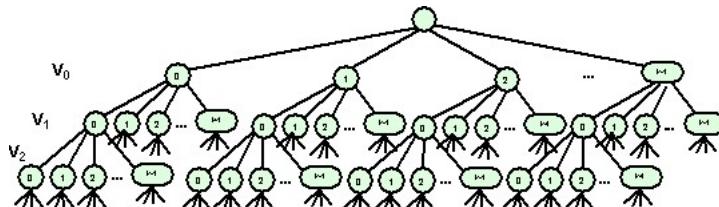
Each try involves a call to `noConflicts`, so the overall algorithm is  $O(\text{NumClasses} * \text{NumTimes}^{\text{NumClasses}})$ . We say that this routine is [exponential](#) in `NumClasses`. Such exponential growth is worse, for sufficiently large inputs, than [any](#) polynomial big-O.

## 4.4 An Iterative Framework for Backtracking

Can we do this without the recursion?

Yes.

Let's return again to the tree view of the possible solutions to a backtracking problem.



Let's assume our tree has 5 levels, corresponding to  $v_0$  through  $v_4$ , and that each variable can take on any of 3 values. Now, suppose we list the possible solutions from left to right in this tree, listing each one from  $v_{n-1}$  down to  $v_0$ . The leftmost path in the tree is 00000, then 00001, then 00002, 00010, 00011, 00012, 00020, ... . You can see that all we are doing is counting in "base 3". If our variables could have taken on 10 different values instead of just three, we'd be counting in our "traditional" base-10 number system. [^ And if the variables could take on different numbers of values, the numbering system might look strange, but we could still recognize it as a kind of counting.]

So, if we had an ADT for counting N digit numbers in an arbitrary base, we could use that as a kind of generator for backtracking solutions.

### 4.4.1 The BackTrack class

The code shown here represents a backtracking state generator.

```

class BackTrack {
public:
    BackTrack (unsigned nVariables, unsigned arity=2);
    // Create a backtracking state for a problem with
    // nVariables variables, each of which has the same
    // number of possible values (arity).

    template <class Iterator>
    BackTrack (Iterator arityBegin,
               Iterator arityEnd);
    // Create a backtracking state in which each variable may have
    // a different number of possible values. The values are obtained
    // as integers stored in positions arityBegin .. arityEnd as per
    // the usual conventions for C++ iterators. The number of
    // variables in the system are inferred from the number of
    // positions in the given range.

    unsigned operator[] (unsigned variableNumber) const;
    // Returns the current value associated with the indicated
    // variable.

    unsigned numberOfVariables() const;
    // Returns the number of variables in the backtracking system.

    unsigned arity (unsigned variableNumber) const;
    // Returns the number of potential values that can be assigned
    // to the indicated variable.
}

```

```

bool more() const;
// Indicates whether additional candidate solutions exist that
// can be reached by subsequent ++ or prune operations.

void prune (unsigned level);
// Indicates that the combination of values associated with
// variables 0 .. level-1 (inclusive) has been judged unacceptable
// (regardless of the values that could be given to variables
// level..numberOfVariables()-1. The backtracking state will advance
// to the next solution in which at least one of the values in the
// variables 0..level-1 will have changed.

BackTrack& operator++();
// Indicates that the combination of values associated with
// variables 0 .. nVariables-1 (inclusive) has been judged unacceptable.
// The backtracking state will advance
// to the next solution in which at least one of the values in the
// variables 0..level-1 will have changed.

BackTrack operator++(int);
// Same as other operator++, but returns a copy of the old backtrack state

private:
    bool done;
    vector<unsigned> arities;
    vector<unsigned> values;
};


```

The constructors allow us to indicate how many variables we need and what the “arity” (number of possible values) for them are. For example

```
BackTrack problem(9, 3);
```

would create backtracking state problem with 9 variables, each of which can take on the values 0, 1, or 2.

The current value of any variable can be read using the square brackets operator:

```

cout << "Variable " << i
    << " has value " << problem[i]
    << endl;
```

We can advance to the next possible state using the `++` operator, and the `more()` function tells us if we have tried every possible combination. For example, if we were to simply loop through all possible states:

```

BackTrack problem(4, 3);
// 4 questions, each with 3 possible answers
while (problem.more())
{
    for (int i = 0; i < 4; ++i)
        cout << problem[i] << ',';
    cout << endl;
    ++problem;
}
```

the output would be: 0000, 0001, 0002, 0010, 0011, 0012, 0020, 0021, 0022, 0100, ..., 2222.

A typical backtracking problem can then be solved this way:

```

BackTrack problem(nVariables, nValues);
bool solved = false;
while (!solved) && problem.more()
{
    solved = checkSolution(problem);

    if (!solved)
        ++problem;
}
```

where `checkSolution` is a function that returns true if the current problem state is an acceptable solution.

#### 4.4.2 Example: Iterative Solution of 3-Houses

Let's look at how to use this class to solve the 3-houses puzzle. In this puzzle, we have a total of 9 variables. We give them names here, assigning those names to the integers from 0 to 8.

```

enum Questions {colorOfHouse1=0,
    colorOfHouse2=1,
    colorOfHouse3=2,
    flowersAtHouse1=3,
    flowersAtHouse2=4,
    flowersAtHouse3=5,
    occupantOfHouse1=6,
    occupantOfHouse2=7,
    occupantOfHouse3=8};
```

Although each group of three variables represents something different, it so happens that all the variables can take on three possible values.

```
enum Colors {red=0, blue=1, green=2};
const char* colorNames[] = {"red", "blue", "green"};

enum Occupants {bob=0, pat=1, sue=2};
const char* occupantNames[] = {"Bob", "Pat", "Sue"};

enum Flowers {daisies=0, lilies=1, roses=2};
const char* flowerNames[] = {"daisies", "lilies", "roses"};
```

Again, we'll give these descriptive names, but bind them to the integers 0, 1, and 2.

The related arrays of character strings are useful for output purposes:

```
void describeHouse(int houseNumber,
                   Occupants occupant,
                   Colors c, Flowers f)
{
    cout << occupantNames[occupant]
        << " lives in house " << houseNumber
        << ", which is painted " << colorNames[c]
        << " and has "
        << flowerNames[f] << endl;
}
```

Now we're ready to set up the main routine for the solution. This just adds a bit of output to the basic solution loop.

```
int main()
{
    BackTrack problem(9, 3);
    // 9 questions, each with 3 possible answers
    bool solved = false;
    while ((!solved) && problem.more())
    {
        for (int i = 0; i < 9; ++i)
            cout << problem[i] << ' ';
        cout << endl;

        solved = checkSolution(problem);

        if (!solved)
            ++problem;
    }

    if (solved)
    {
        describeHouse(1,
                      (Occupants)problem[occupantOfHouse1],
                      (Colors)problem[colorOfHouse1],
                      (Flowers)problem[flowersAtHouse1]);
        describeHouse(2,
                      (Occupants)problem[occupantOfHouse2],
                      (Colors)problem[colorOfHouse2],
                      (Flowers)problem[flowersAtHouse2]);
        describeHouse(3,
                      (Occupants)problem[occupantOfHouse3],
                      (Colors)problem[colorOfHouse3],
                      (Flowers)problem[flowersAtHouse3]);
    }
    else
        cout << "Problem has no solution" << endl;
}
```

Of course, all the “good stuff” is in the `checkSolution` function - that's where we have to determine whether the current set of numbers represent a possible solution to the puzzle or not.

We'll take this in stages.

```
bool checkSolution(const BackTrack& bt)
// Check the state of bt to see if it
//   represents a valid solution.
{
    bool OK = true;

    // Implicit rule: each house is different
    if (bt[occupantOfHouse1] == bt[occupantOfHouse2])
        OK = false;
    if (bt[occupantOfHouse1] == bt[occupantOfHouse3])
        OK = false;
    if (bt[occupantOfHouse2] == bt[occupantOfHouse3])
        OK = false;
    :
}
```

`ok` will represent the result of our tests overall. It starts out `true` but will be set to `false` the moment we find a contradiction between the current problem state and the rules of the puzzle.

The first rules to be checked are implicit ones. Bob can't live in more than one house in this problem, and neither can Sue or Pat. So if any two houses have the same occupant, set OK to false. Note that `bt[occupantOfHouse1]` denotes the current assignment, in `bt`, to the `occupantOfHouse1` variable. It will either be bob (0), pat (1), or sue (2).

We have similar rules for the house colors and the flowers.

```

if (bt[colorOfHouse1] == bt[colorOfHouse2])
OK = false;
if (bt[colorOfHouse1] == bt[colorOfHouse3])
OK = false;
if (bt[colorOfHouse2] == bt[colorOfHouse3])
OK = false;

if (bt[flowersAtHouse1] == bt[flowersAtHouse2])
OK = false;
if (bt[flowersAtHouse1] == bt[flowersAtHouse3])
OK = false;
if (bt[flowersAtHouse2] == bt[flowersAtHouse3])
OK = false;
...

```

```
// 1. Bob does not live in the green house.  
int bobsHouse = indexOf(bt, bob,  
                      occupantOfHouse1,  
                      occupantOfHouse2,  
                      occupantOfHouse3);  
  
int greenHouse = indexOf(bt, green,  
                        colorOfHouse1,  
                        colorOfHouse2,  
                        colorOfHouse3);  
  
if (bobsHouse - occupantOfHouse1 == greenHouse - colorOfHouse1)  
    OK = false;  
:  
:
```

Now we're ready to consider the explicit rules of the puzzle. For rule 1, we need to know which house bob lives in and which house is green.

We'll need to do a lot of this kind of lookup in this function, so we'll introduce a utility function, `indexof`, that tells us which of three variable numbers contains a desired value.

```
int indexOf (const BackTrack& bt, int value,
             int candidate1, int candidate2,
             int candidate3)
{
    if (bt[candidate1] == value)
        return candidate1;
    else if (bt[candidate2] == value)
        return candidate2;
    else
        return candidate3;
}
```

Note that the `indexOf` calls give us variable numbers (in this case, in the range 0 ... 8). To get the house number, we need to subtract the variable number of the first of each group of three variables. Thus `bobsHouse` actually contains the index of the variable describing Bob's location. To get the actual house number, we subtract `occupantOfHouse1`.

```
// 2. Pat lives between Bob and Sue.  
if (bt[occupantOfHouse2] != pat)  
    OK = false;  
:  
:
```

The next rule is easy. If Pat lives between them, Pat must be in the second house.

The code for the third rule looks much like the code for the first rule.

```
:  
    // 3. Sue planted daisies.  
    int suesHouse = indexOf(bt, sue,  
                           occupantOfHouse1,  
                           occupantOfHouse2,  
                           occupantOfHouse3);  
    int daisiesHouse = indexOf(bt, daisies,  
                               flowersAtHouse1,  
                               flowersAtHouse2,  
                               flowersAtHouse3);  
  
    if (suesHouse - occupantOfHouse1 != daisiesHouse - flowersAtHouse1)  
        OK = false;  
    :  
:
```

```
:  
// 4. Lilies are planted at the blue house.  
int liliesHouse = indexOf(bt, lilies,  
                           flowersAtHouse1,  
                           flowersAtHouse2,  
                           flowersAtHouse3);  
int blueHouse = indexOf(bt, blue,  
                        colorOfHouse1,  
                        colorOfHouse2,  
                        colorOfHouse3);  
  
if (liliesHouse - flowersAtHouse1 != blueHouse - colorOfHouse1)  
    OK = false;  
:  
:
```

And so does the 4th rule.

```
:  
// 5. The leftmost house has roses.  
if (bt[flowersAtHouse1] != roses)  
    OK = false;  
  
return OK;  
}
```

And the fifth and final rule is easy.

If we throw all this together and run it we would see the following output:

```
0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 1  
0 0 0 0 0 0 0 0 2  
0 0 0 0 0 0 0 1 0  
0 0 0 0 0 0 0 1 1  
0 0 0 0 0 0 0 1 2  
0 0 0 0 0 0 0 2 0  
0 0 0 0 0 0 0 2 1  
0 0 0 0 0 0 0 2 2  
0 0 0 0 0 0 1 0 0  
:  
:
```

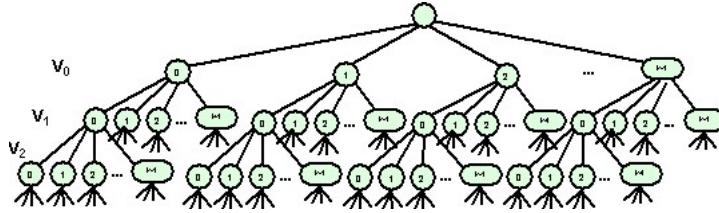
... and, about 4000 lines later ...

```
:  
0 1 2 2 1 0 0 0 2  
0 1 2 2 1 0 0 1 0  
0 1 2 2 1 0 0 1 1  
0 1 2 2 1 0 0 1 2  
Bob lives in house 1, which is painted red and has roses  
Pat lives in house 2, which is painted blue and has lilies  
Sue lives in house 3, which is painted green and has daisies
```

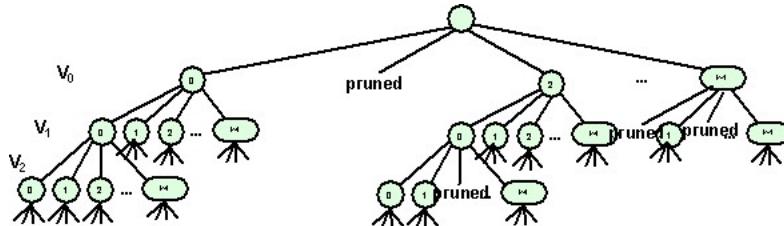
## 4.5 Pruning

Backtracking, despite its exponential-time behavior, continues to arise in problems for which no better solution method is known. A lot of effort goes into finding ways to speed up its average performance.

One of the most common improvements is to use some kind of “early” test to determine if a particular assignment to a particular variable is plausible rather than always waiting until all variables have been assigned to do any checking. For example, in the 3-houses puzzle, we can immediately reject any proposed solution that has Pat in houses 1 or 3, *and there’s no point cycling through all the combinations of the other variables once we see this*.



Again, if we think of the set of possible solutions in terms of a tree, as shown here, then the use of these early “plausibility” tests may have the effect of pruning entire branches of the tree, meaning that we don’t actually need to explore all those alternatives.



Note that the effect of “pruning” is most significant when it can occur early in the tree. A lot of research and design effort has gone into ideas like trying to reorder the variables and/or values so that pruning will occur as early as possible.

#### 4.5.1 Pruning and the BackTrack class

Our backtracking state generator can easily accommodate pruning.

```
class BackTrack {
public:
:
void prune (unsigned level);
// Indicates that the combination of values associated with
// variables 0 .. level-1 (inclusive) has been judged unacceptable
// (regardless of the values that could be given to variables
// level..numberOfVariables()-1. The backtracking state will
// advance to the next solution in which at least one of the
// values in the variables 0..level-1 will have changed.
:
```

In fact, the “normal” `++` operator is actually implemented as `prune(numberOfVariables())` (which is no pruning at all.)

Let’s look, briefly, at just how the backtracking generator works.

```
class BackTrack {
public:
:
private:
    bool done;
    vector<unsigned> arities;
    vector<unsigned> values;
};
```

We use a vector to hold the current values for all of our variables. We also have a vector, `arities`, that indicates the number of different values each variable can take on.

In the problems we have been looking at, all the variables have the same arity, but this isn’t true for all backtracking problems.

When we create a new backtracking state, we start with all the variable values set to zero.

```
BackTrack::BackTrack (unsigned nVariables, unsigned arity=2)
// Create a backtracking state for a problem with
// nVariables variables, each of which has the same
// number of possible values (arity).
: arities(nVariables, arity), values(nVariables, 0), done(false)
{}
```

Pruning is accomplished by treating variables  $0 \dots \text{level-1}$  as a  $\text{level}$ -digit number, adding 1 to it using the same “add and carry to the left” procedure you learned in grade-school arithmetic.

```

void BackTrack::prune (unsigned level)
    // Indicates that the combination of values associated with
    // variables 0 .. level-1 (inclusive) has been judged unacceptable
    // (regardless of the values that could be given to variables
    // level..numberOfVariables()-1. The backtracking state will advance
    // to the next solution in which at least one of the values in the
    // variables 0..level-1 will have changed.
{
    level = (level > numberOfVariables()) ? numberOfVariables() : level;
    fill (values.begin()+level, values.end(), 0);

    // Treat the top level-1 values as a level-1 digit number. Add one
    // to the rightmost "digit". If this digit goes too high, reset it to
    // zero and "carry one to the left".
    int k = level-1;
    bool carry = true;
    while (k >= 0 && carry)
    {
        values[k] += 1;
        if (values[k] >= arities[k])
            values[k] = 0;
        else
            carry = false;
        --k;
    }
    done = carry;
}

```

The variables being “pruned”, the ones in positions level ... numberOfVariables(), are simply set to zero.

The normal ++ is simply a special case in which no variables are actually pruned, but the state is advanced to the next “number”.

```

BackTrack& BackTrack::operator++()
    // Indicates that the combination of values associated with
    // variables 0 .. nVariables-1 (inclusive) has been judged unacceptable.
    // The backtracking state will advance
    // to the next solution in which at least one of the values in the
    // variables 0..level-1 will have changed.
{
    prune(numberOfVariables());
    return *this;
}

```

### Applying Pruning

Now, to actually use pruning in a problem requires a bit more effort when we check the solution state. Instead of simply indicating whether the state is acceptable, we now want to indicate the smallest variable number that we want to change. Basically, for any rule that involves variables  $v_j$  and  $v_k$ , if we find that the rule is violated, then the larger of  $j$  or  $k$  will need to be changed next.

## 4.6 Example: Pruning in 3-Houses

Instead of a simple boolean OK variable, we now use an integer to track the variable that we want changed on the next step.

```

int checkSolution(const BackTrack& bt)
    // Check the state of bt to see if it represents a valid solution.
    // Return -1 if not a solution. If not a solution, return the smallest
    // number of any problem variable known to be incorrect.
{
    int knownIncorrect = bt.numberOfVariables();

    // Implicit rule: each house is different
    if (bt[occupantOfHouse1] == bt[occupantOfHouse2])
        knownIncorrect = min(knownIncorrect,
                            max(occupantOfHouse1, occupantOfHouse2));
    :

```

If we fail the test, we replace that integer with the larger of the variable numbers involved in the test, provided that we haven’t already found a lower-numbered variable for pruning.

The same pattern gets repeated many times for the implicit rules.

```

:
if (bt[occupantOfHouse1] == bt[occupantOfHouse3])
    knownIncorrect = min(knownIncorrect,
                        max(occupantOfHouse1,
                            occupantOfHouse3));
if (bt[occupantOfHouse2] == bt[occupantOfHouse3])
    knownIncorrect = min(knownIncorrect,
                        max(occupantOfHouse2,
                            occupantOfHouse3));

```

and for the explicit rules as well.

```
:  
// 1. Bob does not live in the green house.  
int bobsHouse = indexOf(bt, bob,  
                        occupantOfHouse1,  
                        occupantOfHouse2,  
                        occupantOfHouse3);  
int greenHouse = indexOf(bt, green,  
                        colorOfHouse1,  
                        colorOfHouse2,  
                        colorOfHouse3);  
  
if (bobsHouse - occupantOfHouse1 == greenHouse - colorOfHouse1)  
    knownIncorrect = min(knownIncorrect,  
                         max(bobsHouse, greenHouse));  
  
// 2. Pat lives between Bob and Sue.  
if (bt[occupantOfHouse2] != pat)  
    knownIncorrect = min(knownIncorrect, occupantOfHouse2);  
:  
:
```

until we have checked all the rules and are ready to return.

```
:  
if (knownIncorrect >= bt.numberOfVariables())  
    return -1;  
else  
    return knownIncorrect;  
}
```

```
BackTrack problem(9, 3);  
// 9 questions, each with 3 possible answers  
bool solved = false;  
while ((!solved) && problem.more())  
{  
    for (int i = 0; i < 9; ++i)  
        cout << problem[i] << ' ';  
    cout << endl;  
  
    int pruneAt = checkSolution(problem);  
    if (pruneAt < 0)  
        solved = true;  
    else  
        problem.prune(pruneAt+1);  
}
```

The main program loop now is altered to use `prune` instead of `++`.

#### 4.6.1 Output After Pruning

If we run this version, the output is

```
0 0 0 0 0 0 0 0 0  
0 1 0 0 0 0 0 0 0  
0 1 1 0 0 0 0 0 0  
0 1 2 0 0 0 0 0 0  
0 1 2 1 0 0 0 0 0  
0 1 2 2 0 0 0 0 0  
0 1 2 2 0 1 0 0 0  
0 1 2 2 0 2 0 0 0  
0 1 2 2 1 0 0 0 0  
0 1 2 2 1 0 0 1 0  
0 1 2 2 1 0 0 1 1  
0 1 2 2 1 0 0 1 2  
Bob lives in house 1, which is painted red and has roses  
Pat lives in house 2, which is painted blue and has lilies  
Sue lives in house 3, which is painted green and has daisies
```

That's the whole thing! Only 12 potential solutions were examined, instead of more than 4000.

This shows that pruning can be critical to making backtracking work efficiently. But don't get carried away. The amount of reduction you get from pruning varies considerably (we were probably a bit lucky here) and, even with pruning, the worst case is still exponential time.

### 4.7 Simpler Pruning

The min-max combinations in the previous example can be confusing. In many problems, an easier way to do backtracking with pruning is to work variable by variable, comparing each variable for conflicts with the earlier variables only:

```
bool solved = false;
while ((!solved) && problem.more())
{
    for (int i = 0; i < 9; ++i)
        cout << problem[i] << ' ';
    cout << endl;

    solved = true;
    for (int variable = 0; variable < 9; ++variable)
    {
        if (isInconsistent(problem, variable))
        {
            problem.prune(variable+1);
            solved = false;
            break;
        }
    }
}
```

This requires us to replace the `checkSolution` function with a function that examines a single variable at a time, comparing it only to lower-numbered variables in the problem.

That's actually not particularly easy to do in the 3 houses problem, though in other problems it's simpler than the full `checkSolution` function.

```
inconsistent.cpp +
```

```
// Return true if that value in variable varNum cannot be
// reconciled with the values of the earlier variables.
bool isInconsistent (const BackTrack& bt, int varNum)
{
    switch (varNum)
    {
        case colorOfHouse1:
            break;
        case colorOfHouse2:
            if (bt[colorOfHouse2] == bt[colorOfHouse1])
                return true;
            break;
        case colorOfHouse3:
            if (bt[colorOfHouse3] == bt[colorOfHouse1])
                return true;
            if (bt[colorOfHouse3] == bt[colorOfHouse2])
                return true;
            break;
        case flowersAtHouse1:
            if ((bt[flowersAtHouse1] == lilies)      // Lilies at the blue house
                != (bt[colorOfHouse1] == blue))
                return true;
            if (bt[flowersAtHouse1] != roses) // Leftmost house has roses
                return true;
            break;
        case flowersAtHouse2:
            if (bt[flowersAtHouse2] == bt[flowersAtHouse1])
                return true;
            if ((bt[flowersAtHouse2] == lilies)      // Lilies at the blue house
                != (bt[colorOfHouse2] == blue))
                return true;
            break;
        case flowersAtHouse3:
            if (bt[flowersAtHouse3] == bt[flowersAtHouse1])
                return true;
            if (bt[flowersAtHouse3] == bt[flowersAtHouse2])
                return true;
            if ((bt[flowersAtHouse3] == lilies)      // Lilies at the blue house
                != (bt[colorOfHouse3] == blue))
                return true;
            break;
        case occupantOfHouse1:
            if (bt[occupantOfHouse1] == bob      // bob cannot live in green
                && bt[colorOfHouse1] == green)
                return true;
            if ((bt[occupantOfHouse1] == sue)     // Sue planted daisies
                != (bt[flowersAtHouse1] == daisies))
                return true;
            break;
        case occupantOfHouse2:
            if (bt[occupantOfHouse2] == bt[occupantOfHouse1])
                return true;
            if (bt[occupantOfHouse2] == bob      // bob cannot live in green
                && bt[colorOfHouse2] == green)
                return true;
            if (bt[occupantOfHouse2] != pat) // pat must live between bob and sue
                return true;
            if ((bt[occupantOfHouse2] == sue)     // Sue planted daisies
                != (bt[flowersAtHouse2] == daisies))
                return true;
            break;
    }
}
```

```

        case occupantOfHouse3:
            if (bt[occupantOfHouse3] == bt[occupantOfHouse1])
                return true;
            if (bt[occupantOfHouse3] == bt[occupantOfHouse2])
                return true;
            if (bt[occupantOfHouse2] == bob // bob cannot live in green
                && bt[colorOfHouse2] == green)
                return true;
            if ((bt[occupantOfHouse3] == sue) // Sue planted daisies
                != (bt[flowersAtHouse3] == daisies))
                return true;
        };
    }
}

```

Notice that each case works either with the current variable in isolation with the others, e.g.,

```

case occupantOfHouse2:
:
if (bt[occupantOfHouse2] != pat) // pat must live between bob and sue
    return true;

```

or compares with earlier variables *only*:

```

case colorOfHouse1: ①
break;
case colorOfHouse2:
if (bt[colorOfHouse2] == bt[colorOfHouse1]) ②
    return true;
break;

```

- ① does not check to see if house1 has the same color as any other houses, because in our ordering of variables, `colorOfHouse1` is the very first variable, with index 0.
- ② checks to make sure house2 and house1 do not have the same color. It does not, however, check house2 against house3, because `colorOfHouse3` is a later variable.

This is probably messier than the earlier solution, but that's because the variables that make up the 3 houses problem are so different from one another, and each one needs "custom" handling. In problems where the nature of the variables is more uniform, we might be able to replace the multi-way switch with a much simpler structure.

## 5 Convergent

In contrast to "generate-and-test", some algorithms work by "generate-and-improve". An initial guess is made at a solution, which is successively improved by some process that, it is hoped, eventually "converges" to the correct solution.

Convergent algorithms are most common in numerical processing. For example, the code shown here is a convergent algorithm for computing square roots, given an initial guess. [^ Calculus die-hards may recognize this as Newton's Method.]

```

double sqrt (double x, double initialGuess)
{
    double root = initialGuess;
    do {
        double last = root;
        root = 0.5 * (root + x / root);
    } while (abs(root - last) > 0.0001);
    return root;
}

```

If we were to call, for example, `sqrt(2.0, 1.0)` (find the square root of 2 using an initial guess of 1), the successive values of `root` would be: 1, 1.5, 1.416667, 1.414216, 1.414214, at which point the algorithm would return.

Many scientific and engineering programs depend upon convergent algorithms.

## 6 Dynamic programming

Dynamic programming is a variant on recursion that is useful when a recursive solution would repeatedly solve the same small subproblems.

The basic idea is to go ahead and solve the smaller problems, but to save the results in a data structure so that, if you need the same subproblem solved again, you can just fetch the prior answer from the data structure.

### 6.1 Example: combinations

Suppose we have a set of  $n$  distinct items, and need to select  $k$  of them at random (e.g., dealing 5 cards from a deck of 52). Assume that we aren't concerned about the order in which we obtained the selected items, but we want to know how many different possible sets of items we could possibly obtain.

The number of different combinations we can obtain by selecting  $k$  items from a set of  $n$ , without replacement, is written as  $\binom{n}{k}$  or sometimes as  $C(n, k)$ .

It can be evaluated as

$$C(n, n) = C(n, 0) = 1$$

$$C(n, k) = C(n - 1, k - 1) + C(n - 1, k), \quad 0 < k < n$$

A straightforward recursive solution would be the code shown here.

```
long C(int n, int k)
{
    if (k==0 || k==n)
        return 1;
    else
        return C(n-1, k-1) + C(n-1, k);
}
```

But this wastes effort re-evaluating smaller problem over and over again. For example,

$$C(5, 2) = C(4, 1) + C(4, 2)$$

No problem so far, but let's expand each of the items on the right

$$C(4, 1) = C(3, 0) + C(3, 1)$$

$$C(4, 2) = C(3, 1) + C(3, 2)$$

and you can start to see that one subproblem  $C(3, 1)$  has already arisen in two different places.

Furthermore, if we were to expand these again, we would find that both occurrences of  $C(3, 1)$  and  $C(3, 2)$  will need the value of  $C(2, 1)$ . In fact, most of the run time of this algorithm will be devoted to solving and re-solving the same smaller problems, over and over.

How can we avoid this repetition? Well, let's introduce a “2-dimensional” vector, `savedResults`, to store answers that we have already computed. `savedResults(n, k)` will store the result of a previous evaluation of  $C(n, k)$ , or -1 if we haven't solved that problem yet.

```
long C(int n, int k)
{
    Matrix<long> savedResults(n+1, k+1)
    for (int i = 0; i <= n; ++i)
        for (int j = 0; j <= k; ++j)
            savedResults(i, j) = -1;

    return dynamic (n, k, savedResults);
}
```

Now our new version of this function first checks `savedResults` to see if we already know the answer to a problem. If not, it gets the answer the “old-fashioned” way – by recursive calls.

```
long dynamicC (int n, int k, Matrix<long>& savedResults)
{
    if (k==0 || k==n)
        return 1;
    else if (savedResults(n, k) >= 0)
        return savedResults(n, k);
    else
    {
        long theAnswer = dynamicC(n-1, k-1) + dynamicC(n-1, k);
        savedResults(n, k) = theAnswer;
        return theAnswer;
    }
}
```

Also, whenever we have computed a new solution to a problem, we save that solution in `savedResults`.

This approach will significantly speed up the calculation of  $C(n, k)$  for large values of  $n$  and  $k$ .

The idea of storing prior results of a function in a data structure and checking to see if we have already computed a particular function value is called [memoization](#) (i.e., taking a memo).

## 6.1.1 Reversing the Calculation

But wait — there's more!

Those recursive calls, when they take place, always ask for the solution to a “smaller” problem (either  $n$ , or  $k$ , or both are smaller). That's typical of recursion – the recursive calls always need to be approaching some base case.

But in this problem, there are only so many possible smaller problems. (Namely, as many as we have slots for in `savedResults`). Because we never need the answer to a larger problem before we can answer a smaller one, we can turn the whole evaluation inside-out and simply start by evaluating the simple problems, then the

larger problems that can be solved from those, and so on, until we reach the original problem that we were really interested in.

```
long C(int n, int k)
{
    Matrix<long> savedResults(n+1, k+1)
    for (int i = 0; i <= n; ++i)
        for (int j = 0; j <= k; ++j)
            savedResults(i,j) = -1;

    // Solve the easy problems first, working up towards
    // the harder ones
    for (int n1 = 1; n1 <= n; ++n1)
    {
        savedResults[n1][0] = 1;
        savedResults[n1][n1] = 1;
        for (int k1 = 1; k1 < n1; ++k1)
            savedResults(n1,k1) = savedResults(n1-1, k1-1) +
                savedResults(n1-1, k1);
    }

    return savedResults(n, k);
}
```

And suddenly, there's no recursion left at all!

This idea of computing and storing the results of all subproblems of the problem we really want to solve, in order from simplest/smallest up to the one we really want, is called [dynamic programming](#).

Dynamic programming frequently not only prevents repeated recursion over common sub-problems, but often introduces a structure over which a purely iterative solution is possible.

## 6.2 Example: Edit Distance

One of the classic problems solvable by dynamic programming is computing the “edit distance” between two strings: the minimum number of single-character insertions, deletions, or replacements required to convert one string into another.

For example, the edit distance between “Hello” and “Jello” is 1. The edit distance between “good” and “goodbye” is 3. The edit distance between any string and itself is 0.

The edit distance can be used for such purposes as suggesting, in a spell checker, a list of plausible replacements for a misspelled word. For each word not found in the dictionary (and therefore presumably misspelled), list all words in the dictionary that are a small edit distance away from the misspelling.

### 6.2.1 Breaking the Problem Down

We can compute the edit distance between two strings by working from the ends. The three operations available to us are

- Add one character to the end of a string
- Remove one character from the end of a string
- Change the character at the end of a string.

Suppose, for example, that we wanted to compute the edit distance between “Zeil” and “trials” (make up your own joke here). Starting with “Zeil”, we consider what would have to be done to get “trials” if the last step taken were “add”, “remove”, or “change”, respectively:

- If we knew how to convert “Zeil” to “trial”, we could *add* “s” to get the desired word.
- If we knew how to convert “Zei” to “trials”, then we would actually have “trials!” and we could *remove* that last character to get the desired word.
- If we knew how to convert “Zei” to “trial”, then we would actually have “trall” and we could *change* the final “l” to “s” to get the desired word.

Now, being intelligent humans that can actually look ahead intuitively at a problem, you and I may think one or two of these alternatives are obvious losers, but bear with me. We’re developing a general algorithm for not-so-intelligent computers.

Notice that what we have done is to reduce the original problem to 3 “smaller” problems: convert “Zeil” to “trial”, or “Zei” to “trials”, or “Zei” to “trial”.

### 6.2.2 Breaking it Down - Stage 2

We continue, recursively, to break down these problems:

- Convert “Zeil” to “trial”, then *add* “s” to get the desired word.

To convert “Zeil” to “trial”,

#### Add

Convert “Zeil” to “tria”, then add “l”

#### Remove

Convert “Zei” to “trial”, giving “trall”, then remove. Change

Convert “Zei” to “tria”, giving “trial”, and no change is actually needed.

- Convert “Zei” to “trials”, giving “trials”, then *remove* the last character.

To convert “Zei” to “trials”,

**Add**

Convert “Zei” to “trial”, then add “s”

**Remove**

Convert “Ze” to “trials”, giving “tralsi”, then remove.

**Change**

Convert “Ze” to “trial”, giving “trali”, and change the final character.

- Convert “Zei” to “trial”, giving “trall”, then *change* the final “l” to “s”.

To convert “Zei” to “trial”,

**Add**

Convert “Zei” to “tria”, then add “l”

**Remove**

Convert “Ze” to “trial”, giving “trali”, then remove.

**Change**

Convert “Ze” to “tria”, giving “trai”, and change the final character.

Now we have nine subproblems to solve, but note that the strings involved are getting shorter. Eventually we will get down to subproblems involving an empty string, such as

Convert “” to “xyz”,

which can be trivially solved by a series of “Adds”.

### 6.2.3 A Recursive Solution

Here you see the recursive implementation of the edit distance calculation.

```
int editDistance (string x, string y)
{
    if (x == "")
        return y.length(); // base case
    else if (y == "")
        return x.length(); // base case
    else
    {
        int addDistance = editDistance(x, y.substr(0,y.length()-1)) + 1;
        int removeDistance = editDistance(x.substr(0,x.length()-1), y) + 1;
        int changeDistance = editDistance(x.substr(0,x.length()-1),
                                         y.substr(0,y.length()-1))
            + (x[x.length()-1] == y[y.length()-1]) ? 0 : 1;
        return min(min(addDistance, removeDistance), changeDistance);
    }
}
```

In the main portion, we don’t know, off hand, whether the cheapest way to convert one string into another involves a final add, remove, or change, so we evaluate all three possibilities and return the minimum distance from among the three.

In each case, we recursively compute the distance (number of adds, removes, and changes) required to “set up” a final add, a final remove, or a final change. We add one to the add distance and the remove distance to account for the final add or remove. For the change distance, we add one only if the final characters in the strings are different (if not, no final change is required).

Interestingly enough, we can also solve the problem this way, by using recursion to first convert the trailing portions of the strings, then figure out what must be done at the start of the string.

```
int editDistance (string x, string y)
{
    if (x == "")
        return y.length(); // base case
    else if (y == "")
        return x.length(); // base case
    else
    {
        int addDistance = editDistance(x, y.substr(1)) + 1;
        int removeDistance = editDistance(x.substr(1), y) + 1;
        int changeDistance = editDistance(x.substr(1), y.substr(1))
            + (x[0] == y[0]) ? 0 : 1;
        return min(min(addDistance, removeDistance), changeDistance);
    }
}
```

### 6.2.4 A Dynamic Programming Solution

We can solve the same problem via dynamic programming by, again, reversing the direction so that we work the smaller subproblems first, keeping the answers in a table.

For example, in converting “Zeil” to “trials”, we start by forming a table of the cost (edit distance) to convert ““ to ““, “t“, “tr“, “tri”, etc.:

	$\lambda$	t	r	i	a	l	s
$\lambda$	0	1	2	3	4	5	6

(The symbol  $\lambda$  denotes the empty string.)

In other words, we need 0 steps to convert ““ to ““, 1 to convert ““ to “t“, 2 to convert ““ to “tr“, and so on.

Next, we add a row to describe the cost of converting “Z” to ““, “t“, “tr“, … , “trials”:

	$\lambda$	t	r	i	a	l	s
$\lambda$	0	1	2	3	4	5	6
Z	1	1	2	3	4	5	6

OK, clearly we need 1 step to convert “Z” to ““. How are the other entries in this row computed?

### Looking Closer

Let’s back up just a bit:

	$\lambda$	t	r	i	a	l	s
$\lambda$	0	1	2	3	4	5	6
Z	1	?					

What’s the minimum cost to convert “Z” to “t”? It’s the smallest of the three values computed as

#### Add

1 plus the cost of converting “Z” to ““ (we get this cost by looking in the table to the *left* one position).

#### Remove

1 plus the cost of converting ““ to “t“, giving “tZ” (we get this cost by looking *up* one position).

#### Change

1 (because “Z” and “t” are different characters) plus the cost of converting ““ to ““ (we get this cost by looking *diagonally up and to the left* one position).

The last of these yields the minimal distance: 1.

### Applying the General Rule

	$\lambda$	t	r	i	a	l	s
$\lambda$	0	1	2	3	4	5	6
Z	1	1	?				

What’s the minimum cost to convert “Z” to “tr”? It’s the smallest of the three values computed as

#### Add

1 plus the cost of converting “Z” to “t” (we get this cost by looking to the *left* one position).

#### Remove

1 plus the cost of converting ““ to “tr“, giving “trZ” (we get this cost by looking *up* one position).

#### Change

1 (because “Z” and “t” are different characters) plus the cost of converting ““ to “t” (we get this cost by looking *diagonally up and to the left* one position).

The last of these yields the minimal distance: 2.

Got the idea?

**Question:** Can you fill in the rest of that row before reading further (or looking back to where I showed it earlier)?

\*\*Answer\*\* +

	λ	t	r	i	a	l	s
λ	0	1	2	3	4	5	6
Z	1	1	2	3	4	5	6

### Filling In The Table

And we add the next row, using the same technique:

	λ	t	r	i	a	l	s
λ	0	1	2	3	4	5	6
Z	1	1	2	3	4	5	6
e	2	2	2	3	4	5	6

The row after that becomes a bit more interesting. When we get this far ...

	λ	t	r	i	a	l	s
λ	0	1	2	3	4	5	6
Z	1	1	2	3	4	5	6
e	2	2	2	3	4	5	6
i	3	3	3	?			

... we are looking at the cost of converting “Zei” to “tri”. It’s the smallest of the three values computed as

#### Add

1 plus the cost of converting “Zei” to “tr” (we get this cost by looking to the *left* one position).

#### Remove

1 plus the cost of converting “Ze” to “tri”, giving “trii” (we get this cost by looking *up* one position).

#### Change

*Zero* (because “i” and “i” are the same character) plus the cost of converting “Ze” to “tr” (we get this cost by looking *diagonally up and to the left* one position).

The last of these yields the minimal cost of 2.

	λ	t	r	i	a	l	s
λ	0	1	2	3	4	5	6
Z	1	1	2	3	4	5	6
e	2	2	2	3	4	5	6
i	3	3	3	2	?		

and then we can fill out the rest of the row:

	λ	t	r	i	a	l	s
λ	0	1	2	3	4	5	6
Z	1	1	2	3	4	5	6
e	2	2	2	3	4	5	6
i	3	3	3	2	3	4	5

and finally, the last row of the table:

	λ	t	r	i	a	l	s
λ	0	1	2	3	4	5	6
Z	1	1	2	3	4	5	6
e	2	2	2	3	4	5	6
i	3	3	3	2	3	4	5
l	4	4	4	3	3	3	4

Note that this last row, again, has a situation where the cost of a change is zero plus the subproblem cost, because the two characters involved are the same (“l”).

From the lower right hand corner, then, we read out the edit distance between “Zeil” and “trials” as 4.

Here is an implementation of this dynamic programming approach:

```
int editDistance (const string& x, const string& y)
{
    Matrix<int> distances (x.size() + 1, y.size() + 1);

    // Initialize the matrix
    for (int ix = 0; ix <= x.size(); ++ix)
        distances[ix][0] = ix;
    for (int iy = 0; iy <= y.size(); ++iy)
        distances[0][iy] = iy;

    // Fill in rest of the matrix from left to right and top to bottom
    for (int iy = 1; iy <= y.size(); ++iy)
        for (int ix = 1; ix <= x.size(); ++ix)
```

```

    {
        int costByDeleteOrInsert = min(1+distances[ix-1][iy],
                                       1+distances[ix][iy-1]);

        int costByReplacement = distances[ix-1][iy-1] +
            ((x[ix-1] == y[iy-1]) ? 0 : 1);

        distances[ix][iy] = min(costByDeleteOrInsert, costByReplacement);
    }

    return distances[x.size()][y.size()];
}

```

This uses the 2-D [Matrix template](#) from much earlier in the semester. I could have used 2-D arrays or vectors of vectors, but that might have obscured some of the algorithm behind the code to set up and tear down those structures.

It's pretty clear that this algorithm is  $O(x.size() * y.size())$ , both in time and in memory use.

We can actually reduce the memory use quite a bit. If you think about it, you never actually reach back more than one column to the left during these calculations. So we don't really need the whole matrix, just the column that we are currently filling in and the column just to the left of that one. That observation leads to this version:

```

int editDistance (const string& x, const string& y)
{
    int* priorColumn = new int[x.size()+1];
    int* currentColumn = new int[x.size()+1];

    // initialize column 0
    for (int ix = 0; ix <= x.size(); ++ix)
        priorColumn[ix] = ix;

    // Fill in rest of the matrix from left to right and top to bottom
    for (int iy = 1; iy <= y.size(); ++iy)
    {
        currentColumn[0] = iy;
        for (int ix = 1; ix <= x.size(); ++ix)
        {
            int costByDeleteOrInsert = min(1+currentColumn[ix-1],
                                           1+priorColumn[ix]);

            int costByReplacement = priorColumn[ix-1] +
                ((x[ix-1] == y[iy-1]) ? 0 : 1);

            currentColumn[ix] = min(costByDeleteOrInsert, costByReplacement)a;
        }
        swap (priorColumn, currentColumn);
    }

    int distance = priorColumn[x.size()];
    delete [] priorColumn;
    delete [] currentColumn;
    return distance;
}

```

This algorithm is still  $O(x.size() * y.size())$  in time but only  $O(x.size())$  in memory use.

I often see this algorithm abused in practice. In a lot of cases, we want to know whether two strings differ from one another by no more than 2, 3, or some other small number of characters. Using this algorithm for that purpose on strings that are clearly very different from one another can waste a lot of time. So a more practical version puts an upper limit on how many changes are tolerable, and stops the process when that limit is exceeded in an entire column:

```

int editDistance (const string& x, const string& y, int upperLimit)
{
    int* priorColumn = new int[x.size()+1];
    int* currentColumn = new int[x.size()+1];

    // initialize column 0
    for (int ix = 0; ix <= x.size(); ++ix)
        priorColumn[ix] = ix;

    // Fill in rest of the matrix from left to right and top to bottom
    bool keepGoing = true;
    for (int iy = 1; keepGoing && iy <= y.size(); ++iy)
    {
        currentColumn[0] = iy;
        bool keepGoing = false;
        for (int ix = 1; ix <= x.size(); ++ix)
        {
            int costByDeleteOrInsert = min(1+currentColumn[ix-1],
                                           1+priorColumn[ix]);

            int costByReplacement = priorColumn[ix-1] +
                ((x[ix-1] == y[iy-1]) ? 0 : 1);

            currentColumn[ix] = min(costByDeleteOrInsert, costByReplacement)a;
            keepGoing = keepGoing || (currentColumn[ix] <= upperLimit);
        }
    }

}

```

```
        swap (priorColumn, currentColumn);
    }

int distance = (keepGoing) ? priorColumn[x.size()] : upperLimit+1;
delete [] priorColumn;
delete [] currentColumn;
return distance;
}
```

# Exponential and NP Algorithms

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 A Hard Problem - Graph Coloring](#)
  - [1.1 Graph Coloring: A Backtracking Solution](#)
- [2 NP Problems](#)
  - [2.1 A Parallel Approach to Graph Coloring](#)
  - [2.2 Nondeterministic Machines](#)
  - [2.3 NP-Complete Problems](#)
  - [2.4 So, is P = NP?](#)
  - [3 If you can't get across, go around.](#)

We've spent a lot of time and effort this semester worrying about finding efficient algorithms, and learning how to tell if they are efficient in the first place. It's fitting that we end up by noting that there are some very practical problems for which no known efficient solution exists, and for which there is considerable reason to doubt that an efficient solution is possible.

## 1 A Hard Problem - Graph Coloring

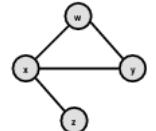
An optimizing compiler may try to save storage by storing variables in the same memory locations if their values are never needed simultaneously:

```
{  
    int w, x, y, z;  
    cin >> w >> x;  
    y = x + w;  
    cout << y;  
    z = x - 1;  
    cout << z << x;  
}
```

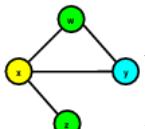
`z` could share a location with `w` or `y`. No other sharing is possible.

The compiler has enough information to construct an “interference graph” in which

- vertices represent variables
- an edge connects two vertices if the corresponding variables cannot share storage



We then try to color the graph, using as few colors as possible, so that no two adjacent vertices have the same color.



Colors represent storage locations. Two vertices with the same color represent variables that can be stored at the same location without interfering with each other.

Graph coloring problems arise in a number of scheduling and resource allocation situations. Similar problems include scheduling a series of meetings so that people attending the meetings are never scheduled to be in two places at once, assigning classrooms to courses, etc.

### 1.1 Graph Coloring: A Backtracking Solution

The most obvious solution to this problem is arrived at through a design referred to as [backtracking](#).

Recall that the essence of backtracking is:

- Number the solution variables  $[v_0, v_1, \dots, v_{n-1}]$ .
- Number the possible values for each variable  $[c_0, c_1, \dots, c_{k-1}]$ .
- Start by assigning  $c_0$  to each  $v_i$ .
- If we have an acceptable solution, stop.
- If the current solution is not acceptable, let  $i = n-1$ .
- If  $i < 0$ , stop and signal that no solution is possible.
- Let  $j$  be the index such that  $v_i = c_j$ . If  $j < k-1$ , assign  $c_{j+1}$  to  $v_i$  and go back to step 4.
- But if  $j \geq k-1$ , assign  $c_0$  to  $v_i$ , decrement  $i$ , and go back to step 6.

Although this approach will find a solution eventually (if one exists), it isn't speedy. Backtracking over n variables, each of which can take on k possible values, is  $O(k^n)$ .

For graph coloring, we will have one variable for each node in the graph. Each variable will take on any of the available colors.

To do a backtracking solution to the graph coloring problem, we start with the plausibility test. The function shown here assumes that the vertices have been assigned sequentially numbered identifiers stored in a map called `vertexNumbers`, and that the colors are represented by integers stored in a backtracking state generator (from our [earlier lesson](#) on backtracking).

```
typedef unordered_map<Vertex, int, VertexHash> VertexToIntegers;

int clashingColors (Graph& g,
                    const VertexToIntegers& vertexNumbers,
                    const BackTrack& colors)
{
    // Test to see if the current coloring is OK. If not, return the
    // lowest number of a vertex that we want to change for the next
    // potential solution.

    int vertexToChange = vertexNumbers.size();
    for (AllVertices v = g.vbegin(); v != g.vend(); ++v)
    {
        int vNum = vertexNumbers[*v];
        int vColor = colors[vNum];
        // Check to see if *v is adjacent to a
        // vertex of the same color.
        // If so, one of them has to change.
        int clashingVertex = -1;
        for (AllOutgoingEdges e = g.outbegin(*v));
            (clashingVertex < 0) && (e != g.outend(*v)); ++e)
        {
            Vertex w = (*e).dest();
            int wNum = vertexNumbers[w];
            int wColor = colors[wNum];
            if (vColor == wColor)
                clashingVertex = max(vNum, wNum);
        }
        if (clashingVertex >= 0)
            vertexToChange = min(vertexToChange, clashingVertex);
    }
    return vertexToChange;
}
```

This function checks all the vertices in the graph to see if any are adjacent to another of the same color. To facilitate pruning of the backtrack search, when it finds clashing assignments, it returns the smallest vertex number that we can safely assume must be changed to yield a solution.

The main routine is a fairly straightforward use of the backtracking generator with pruning.

```
void colorGraph_backtracking (Graph& g, unsigned numColors,
                             ColorMap& colors)
{
    VertexToIntegers vertexNumbers;
    for (AllVertices v = g.vbegin(); v != g.vend(); ++v)
    {
        vertexNumbers[v] = vertexNumbers.size();

        // Search for acceptable solution via backtracking
        BackTrack problem (vertexNumbers.size(), numColors);
        bool solved = false;
        while (!solved && problem.more())
        {
            int pruneAt = clashingColors(g, vertexNumbers, problem);
            if (pruneAt >= vertexNumbers.size())
                solved = true;
            else
                problem.prune(pruneAt+1);
        }

        colors.clear();
        if (solved)
        {
            // Construct the solution (map of vertices to colors)
            for (AllVertices v = g.vbegin(); v != g.vend(); ++v)
                colors[*v] = problem[vertexNumbers[*v]];
        }
    }
}
```

The main while loop uses an `AllVertices` iterator to track which variable (vertex) we are trying to assign (color).

Try [running the backtracking form](#) of this algorithm. Try it at least once where you use fewer colors than are actually possible for that graph, to give yourself a feel for how backtracking (mis)behaves when searching for a subtle or non-existent solution.

Before moving on, it's worth remembering that backtracking is often (usually?) written in recursive form.

```
bool colorGraph_backtracking (Graph& g, unsigned numColors,
                             ColorMap& colors)
{
```

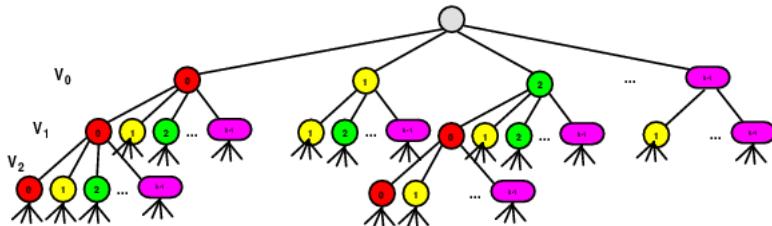
```

for (AllVertices v = g.vbegin(); v != g.vend(); ++v)
{
    colors[*v] = -1;
}
colorGraph_rec (g, g.vbegin(), numColors, colors);
}

bool colorGraph_rec (Graph& g,
                     AllVertices v,
                     unsigned numColors,
                     ColorMap& colors)
{
    if (v != g.vend())
    {
        // Try to color vertex *v, then recursively color the rest.
        c = chooseColor(g, *v, numColors, colors, -1);
        while (c >= 0 && c < numColors)
        {
            colors[*v] = c;
            AllVertices w = v;
            ++w;
            if (colorGraph_rec(g,w,numColors,colors))
                return true; // A solution has been found
            else
                // We have backtracked to here - try a different color for v
                c = chooseColor(g, *v, numColors, colors, c);
        }
        // If we exit the above loop, no solution is possible given the
        // colors that had already been assigned to g.vbegin()..v
        colors[*v] = -1;
        return false;
    }
    else
        return true;
}

```

If the solution variables for a problem form a graph or tree, working with adjacent variables may cause pruning to take place earlier, since the plausibility tests are often framed in terms of properties of adjacent variables.



The solutions considered by this recursive routine are the same as in the iterative case, but now this tree could also be considered to represent the recursive calls made, with each vertex representing an activation (call) to the recursive routine from its parent.

The algorithms I have shown you for coloring are exponential time.

Can we do better?

- Yes, but only in the constant multipliers.
- There is no known algorithm for graph coloring that is not worst-case exponential time.
- There is considerable reason to believe that *no polynomial-time algorithm is possible* for this problem.

## 2 NP Problems

We call the set of programs whose worst case is a polynomial order the class P.

- Suppose that we had an infinite number of computers at our disposal,
- and could spawn off problems to any number of them in constant time.
- These computers could then run in parallel with each other.

### 2.1 A Parallel Approach to Graph Coloring

This procedure works by asking different computers to separately consider the different colors for v.

```

bool colorGraph_rec (Graph& g,
                     AllVertices v,
                     unsigned numColors,
                     ColorMap& colors)

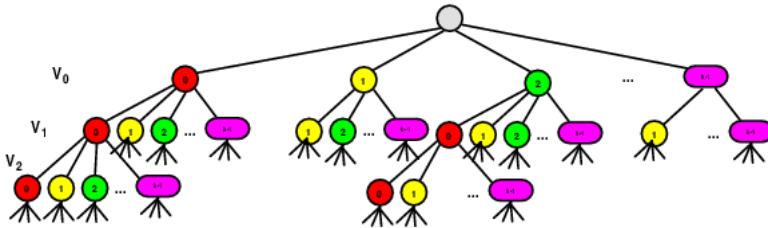
```

```

{
    if (v != g.vend())
    {
        // Try to color vertex *v, then recursively color the rest.
        c = chooseColor(g, *v, numColors, colors, -1);
        while (c >= 0 && c < numColors)
        {
            colors[*v] = c;
            AllVertices w = v;
            ++w;
            spawn colorGraph_rec(g,w,numColors,colors);
            c = chooseColor(g, *v, numColors, colors, c);
        }
        if (any spawned process succeeds)
            return true; // A solution has been found
        else
            return false;
    }
}

```

The solutions being considered would still look like a (pruned) tree:



But now, all nodes at the same level of the call trees are being executed in parallel.

The running time is therefore proportional the depth of the call tree times the cost per call:  $O(|V|)$ . And that's a polynomial!

## 2.2 Nondeterministic Machines

We call this mythical machine that allows us to spawn off parallel computations at no cost a [nondeterministic machine](#).

The set of programs that can be run in [Polynomial](#) time on a [Nondeterministic](#) machine is called the [NP](#) algorithms, and the set of problems solvable by an algorithm from that set are [NP problems](#).

Note that any problem that can be solved in polynomial time on a single, conventional processor, can certainly be solved in polynomial time on an infinite number of processors, so  $P \subseteq NP$ .

There are some exponential time algorithms that can't be solved in polynomial time even on a nondeterministic machine. So the NP problems clearly occupy an intermediate niche between these really nasty exponential problems and the polynomial time P problems.

This leads to one of the more famous unresolved speculations in computer science:

Is  $P = NP$  ?

In other words, is an NP problem one for which a polynomial-time solution on a conventional processor exists, but we just haven't discovered it yet? Or are at least some NP problems truly exponential-time, with no polynomial-time solution possible?

There's another interesting thing about the NP problems. They all have the curious property that it's "easier" to tell *if* we have a solution than it is to compute such a solution in the first place. In fact, this is how the NP problems are usually defined: the set of problems for which we can determine, in polynomial time, whether a proposed solution is correct or not.

You can see how that idea relates to graph coloring. It's hard to *find* a coloring for graph. But if you show me a colored graph, I can traverse the graph in polynomial time, comparing each vertex to the ones adjacent to it, and determine whether any two adjacent vertices have the same color. So I can tell, in polynomial time, *whether* a proposed coloring is correct, but I need exponential time to *find* such a coloring.

## 2.3 NP-Complete Problems

The speculation about  $P = NP$  is particularly interesting in view of the discovery of a special subset of the NP problems.

Among the NP problems, there is a collection of problems called [NP-complete problems](#) that have the property

- for any NP-complete problem A, and
- for any other NP-complete problem B

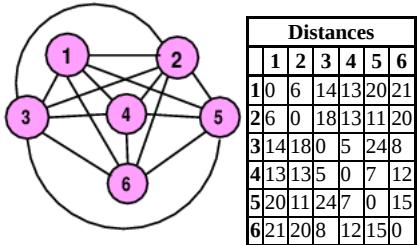
B can [reduced](#) (converted) to A in polynomial time.

Consequently, if we could find a polynomial-time solution to even one NP-complete problem, we would have a polynomial-time solution to *all* the NP-complete problems.

Graph coloring is NP-complete.

### 2.3.1 The Traveling Salesman Problem

The best known NP-complete problem is the [Traveling Salesman Problem](#): Given a complete graph with edge costs, find the cheapest simple cycle that visits each node exactly once.



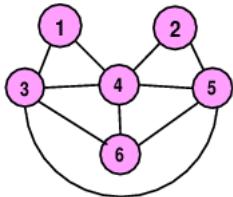
A lot of practical resource allocation and scheduling problems can be shown to be equivalent to the traveling salesman problem, which is somewhat unfortunate because, like all NP-complete problems, we only know exponential-time algorithms for it.

### 2.3.2 Hamiltonian Cycles

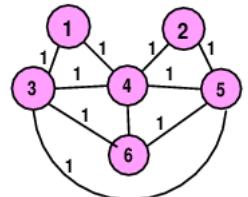
By way of illustration of this idea of reducing one problem to another, consider the [Hamiltonian Cycle](#) problem:

Given a connected graph, is there a simple cycle visiting each node?

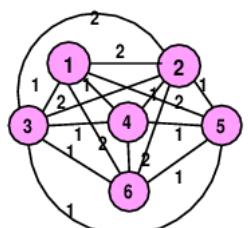
This can be converted into the Traveling Salesman problem in  $O(|V|^2)$  time.



For example, given this graph (This has Hamiltonian cycle [3,1,4,2,5,6,3].)

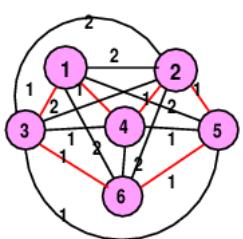


We assign each edge a cost 1:



and then fill in the remaining edges, giving each new edge the cost 2:

The original graph has a Hamiltonian cycle if and only if this graph has a Traveling Salesman solution with cost =  $|V|$ .



For example, if we solve the traveling salesman problem for this graph, we see that the cheapest cycle involves only edges from the original graph.

There are  $|V|$  of them (since the cycle must visit every vertex), so the total cost is  $|V|$ .

If the cost had come out higher, we would know that there had to be at least one cost=2 edge, meaning that no Hamiltonian cycle existed.

So, if we *could* solve the Traveling Salesman problem in polynomial time, we would be able to solve the Hamiltonian Cycle problem in polynomial time as well.

This is typical of the kind of reduction that relates NP-complete problems to the rest of the NP problems.

## 2.4 So, is P = NP?

- To date, no one has found a polynomial time algorithm for any NP-complete problem.
- No one has been able to prove that no such polynomial-time algorithm exists.
- The answer to the question “is P = NP?” is generally believed to be, “No”.

## 3 If you can't get across, go around.

In practice, when faced with an NP or exponential problem, we often resort to approximate solutions or to algorithms that do not always give the best solution.

- For example, polynomial time algorithms are known that can solve the traveling salesman problem within some error factor.

A heuristic is an approach to a solution that often, but not always, succeeds. Heuristic algorithms often work by exploring a small number of the most common or most likely possibilities in any given situation. For example, a common heuristic approach for many algorithms is the “greedy” approach: always try to take the largest possible step towards a likely overall solution.

A heuristic algorithm for coloring a graph using k colors:

- Discard all vertices of degree less than k, keeping them on a stack.
  - If the rest is successful, we can come back to these and trivially color them.
- Keep remaining vertices sorted by the number of colors already assigned to adjacent vertices. (We call this the “number of constraints” on the vertex.)
- Repeatedly pick the most constrained vertex and assign it any legal color. (This is the “greedy” part, as we are hoping that by guessing early at the “hardest” parts of the overall problem, the rest will all work out.)
- If all the remaining vertices are colored, start popping discarded vertices from the stack, assigning them any legal color as you do.

Run the heuristic form of graph coloring. Note how much better it behaves when faced with problems where you have allowed too few colors to actually color the graph.

Now, this algorithm is not guaranteed to always find a solution, even if one is possible. But it's not all that easy to come up with a graph on which this algorithm will fail but the much more expensive backtracking approach would succeed. Try it!

# Priority Queues

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 The std Priority Queue Interface](#)
  - [1.1 priority\\_queue is an Adapter](#)
- [2 Applications of Priority Queues](#)

Problem: Given a collection of elements that carry a numeric “score”, find and remove the element with the smallest [largest] score. New elements may be added at any time.

This collection is called a *priority queue* because,

- like queues, it is used to simulate objects awaiting a service.
- But instead of FIFO, the processing order is determined by the object’s individual “priority” or score.

## 1 The std Priority Queue Interface

This is the entire priority queue interface.

```
template <class T, class Container, class Compare>
class priority_queue
{
public:
    typedef T value_type;
    typedef Container::size_type size_type;

protected:
    Container c;
    Compare comp;

public:
    priority_queue(const Compare& x = Compare());

    template <class Iterator>
    priority_queue(Iterator first,
                  Iterator last,
                  const Compare& x = Compare());

    bool empty() const;
    size_type size() const;
    value_type& top();
    const value_type& top() const;
    void push(const value_type& x);
    void pop();
};
```

- We can check the size of the priority queue or ask if it is empty.
- We can look at the top (largest) element.
- We can remove the largest element by pop’ping.
- We can push a new element into the priority queue. Unlike pushing onto a stack or queue, however, the element does not automatically become the first or last thing we will next retrieve. Exactly when we will see this element again depends on its priority value.

### 1.1 priority\_queue is an Adapter

```
template <class T, class Container, class Compare>
class priority_queue
{
public:
    :
```

Like std::stack and std::queue, 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> > myPQueue;
```

## 2 Applications of Priority Queues

Priority queues tend to crop up in a lot of algorithms as “helpers” for working with other data structures. In particular, we’ll see them in connection with a lot of algorithms for finding optimal paths and doing other [work with graphs](#).

Priority queues are often useful in scheduling. For example, in [discrete event simulation](#), we simulate real-world systems as a series of [events](#) that are scheduled to take place at some future time. For example, if we wanted to simulate automobile traffic on downtown streets, we might create a whole collection of events, some of which spawn other future events:

- Each traffic light in our simulation would have an event scheduled for its next change. When a light changes to red, we schedule it’s next change 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 average size of the car-generation interval would 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.

All scheduled events could be kept in a (min) priority queue ordered on the time at which the event is scheduled. The main logic of the simulation is simply

```
priority_queue<Event, vector<Event> > futureEvents;
:
while (simulation has not ended)
{
    get next event from futureEvents;
    trigger that event;
}
```

Each distinct type of event would have its own “trigger” function, many of which are likely to add one or more additional events to the priority queue. What’s essential is that we always be able to get to the next scheduled event each time around that main loop. When a new event gets added, it might stay far down in the queue (if the new event is a long time in the future) or might spring to the front of the queue if there is nothing more immediate. For example, suppose we have a single car speeding down Main Street at 3AM. Each of its “will-reach-next-intersection” events is likely to go immediately to the first position in the queue, unless the traffic lights are cycling very quickly.

# Heaps

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Recap: the std Priority Queue Interface](#)
  - [1.1 The Priority Queue Implementation](#)
- [2 Implementing Priority Queues - the Heap](#)
  - [2.1 Binary Heaps](#)
  - [2.2 Heaps are complete trees](#)
  - [2.3 Children's Values are Smaller than Their Parent's](#)
  - [2.4 Bubbling and Percolating](#)
  - [2.5 Inserting into a heap](#)
  - [2.6 Removing from Heaps](#)
- [3 Analysis](#)
  - [3.1 Building a Heap](#)
- [4 Implementing the Heap with Iterators](#)
  - [4.1 push\\_heap](#)
  - [4.2 pop\\_heap](#)
  - [4.3 percolateDown](#)
  - [4.4 make\\_heap](#)

Problem: Given a collection of elements that carry a numeric “score”, find and remove the element with the smallest [largest] score. New elements may be added at any time.

In an [earlier lesson](#), we saw that this collection is called a [priority queue](#). Now we will look at an efficient way of implementing it.

## 1 Recap: the std Priority Queue Interface

This is the entire priority queue interface.

```
template <class T, class Container, class Compare>
class priority_queue
{
public:
    typedef T value_type;
    typedef Container::size_type size_type;

protected:
    Container c;
    Compare comp;

public:
    priority_queue(const Compare& x = Compare());
    template <class Iterator>
    priority_queue(Iterator first,
                  Iterator last,
                  const Compare& x = Compare());

    bool empty() const;
    size_type size() const;
    value_type& top();
    const value_type& top() const;
    void push(const value_type& x);
    void pop();
};
```

- We can check the size of the priority queue or ask if it is empty.
- We can look at the top (largest) element.
- We can remove the largest element by popping.
- We can push a new element into the priority queue.

Unlike pushing onto a stack or queue, however, the element does not automatically become the first or last thing we will next retrieve. Exactly when we will see this element again depends on its priority value.

### 1.1 The Priority Queue Implementation

```

template <class T, class Container, class Compare>
class priority_queue
{
public:
    typedef T value_type;
    typedef Container::size_type size_type;

protected:
    Container c;
    Compare comp;

public:
    priority_queue(const Compare& x = Compare())
        : c(), comp(x) {}

    template <class Iterator>
    priority_queue(Iterator first,
                   Iterator last,
                   const Compare& x = Compare())
        : c(first, last), comp(x)
    {
        make_heap(c.begin(), c.end(), comp); ①
    }

    bool empty() const { return c.empty(); }

    size_type size() const { return c.size(); }

    value_type& top() { return c.front(); }

    const value_type& top() const
    { return c.front(); }

    void push(const value_type& x)
    {
        c.push_back(x);
        push_heap(c.begin(), c.end(), comp); ②
    }

    void pop()
    {
        pop_heap(c.begin(), c.end(), comp); ③
        c.pop_back();
    }
};

```

Most of the work is in the 3 algorithm fragments, `make_heap` ①, `push_heap` ②, and `pop_heap` ③. These are template functions from the standard library that are used to implement a “heap”, a new data structure that we will look at in detail shortly.

First, though, let’s look at what we could do to implement priority queues using the data structures we already know.

One possibility would be to use a sorted sequential structure (array, vector, or list). For example, using a `vector`, we would try to keep the elements in ascending order by priority. Then we could get the `top()` of the priority queue as the `back()` of the implementing `vector`.

**Question:** With this data structure, what would the complexities of the priority queue `push` and `pop` operations be?

- O(1) and O(1)
- O(1) and O(n)
- O(n) and O(1)
- O(n) and O(n)

Answer +

## Priority Queues via Ordered Insert

The priority queue push and pop operations would be  $O(n)$  and  $O(1)$ , respectively.

To push a new element onto the priority queue, we must do an ordered insert, which we know is  $O(n)$ .

To pop the priority queue, we remove its smallest element, which we have stored at the end of the vector. We can remove an element from the end of a vector with the `pop_back` operation in  $O(1)$  time. (That's why we're storing the elements in ascending order of priority rather than in descending order. If we had used descending order, the largest element would be at the front instead of at the back, and removing it would be  $O(n)$ .)

Note that, if we used lists or deques instead of vectors, we could store the items in either ascending or descending order and get an  $O(1)$  pop, but push would remain  $O(n)$ .

We can do better than that.

We might consider instead using a balanced binary search tree to store the priority queue. This time, it will be a little easier if we store the items in *descending* order by priority. We could get the `top()` of the priority queue as `*begin()`.

### Question:

Using a balanced binary search tree as the underlying data structure, what would the complexities of the priority queue push and pop operations be?

- $O(1); O(\log n)$
- $O(\log n); O(1)$
- $O(\log n); O(\log n)$
- $O(\log n); O(n)$
- $O(n); O(\log n)$
- $O(n); O(n)$

Answer +

### Priority Queues via Binary Search Trees

The priority queue push and pop operations would be  $O(\log n)$ ;  $O(\log n)$ .

We would implement push by simply inserting into the balanced search tree, which is  $O(\log n)$ .

pop would be implemented as `erase(begin())`, which is also  $O(\log n)$ .

These are both the worst case and the average complexities.

That sounds pretty good.

We can't actually hope to improve on the worst case times offered by balanced search trees, we can match those worst case times (and improve on the multiplicative constant) and actually achieve  $O(1)$  average case complexities by the use of a new data structure, called a "heap".

## 2 Implementing Priority Queues - the Heap

We can implement priority queues using a data structure called a (binary) [heap](#).

### 2.1 Binary Heaps

A binary heap is a binary tree with the properties:

- The tree is [complete](#) (entirely filled, except possibly on the lowest level, which is filled from left to right).
- Each non-root node in the tree has a smaller (or equal) value than its parent.

\*\*Important: A heap is a binary tree, but *not* a binary [search](#) tree. The ordering rules for heaps are different from those of binary search trees.

What I have defined here is sometimes called a [max-heap](#), because the largest value in the heap will be at the root. We can also have a [min-heap](#), in which every child has a value larger than its parent.

Max-heaps always have their very largest value in their root. Min-heaps always have their smallest value in the root.

In this course, we will always assume that a "heap" is a "max-heap" unless explicitly stated otherwise.

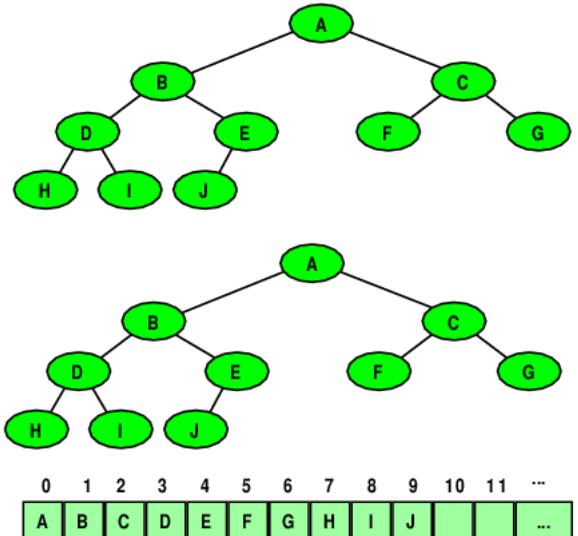
Let's look at the implications of each of these two properties.

### 2.2 Heaps are complete trees

Here's an example of a complete binary tree.

Complete binary trees have a very simple linear representation, allowing us to implement them in an array or vector with no pointers.

- The parent of node  $i$  is in slot  $\left\lfloor \frac{i-1}{2} \right\rfloor$ .
- The children of node  $i$  are in  $2i + 1$  and  $2i + 2$ .

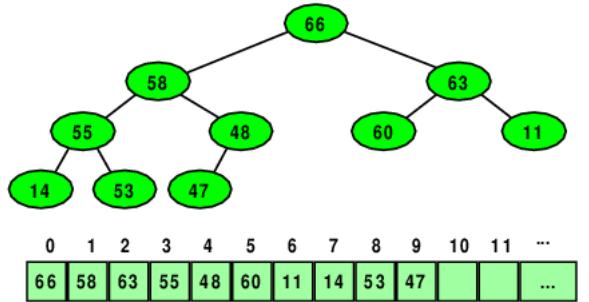


### 2.3 Children's Values are Smaller than Their Parent's

Using the same tree shape, we can fill in some values to show an example of a heap.

Each parent has a value larger than its children's values (and, therefore, larger than the values of any of its descendants).

So when we ask for the front (largest value) of a priority queue, we find it in the root of the heap, which in turn will be in position 0 of the array/vector.



## 2.4 Bubbling and Percolating

Before looking in detail at how to add and delete elements from a heap, let's consider a situation in which we have a "damaged" heap with one node out of position.

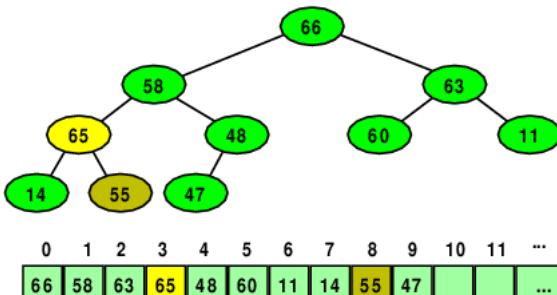
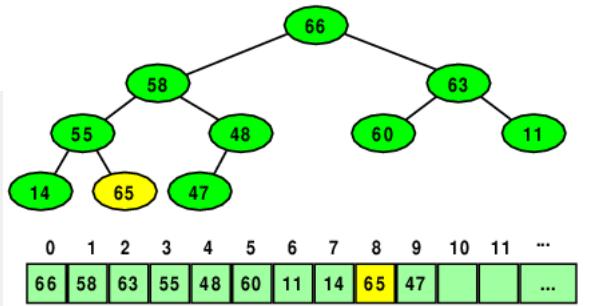
How do we "fix" the heap? There are two cases to consider.

- The out-of-place node is too large (i.e., larger than its parent).
- The out-of-place node is too small (i.e., smaller than one or both of its children).

### 2.4.1 Bubble Up

When we have a node that is larger than its parent, we bubble it up by swapping it with its parent until it has reached its proper position.

```
void bubbleUp (vector<T>& heap,
               unsigned nodeToBubble)
{
    unsigned parent = (nodeToBubble - 1) / 2;
    while (node > 0 && heap[nodeToBubble] > heap[parent])
    {
        swap(heap[nodeToBubble], heap[parent]);
        nodeToBubble = parent;
        parent = (nodeToBubble - 1) / 2;
    }
}
```



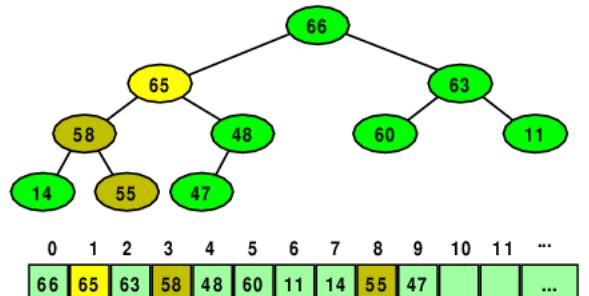
In this case, starting with *nodeToBubble* = 8, we swap node 8 with its parent 3 ...



... and then because node 3 is still greater than its parent (1), we swap again.

Now, node 1 is not greater than its parent (0), so we are done bubbling.

Note that we have repaired the heap. The final arrangement satisfies the ordering requirements for a heap.



### 2.4.2 Percolate Down

When we have a node that is smaller than one or both of its children, we percolate it down by swapping it with the larger of its children until it has reached its proper position.

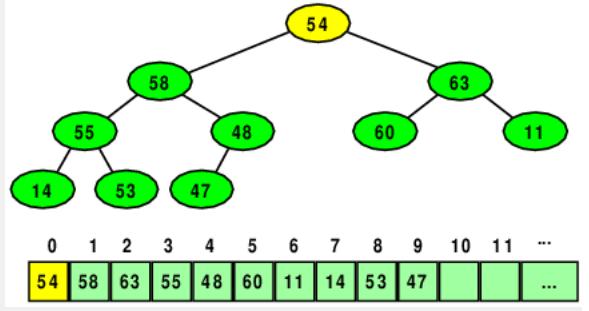
```
void percolateDown (vector<T>& heap,
                     unsigned nodeToPerc)
```

```

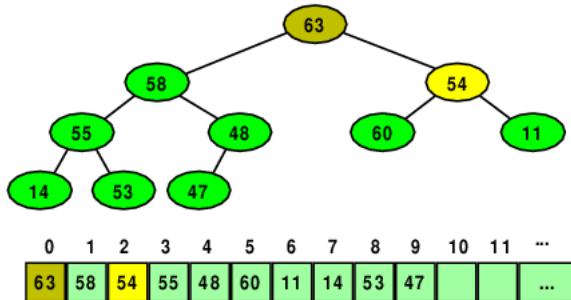
{
    while (2*nodeToPerc+1 < heap.size())
    {
        unsigned child1 = 2*nodeToPerc +1;
        unsigned child2 = child1+1;

        unsigned largerChild = child1;
        if (child2 < heap.size()
            && heap[child2] > heap[child1])
            largerChild = child2;
        if (heap[largerChild] > heap[nodeToPerc])
        {
            swap (heap[nodeToPerc], heap[largerChild]);
            nodeToPerc = largerChild;
        }
        else
            nodeToPerc = heap.size();
    }
}

```



This is only a little more complicated than bubbling up. 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.



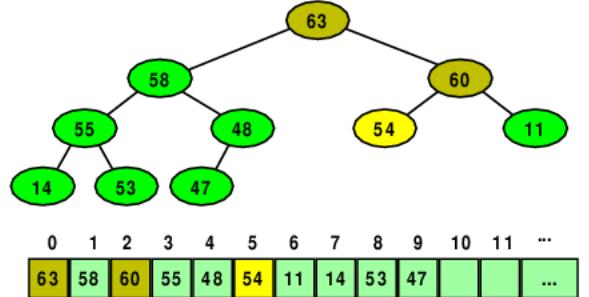
In this case, starting with *nodeToPerc* = 0, we swap node 0 with its larger child, 2 ...

... and then, because node 2 is still less than one of its children (5), we swap again.

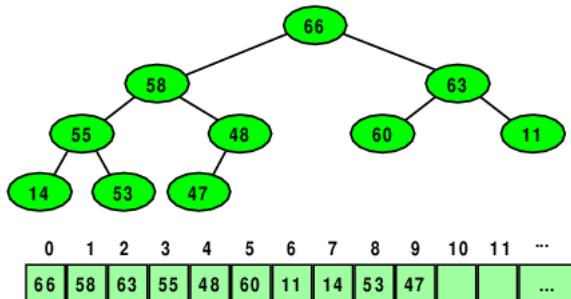
Now, node 5 has no children (*child1* >= *heap.size()*), so we are done percolating.

Note that we have again repaired the heap. The final arrangement satisfies the ordering requirements for a heap.

If you understand the ideas of bubbling up and percolating down, then almost all the things you would want to do to a heap become a variant of those two ideas.



## 2.5 Inserting into a heap



Suppose we have this heap and we want to add a new item to it.

Now, after we add an item to the heap, it will have one more tree node than it currently does. Because heaps are complete trees, we know exactly how the shape of the tree will change, even if we can't be sure how the data values in the tree might be rearranged.

**Question:** How will the shape of the tree shown above change?

- A new child will be added to the node that currently contains 48.
- A new child will be added to the one of the nodes that currently contain 48, 60, or 11.
- A new child will be added to one of the current leaves.
- None of the above.

**Answer:**

- A new child will be added to the node that currently contains 48.

The heap is a complete tree, so new nodes will continue to go into the bottom level, filling it from left to right. Consequently, the next node to be added will be a child of the node that currently holds 48.

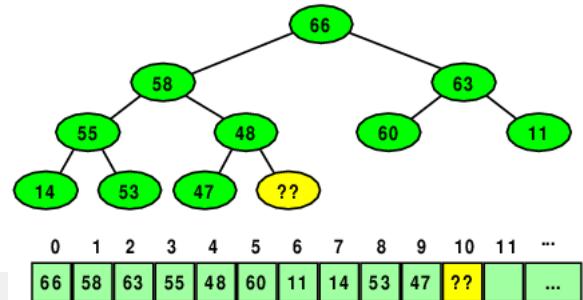
Well, suppose that we just go ahead and put the new value into that position.

We've got two possibilities.

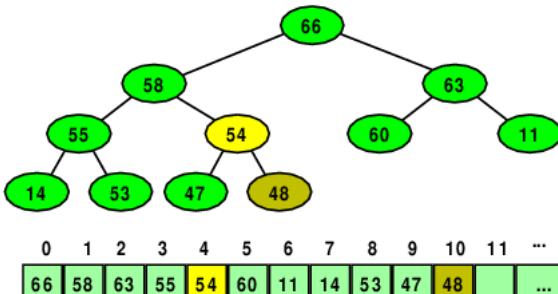
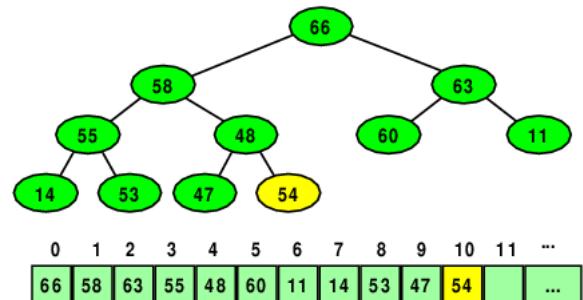
- We might get lucky – maybe this is where the new value belongs.
- If the new value is out of position, it must be because it is larger than its parent.

It would be the *only* node that was out of position, and we know how to “repair” a heap with a single node out of position that is larger than its parent — we bubble up!

```
void add_to_heap (vector<T>& heap, const T& newValue)
{
    heap.push_back (newValue); // add newValue to complete tree
    bubbleUp (heap, heap.size()-1); // repair the heap
}
```



For example, suppose we wanted to add 54 to the heap. First we would push 54 onto the end of the vector, in effect adding it to the complete tree.



When bubbleUp is called, we note that 54 is greater than its parent, so we swap ...

... and since 54 is no longer greater than its new parent, we are done.

## 2.6 Removing from Heaps

When we pop, or remove, from a heap, we know that the value being removed is the value currently in the root.

We also know how the tree shape will change. The rightmost node in the bottom level will disappear.

Now, unless the heap only has one node, the node that's disappearing does not contain the value that we're actually removing. So, we have two problems:

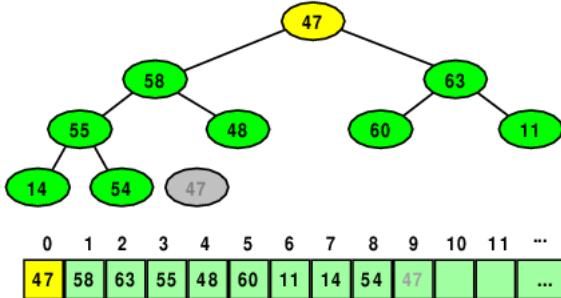
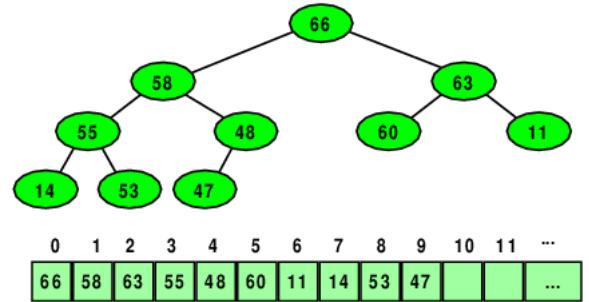
- What value goes into the root to replace the one being removed?
- What do we do with the value currently in the node that's going to disappear?

So, we've got a node with no data, and data that needs a node. The natural thing to do is to put the data in that node.

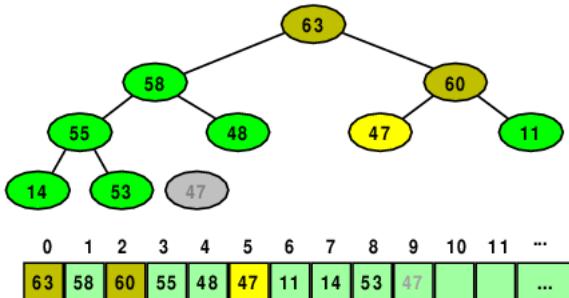
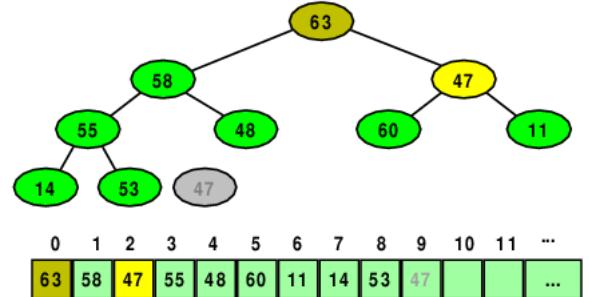
That data value will almost certainly be out of position, being smaller than one or both of its children, but, again, that's only a single node that's out of position. We know how to fix that.

```
void remove_from_heap (vector<T>& heap)
{
    heap[0] = heap[heap.size()-1]; // replace root value
    heap.pop_back(); // remove the duplicate node
    percolateDown (heap, 0); // repair the heap
}
```

Suppose we wanted to remove the maximum value from this heap.



When `percolateDown` is called, we swap 47 with the larger of its children.



## 3 Analysis

A binary heap has the same shape as a balanced binary search tree.

Therefore its height, for  $n$  nodes, is  $\lfloor \log(n) \rfloor$ .

`push_heap` and `pop` do  $O(1)$  work on each node along a path that runs, at worst, between a single leaf and the root.

Hence both operations are  $O(\log n)$ , worst case.

The *average* case for `push_heap` is  $O(1)$ . The proof of this is beyond scope of this class.

### 3.1 Building a Heap

A single insertion is  $O(\log n)$  worst case and  $O(1)$  average.

What happens if we start with an empty heap and do  $n$  inserts? The resulting total could be  $O(n \log n)$ .

As it happens, we can do better with a special `make_heap` operation to build an entire heap from an array (or array-like structure such as a vector).

```

void build_heap (vector<T>& heap)
{
    unsigned i = (heap.size()-1)/2;
    do {
        percolateDown (heap, i);
        --i;
    } while (i >= 0);
}

```

- Start with the data in any order.
- Force heap order by percolating each non-leaf node.

Since each `percolateDown` takes, in worst case, a time proportional to the height of the node being percolated, the total time for `build_heap` is proportional to the sum of the heights of all the nodes in a complete tree. It is possible to show that this sum is itself  $O(n)$ , where  $n$  is the number of nodes in the tree. Therefore `build_heap` is  $O(n)$ .

So it's cheaper to build a heap all at once than to do it one push at a time, although neither approach is terribly expensive.

## 4 Implementing the Heap with Iterators

I've shown the operations `add_to_heap`, `remove_from_heap`, and `build_heap` in a simplified form so far. I've used `[ ]` indexing to get access to the elements because doing so makes the child/parent position calculations easier to read.

But the actual `std` function templates for heap manipulation use iterators, so let's look at the "real thing" now.

### 4.1 push\_heap

```

template <class RandomIterator, class Compare>
void push_heap (RandomIterator first,
                RandomIterator last,
                Compare comp)
//Pre: The range [first, last-1) is a valid heap.
//Post: Places the value in the location last-1 into the resulting
//      heap [first, last).
{
    RandomIterator nodeToBubble = last - 1;
    RandomIterator parent = first + (nodeToBubble - first - 1)/2;
    while (nodeToBubble != first && comp(*parent, *nodeToBubble))
    {
        swap(*nodeToBubble, *parent);
        nodeToBubble = parent;
        parent = first + (nodeToBubble - first - 1)/2;
    }
}

```

```

void bubbleUp (vector<T>& heap,
               unsigned nodeToBubble)
{
    unsigned parent = (nodeToBubble - 1) / 2;
    while (node > 0 && heap[nodeToBubble] > heap[parent])
    {
        swap(heap[nodeToBubble], heap[parent]);
        nodeToBubble = parent;
        parent = (nodeToBubble - 1) / 2;
    }
}

```

Compare that code for `push_heap` to our earlier code for `bubbleUp`. You can see that `push_heap` is pretty much a 1-to-1 translation of `bubbleUp` in which we have made the following changes:

- Replaced unsigned indices by iterators
- Replaced indexing `[ ]` by the iterator `*`
- The `parent` calculations are offset by the starting position `first`
- The comparison function (or functor) `comp` is used to compare items instead of `<`.

### 4.2 pop\_heap

The `std::pop_heap` function relies on a utility, `_percolateDown`. [<sup>^</sup> The “\_” on the front of the function name `_percolateDown` is a signal that this function name is not part of the C++ standard, and so may not actually exist in your compiler's standard library, unlike `pop_heap` or `push_heap`, which are part of the standard.]

```

template <class RandomIterator, class Compare>
void pop_heap (RandomIterator first,
               RandomIterator last,
               Compare comp)
//Pre: The range [first, last) is a valid heap.
//Post: Swaps the value in location first with the value in the location
//      last-1 and makes [first, last-1) into a heap.
{
    swap (*first, *last);
    _percolateDown (first, last, comp, 0, last-first);
}

```

(Look carefully at the pre and post conditions for `pop_heap`, which are copied from the C++ language standard. `pop_heap` may be the only function in the entire C++ standard library for which *last* is an *inclusive* bound on a range of positions.)

## 4.3 percolateDown

The utility function `_percolateDown` ...

```
template <class RandomIterator, class Compare, class Distance>
void _percolateDown
(RandomIterator first,
 RandomIterator last,
 Compare comp,
 Distance nodeToPerc,
 Distance heapSize)
{
    while (2*nodeToPerc+1 < heapSize)
    {
        Distance child1 = 2*nodeToPerc +1;
        Distance child2 = child1+1;
        Distance largerChild = child1;
        if (child2 < heapSize
            && *(first + child2) > *(first+child1))
            largerChild = child2;
        if (*(first + largerChild) > *(first + nodeToPerc))
        {
            swap (*(first + nodeToPerc), *(first + largerChild));
            nodeToPerc = largerChild;
        }
        else
            nodeToPerc = heapSize;
    }
}
```

... differs from our earlier `percolateDown` ...

```
void percolateDown (vector<T>& heap,
                    unsigned nodeToPerc)
{
    while (2*nodeToPerc+1 < heap.size())
    {
        unsigned child1 = 2*nodeToPerc +1;
        unsigned child2 = child1+1;

        unsigned largerChild = child1;
        if (child2 < heap.size()
            && heap[child2] > heap[child1])
            largerChild = child2;
        if (heap[largerChild] > heap[nodeToPerc])
        {
            swap (heap[nodeToPerc], heap[largerChild]);
            nodeToPerc = largerChild;
        }
        else
            nodeToPerc = heap.size();
    }
}
```

... by the same changes as we saw for `push_heap` and `bubbleUp`.

- Replaced `unsigned` indices by iterators
- Replaced indexing `[ ]` by the iterator `*`
- The parent/child calculations are offset by the starting position `first`
- The comparison function (or functor) `comp` is used to compare items instead of `<`.

## 4.4 make\_heap

`make_heap` uses the same `_percolateDown` utility as does `pop_heap`.

```
template <class RandomIterator, class Compare>
void make_heap (RandomIterator first,
                RandomIterator last,
                Compare comp)
//Pre:
//Post: Arranges the values in [first, last) into a heap.
{
    if (first != last)
    {
        RandomIterator i = first+(last-first-1)/2 + 1;
        while (i != first)
        {
            --i;
            _percolateDown (first, last, comp, i-first, last-first);
```

```
        }  
    }  
}
```

It invokes `_percolateDown` on each non-leaf node, working back towards the root.

Try running the [heap algorithms](#).

# Heapsort

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Heapsort – Conceptual](#)
- [2 Implementing HeapSort](#)
- [3 Analysis of HeapSort](#)
- [4 Introspective Sort](#)

Priority queues are useful structures in their own right. They often arise in scheduling problems where things must be serviced based upon their importance. We'll also see, in a later section, that they are valuable in many graph manipulation algorithms.

Hulls have another important application besides as implementations of priority queues. They can be used to provide a simple and efficient sorting algorithm.

## 1 Heapsort – Conceptual

```
template <class RandomIterator, class Compare>
void pop_heap (RandomIterator first,
               RandomIterator last,
               Compare comp)
//Pre: The range [first,last) is a valid heap.
//Post: Swaps the value in location first with the value in the location
//       last-1 and makes [first, last-1) into a heap.
```

Take another look at the description of the `std::pop_heap` function.

The position `last-1` winds up holding the former root of the heap (the largest element that had been in the heap).

```
template <class Container, class Compare>
class priority_queue
{
    :
    void pop()
    {
        pop_heap(c.begin(), c.end(), comp);
        c.pop_back();
    }
};
```

When we use this routine to implement a priority queue, we simply discard that element.

But suppose that we kept it instead of throwing it away.

```
pop_heap (c.begin(), c.end(), comp);
pop_heap (c.begin(), c.end()-1, comp);
pop_heap (c.begin(), c.end()-2, comp);
    :
```

Assume that `c` contains a heap.

The first call here would put the largest element of the container `c` in position `c.end() - 1`.

The second call would put the largest remaining element in the container `c` (the second largest in the original container) in position `c.end() - 2`.

The third call would put the largest remaining element in the container `c` (the third largest in the original container) in position `c.end() - 3`.

If we keep this up, we will eventually wind up having sorted all the elements in `c`.

## 2 Implementing HeapSort

[hs1.cpp](#) +

```
template <class Iterator, class Compare>
void heapsort (Iterator first, Iterator last, Compare comp)
{
    make_heap (first, last, comp);
    while (last != first)
    {
        pop_heap (first, last-1, comp);
        --last;
    }
}
```

```
| }
```

A heap sort is really pretty simple. First we form the array into a heap. Then we repeatedly pop the heap, collecting the successive maximum values at the end of the container.

Try running [this algorithm](#).

## 3 Analysis of HeapSort

```
template <class Iterator, class Compare>
void heapsort (Iterator first, Iterator last, Compare comp)
{
    make_heap (first, last, comp);
    while (last != first)
    {
        pop_heap (first, last-1, comp);
        --last;
    }
}
```

As always, we work from the inside out. `--last` is obviously  $O(1)$ .

**Question:** What is the worst-case complexity of the call to `pop_heap`?

- $O(1)$
- $O(\log N)$
- $O(\log (\text{last}-\text{first}))$
- $O(N)$
- $O(\text{last} - \text{first})$

*Answer:* +

`pop_heap` runs in time proportional to the log of the number of elements in the heap, so the worst-case complexity of the call to is  $O(\log(\text{last}-\text{first}))$

```
template <class Iterator, class Compare>
void heapsort (Iterator first, Iterator last, Compare comp)
{
    make_heap (first, last, comp);
    while (last != first)           // O(1)
    {
        pop_heap (first, last-1, comp); // O(log (last-first))
        --last;                      // O(1)
    }
}
```

The value of `last` changes each time around the loop, so we can't use the multiplicative shortcut. But let  $n$  stand for the value of `last - first` when we first entered the `heapsort` function. Then it's clear that, each time around the loop,  $\text{last} - \text{first} \leq n$ .

At some risk, therefore, of obtaining an overly loose complexity bound, we can treat the body as  $O(\log n)$ . So, we have ...

```
template <class Iterator, class Compare>
void heapsort (Iterator first, Iterator last, Compare comp)
{
    make_heap (first, last, comp);
    while (last != first)           // O(1) n*
    {
        // O(log n)
    }
}
```

... and the loop reduces to ...

```
template <class Iterator, class Compare>
void heapsort (Iterator first, Iterator last, Compare comp)
{
    make_heap (first, last, comp);
    while (last != first)           // O(1) n* = O(n * log n)
    {
        // O(log n)
    }
}
```

Now, we just need to figure out the cost of the `make_heap` call.

**Question:** What is the worst-case complexity of the `make_heap` call?

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$

*Answer:* +

`make_heap` runs in time proportional to the number of elements, which we have called  $n$ , so the complexity of that call is  $O(n)$ .

```
template <class Iterator, class Compare>
void heapsort (Iterator first, Iterator last, Compare comp)
{
    make_heap (first, last, comp); // O(n)
    O(n*log n)
}
```

The  $O(n)$  time for `make_heap` is dominated by the  $O(n \log n)$  time for the loop, so the entire algorithm is  $O(n \log n)$ .

- We can now confirm our earlier decision to treat the loop body as  $O(\log n)$  rather than actually summing up all the different values of  $O(\log(\text{last} - \text{first}))$ . Normally we have to be a little bit way of replacing a computed bound by something looser (larger). But, since no pairwise sorting algorithm can run faster than  $O(n \log n)$ , we could not possibly have obtained a tighter bound.

Heapsort has an advantage over the merge sort (which also has an  $O(n \log n)$  worst case) in that heapsort has a negligible memory overhead, while merge sort has  $O(n)$  overhead.

Heapsort has a better worst case complexity than quick sort, but experiment has shown that heapsort tends to be slower on average because it moves more elements than does quick sort.

## 4 Introspective Sort

The sorting algorithm used in most implementations of the C++ `std::sort` function is an algorithm called the [introspective sort](#).

Introspective sorts combine two algorithms we have already studied:

- Heapsort, which has a worst-case and average-case complexity of  $O(N \log(N))$ .
- Quicksort, which also has an average-case complexity of  $O(N \log(N))$ . Quicksort runs somewhat faster on average than heapsort (by a constant multiplier) but has a much slower worst-case  $O(N^2)$ .

An introspective sort starts as an ordinary quicksort, but monitors the size of the stack used to control the quicksort recursion. If the stack grows much larger than  $\log(N)$ , the sort switches over to the heapsort.

The net result is a sorting algorithm that has a worst-case and average-case complexity of  $O(N \log(N))$  but that usually runs with the fast, low-constant-multiplier of quicksort.

# Graphs --- the Basics

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

### [1 Definitions](#)

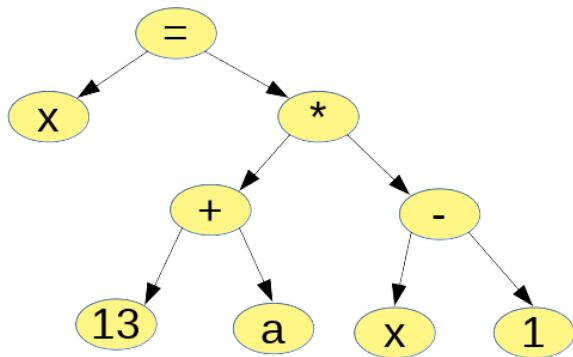
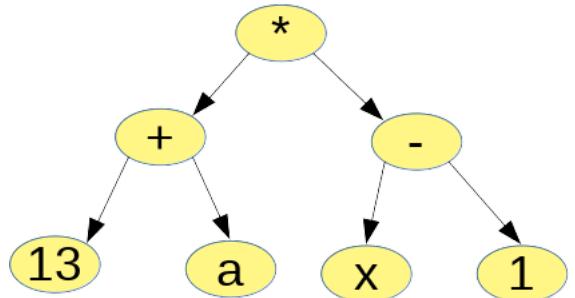
#### [1.1 Paths](#)

### [2 Data Structures for Implementing Graphs](#)

We have previously seen that compilers often represent code as trees.

For example, in this case, the expression is  $(13 + a) * (x - 1)$ .

Each non-leaf node represents the application of some operator to one or more subexpressions.



As good C++ programmers, we know that assignment and many other “built-in” parts of C++ are just more operators, so we can easily extend this idea to entire statements ...

`x = (13 + a) * (x - 1);`

This idea can be extended to other kinds of statements as well, if we’re willing to be a bit loose in our interpretation of what “operator” is.

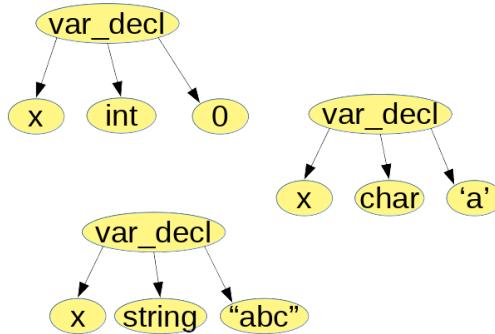
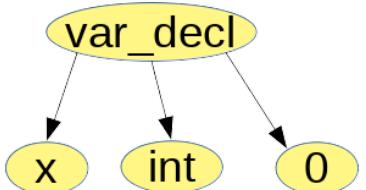
Here, for example, is a structure that might be used to represent a declaration of a variable

`int x = 0;`

Now, let’s consider what happens just a little bit later in the compilation. We have trees for expressions and statements, and trees for declarations. All of these trees are actually joined together as subtrees of larger trees representing entire functions, classes, and other larger C++ constructs.

So, we have a tree for, say, the assignment

`x = (13 + a) * (x - 1);`



Now the compiler wants to know just what “x” actually refers to. In a typical C++ program, we may have lots of objects named “x”, occurring in different functions, as data members of different classes and structs, etc. Each of these “x” objects has a unique tree representing its declaration, but which of those declarations are this assignment statement’s x’s referring to?

Well, the language has various rules allowing the compiler to resolve this question, and typically once the compiler has figured out the answer, it records that answer by adding a pointer from the uses of "x" to the appropriate declaration, as shown here.

We add pointers

- from each use of a name
- to the declaration to which it refers.

So here we have a perfectly useful data structure. But what is it?

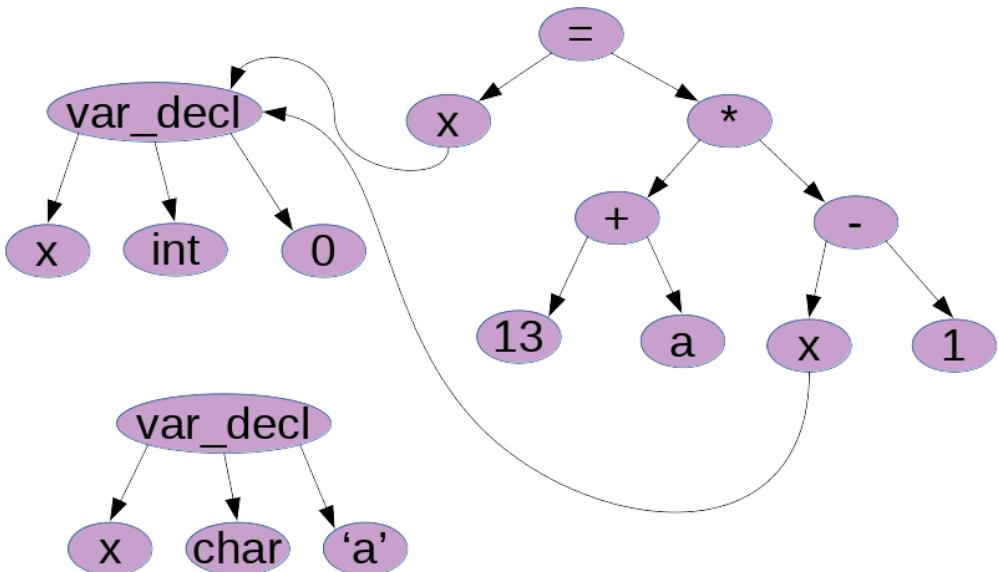
Whatever this is, it's not a tree anymore!

This structure is an example of a [graph](#).

The compiler must traverse this structure, generating code for each node of the tree. Processing graphs requires different kinds of algorithms from what we used for trees.

Obviously we don't want to generate code

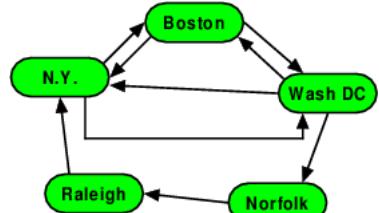
multiple times for the same nodes, but this example shows that in a graph, we can reach the same nodes multiple times using different paths. Even worse, recursion and other constructs can lead to [cycles](#) (loops) in the graph, but we still need to make sure that traversals will terminate.



As another example, consider the map representing the flights offered by a small airline.

This is also a graph. It can be used in such practical problems as

- Given travel time on each route, find the fastest way to travel between two cities.
- Find the fastest way to visit every city in the graph.



## 1 Definitions

A [graph](#)  $G = (V, E)$  consists of a set of [vertices](#) ( $V$ ) and a set of [edges](#) ( $E$ ).

- An edge is an ordered pair  $(v, w)$ , where  $v \in V$  and  $w \in V$ .
- A graph is [undirected](#) if for any vertices  $v$  and  $w$ ,  $(v, w) \in E$  iff (if and only if)  $(w, v) \in E$ .
- Graphs that are not undirected are [directed](#) graphs or [digraphs](#).
  - Both the assignment/declaration graph and the airline graph above are directed graphs.
- A node  $w$  is [adjacent](#) to  $v$  in  $G$  if there exists an edge  $(v, w) \in E$ .
  - In the assignment graph above, the '\*' vertex is adjacent to the '=' vertex. The '=' vertex is *not* adjacent to the '\*' vertex (or to any other vertices).

### 1.1 Paths

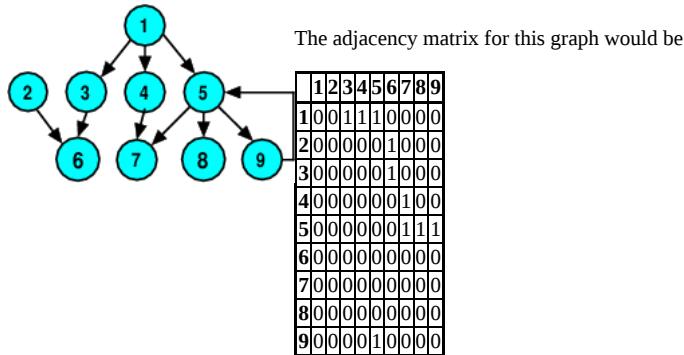
A [path](#) in  $G$  is a sequence of vertices  $[w_1, w_2, \dots, w_n]$  such that  $(w_i, w_{i+1}) \in E$  for all  $i = 1 \dots n - 1$ .

- A path is [simple](#) if no two of its vertices, except possibly the first and last, are the same.
  - In the airline graph above, [Boston, N.Y., Wash DC] is a simple path.
- A [cycle](#) is a path of length 1 or more for which  $w_1 = w_n$ .
  - In the airline graph above, [Boston, N.Y., Wash DC, Boston] is a cycle.
- A directed graph is [acyclic](#) if it contains no cycles.
  - Sometimes called a [DAG](#) for "directed acyclic graph".
  - The assignment/declaration graph above is acyclic.
- An undirected graph is [connected](#) if there is a path from each vertex to each other vertex.
  - A directed graph with this property is called [strongly connected](#).
  - The airline graph above is strongly connected. The assignment/declaration graph is not. Even if we removed the char  $x = 'a'$  declaration structure, that graph would still not be strongly connected. (For example, one cannot reach the '=' vertex from any other vertex.)

## 2 Data Structures for Implementing Graphs

A common approach is to number the vertices and keep them in an array.

An [adjacency matrix](#) indicates which vertices are connected. A 1 indicates the presence of an edge between two vertices, a zero indicates no edge.



The advantage of this structure is that we can determine adjacency in  $O(1)$  time. A minor disadvantage to this structure is the lack of any easy way to remove (or add) vertices.

A more serious problem in many applications is the  $O(|V|^2)$  storage size. This is particularly annoying when the graph is [sparse](#) (when only a small fraction of the adjacency matrix elements are 1), as happens whenever a graph has a fairly small number of edges.

It's often better to use [adjacency lists](#). For each vertex, we keep a list of vertices adjacent to it.

The adjacency list for this graph



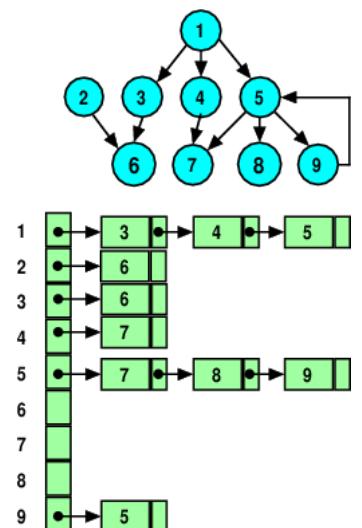
would be this:

The adjacency list

- is more flexible in terms of storage use, but
- requires  $O(|V|)$  time testing to see if  $v_1$  is adjacent to  $v_2$
- makes it easier to iterate over all vertices or all edges

In C++, you can implement an adjacency list as an array or vector of `std::list`. You can actually get something along the lines of an adjacency list by using a multimap:

```
class Node {  
};  
  
typedef std::multimap<Node, Node> Graph;  
Graph g;  
  
g.insert (Graph::value_type(node1, node3));  
g.insert (Graph::value_type(node1, node4));  
g.insert (Graph::value_type(node1, node5));  
g.insert (Graph::value_type(node2, node6));  
g.insert (Graph::value_type(node3, node6));  
⋮
```



or a combination of a map and a set:

```
class Node {  
};  
  
typedef std::map<Node, std::set<Node>> Graph;  
Graph g;  
  
g[node1].insert (node3);  
g[node1].insert (node4);  
g[node1].insert (node5);
```

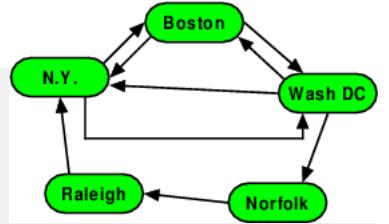
```
g[node2].insert (node6);
g[node3].insert (node6);
:
```

With these approaches, you don't necessarily have to number the nodes. You can use any identifying information that can be inserted into a map or unordered map.

For example, you could do this airline graph as easily as

```
typedef std::multimap<std::string, std::string> Graph;
typedef Graph::value_type Flight;
Graph g;

g.insert (Flight("N.Y.", "Boston"));
g.insert (Flight("Boston", "N.Y."));
g.insert (Flight("Raleigh", "N.Y."));
g.insert (Flight("N.Y.", "Wash DC"));
g.insert (Flight("Wash DC", "N.Y."));
g.insert (Flight("Boston", "Wash DC"));
g.insert (Flight("Wash DC", "Boston"));
g.insert (Flight("Wash DC", "Norfolk"));
g.insert (Flight("Norfolk", "Raleigh"));
```



# Graphs --- ADT and Traversing

Steven J. Zeil

Last modified: Jul 17, 2018

## Contents:

[1 A Graph ADT](#)  
[1.1 Vertex](#)  
[1.2 Edge](#)  
[1.3 DiGraph](#)  
[1.4 Digraph Iterators](#)  
[1.5 Graph](#)  
[2 Traversing a Graph](#)  
[2.1 Spanning Trees](#)  
[2.2 Depth-First Traversals - Trees](#)  
[2.3 Depth-First Traversals of Graphs](#)  
[2.4 Breadth-First Traversal](#)

Your text presents a variety of graph applications and algorithms, but does not present a general-purpose graph ADT. Instead, it works directly with adjacency matrices or adjacency lists.

In fact, most graphs algorithms tend to arise embedded within other data abstractions, so working directly with the data structures used to implement those application-specific ADTs may make sense.

I think, however, that the various graph algorithms are a lot easier to write and understand if we start with a graph ADT that supports the basic style of programming that we use with the C++ std library. So I present one here, not so much for practical use as to make the remaining algorithms more readable.

## 1 A Graph ADT

The ADT I have devised for graphs is designed to take advantage of iterators and the iterator-based style of programming that we have seen in dealing with the standard library. We won't worry too much about a specific implementation of this interface, but it can be implemented using either adjacency matrices or adjacency lists, depending on the performance characteristics that we want.

### 1.1 Vertex

We start with the idea of a vertex in a graph:

```
#ifndef VERTEX_H
#define VERTEX_H

class DiGraph;

class Vertex
{
private:
    const DiGraph* graph;
    unsigned vID;

    Vertex (const DiGraph* g, unsigned theID): graph(g), vID(theID) {}

    friend class DiGraph;
    friend class AllVertices;
    friend class AllEdges;
    friend class AllOutgoingEdges;
    friend class AllIncidentEdges;
    friend class AllIncomingEdges;
    friend class Edge;

public:
    // To get a non-trivial vertex, you must ask a DiGraph to generate one.
    Vertex(): graph(0), vID(0) {}

    unsigned id() const {return vID;}

    bool operator< (const Vertex& v) const
        {return vID < v.vID;}

    bool operator==(const Vertex& v) const
        {return (graph == v.graph) && (vID == v.vID);}

    bool operator!=(const Vertex& v) const
        {return (graph != v.graph) || (vID != v.vID);}

};
```

```

struct VertexHash
{
    unsigned operator() (Vertex v) const {return v.id();}
};

#endif

```

There's really not much to the `Vertex` interface. In particular, note that the only public constructors are the automatically generated

**default constructor**

which creates a vertex that isn't part of any graph, and

**copy constructor**

which copies an existing vertex.

So there is no way to get a vertex that *is* part of some graph from this class. As it happens, the only way to get a vertex within a graph is to ask the graph to create it. We will shortly see that, given a directed graph `dg`, we add a vertex to the graph this way:

```
Vertex v = dg.addVertex();
```

Once we have a vertex, there's not a whole lot we can do with it. We can compare vertices to see if they are equal (or via `a < b` if we want to place vertices in sets or maps), and we can get an integer `id()` that uniquely identifies each node within a given graph. This ID can be used to produce a perfect hash function or, because the ID's are assigned sequentially, even used as indices into an array or vector.

## 1.2 Edge

Next, we consider the idea of an edge:

```

#ifndef EDGE_H
#define EDGE_H

#include "vertex.h"

class DiGraph;

class Edge
{
private:
    const DiGraph* graph;
    unsigned eID;
    unsigned sourceNode;
    unsigned destNode;

    Edge (const DiGraph* g, unsigned theID, unsigned source, unsigned dest)
        : graph(g), eID(theID), sourceNode(source), destNode(dest)
    {}

    friend class DiGraph;
    friend class Graph;
    friend class AllEdges;
    friend class AllIncidentEdges;
    friend class AllIncomingEdges;
    friend class AllOutgoingEdges;

public:
    Edge() {}

    const Vertex source() const {return Vertex(graph, sourceNode);}
    const Vertex dest() const {return Vertex(graph, destNode);}

    unsigned id() const {return eID;}

    bool operator< (const Edge& e) const
    {return eID < e.eID;}

    bool operator== (const Edge& e) const
    {return (graph == e.graph) && (eID == e.eID);}

    bool operator!= (const Edge& e) const
    {return (graph != e.graph) || (eID != e.eID);}

};

struct EdgeHash
{
    unsigned operator() (Edge e) const {return e.id();}
};

#endif

```

Like the `Vertex` class, there's surprisingly little to the `Edge` class. To get new edges, we will ask a graph to add one:

```
Edge e = digraph.addEdge(fromVertex, toVertex);
```

and, to get an existing edge already in a graph, we ask the graph for it:

```
Edge e = digraph.getEdge(fromVertex, toVertex);
```

Once we have an edge, we can compare it to other edges or ask for its `id()`. In addition, we can get the vertex that the edge points from (the `source()`) and the vertex that it points to (`dest()`, the destination).

## 1.3 DiGraph

The `DiGraph` class serves as a container of vertices and edges.

```
class DiGraph
{
public:

    Vertex addVertex();
    void removeVertex(const Vertex& v);

    virtual Edge addEdge(const Vertex& source,
                         const Vertex& dest);
    virtual void removeEdge (const Edge& e);

    unsigned int numVertices() const;
    unsigned int numEdges() const;

    unsigned indegree (Vertex) const;
    unsigned outdegree (Vertex) const;

    virtual bool isAdjacent(const Vertex& v, const Vertex& w) const;

    // Fetch an existing edge. Returns Edge() if no edge from v to w is in
    // the graph.
    Edge getEdge(const Vertex& v, const Vertex& w) const;

    void clear();

    // iterators

    AllVertices vbegin() const;
    AllVertices vend() const;

    AllEdges ebegin() const;
    AllEdges eend() const;

    AllOutgoingEdges outbegin(Vertex source) const;
    AllOutgoingEdges outend(Vertex source) const;

    AllIncomingEdges inbegin(Vertex dest) const;
    AllIncomingEdges inend(Vertex dest) const;

protected:
    ;
};
```

With a digraph, we can

- Add vertices and edges, as discussed earlier.
- Remove vertices and edges.
- Ask how many vertices (`numVertices()`) and edges (`numEdges()`) are in the graph.
- For any vertex `v` in digraph `dg`, you can ask about the number of edges impinging on that vertex.
  - The indegree of a vertex `v` is the number of incoming edges with `v` as their destination. This is obtained as `dg.indegree(v)`.
  - The outdegree of a vertex `v` is the number of outgoing edges with `v` as their source. This is obtained as `dg.outdegree(v)`.
- Given two vertices `v` and `w`, we can ask if `v` is adjacent to `w` (if there exists an edge from `w` to `v`) via the test `dg.isAdjacent(v,w)`

The real heart of the `DiGraph` class, however, lies in the variety of iterators that it provides.

## 1.4 Digraph Iterators

The `DiGraph` class provides four different iterators:

class	starts at	ends at	iterates over
<code>AllVertices</code>	<code>dg.vbegin()</code>	<code>dg.vend()</code>	all vertices in the graph <code>dg</code>
<code>AllEdges</code>	<code>dg.ebegin()</code>	<code>dg.eend()</code>	all edges in the graph <code>dg</code>
<code>AllIncomingEdges</code>	<code>dg.inbegin(v)</code>	<code>dg.inend(v)</code>	all edges in the graph <code>dg</code> that have <code>v</code> as their destination
<code>AllOutgoingEdges</code>	<code>dg.outbegin(v)</code>	<code>dg.outend(v)</code>	all edges in the graph <code>dg</code> that have <code>v</code> as their source

For example, to visit every vertex in a graph `dg`, we would write:

```
for (AllVertices p = dg.vbegin(); p != dg.vend(); ++p)
{
    Vertex v = *p;
    doSomethingTo(v);
}
```

To visit every edge emerging from a vertex `v`, we would write:

```
for (AllOutgoingEdges p = dg.outbegin(v); p != dg.outend(v); ++p)
{
    Edge e = *p;
    doSomethingElseTo(e);
}
```

## 1.5 Graph

A Graph is a DiGraph in which all edges go both ways.

```
#ifndef GRAPH_H
#define GRAPH_H

#include "digraph.h"

class Graph: public DiGraph
{
public:
    virtual Edge addEdge(const Vertex& source,
                         const Vertex& dest);

    virtual void removeEdge(const Edge&);

};

#endif
```

Its interface is identical to that of a `DiGraph`. (In fact, it is a subclass of `DiGraph`, so any function that works on `DiGraphs` can be applied to `Graphs` as well).

But when you add an edge  $(v, w)$  to a graph, you automatically get an edge  $(w, v)$  as well.

The easiest way to accomplish this requirement is to modify the `addEdge` function to add a pair of directed edges:

```
Edge Graph::addEdge(const Vertex& source,
                     const Vertex& dest)
{
    DiGraph::addEdge(dest, source);
    return DiGraph::addEdge(source, dest);
}
```

Of course, we would need to make a similar change to `removeEdge` as well, so that when one edge is removed, its mirror image is removed as well.

## 2 Traversing a Graph

Many problems require us to visit all the vertices of a graph, or to search for vertices with some desired property.

Of course, we can use the `AllVertices` iterator to do this in some cases, but many problems also give a starting vertex for the search and call for us to return the path from the starting vertex to the one that was found. Other problems specifically are interested only in searching through those vertices that can be reached from some starting vertex.

In these circumstances, we can adapt some of the traversal techniques that we learned for trees. This will yield the “depth-first” and “breadth-first” traversals for graphs.

In fact, the easiest way to get started on these kinds of traversals is to recognize the special relation between connected graphs and trees.

Every tree is a graph (i.e., it is a set of vertices and a set of edges connecting those vertices), but not every graph is a tree. A tree is a connected graph in which each vertex is adjacent to at most one other vertex (its parent).

Not every graph is a tree, but we can sometimes find useful trees embedded within graphs.

## 2.1 Spanning Trees

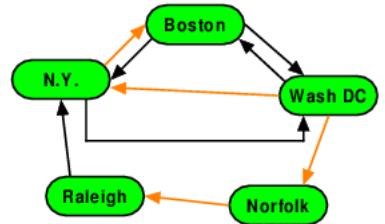
A spanning tree for a connected graph  $G=(V,E)$  is a graph  $G' = (V, E')$  such that  $E' \subseteq E$  and  $G'$  is a tree. The spanning tree is a tree that is “embedded” in the graph.

- They are useful when we need to do something with each vertex, but using as few edges as possible.

### Question:

Is the set of vertices and orange edges shown here a spanning tree for the entire graph? If so, what is its root?

- No.
- Yes. The root is Boston
- Yes. The root is N.Y.
- Yes. The root is Norfolk
- Yes. The root is Raleigh
- Yes. The root is Wash DC



\*\*Answer:\*\*

Yes, this is a spanning tree. The root is Wash DC

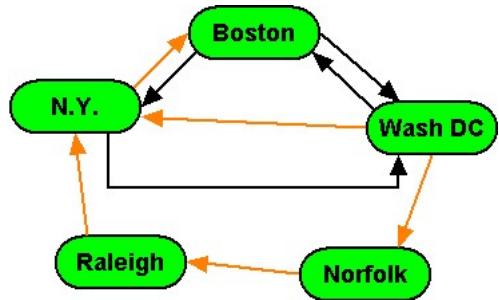
If we choose Wash. DC as the root, then the nodes and edges do form a tree.

**Question:**

Is the set of vertices and orange edges shown here a spanning tree for the entire graph? If so, what is its root?

- No.
- Yes. The root is Boston
- Yes. The root is N.Y.
- Yes. The root is Norfolk
- Yes. The root is Raleigh
- Yes. The root is Wash DC

\*\*Answer:\*\*



No, this is not a spanning tree.

This collection of highlighted edges cannot form a tree, because N.Y. has two “parents”.

For undirected graphs, any acyclic, connected subset of E is a tree.

Note that, while an undirected graph always contains the edge  $(v, w)$  iff it also contains  $(w, v)$ , the spanning tree of an undirected graph cannot contain both  $(v, w)$  and  $(w, v)$ , as those two edges alone would constitute a cycle.

## 2.2 Depth-First Traversals - Trees

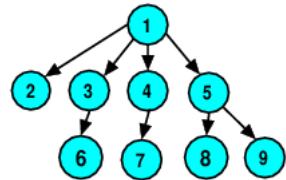
Consider the problem of searching a general *tree* for a given node.

- In a *depth-first traversal*, we investigate one child’s descendants before exploring its right siblings.
- In a *breadth-first traversal*, we explore all nodes at the same depth before moving on to any deeper nodes.

Most of the tree traversals that we looked at (prefix, postfix, and infix) were all variations of the depth-first idea.

**Question:** In what order would a depth-first tree traversal, starting from node 1, visit these nodes?

- 1 2 3 4 5 6 7 8 9
- 1 2 3 6 4 7 5 8 9
- 2 3 6 1 4 7 8 5 9
- None of the above.



\*\*Answer:\*\*

The nodes would be visited in the order: 1 2 3 6 4 7 5 8 9

## 2.3 Depth-First Traversals of Graphs

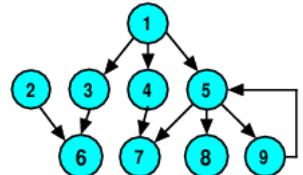
The prototypical code for depth first *tree* traversal is

```
void depthFirst (TreeNode* t)
{
    if (t != 0)
        for (int i = 0; i < t->numChildren; ++i)
            depthFirst (t->child[i]);
}
```

We convert this into a preorder or postorder process depending upon whether we process a node before or after visiting its children.

Now, if we apply this same idea to a graph instead of a tree, starting with vertex 1:

```
void depthFirst (Digraph& dg, Vertex v)
{
    for (AllOutgoingEdges e = dg.outbegin(v);
         e != dg.outend(v); ++e)
    {
        Vertex w = (*e).dest();
        depthFirst (dg, w);
    }
}
```



we can see some problems:

- We never reach vertex 2, because there is no path to it from vertex 1.
  - That's probably OK. We are presumably only interested in the vertices that are reachable from vertex 1, otherwise we wouldn't be doing a depth-first traversal in the first place. Instead we would use the AllVertices iterator.
- Vertex 7 will be visited once as a "child" of 4, and visited again when we reach vertex 5.
- Worst of all, we will eventually go from 5 to 9, from 9 back to 5, then to 9 again, and so on, recursing forever (or until we run out of memory for the activation stack).

### 2.3.1 Using Sets to Cope with Cycles

We can adapt the tree algorithm for use in graphs by using some sort of data structure to keep track of which nodes have already been visited:

```
void depthFirst (Digraph& dg, Vertex v, set<Vertex>& visited)
{
    visited.insert (v);
    for (AllOutgoingEdges e = dg.outbegin(v);
         e != dg.outend(v); ++e)
    {
        Vertex w = (*e).dest();
        if (visited.count(w) == 0)
            depthFirst (dg, w, visited);
    }
}
```

The *visited* set records the vertices that we have already seen. When we are examining adjacent vertices to recursively visit, we simply pass over any that we have already visited.

The use of the set will slow this traversal a little bit, though we know that the `std::set` operations used here are only  $O(\log |V|)$ . We could get even faster average time by using hashing (`unordered_set`). It's also possible to devise a vector-based set that would use the vertex's `id()` to index into the vector, thus achieving true  $O(1)$  time for lookups and amortized  $O(1)$  for insertions.

Try [running a depth-first search](#).

- Note that searching for a node that can't be reached from the starting point is equivalent to doing a complete depth-first traversal.

The edges that are actually followed during the traversal are highlighted in red.

#### Question:

What can you say about this set of edges at the end of a full traversal?

- They are cyclic.
- They form a spanning tree of the entire graph.
- They form a spanning tree of the portion of the graph reachable from the start

- They are not connected.

**\*\*Answer:\*\***

- They form a spanning tree of the portion of the graph reachable from the start.

In fact, the set of edges that could be collected during a depth-first traversal is called a [depth-first spanning tree](#).

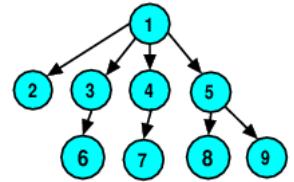
## 2.4 Breadth-First Traversal

Breadth-first visits each node at the same depth (distance from the starting node) before moving on to more distant nodes.

In trees, this is also called “Level-Order” traversal.

**Question:** In what order would a breadth-first tree traversal, starting from node 1, visit these nodes?

- 1 2 3 4 5 6 7 8 9
- 1 2 3 6 4 7 5 8 9
- 2 3 6 1 4 7 8 5 9
- None of the above.



**\*\*Answer:\*\***  +

The nodes would be visited in the order: 1 2 3 4 5 6 7 8 9

The prototypical code for breadth first tree traversal is

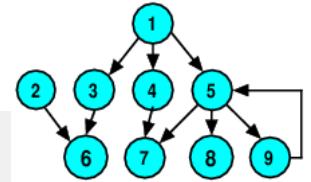
```
void breadthFirst (TreeNode* root)
{
    queue<TreeNode*, list<TreeNode*> > q;
    q.push (root);
    while (!q.empty())
    {
        v = q.front ();
        q.pop ();
        for (int i = 0; i < v->numChildren(); ++i)
        {
            TreeNode* w = v->child[i];
            if (w != 0)
                q.push (w);
        }
    }
}
```

We use a queue to receive the list of vertices to be visited, starting with the root, then the root's children, then the root's grandchildren, and so on.

Again, that tree code would have problems (including going into an infinite loop) if applied to more general graphs.

But we can use the same idea of a set of already-visited vertices to adapt this idea to traversing graphs.

```
void breadthFirstTraversal (const DiGraph& g,
                           const Vertex& start)
{
    queue<Vertex, list<Vertex> > q;
    set<Vertex, less<Vertex> > visited;
    q.push (start);
    visited.insert (start);
    while (!q.empty())
    {
        Vertex v = q.front();
        q.pop();
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e)
        {
            Vertex w = (*e).dest();
            if (visited.count(w) == 0)
            {
                q.push (w);
                visited.insert (w);
            }
        }
    }
}
```



Almost every graph algorithm is based upon either depth-first or breadth-first search.

- Depth-first is easier to program, does not require an additional ADT (the queue)
- Breadth-first (or depth-first using an explicit stack) is slightly faster

The better choice often depends upon the nature of the search and what you are trying to accomplish with your particular algorithm.

Try [running a breadth-first search](#).

Note that this also visits a spanning tree of the portion of the graph reachable from the start. This shorter, broader tree is called a [breadth-first spanning tree](#).

# Graphs --- Sample Algorithms

Steven J. Zeil

Last modified: Jun 29, 2018

## Contents:

- [1 Partial Orders and the Topological Sort](#)
  - [1.1 Partial Orders](#)
  - [1.2 Topological Sort](#)
- [2 Path-Finding](#)
  - [2.1 Shortest \(unweighted\) Path](#)
- [3 Weighted Shortest Paths](#)
  - [3.1 Dijkstra's Algorithm](#)
  - [3.2 Analysis of Dijkstra's Algorithm](#)
  - [3.3 Dijkstra's Algorithm and Standard Priority Queues](#)
- [4 Minimum Spanning Trees](#)
  - [4.1 Prim's Algorithm](#)

The area of graph-processing algorithms is more than large enough to devote an entire course to. In this section we'll look at just a few of the "classics", which will, I hope, give you a feel for some of the common patterns shared by most graph processing.

## 1 Partial Orders and the Topological Sort

Sometimes we need to arrange things according to a *partial order*: a transitive ordering relation in which for any two elements it is possible that

- $a < b$
- $a = b$
- $a > b$
- $a$  is incomparable to  $b$

This last possibility distinguishes partial order operations from the more familiar *total order* operations, such as  $<$  on integers, that are guaranteed to use only the first three of the above four options.

The possibility that some pairs of elements may be incomparable to one another makes sorting via a partial order very different from conventional sorting.

### 1.1 Partial Orders

#### 1.1.1 Example: Course Pre-requisites

Consider the relation among college courses defined as:

- $a < b$  if course  $a$  is listed as a prerequisite for  $b$
- $a == b$  if  $a$  and  $b$  are the same course

For example, in the ODU CS dept., we have

```
cs150<cs250, cs170<cs270, cs150<cs381, cs250<cs361, cs361<cs350, cs250<cs355, cs381<cs355, cs381<cs390, cs270<cs471, cs361<cs471
```

We can represent this as the graph shown here. Each directed edge goes from a prerequisite to a course dependent upon it.

Now, the prerequisite relation holds only between selected pairs of courses.  $cs150$  is *not* a prerequisite for  $cs361$ , but the collection of prerequisites makes it clear that one must take  $cs150$  before taking  $cs361$ .

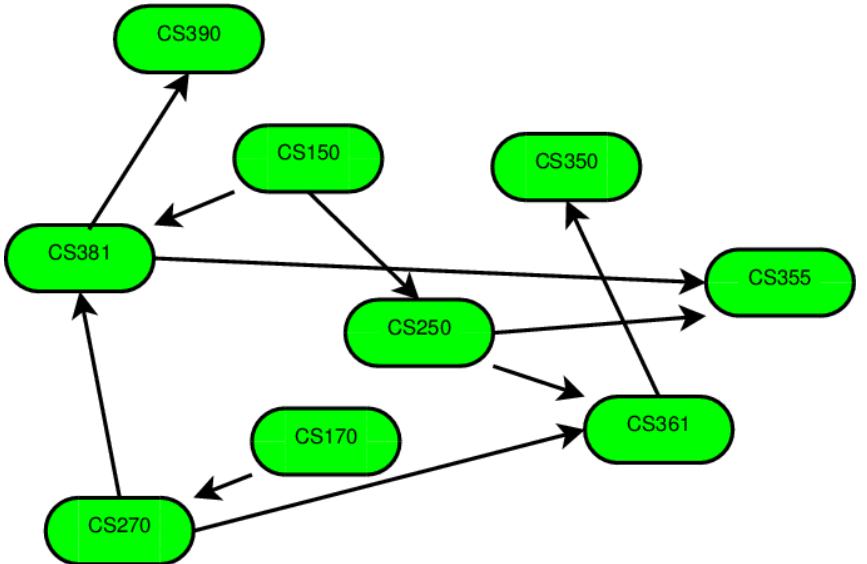
We can define a new relationship "must-be-taken-before" (which I'll symbolize as " $\prec$ ") by starting with the prerequisite relationship, and then insisting that:

$$a \prec c \text{ if } a \prec b \wedge b \prec c.$$

You may recognize the above rule as a statement that  $\prec$  is transitive. This transitive rule adds new, implied, orderings to the "must-be-taken-before" relation. Whenever we have a rule for adding items to a set (based on the items already in there) and we apply it over and over again until we can get no more new items, we call that taking the *closure* of the rule on that set. So "must-be-taken-before" is called the *transitive closure* of the prerequisite relation.

We can interpret the transitive closure of a partial order in terms of the graph of that partial order. A pair of vertices  $v$  and  $w$  are related by  $v <_{\text{closed}} w$  in the transitive closure if there is a path from  $v$  to  $w$  in the graph of the original  $<$  relation.

Transitive closures of orderings are often quite useful. For example, the transitive closure of the prerequisite order can be used to determine if a student's planned sequence for taking CS courses is "legal". If a student plans to take course a before taking b, this is legal only if !( $b \prec$ closed  $a$ ).



### 1.1.2 Example: Spreadsheet Formula Dependencies

As another example, consider the set of formulas that could be entered into a spreadsheet (e.g., Microsoft's Excel or OpenOffice's Calc). Spreadsheets store formulas in "cells". Each cell is identified by its column (using letters) and row (using numbers).

```

a1 = 10
a2 = 20
a3 = a1 + a2
b1 = 2*a1
b2 = 2*a2
b3 = b1 + b2 + c1
c1 = a3 / a1 / a2
  
```

A practical problem faced by all spreadsheet implementors is what order to process the formulas in. If, for example, we evaluate  $b_3$  before  $c_1$  has been evaluated, we can't expect to get meaningful results.

Define a partial order as follows:

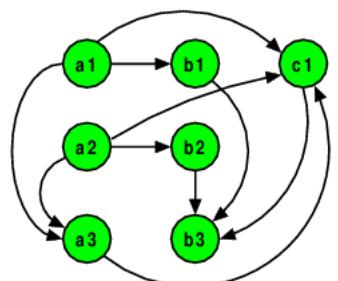
Let  $x$  and  $y$  be any two formulas

- $x \lessdot y$  if the left-hand side variable of  $x$  appears on the right-hand side of  $y$
- $x == y$  if they are the same formula

```

a1 = 10
a2 = 20
a3 = a1 + a2
b1 = 2*a1
b2 = 2*a2
b3 = b1 + b2 + c1
c1 = a3 / a1 / a2
  
```

The graph here captures that partial order. Again, the transitive closure of this order is interesting, as it defines a "must-be-evaluated-before" relation.



## 1.2 Topological Sort

A [topological sort](#) is an ordered list of the vertices in a directed acyclic graph such that, if there is a path from  $v$  to  $w$  in the graph, then  $v$  appears before  $w$  in the list.

A topological sort of the course prerequisite graph would be a possible sequence in which a student might take classes. A topological sort of the spreadsheet graph would be an order in which the formulas could be evaluated.

### 1.2.1 The Algorithm

Define the [indegree](#) of a vertex as the number of edges pointing to it.

- A node of indegree 0 can be placed at the start of the sorted order, since there is clearly nothing that must precede it.
- There must be at least one vertex of indegree 0.

- If not, the graph has a cycle and no topological sort exists.
- Once we have placed all nodes of indegree 0 in the list, we can then add all nodes whose indegree would be zero except for edges from the nodes already placed.

Repeating this process yields a topological sort.

Here is the code for a topological sort.

```
// A topological sort of a directed graph is any listing of the vertices
// in g such that v1 precedes v2 in the listing only if there exists no
// path from v2 to v1.
//
// The following routine attempts a topological sort of g. If the sort is
// successful, the return value is true and the ordered listing of
// vertices is placed in sorted. If no topological sort is possible
// (because the graph contains a cycle), false is returned and sorted will
// be empty.
//
bool topologicalSort (const DiGraph& g, list<Vertex>& sorted)
{
    // Step 1: get the indegrees of all vertices. Place vertices with
    // indegree 0 into a queue.
    hash_map<Vertex, unsigned, VertexHash> inDegree;
    queue<Vertex, list<Vertex> > q;
    for (AllVertices v = g.vbegin(); v != g.vend(); ++v)
    {
        unsigned indeg = g.indegree(*v);
        inDegree[*v] = indeg;
        if (indeg == 0)
            q.push(*v);
    }

    // Step 2. Take vertices from the q, one at a time, and add to sorted.
    // As we do, pretend that we have deleted these vertices from the graph,
    // decreasing the indegree of all adjacent nodes. If any nodes attain an
    // indegree of 0 because of this, add them to the queue.
    while (!q.empty())
    {
        Vertex v = q.front();
        q.pop();

        sorted.push_back(v);

        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e)
        {
            Vertex adjacent = (*e).dest();
            inDegree[adjacent] = inDegree[adjacent] - 1;
            if (inDegree[adjacent] == 0)
                q.push(adjacent);
        }
    }

    // Step 3: Did we finish the entire graph?
    if (sorted.size() == g.numVertices())
        return true;
    else
    {
        sorted.clear();
        return false;
    }
}
```

Let's consider it a step at a time.

```
bool topologicalSort (const DiGraph& g, list<Vertex>& sorted)
{
    // Step 1: get the indegrees of all vertices. Place vertices with
    // indegree 0 into a queue.
    unordered_map<Vertex, unsigned, VertexHash> inDegree;
    queue<Vertex, list<Vertex> > q;
    for (AllVertices v = g.vbegin(); v != g.vend(); ++v)
    {
        unsigned indeg = g.indegree(*v);
        inDegree[*v] = indeg;
        if (indeg == 0)
            q.push(*v);
    }
```

In step 1, we get the indegrees of all the vertices, putting them into a map whose key type is `Vertex` and whose associated data is `unsigned`. I've chosen to use an `unordered_map` for this code. Alternatively, I could (and have, in the past) implemented a vector-based implementation of the map interface that simply uses the `Vertex::id()` function to index into the vector, which would give me true O(1) access time.

As we do this, we also add any vertices whose indegree is zero into a queue, `q`.

```
// Step 2. Take vertices from the q, one at a time, and add to sorted.
// As we do, pretend that we have deleted these vertices from the graph,
// decreasing the indegree of all adjacent nodes. If any nodes attain an
```

```

// indegree of 0 because of this, add them to the queue.
while (!q.empty())
{
    Vertex v = q.front();
    q.pop();

    sorted.push_back(v);

    for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e)
    {
        Vertex adjacent = (*e).dest();
        inDegree[adjacent] = inDegree[adjacent] - 1;
        if (inDegree[adjacent] == 0)
            q.push(adjacent);
    }
}

```

In step 2, we repeatedly remove vertices from the queue and add them to the sorted list output, `sorted`. We can do this because we know that there is nothing in the graph that needs to come before these vertices.

We then look at the outgoing edges of each vertex, and reduce the `inDegree` values of the neighboring vertices to simulate having removed  $v$  from the graph. If doing this causes any of their (simulated) indegrees to become zero, we add them to the queue, because we know that there is nothing *remaining* in the graph that needs to come before these vertices.

```

// Step 3: Did we finish the entire graph?
if (sorted.size() == g.numVertices())
    return true;
else
{
    sorted.clear();
    return false;
}

```

Finally, in step 3, we check to see if all the vertices have been “removed” from the graph and placed into the sorted list. If so, we have successfully found a topological sort. If not, then no topological sort is possible (the graph must have a cycle).

Try [running a topological sort](#).

## 1.2.2 Analysis

In analyzing this algorithm, we will assume that the graph is implementing using adjacency lists, and that the `inDegree` map is implemented using a vector-like structure.

```

bool topologicalSort (const DiGraph& g, list<Vertex>& sorted)
{
    // Step 1: get the indegrees of all vertices. Place vertices with
    // indegree 0 into a queue.
    hash_map<Vertex, unsigned, VertexHash> inDegree;
    queue<Vertex, list<Vertex> > q;
    for (AllVertices v = g.vbegin(); v != g.vend(); ++v)
    {
        unsigned indeg = g.indegree(*v); // O(1)
        inDegree[*v] = indeg; // O(1)
        if (indeg == 0) // cond: O(1) total: O(1)
            q.push(*v); // O(1)
    }
    :

```

In step 1, therefore, we see that everything in the loop body is  $O(1)$ . The loop itself goes around once for every vertex in the graph. Following our definition that says that a graph  $G = (V, E)$  is a set  $V$  of vertices and a set  $E$  of edges, we can say that the number of iterations of this loop is  $|V|$ , the number of vertices.

```

bool topologicalSort (const DiGraph& g, list<Vertex>& sorted)
{
    // Step 1: get the indegrees of all vertices. Place vertices with
    // indegree 0 into a queue.
    hash_map<Vertex, unsigned, VertexHash> inDegree;
    queue<Vertex, list<Vertex> > q;
    for (AllVertices v = g.vbegin(); v != g.vend(); ++v) // cond: O(1) #: |V|*
    {
        unsigned indeg = g.indegree(*v); // O(1)
        inDegree[*v] = indeg; // O(1)
        if (indeg == 0) // cond: O(1) total: O(1)
            q.push(*v); // O(1)
    }
    :

```

We therefore conclude that the entire step 1 loop is  $O(|V|)$ .

(We could also write this as  $O(g.numVertices())$ , but  $|V|$  is shorter and is the usual way that people describe graphs.)

```

bool topologicalSort (const DiGraph& g, list<Vertex>& sorted)
{
    O(|V|)

```

```

// Step 2. Take vertices from the q, one at a time, and add to sorted.
// As we do, pretend that we have deleted these vertices from the graph,
// decreasing the indegree of all adjacent nodes. If any nodes attain an
// indegree of 0 because of this, add them to the queue.
while (!q.empty())
{
    Vertex v = q.front();
    q.pop();

    sorted.push_back(v);

    for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e)
    {
        Vertex adjacent = (*e).dest();
        inDegree[adjacent] = inDegree[adjacent] - 1;
        if (inDegree[adjacent] == 0)
            q.push(adjacent);
    }
}
:

```

Looking at the inner loop of step 2, we see that everything in the body is O(1).

```

bool topologicalSort (const DiGraph& g, list<Vertex>& sorted)
{
    O(|V|)

    // Step 2. Take vertices from the q, one at a time, and add to sorted.
    // As we do, pretend that we have deleted these vertices from the graph,
    // decreasing the indegree of all adjacent nodes. If any nodes attain an
    // indegree of 0 because of this, add them to the queue.
    while (!q.empty())
    {
        Vertex v = q.front(); // O(1)
        q.pop(); // O(1)

        sorted.push_back(v); // O(1)

        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) // cond: O(1)
        {
            O(1)
        }
    }
}
:
```

And the simple statements in the outer loop (everything except for the inner loop) are O(1).

```

bool topologicalSort (const DiGraph& g, list<Vertex>& sorted)
{
    O(|V|)

    while (!q.empty())
    {
        O(1)
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) // cond: O(1)
        {
            O(1)
        }
    }
}
:
```

Now, at this point our normal copy-and-paste approach breaks down. The number of iterations of the inner loop may be different for each vertex visited by the outer loop.

But, let's just stop and think about what's going on here.

- Each vertex goes into the queue at most once
- So, in a successful sort, the outer loop will execute  $|V|$  times, once for each vertex.
- The inner loop simply visits the edges emanating from the vertex being visited by the outer loop.
- So if the outer loop visits every vertex, and the inner one visits every edge leaving that vertex, over the course of all the outer loop iterations, the inner loop will visit every edge in the graph exactly once.

So the statements in the body of the inner loop get executed  $|E|$  times; the other statements in the outer loop get visited  $|V|$  times. Since all of these statements are O(1), the total cost is  $O(|V| + |E|)$ .

```

bool topologicalSort (const DiGraph& g, list<Vertex>& sorted)
{
    O(|V|)
    O(|V| + |E|)

    // Step 3: Did we finish the entire graph?
    if (sorted.size() == g.numVertices())
        return true; // O(1)
}
:
```

```

    else
    {
        sorted.clear(); // O(|V|)
        return false; // O(1)
    }
}

```

In step 3, the only non-trivial operations is clearing the sorted list (done when we can't find a solution). Since this list is actually a list of vertices that we have successfully sorted, it contains at most  $|V|$  elements, and so the clear operation is  $O(|V|)$ .

That makes the worst case for the step 3 “if” statement  $O(|V|)$  as well.

```

bool topologicalSort (const DiGraph& g, list<Vertex>& sorted)
{
    O(|V|)
    O(|V| + |E|)
    O(|V|)
}

```

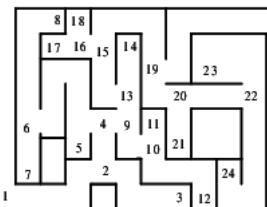
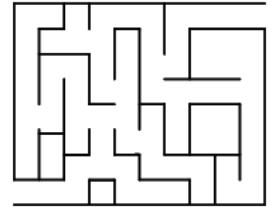
So the total cost of the topological sort is  $O(|V| + |E|)$ .

(Note: this does not mean that topological sorts are faster than conventional sorts — the number of edges can be as high as  $|V|^2$ , so this is actually more comparable to the slowest of our conventional sorting algorithms. That's the penalty we pay for working with partial orders. We don't write the cost of this algorithm as  $O(|V|^2)$ , though, because the number of edges varies widely in practical problems, and we may sometimes know that  $|E|$  will be far less than that maximum, so  $O(|V| + |E|)$  is a more accurate portrayal of the behavior.

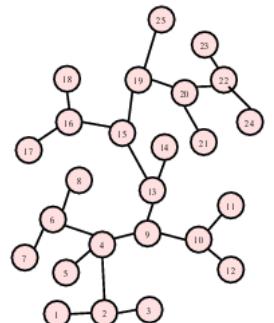
## 2 Path-Finding

Example: Given a maze, find the shortest path from the start to the finish.

To see how this relates to graphs, try numbering each intersection and dead-end in the graph.



Now we can form a graph by assigning a vertex to each numbered location, and connecting two vertices with an edge if it is possible to go directly from one vertex's position to the other without passing through another numbered position.



This graph captures the notion of “adjacent” numbered locations.

And the problem of finding the shortest path through the maze has now been reduced to (changed into) the problem of finding the shortest path between the starting and ending vertex of the graph.

Now, there's lots of ways to find a path through a maze. There's the venerable “keep your right hand on the wall” technique.

- This would take us through positions 1, 2, 3, 2, 4, 9, 10, 12, 10, 11, 10, 9, 13, 14, 13, ...
- This technique often finds a path, but not necessarily the shortest.
- And it's only guaranteed to find a path if the graph (maze) is acyclic (or if, as is the case here, both the entrance and exit are on the outer edge of the maze).

A variant is the “keep your right hand on the wall and trail a string behind you” technique (first recorded in the old Greek tale of Theseus and the Minotaur in the great labyrinth of Crete).

- This avoids problems with cycles, because if you ever come across your own string, you turn around and start back, rewinding the string until you come to a place where you can make a new right turn.
- Still may not find the shortest path

### 2.1 Shortest (unweighted) Path

A better solution is based on the following idea:

- the start vertex is 0 steps from the start.

- Given a list of vertices at distance  $k$  steps from the start, build a list of vertices that are  $k+1$  steps from the start.
- Repeat until the finish vertex is found.

Now let's render this idea into code.

### 2.1.1 The Algorithm

Here is the entire algorithm.

```

typedef unordered_map<Vertex, unsigned, VertexHash> VDistMap;

void findShortestPath (
    DiGraph& g,
    list<Vertex>& path,
    Vertex start,
    Vertex finish)
{
    hash_map<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;

    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi)
    {
        dist[*vi] = g.numVertices();
    }
    dist[start] = 0;
    queue<Vertex, dequeue<Vertex> > q;
    q.push (start);

    // From each vertex in queue, update distances of adjacent vertices
    while (!q.empty() && (dist[finish] == g.numVertices()))
    {
        Vertex v = q.front();
        int d = dist[v];
        q.pop();
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e)
        {
            Vertex w = (*e).dest();
            if (dist[w] > d + 1)
            {
                dist[w] = d + 1;
                q.push (w);
                cameFrom[w] = v;
            }
        }
    }
    // Extract path
    if (dist[finish] != g.numVertices())
    {
        Vertex v = finish;
        while (!(v == start))
        {
            path.push_front(v);
            v = cameFrom[v];
        }
        path.push_front(start);
    }
}

```

Again, let's take it apart, one piece at a time.

```

void findShortestPath (
    DiGraph& g,
    list<Vertex>& path,
    Vertex start,
    Vertex finish)
{
    unordered_map<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;

    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi)
    {
        dist[*vi] = g.numVertices();
    }
    dist[start] = 0;
    queue<Vertex, dequeue<Vertex> > q;
    q.push (start);
}

```

The first loop simply assigns to each vertex a distance value that is larger than any legitimate path could be. The exception is that the start vertex is given a distance of 0.

Then we create a queue, placing the start vertex into it.

```

// From each vertex in queue, update distances of adjacent vertices
while (!q.empty() && (dist[finish] == g.numVertices()))
{
    Vertex v = q.front();
}

```

```

int d = dist[v];
q.pop();
for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e)
{
    Vertex w = (*e).dest();
    if (dist[w] > d + 1)
    {
        dist[w] = d + 1;
        q.push(w);
        cameFrom[w] = v;
    }
}
}

```

The main loop removes vertices, one at a time, from the queue. The shortest known distance to that node is saved in `d`. Then each adjacent vertex is examined. If its distance is bigger than `d+1`, we now know that we can get there “faster” because the distance to the current vertex is `d`, and this adjacent vertex is only one step away from that. So we set its distance to `d+1` and place it on the queue, so that eventually we will examine all the vertices one step away from it.

We keep this up until we have computed a new, shortest distance for the `finish` vertex or until the queue is empty (in which case we conclude that there’s no way to get from the start vertex to the finish).

```

// Extract path
if (dist[finish] != g.numVertices())
{
    Vertex v = finish;
    while (!(v == start))
    {
        path.push_front(v);
        v = cameFrom[v];
    }
    path.push_front(start);
}

```

The final loop traces back from the `finish` by using the information we recorded about how we first reached each vertex.

## 2.1.2 Analysis

Again, I will assume an adjacency list implementation of the graph, and a vector-based implementation of the maps. With this in mind, let’s just start by labeling everything that is obviously  $O(1)$ .

```

void findShortestPath (
    DiGraph& g,
    list<Vertex>& path,
    Vertex start,
    Vertex finish)
{
    unordered_map<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;

    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi) // cond: O(1) #: |V|*
    {
        dist[*vi] = g.numVertices(); // O(1)
    }
    dist[start] = 0; // O(1)
    queue<Vertex, deque<Vertex> > q; // O(1)
    q.push(start); // O(1)

    // From each vertex in queue, update distances of adjacent vertices
    while (!q.empty() && (dist[finish] == g.numVertices())) // cond: O(1)
    {
        Vertex v = q.front(); // O(1)
        int d = dist[v]; // O(1)
        q.pop(); // O(1)
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) // cond: O(1)
        {
            Vertex w = (*e).dest(); // O(1)
            if (dist[w] > d + 1) // O(1)
            {
                dist[w] = d + 1; // O(1)
                q.push(w); // O(1)
                cameFrom[w] = v; // O(1)
            }
        }
    }
    // Extract path
    if (dist[finish] != g.numVertices()) // cond: O(1)
    {
        Vertex v = finish; // O(1)
        while (!(v == start)) // cond: O(1)
        {
            path.push_front(v); // O(1)
            v = cameFrom[v]; // O(1)
        }
        path.push_front(start); // O(1)
    }
}

```

Then, looking at the first loop, we can see that it clearly goes around once for each vertex.

So the entire loop is  $|V| * O(1) = O(|V|)$

Looking at the innermost remaining loop, we can see that its body is  $O(1)$ , as are the other statements of the outer loop body that contains it.

```

void findShortestPath (
    DiGraph& g,
    list<Vertex>& path,
    Vertex start,
    Vertex finish)
{
    unordered_map<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;

    O(|V|)
    O(1)

    // From each vertex in queue, update distances of adjacent vertices
    while (!q.empty() && (dist[finish] == g.numVertices())) // cond: O(1)
    {
        Vertex v = q.front();           // O(1)
        int d = dist[v];              // O(1)
        q.pop();                      // O(1)
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) // cond: O(1)
        {
            Vertex w = (*e).dest();   // O(1)
            if (dist[w] > d + 1)     // cond: O(1)
            {
                dist[w] = d + 1;      // O(1)
                q.push(w);            // O(1)
                cameFrom[w] = v;      // O(1)
            }
        }
    }
    // Extract path
    :
}

void findShortestPath (
    DiGraph& g,
    list<Vertex>& path,
    Vertex start,
    Vertex finish)
{
    unordered_map<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;

    O(|V|)
    O(1)

    // From each vertex in queue, update distances of adjacent vertices
    while (!q.empty() && (dist[finish] == g.numVertices())) // cond: O(1)
    {
        O(1)
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) // cond: O(1)
        {
            O(1)
        }
    }
    // Extract path
    if (dist[finish] != g.numVertices()) // cond: O(1)
    {
        Vertex v = finish;           // O(1)
        while (!(v == start))       // cond: O(1)
        {
            path.push_front(v);     // O(1)
            v = cameFrom[v];        // O(1)
        }
        path.push_front(start);     // O(1)
    }
}

```

We saw this same pattern in the last analysis. The inner loop eventually visits every edge in the graph once, and so its statements are executed a total  $|E|$  times. The remaining statements in the outer loop are executed once per vertex, and so are performed  $|V|$  times.

The entire cost of the two nested loops is therefore  $O(|V| + |E|)$ .

The final loop goes around once for every vertex in the shortest path that we have found. It is possible that the shortest path will include every vertex (think of a linked list-like arrangement), so this loop may execute as many as  $|V|$  times.

```

void findShortestPath (
    DiGraph& g,
    list<Vertex>& path,
    Vertex start,
    Vertex finish)
{
    unordered_map<Vertex, Vertex, VertexHash> cameFrom;

```

```

VDistMap dist;

O(|V|)
O(1)
O(|V|+|E|)
// Extract path
if (dist[finish] != g.numVertices()) // cond: O(1)
{
    Vertex v = finish;           // O(1)
    while (!(v == start))      // cond: O(1) #: |V|* total: O(|V|)
    {
        path.push_front(v);    // O(1)
        v = cameFrom[v];       // O(1)
    }
    path.push_front(start);    // O(1)
}
}

```

So the entire loop is  $O(|V|)$ .

And the entire if statement is  $O(|V|)$ .

```

void findShortestPath (
    DiGraph& g,
    list<Vertex>& path,
    Vertex start,
    Vertex finish)
{
    unordered_map<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;

    O(|V|)
    O(1)
    O(|V|+|E|)
    O(|V|)
}
}

```

And the entire algorithm is  $O(|V| + |E|)$ .

## 3 Weighted Shortest Paths

Now, let's consider a more general form of the same problem. Attach to each edge a weight indicating the cost of traversing that edge.

Find a path between designated start and finish nodes that minimizes the sum of the weights of the edges in the path.

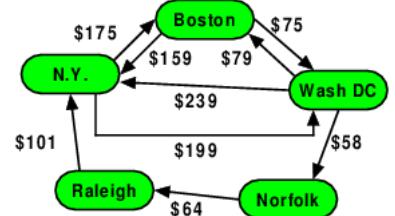
Example: finding the cheapest airline route:

What's the cheapest way to get from D.C. to N.Y?

To deal with this problem, we adapt the unweighted shortest path algorithm.

The algorithm was:

- Keep nodes in a collection (queue).
- From the collection, we extracted a node closest to the start.
- From that node we considered the smallest possible step (always 1), updating the distances of the adjacent nodes accordingly.



### 3.1 Dijkstra's Algorithm

With weighted graphs, we do the same, but the step size is determined by the weights. We use a priority queue to keep track of the nodes closest to the start.

We will use a map (VDistMap) to associate with each vertex the shortest distance (cost, in the airline example) known so far from the start to that vertex. This value starts impossibly high, so that any path we find to that vertex will look good by comparison.

The CompareVerticesByDistance structure provides a comparison operator based upon this shortest-distance-known-so-far. We'll use that operator to maintain a priority queue of the vertices based upon the distance.

```

#include "digraph.h"
#include <unordered_map>

#include <limits.h>

typedef unordered_map<Vertex, unsigned, VertexHash> VDistMap;

struct CompareVerticesByDistance
{

```

```

VDistMap& dist;

CompareVerticesByDistance (VDistMap& dm): dist(dm) {}

bool operator() (Vertex left, Vertex right) const
{
    return dist[left] > dist[right];
}

};

void findWeightedShortestPath (
    DiGraph& g,
    list<Vertex>& path,
    Vertex start,
    Vertex finish,
    unordered_map<Edge, int, EdgeHash>& weight)
{   // Dijkstra's Algorithm
    :
}

```

The edge weights are passed to our algorithm as another map, this one keyed on edges instead of vertices.

The algorithm we will develop is called Dijkstra's algorithm, named for its inventor.

We begin by initializing the distance map and the priority queue.

```

void findWeightedShortestPath (
    DiGraph& g,
    list<Vertex>& path,
    Vertex start,
    Vertex finish,
    unordered_map<Edge, int, EdgeHash>& weight)
{   // Dijkstra's Algorithm
    unordered_map<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;

    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi)
    {
        dist[*vi] = INT_MAX;
    }
    dist[start] = 0;
    adjustable_priority_queue<Vertex, VDistMap, CompareVerticesByDistance>
    pq (g.vbegin(), g.vend(), CompareVerticesByDistance(dist));
    :
}

```

Here is the main part of the algorithm.

```

:
while (!pq.empty())
{
    Vertex v = pq.top(); ①
    int d = dist[v];
    pq.pop();
    for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) ②
    {
        Vertex w = (*e).dest();
        if (dist[w] > d+weight[*e]) ③
        {
            dist[w] = d + weight[*e]; ④
            pq.adjust (w);
            cameFrom[w] = v;
        }
    }
}
:

```

- ①In the heart of the algorithm, we repeatedly remove from the priority queue the vertex,  $v$ , closest to the start. We look in the distance map to get that vertex's minimum distance from the start,  $d$ .
- ②

Then we look at each outgoing edge of  $v$ .

If  $v$  is distance  $d$  from the start, and there is an edge  $*e$  from  $v$  to  $w$  with weight  $weight[*e]$ , then we know we can reach  $w$  in distance  $d+weight[*e]$ .

- ③Now, it's possible that we already know a shorter way to get to  $w$  than that. So we compare the value  $d+weight[*e]$  to the shortest known distance already recorded for  $w$ . If we already know a shorter way to get to  $w$ , we leave it alone.
- ④

If, however, this new distance is shorter, we update the shortest known distance for  $w$ , adjust its position in the priority queue to reflect the change, and record that the shortest known way to reach  $w$  is via  $v$ .

This continues until we have emptied out the priority queue.

The final portion of the algorithm is the same as in the unweighted case.

```
:  
// Extract path  
Vertex v = finish;  
if (dist[v] != INT_MAX)  
{  
    while (!(v == start))  
    {  
        path.push_front(v);  
        v = cameFrom[v];  
    }  
    path.push_front(start);  
}  
}
```

We simply walk backwards along the path using our “how did we get here” information recorded in `cameFrom`. Try [running Dijkstra’s algorithm](#) and see how it works.

## 3.2 Analysis of Dijkstra’s Algorithm

Structurally, Dijkstra’s algorithm is similar to the unweighted shortest path algorithm, so we might expect that the analysis will also be similar. The introduction of the priority queue, however, makes some of the steps more expensive than in the unweighted case.

Again, we will start by marking the obvious O(1) parts.

```
void findWeightedShortestPath (  
    DiGraph& g,  
    list<Vertex>& path,  
    Vertex start,  
    Vertex finish,  
    unordered_map<Edge, int, EdgeHash>& weight)  
{  
    // Dijkstra's Algorithm  
    unordered_map<Vertex, Vertex, VertexHash> cameFrom;  
    VDistMap dist;  
  
    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi) // cond: O(1)  
    {  
        dist[*vi] = INT_MAX; // O(1)  
    }  
    dist[start] = 0; // O(1)  
    adjustable_priority_queue<Vertex, VDistMap, CompareVerticesByDistance>  
    pq (g.vbegin(), g.vend(), CompareVerticesByDistance(dist));  
  
    // Compute distance from start to finish  
    while (!pq.empty()) // cond: O(1)  
    {  
        Vertex v = pq.top(); // O(1)  
        int d = dist[v]; // O(1)  
        pq.pop();  
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) // O(1)  
        {  
            Vertex w = (*e).dest(); // O(1)  
            if (dist[w] > d+weight[*e]) // cond: O(1)  
            {  
                dist[w] = d + weight[*e]; // O(1)  
                pq.adjust (w);  
                cameFrom[w] = v; // O(1)  
            }  
        }  
    }  
    // Extract path  
    Vertex v = finish; // O(1)  
    if (dist[v] != INT_MAX) // cond: O(1)  
    {  
        while (!(v == start)) // cond: O(1)  
        {  
            path.push_front(v); // O(1)  
            v = cameFrom[v]; // O(1)  
        }  
        path.push_front(start); // O(1)  
    }  
}
```

Now, we might as well acknowledge that we’ve seen that final loop before, and we know it’s  $O(|V|)$ . The first loop is also obviously  $O(|V|)$ .

```
void findWeightedShortestPath (  
    DiGraph& g,  
    list<Vertex>& path,  
    Vertex start,  
    Vertex finish,  
    unordered_map<Edge, int, EdgeHash>& weight)  
{  
    // Dijkstra's Algorithm  
    unordered_map<Vertex, Vertex, VertexHash> cameFrom;
```

```

VDistMap dist;
O(|V|)
dist[start] = 0;           // O(1)
adjustable_priority_queue<Vertex, VDistMap, CompareVerticesByDistance>
pq (g.vbegin(), g.vend(), CompareVerticesByDistance(dist));

// Compute distance from start to finish
while (!pq.empty())        // cond: O(1)
{
    Vertex v = pq.top();    // O(1)
    int d = dist[v];        // O(1)
    pq.pop();
    for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) // cond: O(1)
    {
        Vertex w = (*e).dest();          // O(1)
        if (dist[w] > d+weight[*e])    // cond: O(1)
        {
            dist[w] = d + weight[*e]; // O(1)
            pq.adjust (w);
            cameFrom[w] = v;         // O(1)
        }
    }
}
O(|V|)
}

```

The declaration of the priority queue initializes the queue from an existing sequence. That sequence has  $|V|$  items, so this is  $O(|V|)$  (Priority queues can be initialized from existing sequences in linear time).

```

void findWeightedShortestPath (
    DiGraph& g,
    list<Vertex>& path,
    Vertex start,
    Vertex finish,
    unordered_map<Edge, int, EdgeHash>& weight)
{   // Dijkstra's Algorithm
    unordered_map<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;
    O(|V|)
    dist[start] = 0;           // O(1)
    adjustable_priority_queue<Vertex, VDistMap, CompareVerticesByDistance>
    pq (g.vbegin(), g.vend(), CompareVerticesByDistance(dist)); // O(|V|)

// Compute distance from start to finish
while (!pq.empty())        // cond: O(1)
{
    Vertex v = pq.top();    // O(1)
    int d = dist[v];        // O(1)
    pq.pop();
    for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) // cond: O(1)
    {
        Vertex w = (*e).dest();          // O(1)
        if (dist[w] > d+weight[*e])    // cond: O(1)
        {
            dist[w] = d + weight[*e]; // O(1)
            pq.adjust (w);
            cameFrom[w] = v;         // O(1)
        }
    }
}
O(|V|)
}

```

We can summarize all the pre-loop stuff as  $O(|V|)$ .

```

void findWeightedShortestPath (
    DiGraph& g,
    list<Vertex>& path,
    Vertex start,
    Vertex finish,
    unordered_map<Edge, int, EdgeHash>& weight)
{   // Dijkstra's Algorithm
    unordered_map<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;
    O(|V|)
    dist[start] = 0;           // O(1)
    adjustable_priority_queue<Vertex, VDistMap, CompareVerticesByDistance>
    pq (g.vbegin(), g.vend(), CompareVerticesByDistance(dist)); // O(|V|)

// Compute distance from start to finish
while (!pq.empty())        // cond: O(1)
{
    Vertex v = pq.top();    // O(1)
    int d = dist[v];        // O(1)
    pq.pop();
    for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) // cond: O(1)
    {
        Vertex w = (*e).dest();          // O(1)
        if (dist[w] > d+weight[*e])    // cond: O(1)
        {

```

```

        {
            dist[w] = d + weight[*e]; // O(1)
            pq.adjust (w);
            cameFrom[w] = v; // O(1)
        }
    }
O(|V|)
}

```

Then, looking at the innermost portion of the remaining code, we see the `adjust` call on the priority queue. This will do a bubbleUp and/or a percolateDown, so it is  $O(\log |V|)$ .

```

void findWeightedShortestPath (
    DiGraph& g,
    list<Vertex>& path,
    Vertex start,
    Vertex finish,
    unordered_map<Edge, int, EdgeHash>& weight)
{ // Dijkstra's Algorithm
O(|V|)

// Compute distance from start to finish
while (!pq.empty()) // cond: O(1)
{
    Vertex v = pq.top(); // O(1)
    int d = dist[v]; // O(1)
    pq.pop(); // O(log |V|)
    for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) // cond: O(1)
    {
        Vertex w = (*e).dest(); // O(1)
        if (dist[w] > d+weight[*e]) // cond: O(1)
        {
            dist[w] = d + weight[*e]; // O(1)
            pq.adjust (w); // O(log |V|)
            cameFrom[w] = v; // O(1)
        }
    }
}
O(|V|)
}

```

Similarly, the priority queue pop in the outer loop is  $O(\log |V|)$ .

Now, the structure of these two loops is familiar. The inner loop executes a total of  $|E|$  times, the outer  $|V|$  times. So the total cost of adjusting is  $O(|E|\log |V|)$ , and the cost of the rest of the two loops is  $O(|E| + |V|)$ .

The total is  $O(|E| + |V| + |E|\log |V|)$ . But  $|E|\log |V| > |E|$  for sufficiently large values of  $|V|$ , so this simplifies to  $O(|V| + |E|\log |V|)$ .

```

void findWeightedShortestPath (
    DiGraph& g,
    list<Vertex>& path,
    Vertex start,
    Vertex finish,
    unordered_map<Edge, int, EdgeHash>& weight)
{ // Dijkstra's Algorithm
O(|V|)
O(|V| + |E| log |V|)
O(|V|)
}

```

And the total cost of Dijkstra's algorithm is  $O(|V| + |E|\log |V|)$ .

### 3.3 Dijkstra's Algorithm and Standard Priority Queues

Dijkstra's algorithm, in its original form, can be difficult to use in C++. The problem is the assumption that we can “adjust” the value of elements already in the priority queue. The `std::priority_queue` provides no such operation, and adding it adds quite a bit of overhead to a priority queue implementation.

```

struct VertexDist {
    Vertex v;
    int distance;

    VertexDist (Vertex vv, int dist) : v(vv), distance(dist) {}

    bool operator> (Progress right) const
    { return distance > right.distance; }
};

void findWeightedShortestPath (
    DiGraph& g,
    list<Vertex>& path,
    Vertex start,
    Vertex finish,
    unordered_map<Edge, int, EdgeHash>& weight)

```

```

    // Dijkstra's Algorithm
    unordered_map<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;

    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi)
    {
        dist[*vi] = INT_MAX;
    }
    dist[start] = 0;
    std::priority_queue<VertexDist, vector<VertexDist>, greater<VertexDist> > pq; ①

    pq.push(VertexDist(start, 0)); ②
    // Compute distance from start to finish
    while (!pq.empty() && dist[finish] == INT_MAX) ③
    {
        VertexDist vd = pq.top();
        Vertex v = vd.v;
        int d = vd.distance;
        pq.pop();

        if (d < dist[v]) ④
        {
            dist[v] = d; ⑤
            for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e)
            {
                Vertex w = (*e).dest();
                if (dist[w] > d+weight[*e])
                {
                    double wdist = d + weight[*e]; ⑥
                    pq.push(VertexDist(w, wdist)); ⑦
                    cameFrom[w] = v;
                }
            }
        }
    }
    // Extract path
    Vertex v = finish;
    if (dist[v] != INT_MAX)
    {
        while (!(v == start))
        {
            path.push_front(v);
            v = cameFrom[v];
        }
        path.push_front(start);
    }
}

```

In this version, we defer recording the distance to vertex  $dist[v]$  until that  $v$  has bubbled up to the top of the priority queue, at which time we will know that this is the shortest distance to reach that vertex  $v$ .

- At ①, we see the use of a standard priority queue. Instead of just pushing vertices into the queue, we are pushing (vertex,distance) tuples. The priority queue is ordered by the distance component.

Unlike the original Dijkstra algorithm, we may wind up multiple occurrences of the same vertex in the priority queue (as many as one per incoming edge to that vertex). But by the virtue of the priority queue ordering, we will always pop off the one of these with the smallest distance first.

- Also, unlike the original algorithm, the priority queue starts out empty, and we then start things off (②) by pushing an entry indicating that we can reach the start vertex at a total distance of 0.
- At ③, because we will defer updating  $dist$  until we pop the vertex from the queue, and will only do that for the shortest possible distance to each vertex, we can actually exit the main loop early in most cases as soon as we have found our shortest distance to  $finish$ .
- At ④, we check to be sure that we have not already found a shorter way to reach  $v$ .
- If not, we update  $dist$  at ⑤ to indicate that we have found the shortest distance to  $v$ .
- After that, we check each of the outgoing edges from  $v$ , compute their distances (⑥), and, if they are an improvement, push them into the priority queue (⑦).
- The remainder of the algorithm proceeds as before.

How does these modification affect the complexity? The main loop could iterate as many as  $|E|$  times in the worst case (no path from start to finish), same as the original.

The operations of pushing and popping the priority queue are the only thing slowed down. The original algorithm placed  $|V|$  elements into the queue at the beginning, and slowly decreased the size. The new one starts with one element, and the queue grows and shrinks as we process it, but never has more than a total of  $|E|$  elements.

That suggests that, instead of  $O(|V| + |E|\log|V|)$ , we would have  $O(|V| + |E|\log|E|)$ . But  $|E| \leq |V|^2$ , so  $\log|E| \leq 2\log|V|$ , which means that we still simplify to  $O(|V| + |E|\log|V|)$ .

On average, this modified algorithm is probably slightly faster, because we often exit the main loop after many fewer iterations and, for most graphs, and although the average length of the priority queue may be longer, this is offset by the lower overhead of the standard, non-adjustable priority queue.

# 4 Minimum Spanning Trees

Consider the problem of hooking up a number of phone outlets in various locations throughout a building.

- Given the blueprints, we can determine the amount of wire required to connect any two phone jacks.
- We want to connect every jack, using a minimum amount of wire.
- Form a graph with
  - a vertex for each phone jack
  - undirected edges labeled by the wiring distance

This is *not* a minimum path problem, because the phone outlets don't need to be connected in a straight line – they just need to be connected somehow.

We want to find the subgraph of the original graph that

- is connected
- spans* (i.e., has every vertex and some edges from) the original graph
- Involves the edges leading the minimum possible sum of weights

A little thought will show that for any solution to this problem, there will only be one path from any vertex to any other vertex (if there were more than one, we would have a redundant connection somewhere). Any connected graph with that property is a tree, so this problem is known as the “minimum spanning tree” problem.

## 4.1 Prim's Algorithm

One solution, Prim's algorithm, turns out to be almost identical to Dijkstra's. The main difference is that, instead of keeping track of the total distance to each vertex from the start, we base our priority queue on the minimum distance to the vertex from any vertex that has already been processed.

```
void findMinSpanTree (
    DiGraph& g,
    set<Edge>& spanTree,
    Vertex start,
    unordered_map<Edge, int, EdgeHash>& weight)
{   // Prim's Algorithm
    VDistMap dist;
    unordered_map<Vertex, Edge, VertexHash> cameFrom;

    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi)
    {
        dist[*vi] = INT_MAX;
    }
    dist[start] = 0;

    adjustable_priority_queue<Vertex, VDistMap, CompareVerticesByDistance>
        pq (g.vbegin(), g.vend(), CompareVerticesByDistance(dist));

    // Collect edges of span tree
    while (!pq.empty())
    {
        vertex v = pq.top();
        int d = dist[v];
        pq.pop();
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e)
        {
            vertex w = (*e).dest();
            if (dist[w] > weight[*e])
            {
                dist[w] = weight[*e];
                pq.adjust (w);
                cameFrom[w] = *e;
            }
        }
    }
    for (unordered_map<Vertex, Edge, VertexHash>::iterator p = cameFrom.begin();
        p != cameFrom.end(); ++p)
        spanTree.insert ((*p).second);
}
```

Prim's algorithm collects edges in *cameFrom* instead of vertices, and at the end we copy those “shortest edges to this vertex” into the spanning tree output set.

The big change, though is in the innermost loop, where the “distance” associated with each vertex is simply the smallest edge weight seen so far. Try [running Prim's algorithm](#) and see how it works.

### 4.1.1 Analysis of Prim's Algorithm

The analysis of Prim's algorithm is identical to Dijkstra's.

# Sharing Pointers and Garbage Collection

Steven J Zeil

Last modified: Jul 17, 2018

## Contents:

- [\*\*1 Shared Structures\*\*](#)
  - [\*\*1.1 Singly Linked Lists\*\*](#)
  - [\*\*1.2 Doubly Linked Lists\*\*](#)
  - [\*\*1.3 Airline Connections\*\*](#)
- [\*\*2 Garbage Collection\*\*](#)
  - [\*\*2.1 Reference Counting\*\*](#)
  - [\*\*2.2 Mark and Sweep\*\*](#)
  - [\*\*2.3 Generation-Based Collectors\*\*](#)
  - [\*\*2.4 Incremental Collection\*\*](#)
- [\*\*3 Strong and Weak Pointers\*\*](#)
  - [\*\*3.1 Smart Pointers can be Strong or Weak\*\*](#)
- [\*\*4 Java Programmers Have it Easy\*\*](#)

## Swearing by Sharing

We've talked a lot about using pointers to share information, but mainly as something that causes problems.

- We have a [pretty good idea](#) of how to handle ourselves when we have pointers among our data members and *don't* want to share.

In that case, we rely on implementing our own deep copying so that every "container" has distinct copies of all of its components.

Example +

```

class Catalog {
    :
    Catalog (const Catalog& c);
    Catalog& operator= (const Catalog& c);
    ~Catalog();
    :

private:
    Book* allBooks;      // array of books
    int numBooks;
};

Catalog::Catalog (const Catalog& c)
: numBooks(c.numBooks)
{
    allBooks = new Book[numBooks];
    copy (c.allBooks, c.allBooks+numBooks, allBooks);
}

Catalog& Catalog::operator= (const Catalog& c)
{
    if (*this != c)
    {
        delete [] allBooks;
        numBooks = c.numBooks;
        allBooks = new Book[numBooks];
        copy (c.allBooks, c.allBooks+numBooks, allBooks);
    }
    return *this;
}

Catalog::~Catalog()
{
    delete [] allBooks;
}

```

For some data structures, this is OK. If we are using a pointer mainly to give us access to a dynamically allocated array, we can copy the entire array as necessary. In the example shown here, we would want each catalog to get its own distinct array of books. So we would implement a deep copy for the assignment operator and copy constructor, and delete the allBooks pointer in the destructor.

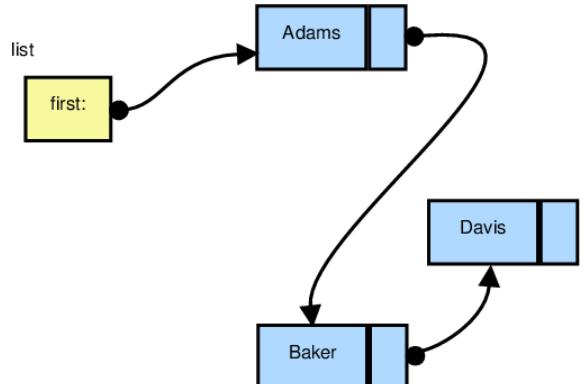
- But not every data structure can be treated this way.
  - Sometimes, sharing is essential to the behavior of the data structure that we want to implement.
  - Most applications that involve graphs rely on lots and lots of pointer-based sharing.

## 1 Shared Structures

In this section, we will call out three examples that we will explore further in the remainder of the lesson. All three involve some degree of essential sharing.

### 1.1 Singly Linked Lists

We'll start with a fairly prosaic example. In its simplest form, a singly linked list involves no sharing, and so we could safely treat all of its components as deep-copied.



#### 1.1.1 SLL Destructors

In particular, we can take a simple approach of writing the destructors — if you have a pointer, delete it:

```

struct SLNode {
    string data;
    SLNode* next;
    :
    ~SLNode () {delete next;}
};

class List {
    SLNode* first;
}

```

```

public:
    ...
~List() {delete first;}
};

```

Problem: stack size is  $O(N)$  where  $N$  is the length of the list.

- If a List object gets destroyed, its destructor will delete its *first* pointer. That node (Adams in the picture) will have its destructor called as part of the delete, and it will delete its pointer to Baker. The Baker node's destructor will delete the pointer to Davis. At the end, we have successfully recovered all on-heap memory (the nodes) with no problems.
- Now, this isn't really ideal. At the time the destructor for Davis is called, there are still partially executed function activations for the Baker and Adams destructors and for the list's destructor still on the call stack, waiting to finish. That's no big deal with only three nodes in the list, but if we had a list of, say, 10000 nodes, then we might not have enough stack space for 10000 uncompleted calls. So, typically, we would actually use a more aggressive approach with the list itself:

#### Alternative: Destroy the List, not the Nodes

```

struct SLNode {
    string data;
    SLNode* next;
    ...
~SLNode () {/* do nothing */}
};

class List {
    SLNode* first;
public:
    ...
~List()
{
    while (first != 0)
    {
        SLNode* next = first->next;
        delete first;
        first = next;
    }
}
};

```

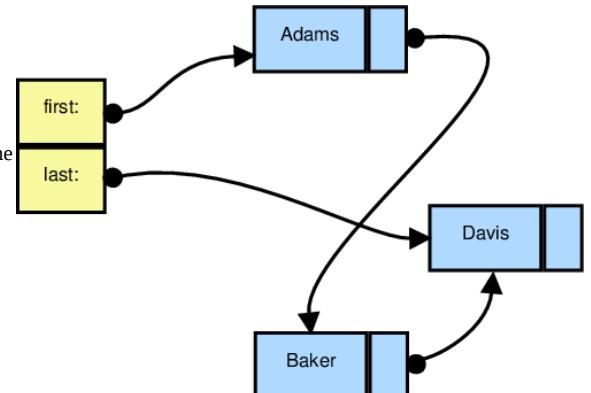
This avoids stacking up large numbers of recursive calls.

- Many operating systems allocate a relatively limited area of memory for the call stack, so this is a non-trivial improvement.

#### 1.1.2 First-Last Headers

But now let's consider one of the more common variations on linked lists.

- If our header contains pointers to both the first and last nodes of this list, then we can do  $O(1)$  insertions at both ends of this list.
- Notice, however, that the final node in the list is now "shared" by both the list header and the next-to-last node.



#### FLH SLL Destructor

So, if we were to extend our basic approach of writing destructors that simply delete their pointers:

```

struct SLNode {
    string data;
    SLNode* next;
    ...
~SLNode () {delete next;}
};

class List {
    SLNode* first;
    SLNode* last;
public:
    ...
~List() {delete first; delete last;}
};

```

```
};
```

Then, when a list object is destroyed, the final node in the list will actually be deleted twice.

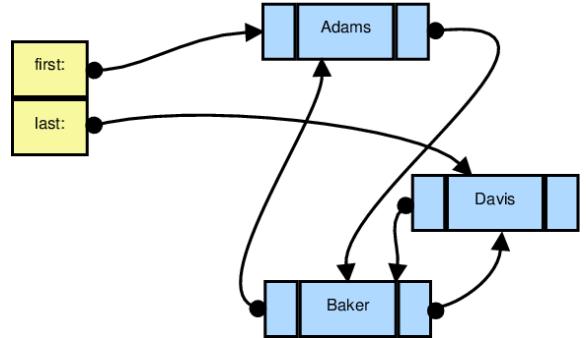
Deleting the same block of memory twice can corrupt the heap (by breaking the structure of the system [free list](#)) and eventually cause the program to fail.

## 1.2 Doubly Linked Lists

### Doubly Linked Lists

Now, let's make things just a little *more* difficult.

If we consider doubly linked lists, our straightforward approach of “delete everything” is really going to be a problem.

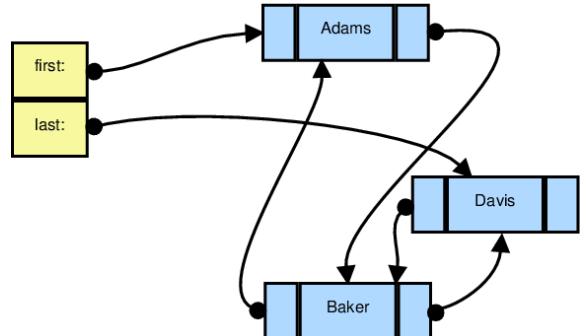


### DLL Aggressive Deleting

```
struct DLNode {
    string data;
    DLNode* prev;
    DLNode* next;
    ...
~DLNode () {delete prev; delete next;}
};

class List {
    DLNode* first;
    DLNode* last;
public:
    ...
~List() {delete first; delete last;}
};
```

- When a list object is destroyed, it will start by deleting the *first* pointer.
  - That node (Adams) will delete its *next* pointer (pointing to Baker).
  - That second node will delete its *prev* pointer (Adams). 7
- Now we've deleted the same node twice, potentially corrupting the heap.
  - But, worse, the Adam node's destructor will be invoked again.
  - It will delete its *next* pointer, and we will have deleted the Baker node a second time.
- Then the Baker node deletes its *prev* pointer *again*.



### Deleting and Cycles

We're now in an infinite recursion,

- which will continue running until either the heap is so badly corrupted that we crash when trying to process a delete,
- or when we finally fill up the activation stack to its maximum possible size.

What makes this so much nastier than the singly linked list?

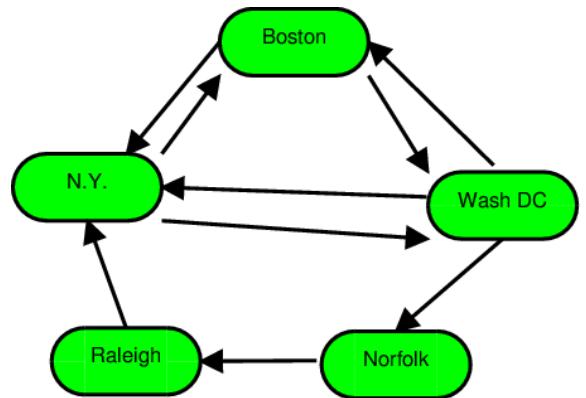
- It's the fact that not only are we doing sharing via pointers, but that the various connections form [cycles](#) in which we can trace a path via pointers from some object eventually back to itself.

## 1.3 Airline Connections

## Airline Connections

Lest you think that this issue only arises in low-level data structures, let's consider how it might arise in programming at the application level.

This graph illustrates flight connections available from an airline.



### Aggressively Deleting a Graph

If we were to implement this airport graph with Big 3-style operations:

```
class Airport
{
    :
private:
    vector<Airport*> hasFlightsTo;
};

Airport::~Airport()
{
    for (int i = 0; i < hasFlightsTo.size(); ++i)
        delete hasFlightsTo[i];
}
```

we would quickly run into a disaster.

### Deleting the Graph

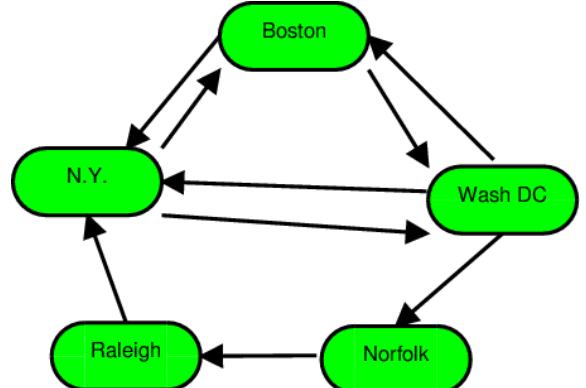
Suppose that we delete the Boston airport.

- Its destructor would be invoked, which would delete the N.Y. airport and Wash DC airports.
  - Let's say, for the sake of example, that the NY airport is deleted first.
- The act of deleting the pointer to the NY airport causes its destructor to be invoked, which would delete the Boston and Wash DC airports.
  - But Boston has already been deleted.
  - If we don't crash right away, we will quickly wind up deleting Wash DC twice.
- In fact, we would wind up, again, in an infinite recursion among the destructors.

This should not be a big surprise. Looking at the graph, we can see that it is possible to form cycles.

- In fact, if there is any node in this graph that *doesn't* participate in a cycle, there would be something very wrong with our airline.

Either we would have planes piling up at some airport, unable to leave; or we would have airports that run out of planes and can't support any outgoing flights.



### The Airline

Now, you might wonder just how or why we would have deleted that Boston pointer in the first place.

So, let's add a bit of context.

- The airport graph is really a part of the description of an airline:

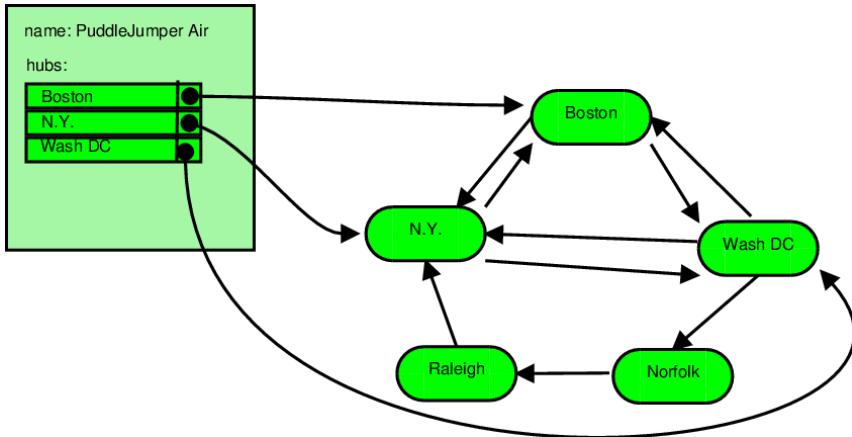
```
class AirLine {
    :
    string name;
    map<string, Airport*> hubs;
};
```

```

AirLine::~Airline()
{
    for (map<string, Airport*>::iterator i = hubs.begin;
         i != hubs.end(); ++i)
        delete i->second;
}

```

### The AirLine Structure



- The map *hubs* provides access to all those airports where planes are serviced and stored when not in flight, indexed by the airport name.
  - Not all airports are hubs.
- An airport that is not a hub is simply one where planes touch down and pick and discharge passengers while on their way to another hub.

Suppose that PuddleJumper Air goes bankrupt.

- It makes sense that when an airline object is destroyed, we would delete those hub pointers.
  - But we've seen that this is dangerous.

### Can We Do Better?

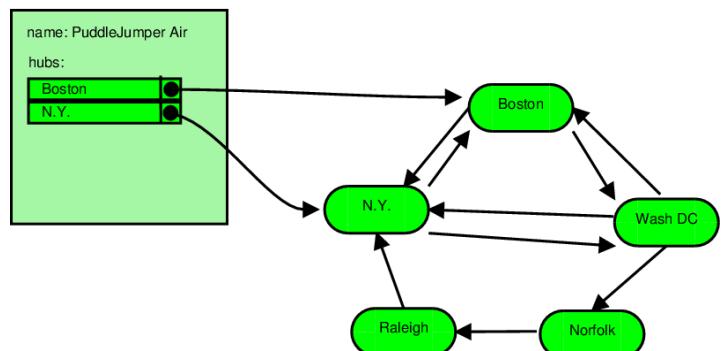
Now, that's a problem. But what makes this example particularly vexing is that it's not all that obvious what would constitute a better approach.

- Let's consider some other changes to the airline structure.

### Changing the Hubs

Suppose that Wash DC were to lose its status as a hub.

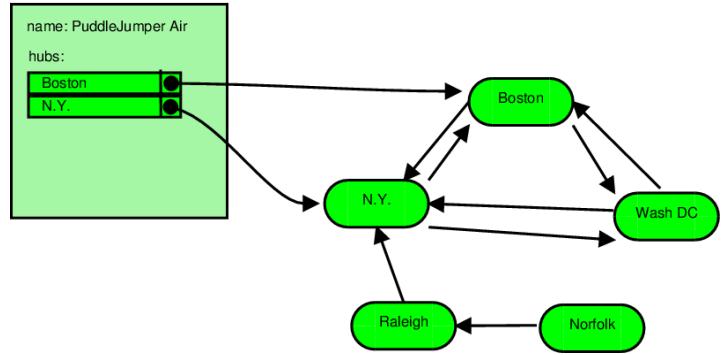
Even though the pointer to it was removed from the *hubs* table, the Wash DC airport needs to remain in the map.



### Changing the Connections

On the other hand, if Wash DC were to drop its service to Norfolk, one might argue that Norfolk and Raleigh should then be deleted, as there would be no way to reach them.

- But how could you write code into your destructors and your other code that adds and removes pointers that could tell the difference between these two cases?

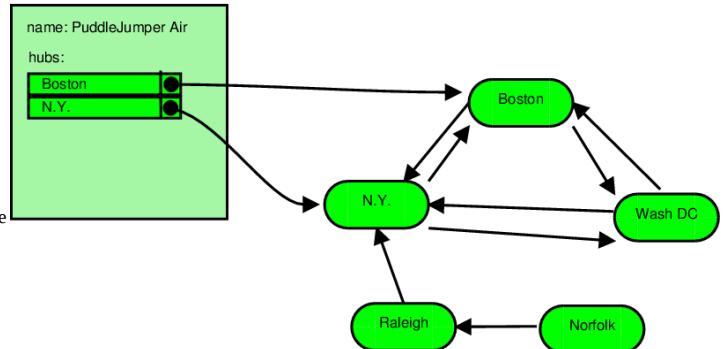


## 2 Garbage Collection

Objects on the heap that can no longer be reached (in one or more hops) from any pointers in the activation stack (i.e., in local variables of active functions) or from any pointers in the static storage area (variables declared in C++ as “static”) are called *garbage*.

### Garbage Example

- In this example, if we assume that the airline object is actually a local variable in some function, then Norfolk and Raleigh appear to be garbage.
  - Unless there's some other pointer not shown in this picture, there's no way to get to either of them.
- Being garbage is not the same thing as “nothing points to it”.
  - Raleigh is garbage even though something is pointing to it. Nonetheless, there is no way to get to Raleigh from the non-heap storage.



### Garbage Collection

Determining when something on the heap has become garbage is sufficiently difficult that many programming languages take over this job for the programmer.

The runtime support system for these languages provides *automatic garbage collection*, a service that determines when an object on the heap has become garbage and automatically *scavenges* (reclaims the storage of) such objects.

### Java has GC

The programming Java, for example, looks very similar to C++. A lot of code written in one of these languages will work in the other.

But, in Java, there is no `delete` operator.

Java programmers use lots of pointers,<sup>1</sup> many more than the typical C++ programmer.

But Java programmers never worry about deleting anything. They just trust in the garbage collector to come along eventually and clean up the mess.

### C++ Does Not Have GC

Automatic garbage collection really can simplify a programmer’s life. Sadly, C++ does not support automatic garbage collection.

- But how is this magic accomplished?
- Why doesn’t C++ support it?
- What can C++ programmers do about it?

That’s the subject of the remainder of this section.

## 2.1 Reference Counting

*Reference counting* is one of the simplest techniques for implementing garbage collection.

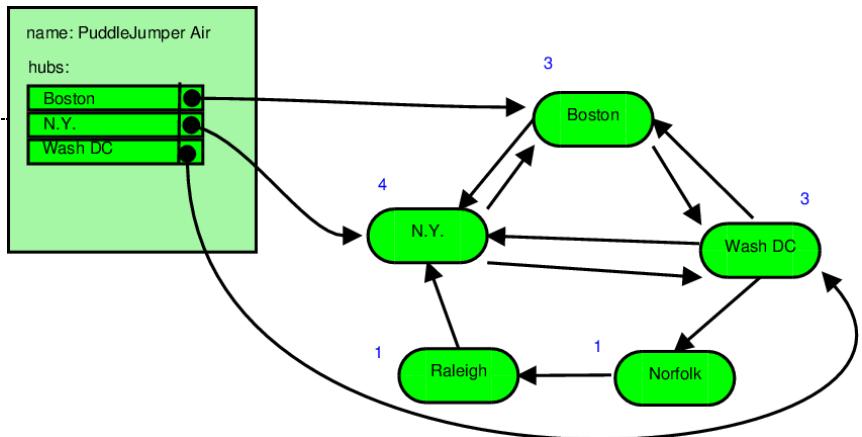
- Keep a hidden counter in each object on the heap. The counter will indicate how many pointers to that object exist.
- Each time we reassign a pointer that used to point at this object, we decrement the counter.
- Each time we reassign a pointer so that it now points at this object, we increment the counter.

- If that counter ever reaches 0, scavenge the object.

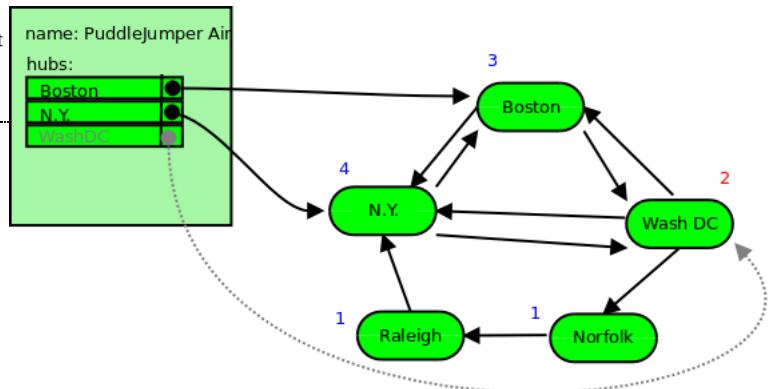
### **2.1.1 Reference Counting Example**

For example, here's our airline example with reference counts.

Now, suppose that Wash DC loses its hub status...

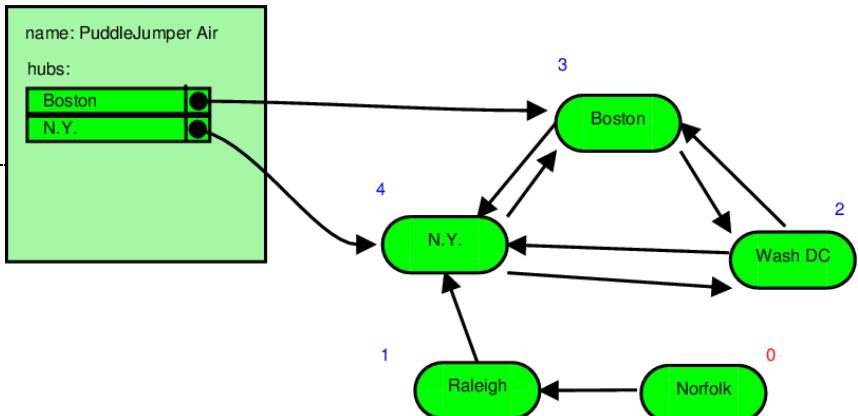


The removal of the pointer from the airline object causes the reference count of Wash DC to decrease by one, but it's still more than zero, so we don't try to scavenge Wash DC.



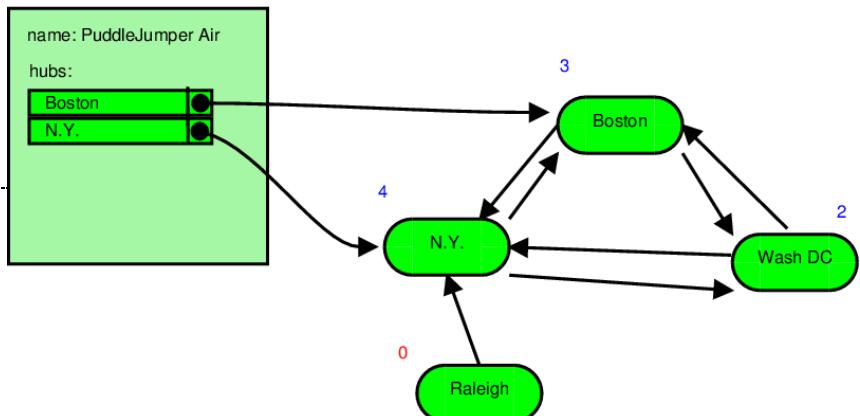
Now, suppose that Wash DC drops its service to Norfolk

- When the pointer from Wash DC to Norfolk is removed, then the reference count of Norfolk decreases. In this case, it decreases to zero.

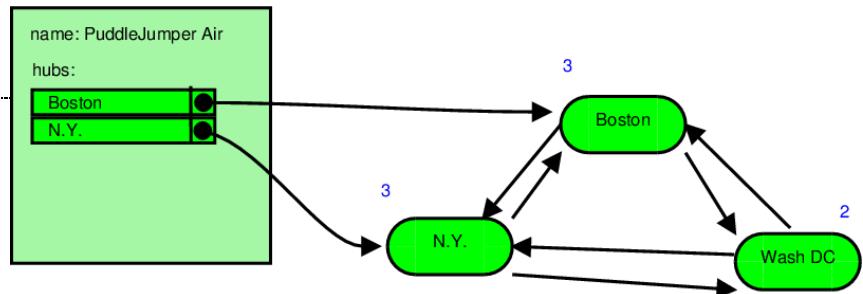


So the Norfolk object can be scavenged.

- When we do so, however, its pointer to Raleigh disappears, reducing Raleigh's reference count to zero.
- That means that Raleigh can be scavenged.



When Raleigh is destroyed, that reduces N.Y.'s reference count, but the count stays above zero, so we don't try to scavenge N.Y.



## 2.1.2 Reference Counted Pointers in C++

Implementing reference counting requires that we take control of pointers.

- To properly update reference counts, we would need to know whenever a pointer is assigned a new address (or null), whenever a pointer is created to point to a newly allocated object, and whenever a pointer is destroyed.
- Now, we can't do that for "real" pointers in C++.
  - But it is quite possible to create an ADT that looks and behaves much like a regular pointer.
  - And, by now, we know how to take control of assignment, copying, and destroying of ADT objects.

C++ now provides such an ADT — they are called "smart" pointers.

```
shared_ptr<T> p (new T());
```

This declares p to be a smart pointer to a reference-counted object of type T.

- Actually two blocks of storage are allocated – one to hold the T object and one to hold an associated reference counter;
- When we copy shared pointers to one another, the reference counters get updated automatically.
- If a reference counter drops to zero, then the object on the heap and its reference counter are destroyed.

**Important:** You have to commit *fully* to using shared pointers on the objects they manage. You cannot have ordinary pointers to an object simultaneously while you also have shared\_ptrs to the same objects.

Mixing ordinary and shared pointers will likely leave you with dangling ordinary pointers when the shared pointer decides to scavenge an object, eventually causing your program to crash.

## 2.1.3 Example: A Reference-Counted Singly Linked List

Starting from:

```
#include <string>
using namespace std;

struct SLNode {
    string data;
    SLNode* next;
    ...
    SLNode (string d = string(), SLNode* nxt = nullptr)
    : data(d), next(nxt)
    {}
    ~SLNode () { }
};

class List {
    SLNode* first;
public:
    ...
    ~List()
    {
        while (first != 0)
        {
            SLNode* next = first->next;
            delete first;
            first = next;
        }
    }

    void add(string s)
    {
```

```

        first = new SLNode(s, first);
    }
};


```

we can change all uses of `SLNode*` to `shared_ptr<SLNode>`:

```

#include <string>
#include <memory>

using namespace std;

struct SLNode {
    string data;
    shared_ptr<SLNode> next;

    SLNode (string d = string(), shared_ptr<SLNode> nxt = nullptr)
        : data(d), next(nxt)
    {}
    ~SLNode () {/* do nothing */}
};

class List {
    shared_ptr<SLNode> first;
public:
    ...
    ~List()
    { }

    void add(string s)
    {
        first = shared_ptr<SLNode>(new SLNode(s, first));
    }
    ...
};


```

And we no longer have to worry about explicitly deleting our unneeded nodes.

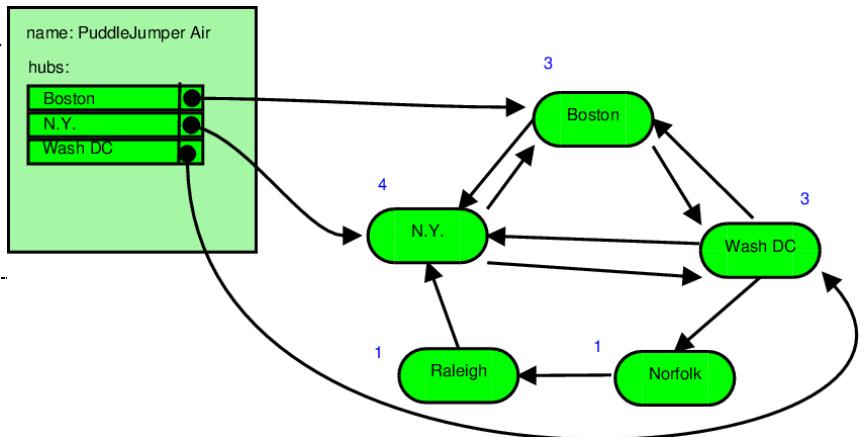
## 2.1.4 Is it worth the effort?

- Reference counting is fairly easy to implement.
  - Unlike the more advanced garbage collection techniques that we will look at shortly, it can be done in C++ because it does not require any special support from the compiler and the runtime system.
- There's a problem with reference counting, though.
  - One that's serious enough to make it unworkable in many practical situations.

### The Case of the Disappearing Airline

Let's return to our original airline example, with reference counts.

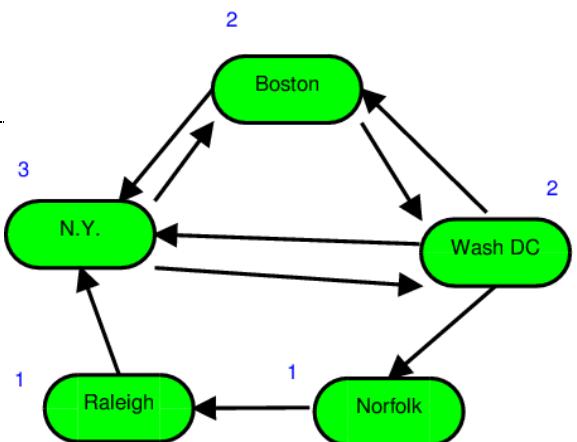
- Assume that
  - the airline object itself is a local variable in a function and that
  - we are about to return from that function.
- That object will therefore be destroyed, and its reference counted pointers to the three hubs will disappear.



Here is the result, with the updated reference counts.

- Now, all of these airport objects are garbage, but none of them have zero reference counts.
  - Therefore none of them will be scavenged.
  - We're leaking memory, big time!

What went wrong? Let's look at our other examples.

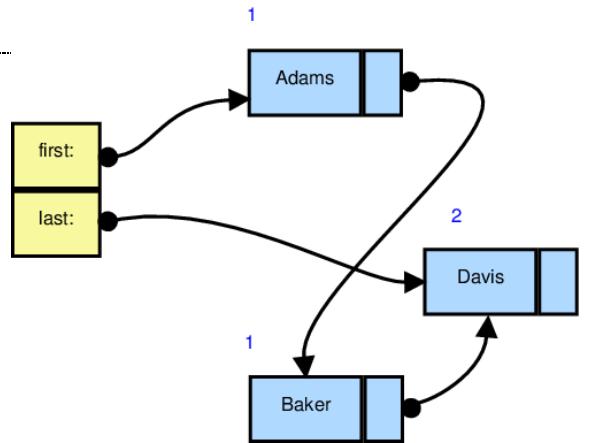


---

#### Ref Counted SLL

Here is our singly linked list with reference counts.

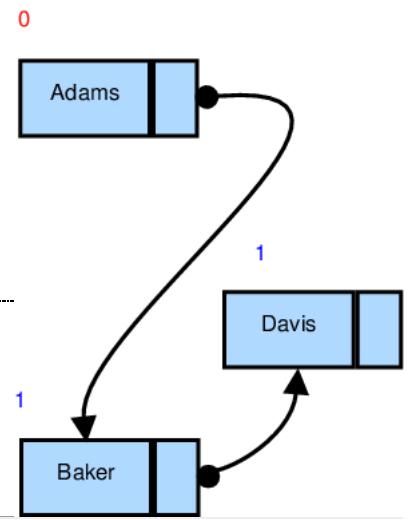
Assume that the list header itself is a local variable that is about to be destroyed.



## Ref Counted SLL II

- When the first and last pointers disappear, the reference count for Adams goes to zero.
  - So Adams can be scavenged.
- When it is, that will cause Baker's reference count to drop to zero.
  - When Baker is scavenged, Davis' count will drop to zero
- and it too will be scavenged.

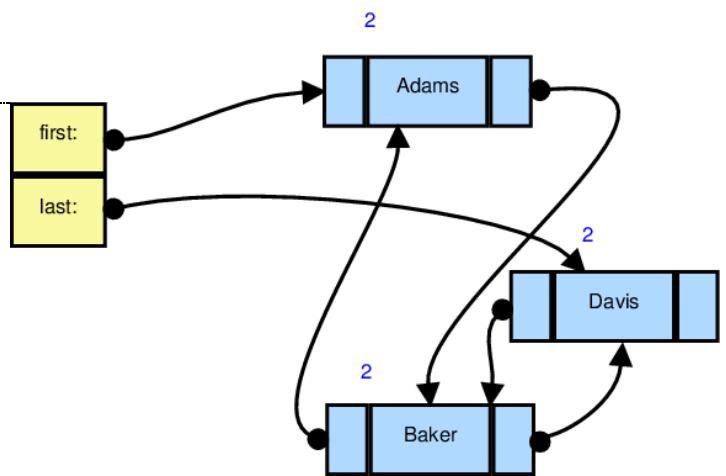
So that works just fine!



## Ref Counted DLL

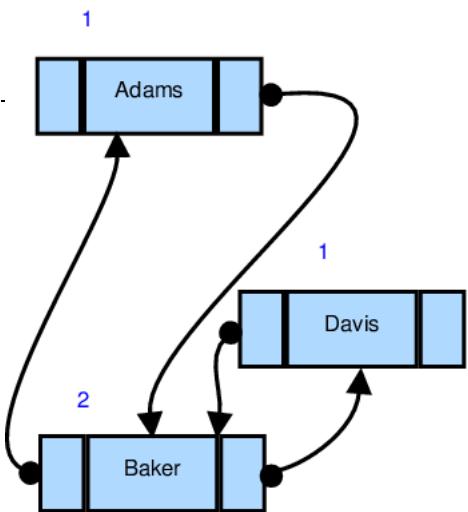
Now let's look at our doubly linked list.

Again, let's assume that the list header itself is a local variable that is about to be destroyed.



Here's the result.

Alas, we can see that none of the reference counters have gone to zero, so nothing will be scavenged, even though all three nodes are garbage.



### Reference Counting's Achilles Heel

So two of our last three examples failed when trying to use reference counting.

What's the common factor between the two failures?

- It's the *cycles*!

If the pointers form a cycle, then the objects in that cycle can never get a zero reference count, and reference counting will fail.

- Reference counting won't work if our data can form cycles of pointers.

◦ And, as the examples discussed here have shown, such cycles aren't particularly unusual or difficult to find in practical structures.

So a more general approach is needed.

## 2.2 Mark and Sweep

Mark and sweep is one of the earliest and best-known garbage collection algorithms.

- It works perfectly well with cycles, but
- requires some significant support from the compiler and run-time support system.

### Assumptions

The core assumptions of mark and sweep are:

- Each object on the heap has a hidden "mark" bit.
- We can find all pointers outside the heap (i.e., in the activation stack and static area)
- For each data object on the heap, we can find all pointers within that object.
- We can iterate over all objects on the heap

### The Mark and Sweep Algorithm

With those assumptions, the mark and sweep garbage collector is pretty simple:

```
markAndSweep.cpp +  
  
void markAndSweep()  
{  
    // mark  
    for (all pointers P on the run-time stack or  
         in the static data area )  
    {  
        mark *P;  
    }  
  
    //sweep
```

```

for (all objects *P on the heap)
{
    if *P is not marked then
        delete P
    else
        unmark *P
}
}

template <class T>
void mark(T* p)
{
    if *p is not already marked
    {
        mark *p;
        for (all pointers q inside *p)
        {
            mark *q;
        }
    }
}

```

The algorithm works in two stages.

- In the first stage, we start from every pointer outside the heap and recursively mark each object reachable via that pointer.

(In graph terms, this is a depth-first traversal of objects on the heap.)

- In the second stage, we look at each item on the heap.
  - If it's marked, then we have demonstrated that it's possible to reach that object from a pointer outside the heap.
    - It isn't garbage, so we leave it alone (but clear the mark so we're ready to repeat the whole process at some time in the future).
  - If the object on the heap is not marked, then it's garbage and we scavenge it.

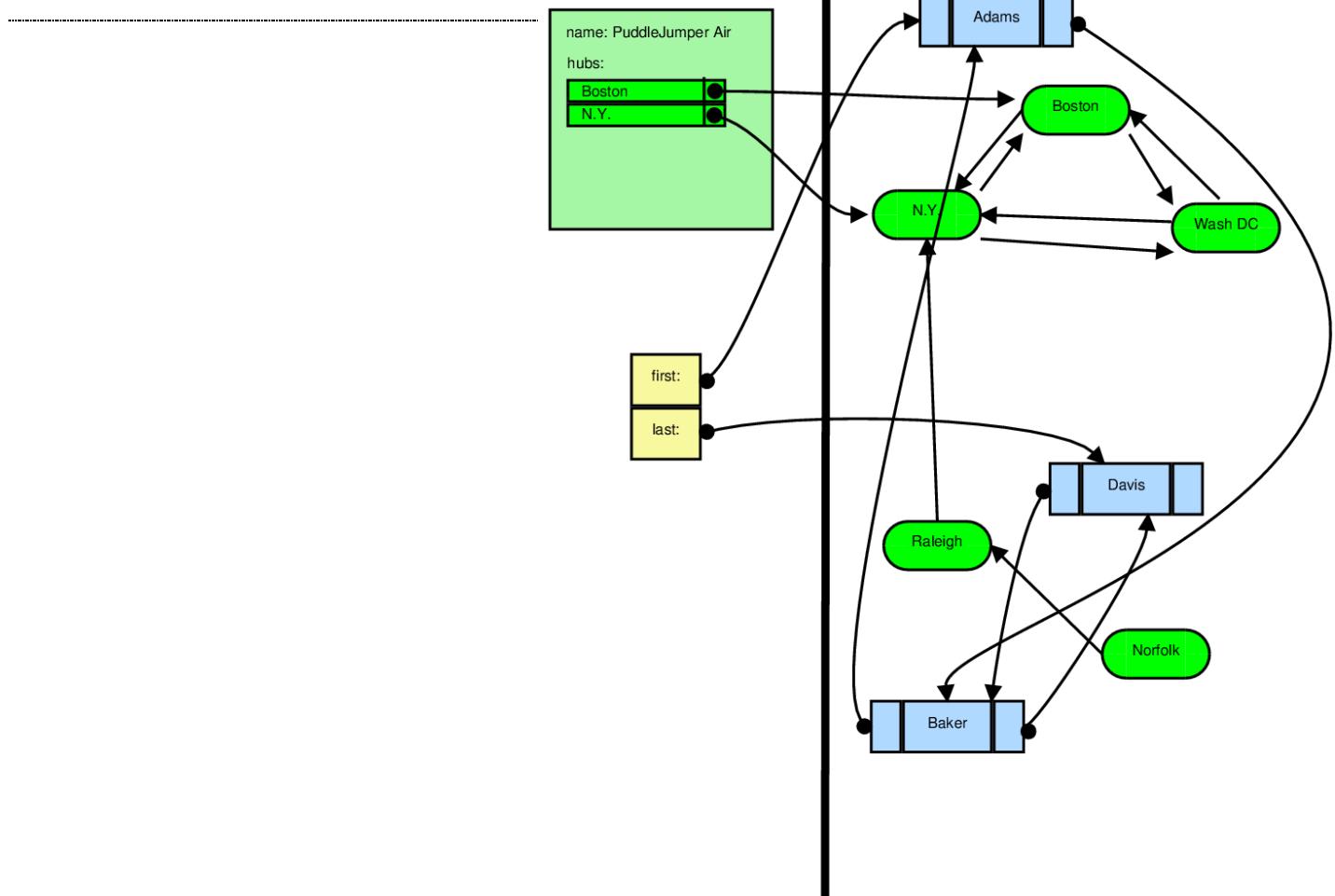
### **Mark and Sweep Example**

As an example, suppose that we start with this data.

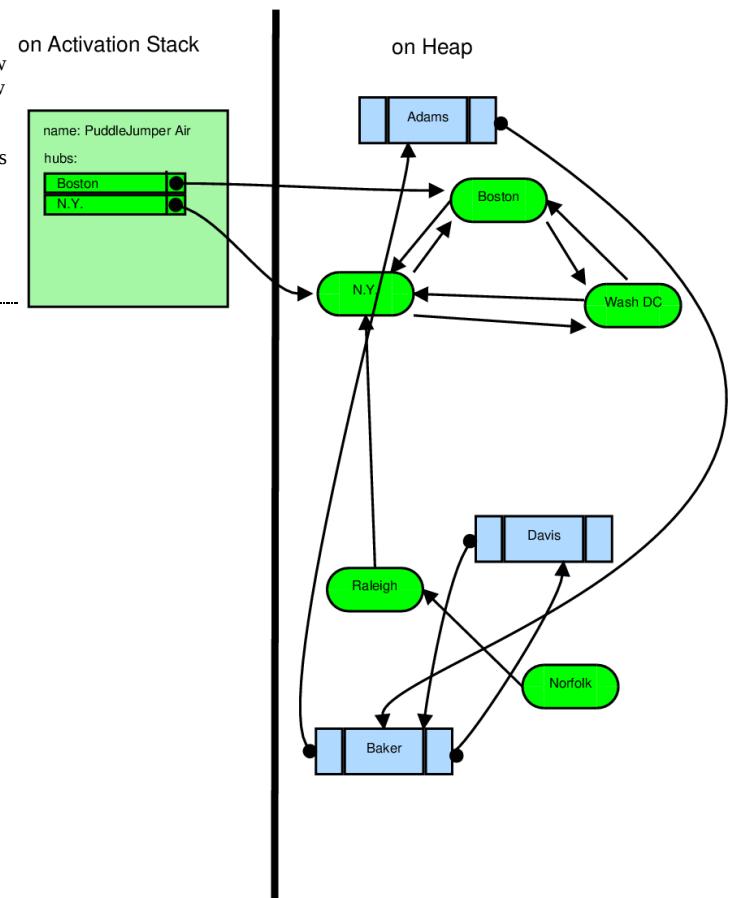
on Activation Stack

on Heap

Then, let's assume that the local variable holding the list header is destroyed.



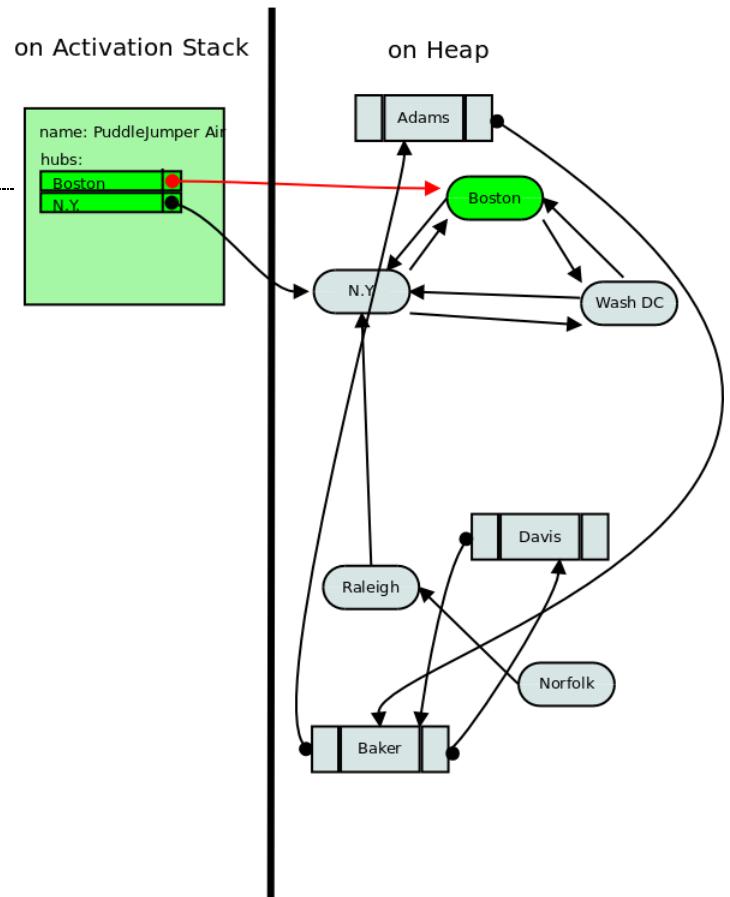
- At some point later in time, the mark and sweep algorithm is started (typically in response to later code of ours trying to allocate something new in memory and the run-time system has discovered that we are running low on available storage.).
- The main algorithm begins the marking phase, looping through the pointers in the activation stack.
  - We have two. The first points to the Boston node. So we invoke the `mark()` function on the pointer to Boston.
    - The Boston node has not been marked yet, so we mark it.



## The Mark Phase

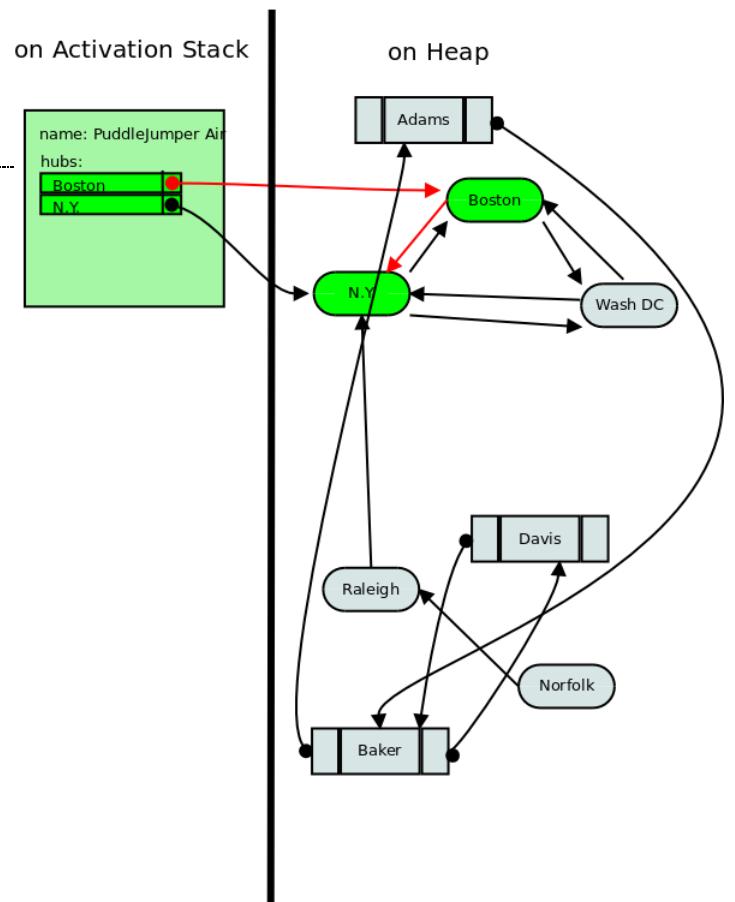
Here I've greyed out the objects on the heap that have not yet been marked. Boston is the only object marked so far, because we have just invoked `mark()` on it.

- Then the `mark()` function iterates over the pointers in the Boston object. It first looks at the N.Y. pointer and recursively invokes itself on that.

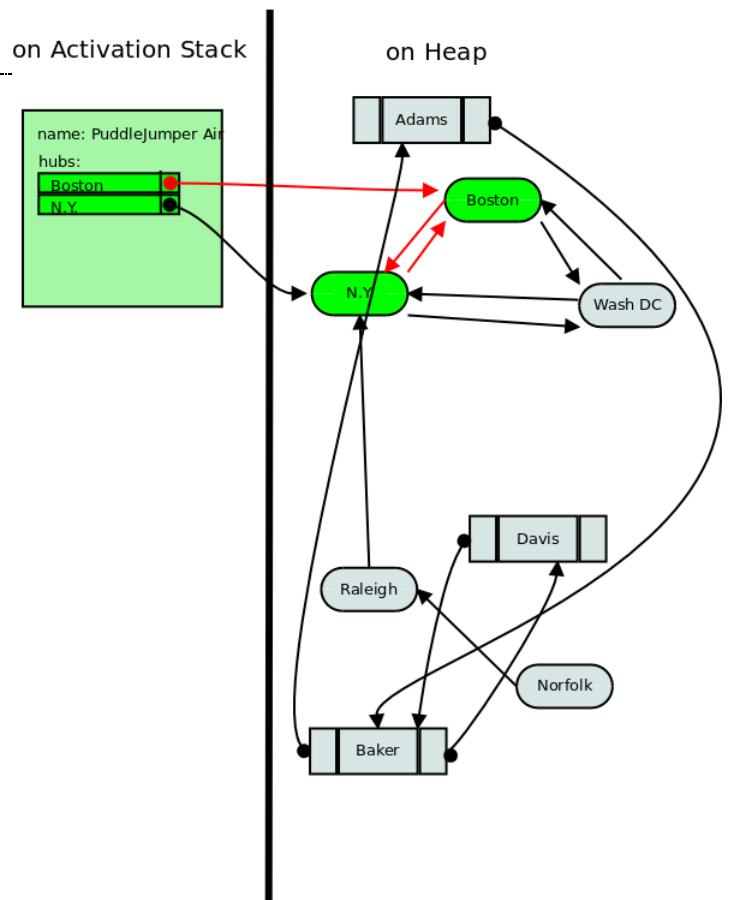


The N.Y. object has not been marked yet, so we mark it and then iterate over the pointers in N.Y.,

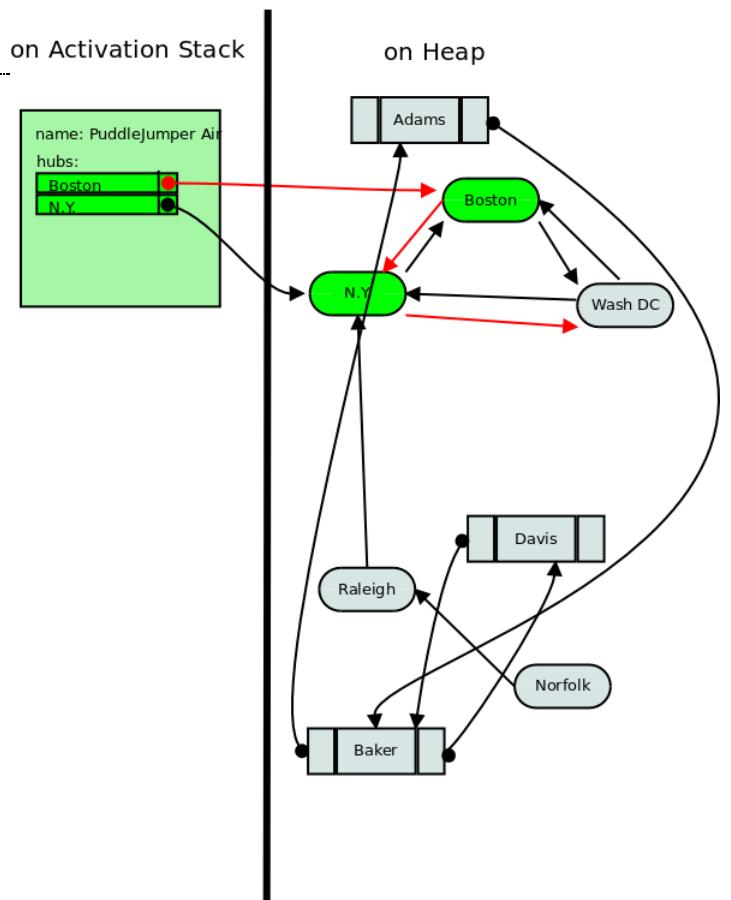
- Looking at the pointers coming out of N.Y., we first come to the pointer to Boston, and recursively invoke mark() on that.



But Boston is already marked, so we return immediately to the N.Y. call.

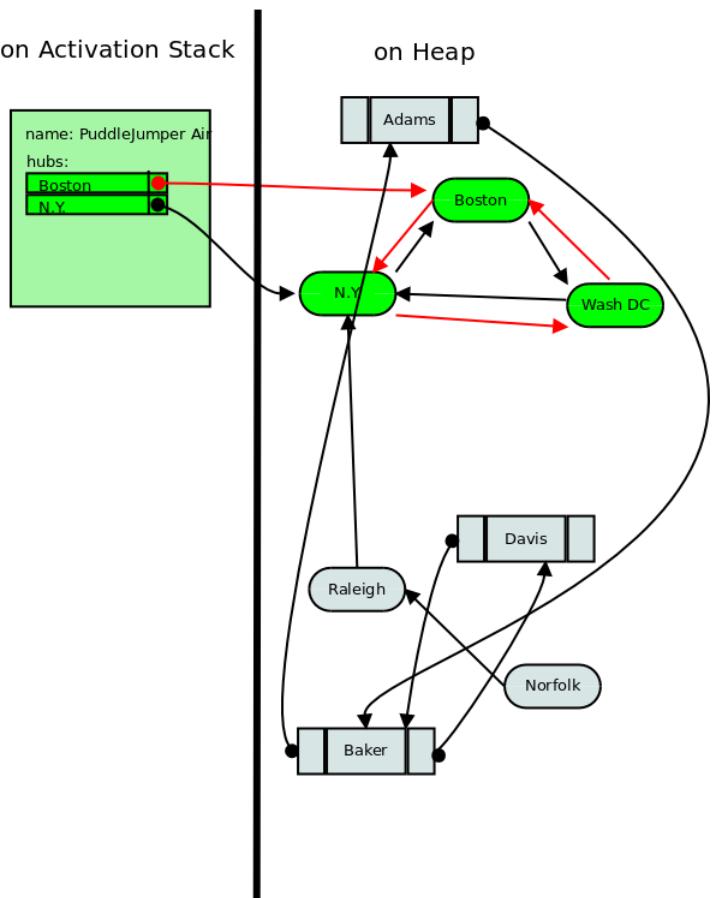


Continuing on, we find a pointer to Wash DC. and invoke mark() on that.



The Wash DC object has not been marked yet, so we mark it and then iterate over **on Activation Stack**

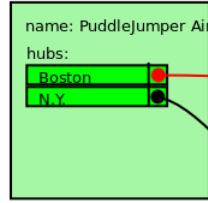
We first come to the pointer to Boston, and recursively /invoke mark() on that.



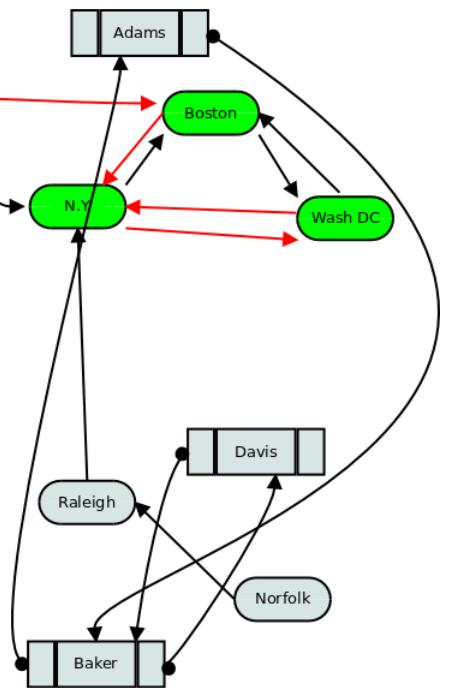
But Boston is already marked, so we return immediately to the Wash DC call.

There we find a pointer to N.Y.

on Activation Stack

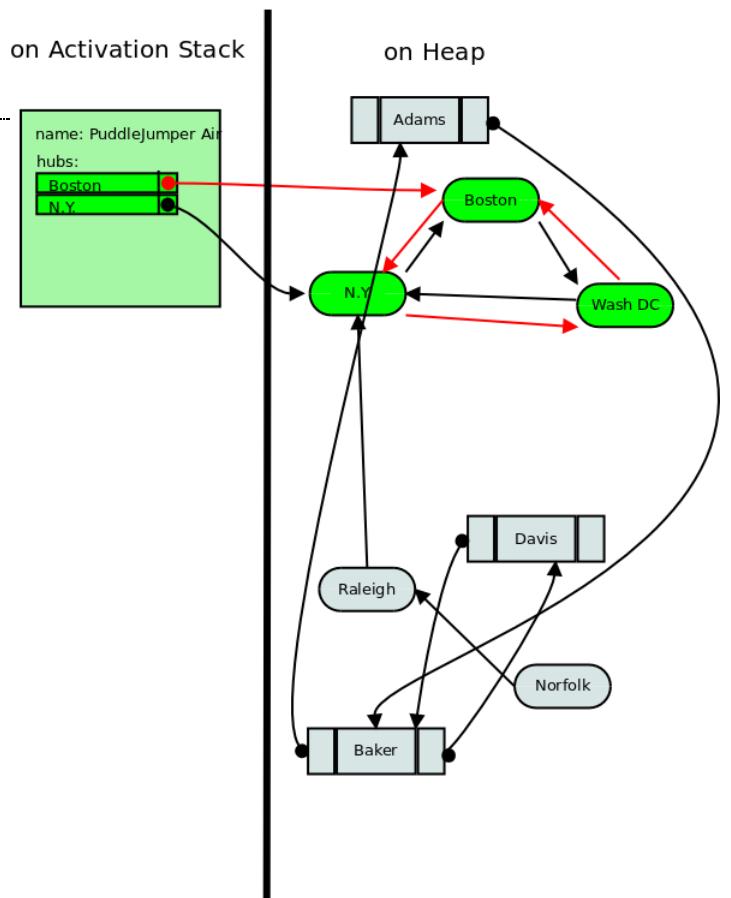


on Heap



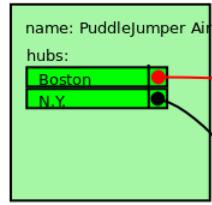
But N.Y is already marked, so we return immediately to the Wash DC call.

We've taken all the pointers out of Wash DC, so we return to the N.Y. call.

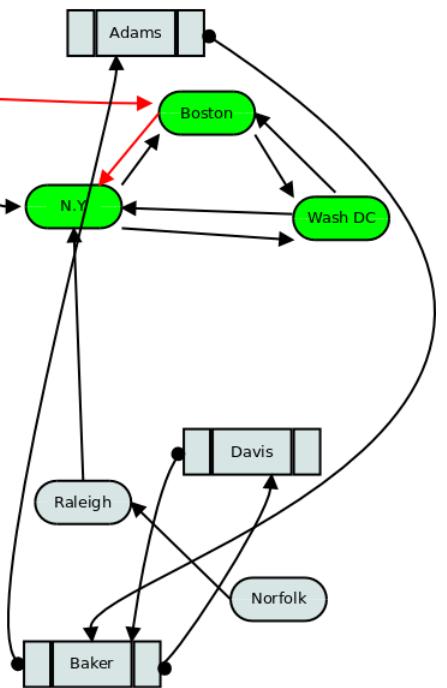


We've taken all the pointers out of N.Y., so we return to the Boston call.

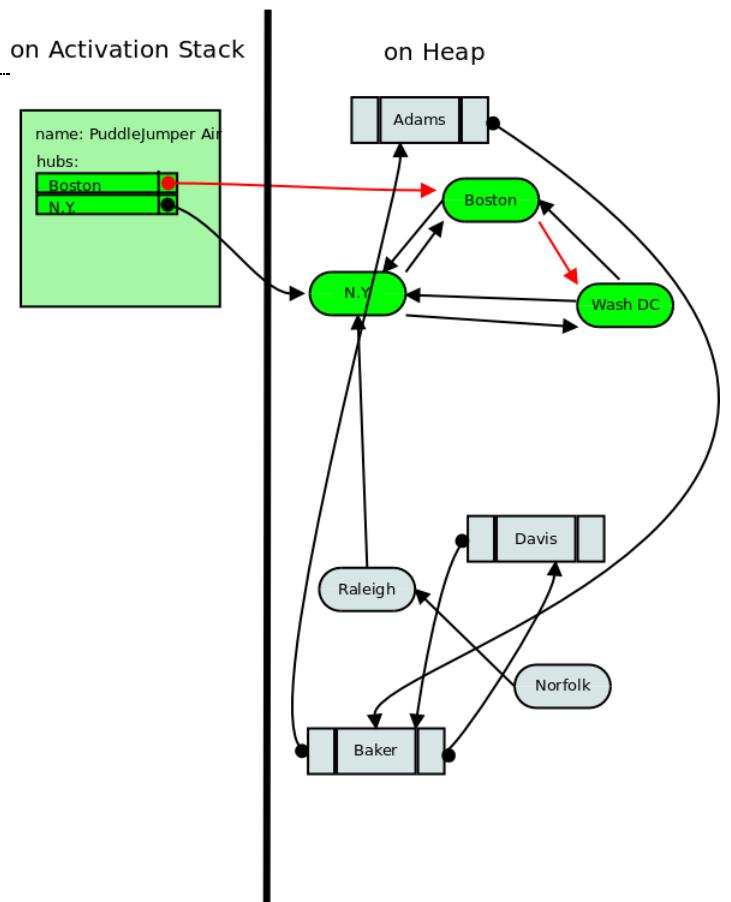
on Activation Stack



on Heap



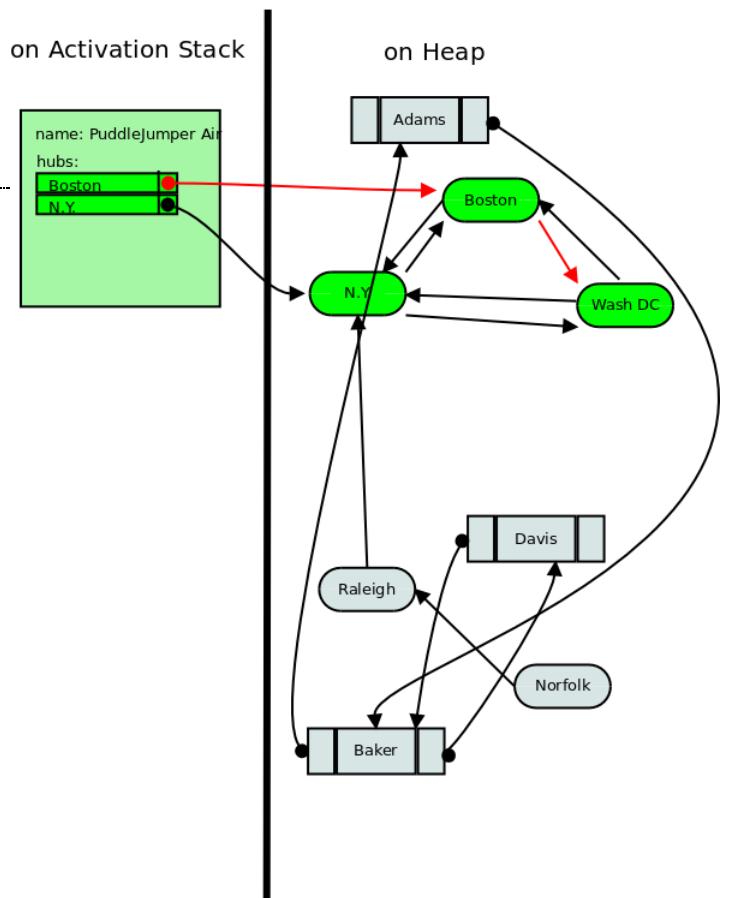
There we have a pointer to Wash DC.



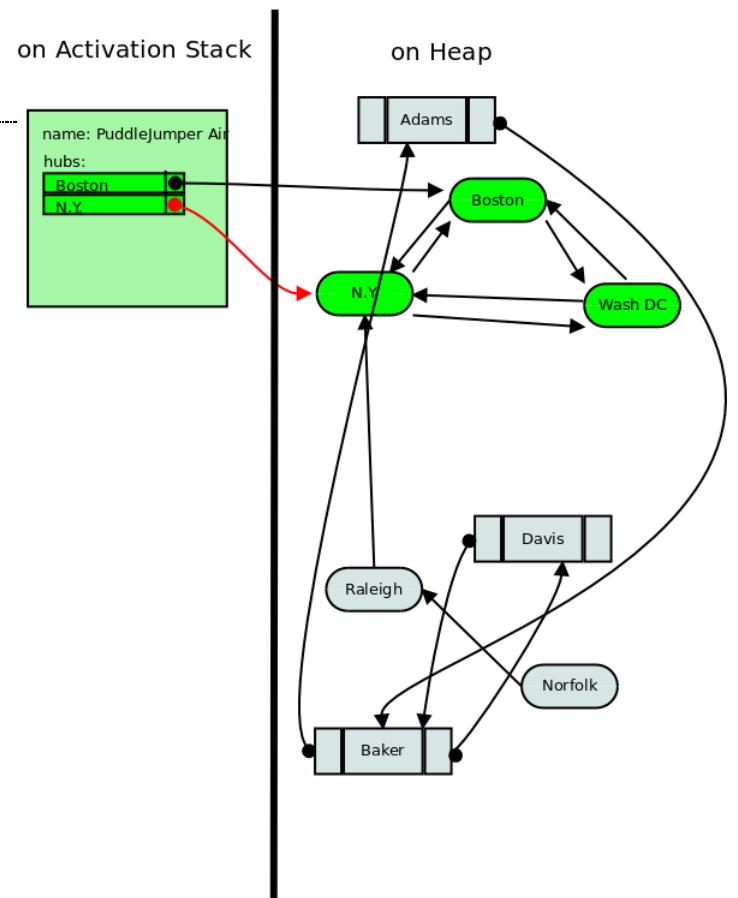
But Wash DC has already been marked, so we return to the Boston call.

We have followed all of the pointers out of Boston, so we return to the call on the first pointer in the hubs map.

We've taken all the pointers out of N.Y., so we return to the Boston call.

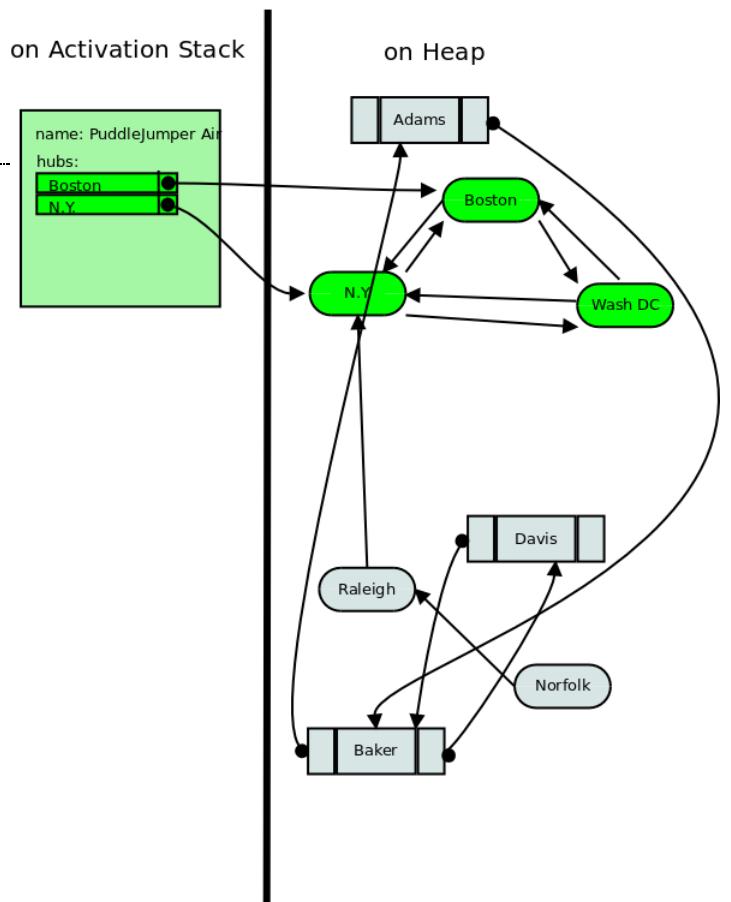


That algorithm continues looking at pointers on the activation stack. We have a pointer to N.Y., and call mark() on that. But N.Y. is already marked, so we return immediately.



Once the mark phase of the main algorithm is complete,

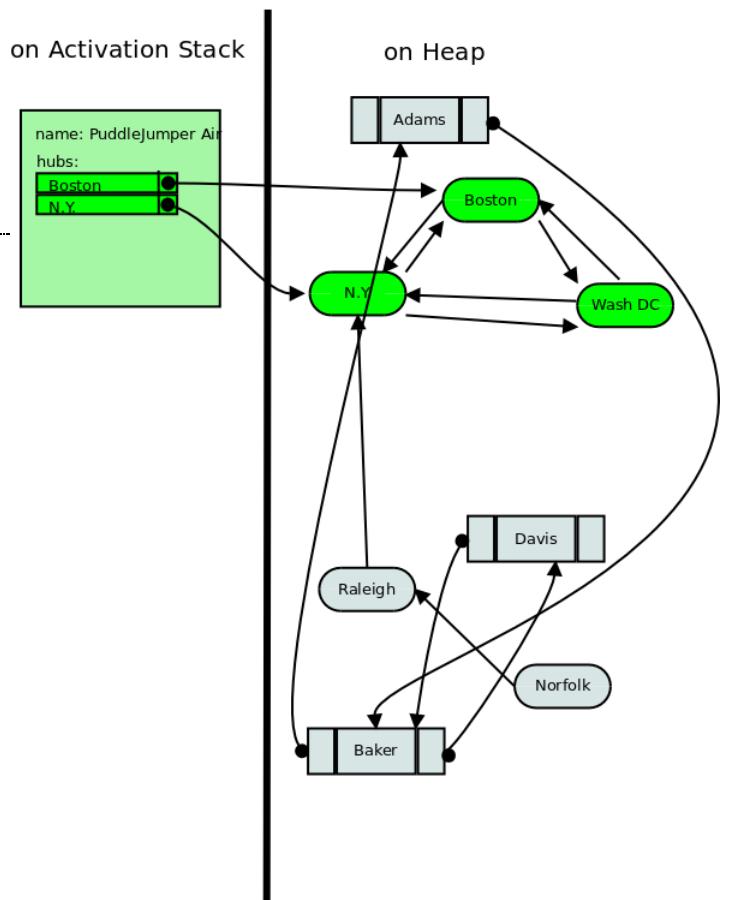
- We have marked the Boston, N.Y., and Wash DC objects.
- The Norfolk, Raleigh, Adams, Baker, and Davis objects are unmarked.



### The Sweep Phase

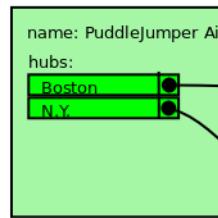
In the sweep phase, we visit each object on the heap.

- The three marked hubs will be kept, but their marks will be cleared in preparation for running the algorithm again at some time in the future.
- All of the other objects will be scavenged.

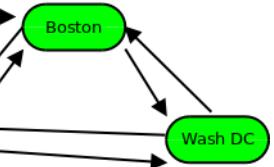


And the garbage has all been collected.

on Activation Stack



on Heap



### Assessing Mark and Sweep

In practice, the recursive form of mark-and-sweep requires too much stack space.

- It can frequently result in recursive calls of the mark() function running thousands deep.
  - Since we call this algorithm precisely because we are running out of space, that's not a good idea.

Practical implementations of mark-and-sweep have countered this problem with an iterative version of the mark function that “reverses” the pointers it is exploring so that they leave a trace behind it of where to return to.

- Even with that improvement, systems that use mark and sweep are often criticized as slow.

The fact is, tracing *every* object on the heap can be quite time-consuming. On virtual memory systems, it can result in an extraordinary number of page faults. The net effect is that mark-and-sweep systems often appear to freeze up for seconds to minutes at a time when the garbage collector is running. There are a couple of ways to improve performance.

## 2.3 Generation-Based Collectors

### Old versus New Garbage

In many programs, people have observed that object lifetime tends toward the extreme possibilities.

- temporary objects that are created, used, and become garbage almost immediately
- long-lived objects that do not become garbage until program termination

### Generational GC

Generational collectors take advantage of this behavior by dividing the heap into “generations”.

- The area holding the older generation is scanned only occasionally.
- The area holding the youngest generation is scanned frequently for possible garbage.

- an object in the young generation area that survives a few garbage collection passes is moved to the older generation area

The actual scanning process is a modified mark and sweep. But because relatively few objects are scanned on each pass, the passes are short and the overall cost of GC is low.

To keep the cost of a pass low, we need to avoid scanning the old objects on the heap. The problem is that some of those objects may have pointers to the newer ones. Most generational schemes use traps in the virtual memory system to detect pointers from “old” pages to “new” ones to avoid having to explicitly scan the old area on each pass.

## 2.4 Incremental Collection

---

### Incremental GC

Another way to avoid the appearance that garbage collection is locking up the system is to modify the algorithm so that it can be run one small piece at a time.

- Conceptually, every time a program tries to allocate a new object, we run just a few mark steps or a few sweep steps,
- By dividing the effort into small pieces, we give the illusion that garbage collection is without a major cost.

There is a difficulty here, though. Because the program might be modifying the heap while we are marking objects, we have to take extra care to be sure that we don’t improperly flag something as garbage just because all the pointers to it have suddenly been moved into some other data structure that we had already swept.

- In languages like Java where parallel processes/threads are built in to the language capabilities, systems can take the incremental approach even further by running the garbage collector in parallel with the main calculation.

Again, special care has to be taken so that the continuously running garbage collector and the main calculation don’t interfere with one another.

## 3 Strong and Weak Pointers

---

### Doing Without

OK, garbage collection is great if you can get it.

- But C++ does not provide it, and C++ compilers don’t really provide the kind of support necessary to implement mark and sweep or the even more advanced forms of GC.
- So what can we, as C++ programmers do, when faced with data structures that need to share heap data with one another?

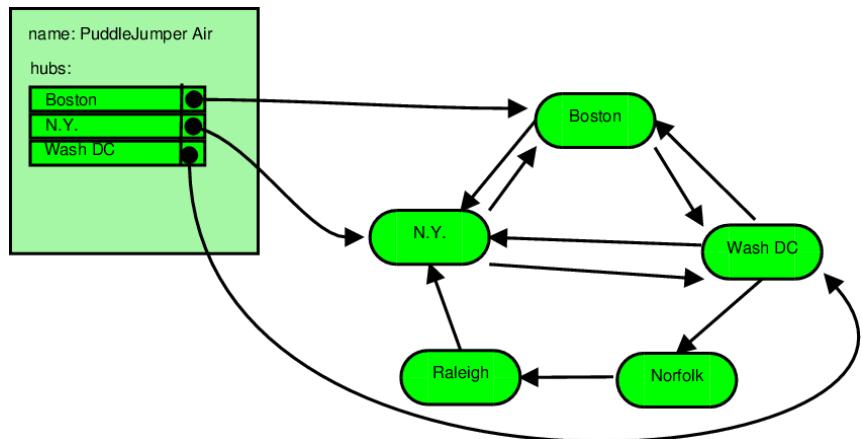
### Ownership

One approach that works in many cases is to try to identify which ADTs are the *owners* of the shared data, and which ones merely use the data.

- The owner of a collection of shared data has the responsibility for creating it, sharing out pointers to it, and deleting it.
- Other ADTs that share the data without owning it should never create or delete new instances of that data.

### Ownership Example

In this example that we looked at earlier, we saw that if both the Airline object on the left and the Airport objects on the right deleted their own pointers when destroyed, our program would crash.



### Ownership Example

We could improve this situation by deciding that the Airline *owns* the Airport descriptors that it uses. So the Airline object would delete the pointers it has, but the Airports would never do so.

```

class Airport
{
    :

private:
    vector<Airport*> hasFlightsTo;
};

Airport::~Airport()
{
    /* for (int i = 0; i < hasFlightsTo.size(); ++i)
        delete hasFlightsTo[i]; */
}

class AirLine {
    :
    string name;
    map<string, Airport*> hubs;
};

AirLine::~Airline()
{
    for (map<string, Airport*>::iterator i = hubs.begin;
        i != hubs.end(); ++i)
        delete i->second;
}

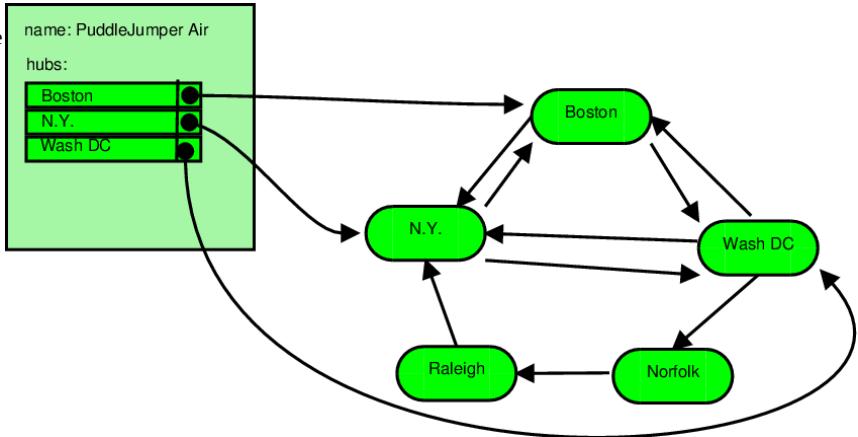
```

## Ownership Example

Thus, when the airline object on the left is destroyed, it will delete the Boston, N.Y., and Wash DC objects.

- Each of those will be deleted exactly once, so our program should no longer crash.
- This solution isn't perfect. The Norfolk and Raleigh objects are never reclaimed, so we do wind up leaking memory.

The problem is that, having decided that the Airline owns the Airport descriptors, we have some Airport objects with no owner at all.



## Asserting Ownership

I would probably resolve this by modifying the Airline class to keep better track of its Airports.

```

class AirLine {
    :
    string name;
    set<string> hubs;
    map<string, Airport*> airportsServed;
};

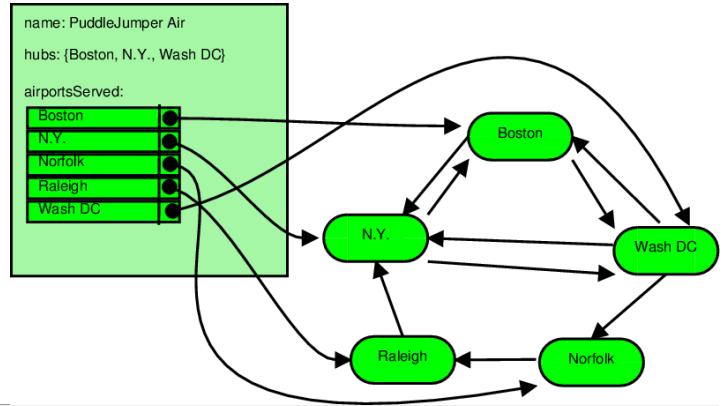
AirLine::~Airline()
{
    for (map<string, Airport*>::iterator i = airportsServed.begin;
        i != airportsServed.end(); ++i)
        delete i->second;
}

```

## Asserting Ownership (cont.)

The new map tracks all of the airports served by this airline, and we use a separate data structure to indicate which of those airports are hubs.

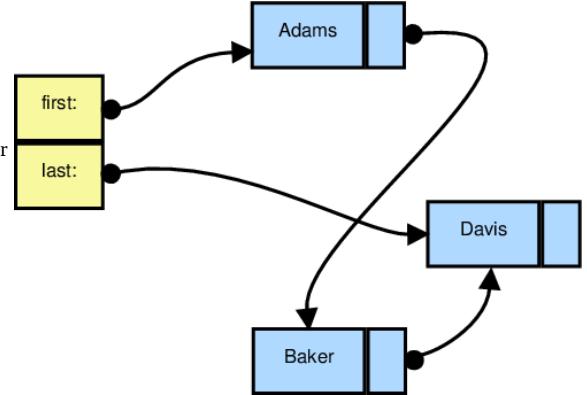
Now, when an airline object is destroyed, all of its airport descriptors will be reclaimed as well.



## Ownership Can Be Too Strong

Ownership is sometimes a bit too strong a relation to be useful.

- In this example, if we simply say that the list header owns the nodes it points to, then we would delete the first and last node and leave Baker on the heap.
- And if we say that the nodes owned the other nodes that they point to *and* that the list header owns the ones it points to, we would delete the last node twice.



## Strong and Weak Pointers

We can generalize the notion of ownership by characterizing the various pointer data members as strong or weak.

- A [strong pointer](#) is a pointer data member that indicates that the object pointed to must remain in memory.
- A [weak pointer](#) is a pointer data member that is allowed to point to data that might have been deleted.
  - (Obviously, we never want to follow a weak pointer unless we are sure that the data has not, in fact, been deleted.)

When an object containing pointer data members is destroyed, it deletes its strong pointer members and leaves its weak ones alone.

## Strong and Weak SLL

In this example, if we characterize the pointers as shown:

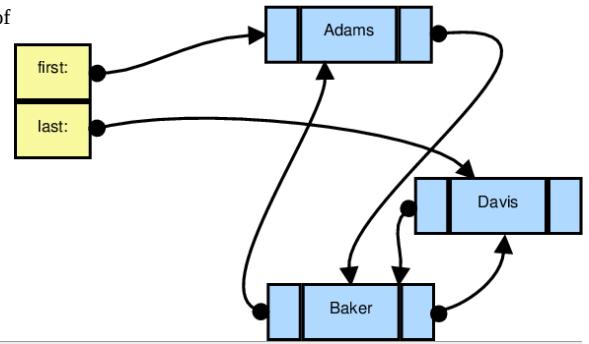
```
struct SLNode {
    string data;
    SLNode* next; // strong
    :
    ~SLNode () {delete next;}
};

class List {
    SLNode* first; // strong
    SLNode* last; // weak
public:
    :
    ~List()
    {
        delete first; // OK, because this is strong
        /*delete last; */ // Don't delete. last is weak.
    }
};
```

then our program will run correctly.

## Picking the Strong Ones

The key idea is to select the smallest set of pointer data members that would connect together all of the allocated objects, while giving you exactly one path to each such object.



### Strong and Weak DLL

Similarly, in a doubly linked list, we can designate the pointers as follows:

```

struct DLNode {
    string data;
    DLNode* prev; // weak
    DLNode* next; // strong
    ...
    ~DLNode () {delete next;}
};

class List {
    DLNode* first; // strong
    DLNode* last; // weak
public:
    ...
    ~List() {delete first;}
};
  
```

and so achieve a program that recovers all garbage without deleting anything twice.

## 3.1 Smart Pointers can be Strong or Weak

C++ smart pointers actually come in two “flavors”

- `shared_ptr` gives us strong pointers that are reference counted
- `weak_ptr` gives us weak pointers that do not affect reference counts, but can be assigned to and from strong pointers.

For example, if we were doing a doubly linked list, this would not be useful:

```

struct DLNode {
    string data;
    shared_ptr<DLNode> prev;
    shared_ptr<DLNode> next;
    ...
    ~DLNode () {delete next;}
};

class List {
    shared_ptr<DLNode> first;
    shared_ptr<DLNode> last;
public:
    ...
};
  
```

because the cycles induced by the `prev` and `next` pointers would prevent any nodes’ reference counts from dropping to zero.

But if we make the back pointer weak:

```

struct DLNode {
    string data;
    weak_ptr<DLNode> prev;
    shared_ptr<DLNode> next;
    ...
    ~DLNode () {delete next;}
};

class List {
    shared_ptr<DLNode> first;
    shared_ptr<DLNode> last;
public:
    ...
};
  
```

then the list should have no cycles, and reference counting should work just fine.

## 4 Java Programmers Have it Easy

Java has included automatic garbage collection since its beginning.

- From a practical point of view, sharing in Java is actually easier (and more common) than deep copying.
- Java programmers typically are unconcerned with many of the memory management errors that C++ programmers must strive to avoid.

C++ programmers may sometimes sneer at the slowdown caused by garbage collection. The collector implementations, however, continue to evolve. In fact, current versions of Java commonly offer multiple garbage collectors, one which can be selected at run-time in an attempt to find one whose run-time characteristics (i.e., how aggressively it tries to collect garbage and how much of the time it can block the main program threads while it is working) that matches your program's needs.

Java programmers sometimes face an issue of running out of memory because they have inadvertently kept pointers to data that they no longer need. This is a particular problem in implementing algorithms that use caches or memoization to keep the answers to prior computations in case the same result is needed again in the future. Because of this, Java added a concept of a weak reference (pointer) that can be ignored when checking to see if an object is garbage and that gets set to null if the object it points to gets collected.

---

[1:](#) Though, somewhat confusingly, they call them “references” instead of “pointers”. But they really are more like C++ pointers than like C++ references because you

- obtain one of these pointer/references by allocating an object on the heap via the operator new,
- can assign the value “null” to one of these to indicate that it isn’t pointing at anything at all, and
- can assign a new address to one of these to make it point at some different object on the heap

All three of these properties are true of C++ pointers but not of C++ references. So Java “references” really are the equivalent of C++ “pointers”.

By renaming them, Java advocates are able to boast that Java is a simpler language because it doesn’t have pointers. That’s more than a little disingenuous, IMO. (If it looks like a duck, swims like a duck, and quacks like a duck,...)

In actual fact, Java programs are absolutely swimming in pointers, but they pointers just aren’t as problematic as they are in C++.

# CS361 : Frequently Asked Questions

Steven J. Zeil

Last modified: May 16, 2020

Contents:

## [1 General Programming Questions](#)

- [1.1 What's all this “foo” and “bar” stuff?](#)
- [1.2 I'm not sure I have a correct algorithm to solve this problem. How do I check before writing and debugging the code?](#)
- [1.3 My program crashed. How do I find out why?](#)
- [1.4 Why do compilers' error messages often give the wrong line number, or even the wrong file?](#)
- [1.5 What are formal and actual parameters?](#)

## [2 The C++ Programming Language](#)

- [2.1 What's the odd expression with the ‘?’ and ‘:’ ?](#)
- [2.2 How do I convert a string to a character array \(or a character array to string\)?](#)
- [2.3 How do I convert a string to a number \(or a number to a string\)?](#)
- [2.4 What is the difference between a declaration and a definition?](#)
- [2.5 What goes in a .h file? What goes in a .cpp file?](#)
- [2.6 What is an inline function?](#)
- [2.7 Why is operator++ sometimes declared with an int parameter?](#)
- [2.8 What is the Rule of the Big 3?](#)
- [2.9 I have other C++ questions.](#)

## [3 C++ Compilers and IDEs](#)

- [3.1 How can I install a compiler and IDE on my own PC?](#)
- [3.2 How do I use a make file in Eclipse or Code::Blocks?](#)
- [3.3 How do I activate C++14 features when compiling?](#)
- [3.4 How do I set compiler flags and options?](#)
- [3.5 How do I supply command line parameters when executing my programs?](#)
- [3.6 How do I get my program to read from a file instead of from the keyboard?](#)

## [4 C++ Error Messages](#)

- [4.1 AddressSanitizer: attempting free on address which was not malloc\(\)-ed](#)
- [4.2 AddressSanitizer: heap-buffer-overflow](#)
- [4.3 AddressSanitizer: heap-use-after-free](#)
- [4.4 AddressSanitizer: NullPointerAccess](#)
- [4.5 AddressSanitizer: SEGV](#)
- [4.6 attempt to subscript container with out-of-bounds index](#)
- [4.7 attempt to dereference a singular iterator.](#)
- [4.8 attempt to dereference a past-the-end iterator.](#)
- [4.9 ...discards qualifiers...](#)
- [4.10 LeakSanitizer: detected memory leaks](#)
- [4.11 need typename before..., also “warning: ... is implicitly a typename” or “warning: implicit typename is deprecated”?](#)
- [4.12 No matching function ... , No match for...](#)
- [4.13 ‘nullptr’ was not identified in this scope](#)
- [4.14 Segmentation error? Segmentation fault? Bus error? Null reference error?](#)
- [4.15 terminate called after throwing an instance of ‘std::out\\_of\\_range’](#)
- [4.16 Undeclared/undefined names and symbols](#)

## [5 Mathematics and Notation](#)

- [5.1  \$\forall \exists\$](#)
- [5.2 Basic Probability \(probably just what you wanted\)](#)
- [5.3 Yucc! Nobody told me I needed to know about logarithms.](#)
- [5.4 Simplifying Summations](#)
- [5.5 How can I type mathematics into Blackboard?](#)

## [6 Miscellaneous](#)

- [6.1 How do I download a file from a link on a web page?](#)
- [6.2 I can't save makefiles \(or other files with no extension or unusual extensions\) without adding an extension.](#)
- [6.3 Will we be able to see solutions to quiz/exam questions? Why is my quiz/exam score so low? etc.](#)

This is a collection of questions (and answers!) that have arisen repeatedly in some of my past classes.

## 1 General Programming Questions

### 1.1 What's all this “foo” and “bar” stuff?

There is a long-standing tradition in computer science of using certain words as sample variable/function/whatever names. Just as a mathematician might use “x” or “y” whenever an arbitrary variable name is needed, computer scientists tend to use “foo”, “bar”, and “baz”, in that order. Check out [this entry](#) in the [Hacker’s Dictionary](#) for a discussion of the origin of these terms.

### 1.2 I'm not sure I have a correct algorithm to solve this problem. How do I check before writing and debugging the code?

The worst thing you can do is to simply start throwing C++ statements around at random trying to guess at an algorithm design.

1. Research - is there a known algorithm for addressing this kind of problem? Check your text and the lecture notes. Check the web.

Even if you find something, you will likely have to adapt it to the details of the problem at hand, but this can get you started.

2. Work at least a few examples by hand so that you have some simple examples of possible inputs and expected outputs at hand.

3. Draw pictures of the input and output data structures (i.e., [desk checking](#)). Take note of all the differences between the input and output pictures. Think about the programming steps that would be required to change the input pictures into the output pictures.

4. Use a systematic procedure for design, such as [stepwise refinement](#), a.k.a. *top-down design*, to derive a new algorithm that solves your problem.

5. As you work, update your desk checking pictures to make sure that your algorithm steps are moving you in the right direction.

## 1.3 My program crashed. How do I find out why?

There's no easy answer to that. Here's what I do, though, when faced with a crash that I just don't understand:

1. Look at the output produced before the crash. That can give you a clue as to where in the program you were when the crash occurred.

2. Run the program from within a [debugger](#) (or [here](#) and [here](#)).

Don't worry about breakpoints or single-stepping or any of that stuff at first. Just run it.

When the crash occurs, the debugger should tell you what line of code in what file was being executed at the moment of the crash.

Actually, it's not quite that simple. There's a good chance that the crash will occur on some line of code you didn't actually write yourself, deep inside some system library function that was called by some other system library function that was called by some other ... until we finally get back to your own code. That crash occurred because you are using a function but passed it some data that was incorrect or corrupt in some way.

Your debugger should let you view the entire runtime stack of calls that were in effect at the moment of the crash. So you should be able to determine **where** the crash occurred. That's not as good as determining **why**, but it's a start.

3. Take a good look at the data being manipulated at the location of the crash. Are you using pointers? Could some of them be null? Are you indexing into an array? Could your index value be too large or negative? Are you reading from a file? Could the file be at the end already, or might the data be in a different format than you expected?

If you used a debugger to find the crash locations, you can probably move up and down the stack and to view the values of variables within each active call. This may give a clue about what was happening.

4. Form some hypotheses (take a guess) as to what was going on at the time of the crash. Then [test your hypothesis!](#)

## 1.4 Why do compilers' error messages often give the wrong line number, or even the wrong file?

A compiler can only report where it *detected* a problem. Where you actually *committed* a mistake may be someplace entirely different.

The vast majority of error messages that C++ programmers will see are

- syntax errors (missing brackets, semi-colons, etc.)
- undeclared symbols
- undefined symbols
- type errors (often “cannot find a matching function” complaints)
- const errors

Let's look at these from the point of view of the compiler.

### 1.4.1 Syntax errors

Assume that the compiler has read part, but not all, of your program. The part that has just been read contains a syntax error. For the sake of example, let's say you wrote:

```
x = y + 2 * x // missing semi-colon
```

Now, when the compiler has read only the first line, it can't tell that anything is wrong. That's because it is still possible, as far as the compiler knows, that the next line of source code will start with a “;” or some other valid expression. So the compiler will *never* complain about this line.

If the compiler reads another line, and discovers that you had written:

```
x = y + 2 * x // missing semi-colon
++i;
```

it still won't conclude that there's a missing semi-colon. For all it knows, the "real" mistake might be that you meant to type "+" instead of "++".

Now, things can be much worse. Suppose that inside a file `foo.h` you write

```
class Foo {  
    :  
    Foo();  
    int f();  
    // missing };
```

and inside another file, `bar.cpp`, you write

```
#include "foo.h"  
  
int g() { ... }  
  
void h(Foo) { ... }  
  
int main() { ... }
```

Where will the error be reported? Probably on the very last line of `bar.cpp`! Why? Because until then, it's still possible, as far as the compiler knows, for the missing "};" to come, in which case `g`, `h`, and `main` would just be additional member functions of the class `Foo`.

So, with syntax errors, you know only that the real mistake occurred on the line reported *or earlier*, possibly even in an earlier-`\#include`'d file.

## 1.4.2 undeclared and undefined symbols

See [this discussion](#).

## 1.4.3 type errors

When you use the wrong object in an expression or try to apply the wrong operator/function to an object, the compiler may detect this as a type mismatch between the function and the expression supplied as the parameter to that function. These messages seem to cause students the most grief, and yet the compiler is usually able to give very precise descriptions of what is going wrong. The line numbers are usually correct, and the compiler will often tell you exactly what is going wrong. That explanation, however, may be quite lengthy, for three reasons:

1. Type names, especially when templates are involved, can be very long and messy-looking.
2. Because C++ allows function overloading (multiple functions with the same name, as long as they take different kinds of parameters), there may be many functions with the same name. The compiler will have to look at each of these to see if any one matches the parameter types you supplied. Some compilers report on each function tried, explaining why it didn't match the parameters in the faulty call.
3. If the function call was itself produced by a template instantiation or an inline function, then the problem is detected at the function call (often inside a C++ standard library routine) but the actual problem lies at the place where the template was used/instantiated. So most compilers will list both the line where the error was detected and all the lines where templates were instantiated that led to the creation of the faulty call.

So, to deal with these, look at the error message on the faulty function call. Note what function/operator name is being complained about. Then look at the line where the faulty call occurred. If it's inside a template or inline function that is not your own code, look back through the "instantiated from" or "called from" lines until you get back into your own code. That's probably where the problem lies.

Here's an example taken from a student's code:

```
g++ -g -MMD -c testapq.cpp  
/usr/local/lib/gcc-lib/sparc-sun-solaris2.7/2.95.2/../../../../include/g++-3/  
stl_relops.h: In function `bool operator <>_Rb_tree_iterator<pair<const  
PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const  
PrioritizedNames,int> *> >(const _Rb_tree_iterator<pair<const  
PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const  
PrioritizedNames,int> *> &, const _Rb_tree_iterator<pair<const  
PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const  
PrioritizedNames,int> *> &):  
adjpq.h:234: instantiated from `adjustable_priority_queue<  
PrioritizedNames,map<PrioritizedNames,int,CompareNames,allocator<int> >,  
ComparePriorities>::percolateDown(unsigned int)'  
adjpq.h:177: instantiated from `adjustable_priority_queue<PrioritizedNames,  
map<PrioritizedNames,int,CompareNames,allocator<int> >,  
ComparePriorities>::makeHeap()'  
adjpq.h:84: instantiated from here  
/usr/local/lib/sparc-sun-solaris2.7/2.95.2/../../../../include/  
g++-3/stl_relops.h:43: no match for `const _Rb_tree_iterator<pair<const  
PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const  
PrioritizedNames,int> *> & < const _Rb_tree_iterator<pair<const  
PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const  
PrioritizedNames,int> *> &'
```

Now, that may look intimidating, but that's mainly because of the long type names (due to template use) and the long path names to files from the C++ standard library. Let's strip that down to the essentials:

```
g++ -g -MMD -c testapq.cpp  
stl_relops.h: In function `bool operator >:  
adjpq.h:234: instantiated from `percolateDown(unsigned int)'
```

```
adjpq.h:177: instantiated from `makeHeap()'  
adjpq.h:84: instantiated from here  
stl_relops.h:43 no match for ... < ...
```

This one is actually worse than most error messages, because it's easy to miss the “<” operator amidst all the < . . . > template markers.

The problem is a “no match for” a less-than operator call in line 43 of a template within the standard library file `stl_relops.h`. But that template is instantiated from the student's own code (`adjpq.h`) and so the thing to do is to look at those three lines (234, 177, and 84) for a data type that is supposed to support a less-than operator, but doesn't.

#### 1.4.4 const errors

Technically, “const”-ness is part of a type, so while sometimes these get special messages of their own, often they masquerade as ordinary type errors and must be interpreted in the same way.

### 1.5 What are formal and actual parameters?

The *formal parameters* of a function are the parameter names that are declared in the function header and that are used when we write the function body. For example, in the code

```
int sequentialInsert (T a[], unsigned& n, const T& x)  
// insert x into sorted position within a,  
// with a already containing n items.  
// Return the position where inserted.  
{  
    int i = n;  
    while ((i > 0) && (x < a[i-1]))  
    {  
        a[i] = a[i-1];  
        i = i - 1;  
    }  
    a[i] = x;  
    ++n;  
    return i;  
}
```

the formal parameters are named `a`, `n`, and `x`. The *actual parameters* of a *call* to a function are the values passed by the caller. So, in the call

```
k = sequentialInsert (myArray, size-1, value);
```

the actual parameters are `myArray`, `size-1`, and `value`. Note that formal parameters are always just names. Actual parameters can be simple names or arbitrarily complicated expressions.

## 2 The C++ Programming Language

### 2.1 What's the odd expression with the ‘?’ and ‘:’ ?

You're probably looking at a *conditional expression*. It's a convenient shorthand for an if-then-else structure inside an ordinary expression. A typical example would be

```
x = (x < 0.0) ? -x : x;
```

The part before the ‘?’ is the condition and is evaluated first. If the condition is true, then the then-part expression (between the ‘?’ and the ‘:’) is evaluated and the result used as the value of the entire expression. If, however, the condition is false, then the else-part (after the ‘:’) is evaluated and that result is used as the value of the entire expression. This assignment, then, replaces `x` by its own absolute value. The whole thing, then, is equivalent to

```
double temp;  
if (x < 0.0)  
    temp = -x;  
else  
    temp = x;  
x = temp;
```

A conditional expression can appear inside other expressions, anywhere a value of the same type as its then-part and else-part might appear:

```
x = sqrt((x < 0.0) ? -x : x);
```

### 2.2 How do I convert a string to a character array (or a character array to string)?

To get a character array from a `std::string`, use the `string`'s `c_str()` function:

```
std::string str;  
:
```

```
char* cstr = str.c_str();
```

You will often see this done with older library functions that were originally designed to work with character arrays and not yet updated to work with strings:

```
copyFile.cpp +
```

```
void copyFile (string fileName)
{
    ifstream input (fileName.c_str()); // open an input file;
    ofstream output ("output.txt"); // No c_str() required - "output.txt" is a
                                    // character array, not a string.
    string line;
    getline (input, line);
    while (input)
    {
        output << line << endl;
        getline (input, line);
    }
}
```

Going in the opposite direction is even simpler. The `string` class has a constructor for building strings from character arrays, so this takes place automatically most of the time:

```
void copyFile (string fileName);
:
copyFile ("input.dat"); // automatically converted to std::string
```

## 2.3 How do I convert a string to a number (or a number to a string)?

There are some quick-and-dirty functions for doing this in `<cstdlib>` for converting to numbers:

```
#include <cstdlib>
using namespace std;
:
int i = atoi("123");
double f = atof("3.14159");
```

Now, these functions actually convert from character arrays, not strings, so if you have a string you need to [convert to a character array](#):

```
#include <cstdio>
#include <string>
using namespace std;
:
string s1 = "123";
string s2 = "3.14159";
:
int i = atoi(s1.c_str());
double f = atof(s2.c_str());
```

If you need to go in the other direction, or if you need to deal with strings with unusual formatting, then there is a more general technique. In fact, you can convert *any* datatype that has `<<` and `>>` I/O operators to and from strings by reading from and writing into a string. The `istringstream` is an input stream that reads from a string:

```
#include <sstream> // provides istringstream and ostringstream
#include <string>
using namespace std;
:
string s1 = "123 3.14159";
:
istringstream in (s1); // create a stream reading from s1
int i;
double f;
in >> i >> f; // i will contain 123 and f will contain 3.14159
```

The `ostringstream` is an output stream that writes into a string:

```
#include <sstream> // provides istringstream and ostringstream
#include <string>
using namespace std;
:
string s1;
ostringstream out (); // create a stream writing into a string
int i = 245;
double f = 1.2;
out << i << ":" << f;
string s = out.str(); // Retrieve the string we have written
cout << s << endl; // Prints "245:1.2"
```

Again, let me point out that this `stringstream` approach can be used to convert between `string` and *any* data type that you can read and write.

## 2.4 What is the difference between a declaration and a definition?

Pretty much everything that has a “name” in C++ must be declared before you can use it. Many of these things must also be defined, but that can generally be done at a much later time. You declare a name by saying what kind of thing it is:

```
const int MaxSize;           // declares a constant
extern int v;                // declares a variable
void foo (int formalParam); // declares a function (and a formal parameter)
typedef Bar* BarPointer;    // declares a type name
using BarPointer = Bar*;     // declares a type name (same as previous example)

class Bar{ ... };           // declares a class
{
:
void bar (int formalParam); // declares a member function within a class
:
int z;                      // declares a data member within a class
};
```

In most cases, once you have declared a name, you can write code that uses it. Furthermore, a program may declare the same thing any number of times, as long as it does so consistently. That’s why a single .h file can be included by several different .cpp files that make up a program — most .h files contain only declarations. You define constants, variables, and functions as follows:

```
const int MaxSize = 1000;      // defines a constant
int v;                        // defines a variable
void foo (int formalParam) {++formalParam;} // defines a function

void Bar::bar(int formalParam); // defines a member function within a class
```

A definition must be seen by the compiler once and only once in all the compilations that get linked together to form the final program.

- data types and classes are only declared, never defined.
- data members of classes are only declared. They do not have separate definitions unless they are declared as `static`.

A definition is itself also a declaration (exception: member functions within classes).

- If you define something that hasn’t been declared yet, that’s OK. The definition will serve double duty as declaration and definition.

However, such declarations/definitions are generally limited to use within the single .cpp file where they appear.

## 2.5 What goes in a .h file? What goes in a .cpp file?

The short answer is that a .h file contains shared declarations, a .cpp file contains definitions and declarations that you don’t want to share..

It’s important that you understand [difference between declarations and definitions](#).

- A .h file is intended to be #included from many different .cpp files that make up a single program. In fact, the earliest stage of compilation, the preprocessor, actually replaces each #include by the full contents of the included file. Consequently, a .h file may be processed many times during the compilation of a single program, and should contain only declarations.
- A .cpp file is intended to be compiled once for any given build of the program. So the .cpp file can have any declarations that it doesn’t need to share with other parts of the program, and it can have definitions. But the main purpose of a .cpp file is to contain definitions that must only be compiled once. The most common example of this would be function bodies.

**Never, ever, ever** name a .cpp file in an #include.

That defeats the whole purpose of a C++ program structure.

- .h files are #included;
- .cpp files are compiled.

## 2.6 What is an inline function?

When we define a function, it is usually compiled into a self-contained unit of code. For example, a function

```
int foo(int a, int b)
{
    return a+b-1;
}
```

would compile into a block of code equivalent to

```
stack[1] = stack[3] + stack[2] - 1;
jump to address in stack[0]
```

where the “stack” is the [runtime stack](#) a.k.a. the [activation stack](#) used to track function calls at the system level, stack[0] is the top value on the stack, stack[1] the value just under that one, and so on. A function call like

```
x = foo(y, z+1);
```

would be compiled into a code sequence along the lines of

```
push y onto the runtime stack;
evaluate z+1;
push the result onto the runtime stack
push (space for the return value) onto the runtime stack
save all CPU registers
push address RET onto the runtime stack
jump to start of foo's body
RET: x = stack[1]
pop runtime stack 4 times
restore all CPU registers
```

As you can see, there’s a fair amount of overhead involved in passing parameters and return address information to a function when making a call. The amount of time spent on this overhead is really all that large. If the function body contains several statements in any kind of loop, then the overhead is probably a negligible fraction of the total time spent on the call.

```
class Foo {
private:
    int bar;
public:
    int getBar ();
    void setBar (int);
};

int Foo::getBar () {return bar;}
void Foo::setBar (int b) {bar = b;}
```

But many ADTs have member functions that are only one or two lines long, and often trivial lines at that. For these functions, the overhead associated with each call may exceed the time required to do the function body itself. Furthermore, because these functions are often the primary means of accessing the ADT’s contents, sometimes these functions get called thousands of times or more inside the application’s loops.

For these kinds of trivial functions, C++ offers the option of declaring them as [inline](#).

```
class Foo {
private:
    int bar;
public:
    int getBar () {return bar;}
    void setBar (int);
};

inline
void Foo::setBar (int b) {bar = b;}
```

An inline function can be written one of two ways. **First**, it can be written inside the class declaration. **Second**, we can place the reserved word **inline** in front of the function definition written in its usual place outside the class declaration. When we make a call to an inline function, the compiler simply replaces the call by a compiled copy of the function body (with some appropriate renaming of variables to avoid conflicts). So, if we have

```
inline int foo(int a, int b)
{
    return a+b-1;
}
```

and we later make a call

```
x = foo(y, z+1);
```

This would be compiled into a code sequence along the lines of

```
evaluate z+1, storing result in tempB
evaluate y + tempB - 1, storing result in x
```

Most of the overhead of making a function call has been eliminated.

Inline functions can reduce the run time of a program by removing unnecessary function calls, but, used unwisely, may also cause the size of the program to explode. Consequently, they should be used only by frequently-called functions with bodies that take only 1 or 2 lines of code. For larger functions, the times savings would be negligible (as a fraction of the total time) while the memory penalty is more severe, and for infrequently used functions, who cares?

Inlining is only a recommendation from the programmer to the compiler. The compiler may ignore an inline declaration and continue treating it as a conventional function if it prefers. In particular, note that inlining of functions with recursive calls is impossible, as is inlining of most virtual function calls. Many compilers will refuse to inline any function whose body contains a loop. Others may have their own peculiar limitations.

## 2.7 Why is operator++ sometimes declared with an int parameter?

The ++ and -- operators are unusual in that they can be written in either prefix (++x, --x) or postfix (x++, x--) form. Whether you write ++x or x++, the value of x is increased by 1. But when you write a ++ expression inside another expression, its return value depends on whether you used the prefix or postfix form.

- The prefix form returns the value *after* the increment or decrement was performed.
- The postfix form returns the value *before* the increment or decrement was performed.

So, for example, the code

```
int i = 0;
int j = 0;
cout << ++i << ' ' << j++ << endl;
cout << i << ' ' << j << endl;
```

would print

```
1 0
1 1
```

Now that's all very well for the builtin ++ for int, but what happens when we write ++ for our own classes? Like most operators, ++ can be thought of as a shorthand for a function named operator++, so it's not too hard to see that we can say:

```
class SomethingWeCanIncrement
{
    ...
    SomethingWeCanIncrement& operator++ ();
    ...
};
```

but, somewhat late in the game, the designers of C++ realized that they had neglected to provide a syntax for indicating whether a unary operator was prefix or postfix. This is only a problem for ++ and --, because these are the only operators that can be written in both forms. The solution they came up with is a complete kludge. If you declare

```
const MyIncrementableClass operator++();
const MyIncrementableClass operator--();
```

you are declaring the *prefix* operators. If you declare

```
MyIncrementableClass operator++(int);
MyIncrementableClass operator--(int);
```

you are declaring the *postfix* operators.

What do you *do* with the int parameter for the postfix operators?

*Absolutely nothing!* It's just a dummy parameter used to distinguish the prefix and postfix forms. Finally, note that the prefix forms return a reference. The postfix forms return a non-reference value. That's because the prefix forms are returning the value that has been incremented/decremented. That value exists, so it's easy to return:

```
class MyIncrementableClass {
    ...
    const MyIncrementableClass operator++() {
        ...
        // do what you need to do to increment it
        ...
        return *this;
    }
    ...
}
```

On the other hand, the postfix form returns the value before the increment/decrement takes place. Usually the only way to do that is to make a copy of that value first, then do the increment, then return the copied value.

```
class MyIncrementableClass {
    ...
    const MyIncrementableClass operator++(int) {
        MyIncrementableClass clone = *this; // save old value
        operator++(); // increment this
        return clone; // return the old value
    }
    ...
}
```

## 2.8 What is the Rule of the Big 3?

The Big 3 in C++ class design are the copy constructor, destructor, and assignment operator.

The *Rule of the Big 3* states that, if you ever find yourself needing to provide your own version of any one of the Big 3, you should provide your own version of all three.

The C++ compiler will provide its own version of these if the programmer does not do so. However, the compiler-generated versions for a class are usually inappropriate if

1. The class has data members that are pointers, and
  2. The pointers denote data that should not logically be shared.
- Failure to provide a destructor when one is needed will result in significant memory leaks.
  - Failure to provide a copy constructor or assignment operator when one is needed will likely result in program crashes or incorrect output due to unwanted data sharing.
  - Providing only one or two of the Big 3 is even more likely to cause program crashes.

Like most such rules, there can be exceptions, but programmers should only violate this rule knowingly and after full consideration of the possible consequences.

## 2.9 I have other C++ questions.

- [cplusplus.com](http://cplusplus.com) has an extensive set of tutorials and reference material on C++
- [cppreference.com](http://cppreference.com) is also an excellent reference site.
- The [C++ FAQ](#) (Frequently Asked Questions) has a wealth of information about object-oriented programming in general and about C++ specifically.

# 3 C++ Compilers and IDEs

## 3.1 How can I install a compiler and IDE on my own PC?

See [Installing a C++/Java IDE on Your Own PC](#)

## 3.2 How do I use a make file in Eclipse or Code::Blocks?

A *make file* contains instructions on how to compile the code of a project. These instructions are interpreted by a program called `make`, which is [covered in CS252](#). `make` is what we call a *build manager*, a program designed to automate the steps in building a software project.

If you have a complicated project, creating a make file can allow you to avoid typing long sequence of commands to compile all of your code.

Alternatively, some instructors provide make files with their assignments. You will recognize these because they are titled `makefile` or `Makefile`. Using an instructor-supplied make file can help guarantee that you are compiling your code using the same settings the instructor will use when grading your code.

But most IDEs have their own build managers “built in”. In many cases that will be good enough, particularly if you take the time to [set your compiler options](#) to the appropriate values.

But sometimes you will really need to or want to use that make file.

### 3.2.1 Using make files from the command line.

Assuming that you are on a machine with the `make` program installed, just type

```
make
```

to build your project. If that does not work, try

```
make all
```

Type

```
make clean
```

to clean up (remove anything produced automatically, leaving the source code unchanged).

“all” and “clean” above are examples of make file *targets*, options written in to the make file to trigger a series of commands. Actually, there aren’t hard and fast rules on the target names. But most people who write make files set them up so that, by default, `make` and/or `make all` builds the project and `make clean` clean up after it. If those don’t work for you, you’ll have to ask the author of the make file or read it yourself.

### 3.2.2 Using make files from Eclipse

When you create a C++ project in Eclipse, among your early options is create an “Executable” or a “Makefile project”. Choose “Makefile project” ... “Empty Project”. (“Empty” in this case means that you already have a makefile and some C++ code and don’t want Eclipse to set up a “hello-world” style starter for you.)

If you have already created your project as an “Executable” (which means that you requested that Eclipse use its own built-in build manager), this is easily remedied.

1. Take note of what directory the project is stored in. If you aren't sure, use Project -> Properties to find out.

2. In the Project Explorer list on the left, right-click on your existing C++ project and select "Delete".

Now, we only want to delete the project settings from Eclipse, not your source code. So when the box pops up to confirm the deletion, make sure that the "Delete project contents on disk" is **not** checked.

3. Now go back and add that same directory as a new C++ Empty Makefile project, as described earlier.

### 3.2.3 Using make files in Code::Blocks

1. Create your C++ project as usual.

2. Then go to Project -> Properties -> Project settings, and by "Makefile:", make sure that the file listed is the appropriate location and spelling of of your make file, then put a check in the "This is a custom Makefile" box.

3. Now go to the "Build targets" tab.

By default, Code::Blocks uses make file targets "Debug" and "Release", but those are rarely seen in practice.

Use the Add button to add the "real" targets of your makefile. (As noted earlier, "all" and "clean" are among the most common, but you may have to read the make file to be sure.)

Optionally, you can use the "Delete" button to remove the "Debug" and "Release" targets.

4. Under "Output filename", enter the name (and location) of the executable produced by the makefile.

- If you don't know how to read make files, you may have to run it once to find out.

5. Click OK to save the setting changes.

To compile your code, go to the Build menu and then Select target to select "all". Then click the usual build button.

To clean up your project directory, go to the Build menu and then Select target to select "clean". Then click the same build button.

## 3.3 How do I activate C++14 features when compiling?

By now, much of the C++ code you are likely to encounter will make use of features introduced in the C++ 2014 standard (there is also a 2017 standard, but compilers have not caught up with that). Most compilers default to the 1998 language definition. You will need to be sure that you are compiling with the appropriate options set.

In the GNU g++ and Clang compiler suites, the relevant compiler option to activate C++14 features is

```
-std=c++14
```

If you are typing your compiler commands directly at the command line, just add this.

But, more likely, you are using some sort of builder, either make or an IDE (Code::Blocks or Eclipse) or a makefile invoked from an IDE.

### 3.3.1 make Files

If you are using make, your makefile will generally have a list of options provided explicitly for your uses of g++. Just add -std=c++14 to that list.

### 3.3.2 CodeBlocks

1. From the Settings menu, select "Compiler..."

2. You should be on the "Global compiler settings" tab. If not, select it.

3. Put a checkmark by the option:

```
Have g++ follow ... C++14 ISO ...
```

4. Click OK to accept the changes. Then from the "Project" menu select "Project Clean", then rebuild your project.

### 3.3.3 Eclipse (default C++ builder)

If you are using Eclipse and specified "C++ project", you have requested that Eclipse use its default builder. You set your compilation options from the project settings menu.

1. Right-click on your project and select "Properties" or select "Properties" from the "Project" menu.

2. Click on "C/C++ Build" then "Settings". Select the "...C++ Compiler" that you are using.

3. Select "Dialect", and choose "ISO C++14".

4. Click OK to accept the changes. Then from the "Project" menu select "Project Clean", then rebuild your project.

### 3.3.4 Eclipse (makefile project)

If you are using Eclipse and specified “Makefile project with existing code”, you have requested that Eclipse run “make” to build your code. Edit your `makefile` as described above.

You may find, however, that the Eclipse editor reports errors as you are editing your code that do not actually arise when you compile/build. That may be because the Eclipse editor is using pre-2014 rules. To fix,

1. Right-click on your project and select “Properties” or select “Properties” from the “Project” menu.
2. Select “C/C++ General” => “Preprocessor Include Paths...”. Look at the “Providers” tab. Look for the entry that seems to most directly describe your compiler (e.g., “CDT GCC Built-in Compiler Settings CygWin” or “CDT GCC Built-in Compiler Settings MinGW”) and use the “MoveUp” button to position this at the top of the list
3. Click OK to accept the changes. Then from the “Project” menu select “C/C++ Index” => “Rebuild”. Repeat the rebuild if necessary.
4. From the “Project” menu select “Project Clean”, then rebuild your project.

## 3.4 How do I set compiler flags and options?

For most assignments, I recommend the following options:

```
-g -std=c++14 -Wall -D_GLIBCXX_DEBUG -fsanitize=address
```

- The `-g` option adds the information necessary to run the debugger later, if you should need to.
- The `-std=` option adds C++14 features as described [here](#how-do-i-activate-c14-features-when-compiling).
- The `-Wall` option adds warning messages for a variety of common errors. They are warnings, not errors, and will not actually prevent your code from compiling. They can alert you to potential problems, but *only if you actually read and consider the warning messages*.
- The `-D...` option causes accesses to the `std::` library containers such as `std::array`, `std::vector`, & `std::deque` to be bounds-checked so that an run-time error message will be issued if you try to access data outside the legal bounds of that container. It also adds protection when using `iterators`, checking to be sure that each iterator actually access a legal position within the container that it refers to.
- The `-fsanitize...` option adds code to your program to test for many common pointer manipulation errors, including memory leaks, multiple deletions of the same address, and some uses of uninitialized pointers.

If you are typing your compiler commands directly at the command line, just add this.

But, more likely, you are using some sort of builder, either `make` or an IDE (Code::Blocks or Eclipse) or a `makefile` invoked from an IDE.

### 3.4.1 make Files

If you are using `make`, your `makefile` will generally have a list of options provided explicitly for your uses of `g++`. Edit them to match the above.

### 3.4.2 CodeBlocks

1. From the Settings menu, select “Compiler...”
2. You should be on the `Global compiler settings` tab. If not, select it. Within that, look for a “Compiler Flags” tab. Again, you are probably already on it. If not, select it.
3. Put a checkmark by each of the following:

```
Have g++ follow ... C++14 ISO ...
Enable all common compiler warnings
Enable extra compiler warnings
Enable Effective-C++ warnings
```

4. Now go to the `#defines` tab. In the text box enter, on a line by itself,

```
_GLIBCXX_DEBUG
```

Be sure to get both of the underscore (\_) characters in there.

5. Click OK to accept the changes. Then from the “Project” menu select “Project Clean”, then rebuild your project.

### 3.4.3 Eclipse (default C++ builder)

If you are using Eclipse and specified “C++ project”, you have requested that Eclipse use its default builder. You set your compilation options from the project settings menu.

1. Right-click on your project and select “Properties” or select “Properties” from the “Project” menu.

2. Click to open the “C/C++ Build” list, then click on “Settings”. Select the “...C++ Compiler” that you are using.
3. Select “Dialect”, and choose “ISO C++14”.
4. Select “Warnings” and check the options for -Wall and -Wextra.
5. Select “Miscellaneous” and add -Weffc++ to the “Other flags”.
6. Select “Preprocessor” and, on the “Defined symbols” box, use the icon with the green plus sign to add

`_GLIBCXX_DEBUG`

Be sure to get both of the underscore (\_) characters in there.

7. Click OK to accept the changes. Then from the “Project” menu select “Project Clean”, then rebuild your project.

### 3.4.4 Eclipse (makefile project)

If you are using Eclipse and specified “Makefile project with existing code”, you have requested that Eclipse run “make” to build your code. Edit your `makefile` accordingly.

You may find, however, that the Eclipse editor reports errors as you are editing your code that do not actually arise when you compile/build. That may be because the Eclipse editor is using pre-2014 rules. To fix,

1. Right-click on your project and select “Properties” or select “Properties” from the “Project” menu.
2. Select “C/C++ General” => “Preprocessor Include Paths...”. Look at the “Providers” tab. Look for the entry that seems to most directly describe your compiler (e.g., “CDT GCC Built-in Compiler Settings CygWin” or “CDT GCC Built-in Compiler Settings MinGW”) and use the “MoveUp” button to position this at the top of the list
3. Click OK to accept the changes. Then from the “Project” menu select “C/C++ Index” => “Rebuild”. Repeat the rebuild if necessary.
4. From the “Project” menu select “Project Clean”, then rebuild your project.

## 3.5 How do I supply command line parameters when executing my programs?

Often you will be writing programs that take “*command line parameters*” as part of their inputs.

Of course, if you are executing a program from the command line, you just type the parameter values on the command line (hence the name!), e.g.,

`./myProgram ../testData/myTestData.txt 23`

But what if you are launching your program from an IDE?

### 3.5.1 Code::Blocks

Before running your program, from the Project menu, select “Set programs’ arguments...”. In the “Program Arguments” box, enter your parameters, one per line. Click OK when done.

Run the program.

### 3.5.2 Eclipse

1. After you have successfully compiled your code, click on your executable binary in the Project Explorer). Then, from the Run menu, select Run configurations...

You’ll see a list of different project types, one of which should be “C/C++ Application”.

- Underneath that, you might already have a configuration for this program (Clicking on a configuration will show you what project and what executable file it is associated with.
  - If not, select “C/C++ Application”, then at the top of the column click on “New launch configuration”.
2. On the “Arguments” tab, enter your command line parameters into the “Program Arguments” box.
  3. Click Run to save these arguments and launch your program.

[Video of this procedure](#) (for Java, but the procedure is similar)

## 3.6 How do I get my program to read from a file instead of from the keyboard?

At the command line, this is called *input redirection* and is done by appending

`< location_of_input_file`

to the end of the command launching the program. You can read [more about this in CS252](#), and it works the exact same way in a Windows command-line session.

What if you are launching your program from an IDE?

### 3.6.1 Code::Blocks

Can't really be done.

What you can, do, however, is to

1. Open your file of input text in the Code::Blocks editor.
2. Run the program.
3. Then, when it pauses for input, instead of typing the input directly, copy-and-paste lines of text from the input file in the editor into the running program.
  - You can usually copy-and-paste many lines of input at once, because the underlying run-time system will simply buffer up unused input until later input statements are reached.
4. Indicate the end of input, if necessary, by typing the appropriate character depending on your operating system: Ctrl-Z for Windows, Ctrl-D (at the start of a new line) for Linux and MacOs,

### 3.6.2 Eclipse

After you have successfully compiled your code, click on your executable binary in the Project Explorer). Then, from the Run menu, select Run configurations...

You'll see a list of different project types, one of which should be "C/C++ Application".

- Underneath that, you might already have a configuration for this program (Clicking on a configuration will show you what project and what executable file it is associated with).
- If not, select "C/C++ Application", then at the top of the column click on "\_New launch configuration".

On the "Common" tab, under "Standard input and output", click to select the "Input" box, then use one of the three buttons on the line below to select the file of input data you want supplied as the standard input.

Click Run to save these arguments and launch your program.

## 4 C++ Error Messages

### 4.1 AddressSanitizer: attempting free on address which was not malloc()-ed

The Address Sanitizer (activated when compiling with a -fsanitize=... option) has detected that the program attempted to delete a block of memory that

- it had already deleted once before, or
- was statically allocated or obtained via application of the address-of (&) operator, or
- was allocated as an array, but not deleted as one (delete [] ...)

Examine the message details for the source code files and locations at which the problem was detected.

### 4.2 AddressSanitizer: heap-buffer-overflow

The Address Sanitizer (activated when compiling with a -fsanitize=... option) has detected that the program attempted to access data in a block of memory that has been properly allocated. This can be caused by any of

- Accessing an uninitialized pointer
- Storing past either end of an array
- Deleting a block of memory allocated as a single variable with the array-style (delete [] ...)

This error is often detected well after the offending action has taken place, making it fairly difficult to debug.

- May also be caused, indirectly, by a violation of the [Rule of the Big 3](#)

If you implement a destructor without providing a working copy constructor and assignment operator at the same time, there is a good chance that you will wind up deleting the same block of memory multiple times due to undesired shallow copies of pointers.

### 4.3 AddressSanitizer: heap-use-after-free

The Address Sanitizer (activated when compiling with a -fsanitize=... option) has detected that the program attempted to access data in a block of memory that has been deleted.

- May also be caused, indirectly, by a violation of the [Rule of the Big 3](#)

### 4.4 AddressSanitizer: NullPointerAccess

The Address Sanitizer (activated when compiling with a `-fsanitize=...` option) has detected that the program attempted to retrieve or store data via a null pointer.

Examine the message details for the source code files and locations at which the problem was detected.

## 4.5 AddressSanitizer: SEGV

SEGV stands for [segmentation error](#).

## 4.6 attempt to subscript container with out-of-bounds index

Indicates that one of the `std::` library container functions has detected an attempt to access or store data that is out of range (e.g., trying to get the N'th item from a vector that only holds N-1 items).

The remainder of the error message will list files and line numbers that may help you trace the point at which the problem was detected.

## 4.7 attempt to dereference a singular iterator.

Indicates that one of the `std::` library container functions has detected an attempt to access or store data via an uninitialized iterator or an iterator that does not denote a valid position within a container.

## 4.8 attempt to dereference a past-the-end iterator.

Indicates that one of the `std::` library container functions has detected an attempt to access or store data via an uninitialized iterator or an iterator that does not denote a valid position within a container.

## 4.9 ...discards qualifiers...

The message

In *some-function-name*, passing const *some-type-name* ... discards qualifiers

occurs when you try to pass a `const` object to a function that might try to change the object's value. For example, if you have a class `C`:

```
class C {
public:
    C();
    int foo ();
    void bar (std::string& s);
    ...
}
```

and you try to compile the following code:

```
void baz (const C& c1, C& c2, const std::string& str)
{
    int i = c1.foo();    // error!
    int j = c2.foo();    // error!
    c2.bar(str);        // error!
    ...
}
```

then a C++ compiler should flag the 1st and 3rd line indicated above. The `g++` compiler will say something along the lines of

```
In function 'void baz(const C&, C&, const std::string&)':
  passing 'const C' as 'this' argument of 'int C::foo()' discards qualifiers

In function 'void baz(const C&, C&, const std::string&)':
  passing 'const std::string' as argument 1 of
  'void C::bar(std::string&)' discards qualifiers
```

The first message complains that you have passed a `const` object as the left-hand parameter (implicitly named `this`) to the function `foo`, which has not promised to leave that parameter unchanged. You have, in effect, tried to discard the “qualifier” (the word “`const`”) in the `const C&` datatype. The second message makes a similar complaint about the string parameter being passed to `bar`. Again, the object being passed is marked as `const`, but the declaration of `bar` suggests that `bar` is allowed to change the string it receives as a parameter. To get rid of this message, you must examine what it is you are trying to do and determine whether:

- The declarations of the functions you are trying to call are incorrect. Perhaps `foo` and `bar` really *should* promise to leave those parameters unchanged. I.e., perhaps they should have been declared like this:

```
class C {
public:
    C();
    int foo () const; // don't change the object it's applied to
```

```
void bar (const std::string& s); // don't change s
{
```

- or, perhaps the application code is declaring things as **const** that it really *does* want to change:

```
void baz (C& c1, C& c2, std::string& str)
{
    int i = c1.foo(); // ok
    int j = c2.foo();
    c2.bar(str);
    :
```

- or, perhaps the application really is not supposed to change those parameters, and you will need to find some other way to accomplish what you need to do. For example,

```
void baz (const C& c1, C& c2, const std::string& str)
{
    C c1Copy = c1;
    int i = c1Copy.foo(); // ok
    int j = c2.foo();
    :
```

## 4.10 LeakSanitizer: detected memory leaks

The LeakSanitizer (activated when compiling with a `-fsanitize=...` option) has detected that the program allocated blocks of memory that were never properly deleted.

Examine the message details for the source code files and locations at which the memory was allocated. (Of course the messages can't tell you where you are missing a `delete` statement, because they can't point to code that you haven't written!)

## 4.11 need typename before..., also “warning: ... is implicitly a typename” or “warning: implicit typename is deprecated”?

This error arises in certain uses of template parameters (or of names that are `typedef`'d in terms of a template parameter). For example,

```
template <class Container, class T>
void fillContainer (Container& c, T value)
{
    Container::iterator b = c.begin();
    Container::iterator e = c.end();
    fill (b, e, value);
}
```

will probably get one of these messages from `g++` complaining about the mentions of “`Container::iterator`”. The fix is

```
template <class Container, class T>
void fillContainer (Container& c, T value)
{
    typename Container::iterator b = c.begin();
    typename Container::iterator e = c.end();
    fill (b, e, value);
}
```

## 4.12 No matching function ..., No match for...

This is a variation on the messages saying a symbol is undeclared. In particular, you will get this message when you call a function, and there are one or more functions with that name, but your set of actual parameters' data types do not match up with the formal parameter list of any of the declared functions.

Either you are correctly calling a function that you have not declared, or you are trying to call a declared function with the wrong kind of parameters.

This can be one of the longer error messages you will ever get, as `g++` tries to list out *all* the functions with the same name that it knows, with the data types of all their parameters, as a way of showing you all your existing options. Although the list can be a bit daunting, if you are running in a support environment (e.g., emacs) that lets you step from message to message while displaying the relevant line of code, this list can actually be quite helpful.

## 4.13 ‘nullptr’ was not identified in this scope

You are compiling a program that uses C++ 2014 features with compiler settings for the 1998 C++ standard.

[Change your compiler settings](#) to use the C++14 standard.

## 4.14 Segmentation error? Segmentation fault? Bus error? Null reference error?

These are all various errors signaled by the underlying operating system when your program tries to retrieve from, store into, or execute instructions from an address that either doesn't exist or is reserved for another program. In practical terms, these kinds of messages almost always arise because of

- a pointer that is null, uninitialized, or pointing to an object that has already been `delete`'d, or
- array indices that are out-of-bounds, or
- iterators that are uninitialized or out of bounds.
- May also be caused, indirectly, by a violation of the [Rule of the Big 3](#)

To debug these problems, concentrate on your pointers, arrays, and iterators. Add debugging output or use a debugger to find out which one is being used when the crash occurs. Then work backwards to figure out why that pointer/index/iterator has an invalid value. Keep in mind that you might not be using a pointer, array, or iterator directly, but might be using (abusing?) a function or class that does.

## 4.15 terminate called after throwing an instance of ‘std::out\_of\_range’

Indicates that one of the `std`: library container functions has detected an attempt to access or store data that is out of range (e.g., trying to get the N'th item from a vector that only holds N-1 items).

The remainder of the error message will list files and line numbers that may help you trace the point at which the problem was detected.

## 4.16 Undeclared/undefined names and symbols

First, look *very* closely at the error messages. Does it say “undeclared” or “undefined”? These are two very different things, and [understanding the difference](#) is the key to fixing the problem.

So, if the compiler says that a function is *undeclared*, it means that you tried to use it before presenting its declaration, or forgot to declare it at all.

The *compiler* never complains about definitions, because an apparently missing definition might just be in some other file you are going to compile as part of the program.

But when you try to produce the executable program by linking all the compiled .o or .obj files produced by the compiler, the *linker* may complain that a symbol is *undefined* (none of the compiled files provided a definition) or is *multiply defined* (you provided two definitions for one name, or somehow compiled the same definition into more than one .o or .obj file).

For example, if you forgot a function body, the linker will eventually complain that the function is undefined. If you put a variable or function definition in a .h file and include that file from more than one place, the linker will complain that the name is multiply defined.

Undefined symbol messages seem to happen most often with functions. The possible causes are:

- You have forgotten to supply a body for a function.
- You misspelled the name of the function in the body, so the compiler thinks you are supplying a body for a completely different function.
- You forgot to compile the .cpp file that has the function body, or forgot to include the resulting .o file when you linked the rest of the program together.
  - If you are working in an IDE, you may have forgotten to add that .cpp file to your project.
- You [gave the wrong compilation command](#), and told g++ to treat a single .cpp file as the entire program even though there are [multiple .cpp files making up the program](#).

# 5 Mathematics and Notation

## 5.1 $\forall$ ? $\exists$ ?

Some of the notation used in logical expressions:

$\forall$	“for all”
$\exists$	“there exists”
$\exists  $	“such that”
$\Rightarrow$	“implies” (Alternatively, you can read $A \Rightarrow B$ as “if A, then B”)

So, for example, the formula

$$\exists c_1, n_0 \mid n > n_0 \Rightarrow t(n) \leq c_1 * (c f(n))$$

reads as “there exists some values for  $c_1$  and  $n_0$  such that, whenever  $n > n_0$ , then  $t_n$  is less than or equal to  $c_1 * c * f(n)$ .”

## 5.2 Basic Probability (probably just what you wanted)

A *probability* is a number between 0 and 1 that expresses the chances of something happening. For example, in rolling an ordinary 6-sided die, the probability of getting a ‘5’ is 1/6. The probability of getting an even number is 3/6 or 0.5. The probability of getting a ‘9’ is 0.0 — it can’t possibly happen with a single roll of an ordinary die. The probability of getting *some* number between 1 and 6 is 1.0.<sup>1</sup>

Some key ideas about probabilities:

- Every probability lies between 0 and 1.
  - 0 means it never happens.
  - 1 means it always happens.
- If you know all the possible events and add up the probabilities, the sum must be 1.0.[^independence]

For example, on a single roll of a six-sided die, the probability of seeing a “1” is  $1/6$ , the probability of seeing a “2” is  $1/6$ , and so on for all six possible outcomes.

So we have six total events, each with a probability of  $1/6$ , which adds up to 1.0.<sup>2</sup>

- If there are only two possible outcomes to a test, and the probability of the first outcome is  $p$ , then the probability of the other is  $1 - p$ . (This follows from the fact that the two probabilities have to add up to 1.)
- If the probability of seeing an event in one try is  $p$ , and we make  $k$  independent tries,

- the probability of seeing that event every time is  $p^k$ .
- The probability of never seeing that event in any of the  $k$  tries is  $(1 - p)^k$ .
- The probability of seeing that event at least once is  $1 - (1 - p)^k$ .

For example, the chances of a die coming up ‘1’ on all of 5 successive throws is  $(1/6)^5 = 0.00013$ . The probability of never seeing a ‘1’ in 5 throws is  $(1 - (1/6))^5 = (0.833)^5 = 0.402$ . The probability of seeing at least one ‘1’ in five throws is  $1 - (1 - 1/6)^5 = 0.598$ .

- Suppose that we are looking at a set of possible numeric values for some quantity  $X$ , and we know the values of  $X$  will be drawn from a set

$$x_1, x_2, \dots, x_n.$$

Suppose that the probability of seeing  $x_1$  is  $p_1$ , the probability of seeing  $x_2$  is  $p_2$ , and so on. Then the average or expected value of  $X$  is

$$X_{\text{average}} = \sum_{i=1}^n p_i x_i$$

(Notice that if all the values are equally likely, then  $p_i = (1/n)$  and this formula simplifies to

$$X_{\text{average}} = \sum_{i=1}^n (1/n) x_i = \frac{\sum_{i=1}^n x_i}{n}$$

which is the “normal” formula for taking an average of  $n$  things.)

- Suppose that we are waiting on an event that has a probability  $p$  of occurring on any one try. On average, we can expect to wait  $1/p$  tries before seeing that event.

For example, if we are throwing a six-sided die and waiting to see a ‘4’, we know that ‘4’ has a  $1/6$  probability of coming up on a single throw. So we should expect to wait, on average,  $\frac{1}{(1/6)} = 6$  throws before seeing a ‘4’.

## 5.3 Yucch! Nobody told me I needed to know about logarithms.

Suppose that  $a^b = c$ . Then we say that  $b$  is the logarithm to the base  $a$  of  $c$ , written as  $\log_a c$ . In other words,  $\log_a c$  is the power that  $a$  must be raised to, in order to produce  $c$ .

Most uses of logarithms outside of Computer Science focus on  $\log_{10}$ , often abbreviated as  $\log$ , and on  $\log_e$  (where  $e$  is the irrational number  $2.71828\dots$ ), often abbreviated as  $\ln$ .

In Computer Science, however, most uses of logarithms arise from considering a situation where we start with  $N$  objects, divide that set of objects in half, divide that half in half, divide that fourth in half, and so on:

$$N, N/2, N/4, \dots, 2, 1$$

The question is: how many divisions in half can we perform until we get down to a single object? The answer to this question comes from considering the sequence of numbers in reverse:

$$1, 2, 4, \dots, N/2, N$$

and realizing that at each step, we are moving to a progressively higher power of 2:  $2^0, 2^1, 2^2, \dots, N/2, N$ . How many such doublings can we do before reaching  $N$ ? This question is the same as asking “to what power can we raise 2 until we reach  $N$ ?” And the answer, by definition, is  $\log_2 N$ .

Logarithms to the base 2 are so common in Computer Science that we use the abbreviation  $\log$  to stand for  $\log_2$ , unlike the rest of the scientific world where that stands for  $\log_{10}$ .

But when we are doing a big-O analysis, the base of the logarithm does not matter, because of the conversion formula between logarithms of different bases:

$$\log_a x = \frac{\log_b x}{\log_b a}$$

So if we have, for example,  $\log_{10} N$ , we can convert this to a base 2 logarithm

$$\log_2 N = \frac{\log_{10} N}{\log_{10} 2}$$

But  $\log_{10} 2$  is a constant, and we can drop constant multipliers within  $O(\dots)$ . So  $O(\log_{10} N) = O(\log_2 N)$ . We can generalize this argument to cover any two bases, and so we just write  $O(\log N)$  without even worrying about the base.

Some useful formulae:

$$\log_a x = \frac{\log_b x}{\log_b a}$$

$$\log(x * y) = \log x + \log y$$

$$\log(x^y) = y \log x$$

## 5.4 Simplifying Summations

Some useful formulae:

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1$$

$$\sum_{i=0}^n a^i = \frac{a^{n+1} - 1}{a - 1}$$

$$\sum_{i=0}^n i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=0}^n (1/i) = O(\log n)$$

$$\sum_{i=0}^{\infty} \frac{1}{2^i} = 2$$

## 5.5 How can I type mathematics into Blackboard?

Let's assume that you are in Blackboard taking a quiz or test, or making a Forum (Discussion Board) post, or working on an assignment. You are in the Blackboard editor and discover that you need to do a bit of math, or maybe a lot of math.

You have options:

- Superscripts and subscripts are easy: Just select the superscript or subscript text and use the **super/subscript** buttons to raise or lower the text. This works fine with things like  $x^2$  or  $x_i$ . It's not as good when you need both superscripts and subscripts on the same expression. You can get  $x^2_i$  or  $x_i^2$  this way, but neither looks quite as good as  $x_i^2$ .
- Sometimes you just need a special character or two. If your PC operating system has a character insertion utility, try it. You can also try using the **HTML** button and insert some special characters directly. For example, I often type  $\&rarr;$  to get  $\rightarrow$ ,  $\&RArr;$  to get  $\Rightarrow$ , and  $\&epsilon;$  to get  $\epsilon$ .



The screenshot shows the Blackboard editor interface. At the top, there is a subject field labeled "Subject" and a "My answer" input field. Below this is a "Message" area with a placeholder "For the toolbar, press ALT+F10 (PC) or ALT+FN+F10 (Mac)". The toolbar below the message area contains various icons for text formatting, including bold, italic, underline, and various alignment and style options. A "Mashups" button is also present. The message area contains the text "I believe the code in question is  $O(n^2)$  because  $\sum_{i=0}^n i = n(n-1)/2$ " and a "Path: p" and "Words: 14" indicator at the bottom right.

**Blackboard error warning:** the Blackboard spell check facility is not your friend here. Basically, whenever it thinks you have misspelled something, it surrounds the “misspelled” word with the HTML element `<span>`. Even if you later correct the spelling, the `<span>`s remain. This can make the underlying HTML so ugly that it is all but impossible to edit.

*Turn off the **spell-checker** before typing any mathematics in Blackboard.*

- When you need more than a character or two, Blackboard has a built-in **equation editor**, similar to the one found in most Word processors.

Most math is actually a mixture of actual mathematics expressions and plain text explaining/discussing those expressions. Never use the equation editor for plain text. Get in and out as you move back and forth from math to text. Typesetting plain text as if it were mathematics often leads to a real mess, because in mathematics, two characters next to each other represents a product of two variables (even if there is a space between them) and a math editor will typeset them accordingly.

- If you have a lot of math, particularly a derivation or a proof with multiple lines of mathematics that change just slightly from line to line, you may find the Blackboard equation editor awkward because copy and pasting, then making slight alterations to the pasted expression, can be awkward.

Another option is to type your expression in LaTeX format between a pair of double dollar signs. For example,  $\sum_{i=0}^n i$  typesets as  $\sum_{i=0}^n i$ .

You can find a brief rundown on LaTeX notation [here](#).

You can easily preview your math expression by placing the cursor anywhere inside it, clicking on  button, then cancelling the preview box that pops up.

**Blackboard error warning:** the Blackboard spell check facility will break your math. Basically, whenever it thinks you have misspelled something, it surrounds the “misspelled” word with the HTML element `<span>`. Even if you later correct the spelling, the `<span>`s remain. And since most math expressions look misspelled from the point of view of an English language spell checker, it puts a **lot** of those `<span>` elements inside most math expressions. This breaks your math badly.

*Turn off the spell-checker before typing any mathematics in Blackboard.*

1: Well, I suppose the die could come to rest on an edge, or a black hole could suddenly open up and swallow the die, so make that probability 0.9999 . . .

2: OK, that's a bit of an over-simplification. Technically, the events that we're talking about have to be “independent” of one another. That's why we add up the probability of the die coming up 1, 2, . . . , 6 but do not also add in the probability of the die coming up with an even number. The “coming up even” event is not independent of the event “coming up ‘2’”: when a die comes up ‘2’ it is also coming up even.)

## 6 Miscellaneous

### 6.1 How do I download a file from a link on a web page?

Left-clicking on the link will often show you the file in the browser, but not provide an easy way to download the file itself.

Instead, right-click on the link and in the pop-up menu, look for something along the lines of “Save link as” (exact phrase depends on what browser you are using). Selecting that should allow you to download the file.

### 6.2 I can't save makefiles (or other files with no extension or unusual extensions) without adding an extension.

This is a Windows-specific problem. Windows used to be stupid about file extensions (the ‘.’ followed by 1-3 letters at the end of a file name), requiring all non-folder files to have one. Although Windows itself has gotten better about this, a lot of Windows software (including popular browsers) remains stupid.

The workaround is simple:

1. Go ahead and add a `.txt` or other extension to the file name when you save it.
2. Then immediately rename the saved file back to its intended name.

### 6.3 Will we be able to see solutions to quiz/exam questions? Why is my quiz/exam score so low? etc.

Most quizzes and exams will contain questions cannot be (reliably) graded automatically. Therefore you will have to wait for the instructor to grade your submission before you have a final score.

- Typically, this will be a week or so after the due date.
- In the meantime, Blackboard may show you partial scores based upon the multiple-choice questions that it could grade on its own, or upon the questions that I have graded so far.

Anything still ungraded will be counted as zero.

Do **not** send me email complaining about your grade or asking about the solutions until Blackboard shows that your quiz/exam has actually been graded.

- You will know that your quiz/exam has been graded when the score shown in the Quiz/Exam Grades area (a.k.a. My Grades in Blackboard) turns into a link.
- Once that happens, you will be able to use that link to see your answers, my answers, your score, and possibly other notes from me.

