



San Diego State University



Department of Computer Science

## INSTRUCTOR NOTES

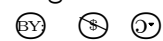
In  
*The C++ Programming Language*

Prepared by SHAWN HEALEY

# CS 310 Data Structures

DRAFT

©2019 Shawn Healey - sjhealey [at] sdsu [dot] edu.  
This work is licensed under a Creative Commons Attribution- ShareAlike  
3.0 License. To view a copy of this license visit:  
<http://creativecommons.org/licenses/by-sa/3.0/legalcode>.



# Contents

<b>I</b>	<b>The Language</b>	<b>1</b>
<b>1</b>	<b>Overview</b>	<b>3</b>
1.1	Systems and Application Programming . . . . .	4
1.2	Development Environment . . . . .	5
1.2.1	Secure Shell . . . . .	7
1.2.2	Visual and Visual Improved . . . . .	11
1.2.3	Secure Copy . . . . .	11
1.3	Compilation . . . . .	12
1.3.1	The Preprocessor . . . . .	12
1.3.2	Header Files . . . . .	15
1.3.3	Compiler . . . . .	18
1.3.4	Linker . . . . .	18
1.4	Exercises . . . . .	20
<b>2</b>	<b>Basic Language Syntax</b>	<b>21</b>
2.1	Data Types . . . . .	21
2.1.1	Variable Names and Standards . . . . .	22
2.1.2	Boolean Values . . . . .	24
2.1.3	Characters . . . . .	26
2.1.4	Integers . . . . .	27
2.1.5	Floating-point Numbers . . . . .	27
2.1.6	Auto . . . . .	29
2.1.7	Type Casting . . . . .	29
2.2	Arrays and Pointers . . . . .	30
2.2.1	Arrays . . . . .	30
2.2.2	Pointers . . . . .	34
2.2.3	Dereferencing . . . . .	35
2.2.4	References . . . . .	37
2.2.5	A Stack Frame . . . . .	38

2.2.6	Buffer Overflow . . . . .	38
2.3	Strings . . . . .	39
2.4	Statements and Expressions . . . . .	40
2.4.1	Conditional Statements . . . . .	41
2.4.2	Iteration Statements . . . . .	42
2.4.3	Evaluating Expressions . . . . .	42
2.5	Exercises . . . . .	42
<b>3</b>	<b>Functions</b>	<b>45</b>
3.1	Parameters . . . . .	47
3.1.1	Passing by Value . . . . .	47
3.1.2	Passing by Reference . . . . .	48
3.1.3	Default Arguments . . . . .	50
3.2	Return Values . . . . .	50
3.3	Recursion . . . . .	51
3.4	Function Pointers . . . . .	51
3.5	Lambdas, Anonymous Functions, and Closures . . . . .	51
3.6	Exercises . . . . .	54
<b>4</b>	<b>Input and Output</b>	<b>57</b>
<b>5</b>	<b>Classes</b>	<b>61</b>
5.1	Defining a Class . . . . .	61
5.1.1	Constructors . . . . .	62
5.1.2	Uniform Initialization . . . . .	63
5.1.3	Copy Constructor . . . . .	63
5.1.4	Destructors . . . . .	63
5.1.5	Access Modifiers and Specifiers . . . . .	63
5.2	Overloading Operators . . . . .	65
5.3	Class Hierarchy . . . . .	65
5.4	Exercises . . . . .	67
<b>6</b>	<b>Scope and Namespaces</b>	<b>73</b>
6.1	Scope . . . . .	73
6.2	Namespaces . . . . .	74
6.2.1	Java Packages . . . . .	77
<b>7</b>	<b>Templates and Genericism</b>	<b>79</b>

<b>II</b>	<b>Analysis</b>	<b>81</b>
<b>8</b>	<b>Complexity</b>	<b>83</b>
8.1	Empirical Analysis . . . . .	83
8.2	Theoretical Analysis . . . . .	84
8.2.1	Complexity Classes . . . . .	84
8.3	Exercises . . . . .	84
<b>9</b>	<b>Sorting and Searching</b>	<b>89</b>
9.1	Binary Search . . . . .	89
9.2	Selection Sort . . . . .	90
9.3	Insertion Sort . . . . .	91
9.4	Merge Sort . . . . .	94
9.5	Radix Sort . . . . .	96
9.6	Quick Sort . . . . .	98
9.7	Exercises . . . . .	100
<b>III</b>	<b>Data Structures: Linear</b>	<b>101</b>
<b>10</b>	<b>Lists</b>	<b>105</b>
10.1	Stack . . . . .	105
10.2	Queue . . . . .	105
10.2.1	Priority Queue . . . . .	105
10.3	Random Access Lists . . . . .	105
10.3.1	Array . . . . .	105
10.3.2	Dynamic Array . . . . .	105
10.3.3	Circular Array . . . . .	105
10.4	Sequential Access Lists . . . . .	105
10.4.1	Singly Linked List . . . . .	106
10.4.2	Doubly Linked List . . . . .	107
10.4.3	Linked List with a Tracer . . . . .	108
10.4.4	Circular Linked List . . . . .	110
10.5	Exercises . . . . .	111
<b>IV</b>	<b>Data Structures: Non-linear</b>	<b>113</b>
<b>11</b>	<b>The Binary Heap</b>	<b>115</b>
11.1	Array-based Trees . . . . .	115
11.2	Insertion . . . . .	115

11.3	Deletion . . . . .	118
11.4	Heapification . . . . .	118
11.5	Performance . . . . .	118
11.6	Exercises . . . . .	121
<b>12</b>	<b>Maps and Sets</b>	<b>123</b>
12.1	Hash Tables . . . . .	123
12.1.1	Open Addressing . . . . .	123
12.1.2	Separate Chaining . . . . .	123
12.1.3	Collisions and Clustering . . . . .	123
12.1.4	Performance . . . . .	123
12.2	Binary Search Trees . . . . .	123
12.2.1	Insertion . . . . .	124
12.2.2	Removal . . . . .	125
12.2.3	Performance . . . . .	127
12.3	Balanced Trees . . . . .	128
12.3.1	Rotating Nodes . . . . .	129
12.3.2	AVL Trees . . . . .	130
12.3.3	Red Black Trees . . . . .	134
12.4	B-Trees . . . . .	137
12.5	Exercises . . . . .	144
<b>V</b>	<b>Algorithms</b>	<b>149</b>
12.5.1	Performance . . . . .	151
12.6	Graph Algorithms . . . . .	151
12.6.1	Depth-First Search . . . . .	151
12.6.2	Breadth-First Search . . . . .	151
12.6.3	Dijkstra's Shortest Path . . . . .	151
12.6.4	A* Search . . . . .	151
12.6.5	Performance . . . . .	151
12.7	Exercises . . . . .	151

**Part I**

**The Language**





# Chapter 1

## Overview

This document details some of the terminology and major concepts relevant for upper-division computer science students working with C++ at San Diego State University (SDSU); as a degree requirement, the department requires undergraduate computer science students complete an upper-division course in data structures and algorithms, and these lecture notes *supplement* the weekly discussion and lectures occurring in the class.

SDSU introduces students to programming using the Java language. Thus, this document accordingly assumes the reader understands prerequisite Java syntax and programming, for it builds upon those concepts. Specifically, students should understand the basics of programming syntax, primitive data types, conditional statements, creating functions, and solving basic problems using a high-level programming language. Although not taught in Java, knowing the language bears fruit when we juxtapose the two languages as we will during discussions of memory management.

That said, students with similar programming experience in C# or Python should achieve an equatable level of comprehension. Having experience in a language with automatic garbage collection, for example, assists the discussion of memory management, but lacking this depth should not prevent a student from understanding C++. On the other hand, students with no prior programming experience will find this document, and the class in general, entirely insufficient for their needs.

**Syntactic Sugar** As programming languages develop, they introduce shorthand and additional language uses for often repeated tasks. Rather than force the developer to include the unchanging boiler-plate code, a language's *syntactic sugar* [1] introduces easier ways to express the idea to the com-

piler. Syntactic sugar provides no functionality one could replicate without the language feature, but it does tend to make a language easier to use when performing those tasks. Typically, as a developer begins to use these features, the code begins taking on *the style of* the language.

## 1.1 Systems and Application Programming

All programming languages possess a target purpose, for none naturally occur. The language's creators address a perceived deficiency in the current offerings. For example, systems and game engine programmers frequently focus on optimization, so they appreciate many of the language features C++ provides. This purpose remains central to their day-to-day coding activities. Systems programmers also tend to allocate and use memory in unconventional ways, and C++ allows developers to directly manipulate and manage memory, so it fits the audience. Java, on the other hand, focuses on cross-platform development. One major selling-point for Java remains its "write once, run anywhere" mentality. The compiled Java bytecode, coupled with a Java Virtual Machine to interpret it, allows someone to write a program on one computer and execute it on any other machine by simply interpreting the compiled bytecode.

Additionally, Java transfers the task of memory management to its built-in *garbage collector*. This frees the chore from the developer, and that simplifies the resultant code. Moreover, it also helps eliminate a frequent source of errors in languages with managed memory: freeing the memory when no longer necessary. Java requires all code reside within a class, but in C++, one may write procedural or object-oriented code. Each language seeks to address deficiencies or limitations its developers perceived in the existing set of available tools.

Many of Java's features make it seem appealing in an educational environment. Its cross-platform development capability lends itself well to the varied computer hardware available to students. That said, Java's compilation and package system require familiarity with the underlying file system. Including external modules requires substantial understanding of the Java build process, and this limits the amount of time one discusses the task of programming. That is, the more time one discusses the particularities and idiosyncrasies of a language, the less time one may discuss path-finding algorithms and persistent data structures. To simplify some Java programming tasks, one might use a third-party build management tool like Maven, but doing so requires developers to understand the Project Object Model in

	<b>C++</b>	<b>Java</b>
Audience	Systems, games, and optimization-centered developers	Cross-platform application development
Inheritance	Multiple (code)	Single (code)
Operator Overloading	Yes	No
Memory Management	Raw pointers	Garbage Collected
Structs	Yes	No
Header Files	Yes	No
Functions	Yes	Everything in a Class
Threading	C++11 and later	Built-into the language

Figure 1.1: Some of the major differences between C++ and Java

Maven to correctly build the program.

Although Java emerged in response to C++, neither language remains unchanged. Standardization committees routinely update the language to include modern programming features. Both Java and C++ evolved to include support for lambda expressions. The *standard library* in C++ and the Java Collections Framework each include updated data structures and algorithms to help simplify the coding process.

## 1.2 Development Environment

An Integrated Development Environment (IDE) greatly assists creating working source code. It combines together several development tools into a unified interface. Not only does an IDE offer compilation services, but they frequently include debugging tools, a linker, and a host of other tools useful to the modern software developer. Some object oriented IDEs, for example, include a convenient and automated method to generate class diagrams from existing source or, more interestingly, develop source code from Unified Modeling Language (UML) diagrams.

On large teams, these tools frequently include task and bug tracking features. Build management tools, profile tools, and connections to version control services now arrive as part of the IDE. A well-designed IDE completes code and helps enhance the development experience. This additional

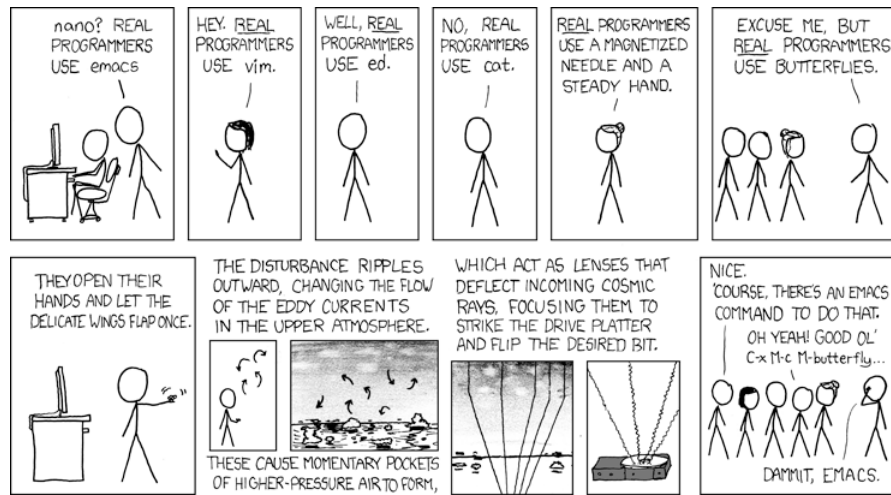


Figure 1.2: *Real Programmers*, courtesy of XKCD and the Creative Commons.

functionality, however, comes with a cost, for machines using an IDE require additional resources. One cannot easily remotely connect to a server and launch an IDE when running from the shell <sup>1</sup>.

On the other hand, one may write both Java and C++ source directly in a text editor and compile it from the command line without additional software. Much of the assistance IDEs offer comes in the form of automating the mundane tasks in computer science, but doing so obscures understanding the underlying concept. Moreover, rarely does an entry-level programming course reach a level of complexity where any of these features truly pay off. Provably, however, the IDE introduces additional complexity. Not only must one download and install the tool, but it introduces an entirely new interface the student must become familiar with before producing reliable code.

Consequently, while acknowledging the general usefulness of an IDE in nearly all professional applications, it impedes comprehension in an educational context. Here, we strive toward achieving a deep, potentially-pedantic, understanding of the fundamentals, for it is *this* knowledge which makes learning additional languages approachable. A developer's knowledge facilitates making educated choices about an appropriate language. Addi-

<sup>1</sup>Tools exist to facilitate this, but one must install them, and that requires time and knowledge.

tionally, coding interviews frequently take place in front of a white board [2] where these features remain inaccessible.

✎ Although students may use the C++ development environment of their choice (e.g., Vim, NetBeans, Eclipse, Visual C++) the instructor and teaching assistants offer *no* assistance with the student's chosen IDE.

None of the examples or projects discussed in this document necessitate the use of an IDE, but using one might significantly simplify the development experience. Alternatively, it could catastrophically-complicate the homework assignments.

**Windows Users** Microsoft's Visual Studio product line remains one of the world's best integrated development environments. With support for the Python, C#, and C++ languages, and coupled with ongoing updates by Microsoft, Visual Studio stands out as a premiere Windows development environment. The corporation offers both professional and enterprise editions, for a substantial licensing fee <sup>2</sup>, yet one may also download a free, Community edition for many of the same features.

**Apple Users** Apple provides its own C++ compiler, Xcode, for developers learning C++ on a Mac computer, and it provides sufficient capability for our needs. Alternatively, Eclipse and Netbeans, IDEs familiar to Java developers, also provide C++ integration.

**Linux Users** Vim. You're already doing it on advanced – commit to the plan and go all in.

### 1.2.1 Secure Shell

The University provides temporary educational accounts on a semester-by-semester basis. The class instructor shall provide each student with an account to use. These accounts allow students to connect to the university's development server either in person in the computer lab or remotely through a secure shell (SSH). Students with experience using a Raspberry Pi may possess some familiarity with the SSH process on their own machines. Connecting to the university server mimics this process.

---

<sup>2</sup>What do you really own?

```
instructor@home:~$ ssh csscXXXX@edoras.sdsu.edu
```

This system is provided by San Diego State University for the use of authorized users only. Individuals using this computer system without authority, or in violation of state or federal laws are subject to having their activities monitored by law enforcement officials.

```
csscXXXX@edoras.sdsu.edu's password:  
[csscXXXX@edoras ~]$
```

Figure 1.3: Establishing an encrypted connection to the university server.

Linux and MacOS users should find the SSH process relatively painless, for the operating systems include support for the feature at the command line. Thus, after opening a terminal window on a remote machine, one may simply execute the `ssh` command with the appropriate parameters and connect to the server. Windows users must either install a third-party tool like PuTTY, or they must enable the Windows Subsystem for Linux which allows them to install Ubuntu or Kali via the Microsoft Store.

✎ The course instructor highly encourages every Windows 10 user to migrate to the Windows Subsystem for Linux, for it enhances the development experience and simplifies interfacing with other machines.

## Account Password

Historically, information technology officers, using standards developed by the National Institute of Standards and Technology (NIST) based upon what security professionals believed to be reasonable assumptions. After countless cracked passwords, however, the failure of this system became obvious. Although using multiple character sets and requiring strange permutations of ASCII symbols allows a computer to produce a wide range of passwords, humans struggle to memorize these patterns.

Consequently, the very shop that developed the original standard mandating these obscure combinations now strongly argues against its continued use. In short, the NIST now encourages people to use password manager tools and follow some general guidelines for memorized secrets[3]:

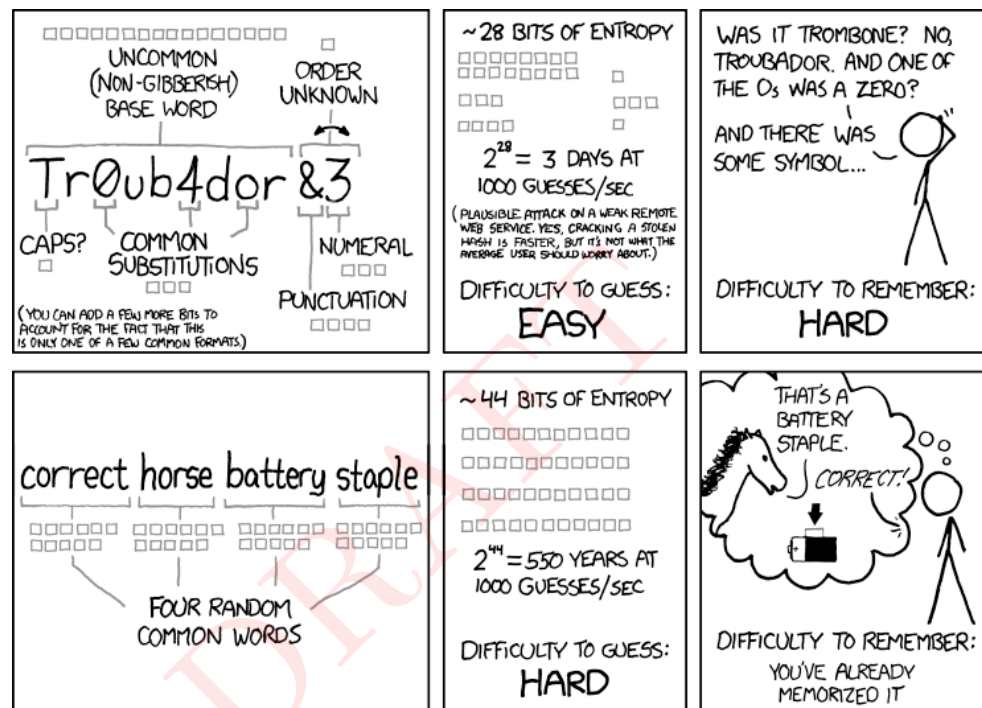
- Remove periodic password change requirements: Why change a password if you do not suspect it is compromised?
- Eliminate arbitrary complexity restrictions.
- Screen new passwords against a list of known leaked passwords, dictionary words, and repetitive sequences.

Furthermore, the NIST recommends several policies to support using a password manager. Every service should use a unique password to limit the danger *awhen* the password becomes public, but forcing users to create non-trivial passwords for every web service quickly becomes untenable, so people begin reusing favorite passwords. This makes it easy to remember the password, but it exposes people to risk when any website using the password experiences a data breach.

Password managers, like LastPass, not only maintain a list of passwords to automatically use when on the appropriate website, but they typically provide tools for generating passwords with arbitrary complexity requirements. With a password manager, users may easily produce unique secret keys for every online service. This added defense motivates the NIST recommendation. Instead of maintaining a list of hundreds of complicated passwords, users need only memorize the lengthy password used to access the password manager. Password managers introduce their own blend of security vulnerability, but they help patch an existing porous security opening.

Finally, these recommendations better reflect how modern computer attacks occur. Brute force password attacks, like the WOPR attempting to crack the United States nuclear launch codes [4], rarely occur. The original, cryptic rules helped prevent this iterative approach, but they also created an unusable system. In practice, with examples of millions of cracked passwords available on the Internet to simply try, the brute force hack makes little sense. Furthermore, many computers and online services quickly detect this attack and they implement strategies to minimize its impact.

**Changing the Account Password** Students *shall* consider the initial password provided with the account as compromised, for it appears in plain text on a document distributed around the classroom. Consequently, students shall change the compromised password in accordance with NIST recommendations. To initiate a password change, students shall type `passwd` at the command prompt after logging into their accounts. The system may require special characters; on many systems, the space character meets these



THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

Figure 1.4: *Password Strength*, courtesy of XKCD and the Creative Commons.



requirements. To change an account's password requires users must know the current password.

### 1.2.2 Visual and Visual Improved

With access to a shell window, users may launch applications on the host machine. After connecting to the university server, students may open one of the editors available on the server. Both Visual (`vi`) and its successor Visual Improved (`vim`), offer basic text editing features. Vim acts as a super-set of `vi`, so it supports the same interface, but it also includes code highlighting and other features useful to developers. These tools also possess the inherent advantage in that they ship on nearly every Linux installation, so learning the interface guarantees the developer may modify essential system files without dependency on a particular program.

That said, the editor's interface possesses a steep learning curve<sup>3</sup>. One must quickly learn the difference between command mode and insert-text mode. Although learning `vi` benefits the computing professional, one need not use it exclusively when developing code in this course. Critically, the chosen editor must generate ASCII files without any additional formatting or hidden text. That is, they must *only* write to the file what they show the user.

For this reason, Microsoft Word documents will not compile. Although the application displays only limited text to the user, the saved file includes a great deal of additional formatting and special characters Word needs to properly display and work with the document. All this additional overhead will cause compilation to fail. Notepad, on the other hand, *may* write files suitable for compilation. Look for a text editor, *not* a word processor.

### 1.2.3 Secure Copy

After completing a program, students must submit their completed work to the university server. For files developed remotely, students must copy them to the server for the grader to access the data. Consequently, students must transfer the data from their local, host machine to the external university server through an encrypted channel. Linux and MacOS generally include this tool by default, so users need only run the appropriate command.

---

<sup>3</sup>How do I exit this program?! Why won't it save? I'm typing, nothing's happening!!!!

```
scp -r ./program1 csscXXXX@edoras.sdsu.edu:~/submit/program1/
```

Figure 1.5: An example of a recursive directory copy of a local folder to the university server through SCP.

## 1.3 Compilation

Computers must convert between the high-level language we use to write software and the low-level, bare-bones machine code necessary to operate the processor. Ultimately, every program must toggle bits on a physical device. It could hold a pin high or toggle the contents of some internal flip-flops. The approach to this *compilation* process varies based upon the language in question.

For example, Java compiles the source file into platform-independent Java bytecode. The platform-dependent Java Virtual Machine (JVM) then interprets this bytecode into the appropriate machine instructions for the host processor. With Java, one need not compile the application on every new system. Instead, one simply writes a unique JVM for the hardware, and this JVM uses the same, compiled bytecode to function.

By contrast, C++ compiles source code into an executable appropriate for the target machine, for it does not include an intermediate bytecode step. The executable file produced after running the C++ compiler specifically works on its target hardware, so one must compile separately for Mac, Linux, and Windows. In C++, the compilation process moves through several distinct phases: pre-processing, compilation, and linking. Each of these steps possesses specific input and output requirements. That is, the source file must meet several syntax requirements before the compiler can use it to produce any meaningful output for the next stage.

### 1.3.1 The Preprocessor

Both C and C++ include a pre-processing step in the compilation process. Before performing any of the actual syntax analysis, the pre-processor performs a lexical substitution within the original source document. It reads each of the *pre-processor directives* included in the source file and takes action based upon the directives contained therein. The pre-processor outputs a *translation unit* for the compiler to use at the next step in compilation.

At the simplest level, a pre-processor directive might include simply substituting the number 1 in the code everywhere the text TRUE appears.

Pre-processor directives generally:

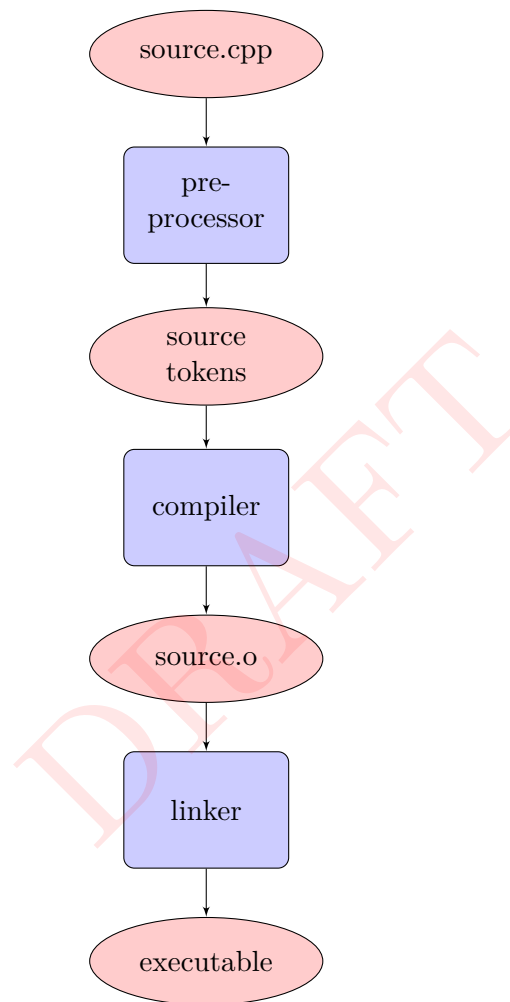


Figure 1.6: A rough overview of the C++ compilation process.

---

```
# 1 "bare.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "bare.c"
int main(){
    return 0;
}
```

---

Figure 1.7: An example of the pre-processor expanded output from a minimal C program.

- Include content from external files
- Define macro functions
- Insert conditional logic

The C++ compiler also uses the pre-processor to insert function definitions and variables defined in external files. Similar to Java's `import` directive, the `#include` pre-processor directive instructs the compiler to paste the contents of the indicated file at the point the `#include` directive appears in the code. Errant use of `include` statements may bloat the output from the pre-processor, for the pre-processor recursively expands each of the files it includes, so if the user's file includes a header file which, itself, includes several header files, the pre-processor will expand those where they appear in the included file.

In addition to inserting text from other sources, say when using the standard library or the output streams, one may actually program constant values and write simple pseudo-functions. Because a pre-processor expansion simply replaces the text with its saved text, C and C++ developers may write what looks like a function but does not require the overhead associated with a function call. Modern C++ development frowns on using a macro for a function like this <sup>4</sup>, for there exists a *double-evaluation* problem. Because macros, even macro functions, simply serve as a text replacement before compilation, one might inadvertently call a function multiple times.

---

<sup>4</sup>The standard library includes a `max` function, so you need not create your own without reason.

---

```
#define max(a,b) (a < b ? b : a)

max(x++, y++);
```

---

Figure 1.8: A basic C macro for finding the maximal value. The pre-processor replaces the macro variable `a` with `x++`, so what values do `x` and `y` have after executing this expression?

---

```
#include <iostream>
#define SWAP(X,Y) {int temp=X; X=Y; Y=temp;}

int main() {
    int a = 2;
    int temp = 17;
    std::cout << a << " " << temp << std::endl;
    SWAP(a, temp);
    std::cout << a << " " << temp << std::endl;
}
```

---

Figure 1.9: An example of *variable capture*. What values for `a` and `temp` does the program print?

Additionally, pre-processor macros remain prone to *variable capture*. What happens when a macro declares a temporary variable with the same name as one currently in use? Because it simply substitutes the text it finds, an incorrectly named variable may introduce devious program errors as 1.9 illustrates.

Finally, the pre-processor includes several directives for controlling how the pre-processor itself works with the file. For example, what happens if two header files included in the current source file each reference a third header file? Should the pre-processor include the header file code twice? Why would one need to insert identical definitions for a swap macro in the same file? Thus, the pre-processor includes some conditional logic to help trim unnecessary tokens from its output.

### 1.3.2 Header Files

When including functions or variables defined in other translation units, like one would do when using a third-party library, one must first *declare*

<code>include</code>	Inserts the text from the specified file at the point the include statement appears in source.
<code>define</code>	Specifies the form of a macro substitution
<code>undef</code>	Remove a current pre-processor macro definition
<code>if</code>	Begin conditional block.
<code>endif</code>	End conditional block.
<code>ifdef</code>	Tests if the following name is a valid macro substitution.
<code>ifndef</code>	Tests if the following name is not a valid macro substitution.
<code>line</code>	Outputs the line in the event of an error.

Figure 1.10: Examples of pre-processor directives included with C++.

the function. Traditionally, a C header file includes global variable and function declarations and typically ends in `.h`; C++ header files include the actual working code and might end in `.hpp`, but the language imposes no requirements on the actual suffix. In C++, one may define and implement a class in a single file, and some teams choose to write code using this style. Others, however, prefer to separate the *interface* logic from the actual *business* logic by placing only function and class declarations in the `.h` file and their definitions in a `.cpp` file.

A *header guard* should prevent the compiler from including identical code, repeatedly into single translation unit. Figure 1.11 illustrates one such approach using pre-processor directives. Alternatively, one may use a `#pragma once` directive, as shown in Figure 1.12, but not every compiler supports this feature. Header guards using the `#ifndef` directive provide backward compatibility with C and work with all C++ compilers.

One may use either angular brackets or quotation marks when including the files, but the two symbols possess subtle distinctions. Angular brackets reference the global include directories; one may view them as external to the project. Files in quotes are relative to the current directory for the source file; these only work with directories in the include path. Consider placing project-specific include files in quotation marks while leaving the major, common libraries in angular brackets.

---

```
#ifndef __USER_SETTING_H
#define __USER_SETTING_H

// The pre-processor only inserts this code if __MY_CUSTOM_H is
// undefined. This only happens the first time through the file.

#include <iostream>
#include "my_project.h"

namespace sdsu
{
    struct UserSettings {
        bool darkMode = true;
        bool sUser = false;
    };
}
#endif
```

---

Figure 1.11: An example of a header guard which only includes the contents of a header file the first time it encounters it in code. This style works with both C and C++.

---

```
#pragma once

#include <iostream>
#include "my_project.h"

namespace sdsu
{
    struct UserSettings {
        bool darkMode = true;
        bool sUser = false;
    };
}
```

---

Figure 1.12: An example of a header guard using the pre-processor's `#pragma once` directive. Not all C++ compilers support this feature.

### 1.3.3 Compiler

After pre-processing each of the source files included in the compilation process, the tool begins translating the expanded output into an intermediate machine code. This *object* file includes the machine code for the statements in the current file with placeholders to statements made in other source units. Each object file remains independent of its peers in the program at this point.

The compiler does not need to know what an external function call does in order to compile the current file, for it only needs the variable or function's *signature*. The machine code compiled for the current file only needs to know what parameters to load on the stack before the current call and what return value to expect. A program does not need to know what happens inside `max` or `min` to know how to prepare for the call and what to do after it completes.

### 1.3.4 Linker

Finally, using the resultant object files produced by the compiler, the linker fills in the final blanks necessary to produce an executable the user may run on the machine. The linker replaces the external references in each of the object files with references to each compiled unit. Now that the machine understands where the `max` and `min` functions reside within the compiled object file, it can replace those placeholder calls with the actual jumps to the appropriate instruction. That is, the compiler cannot include this during its initial generation of code, for it will not know everything until the first stage of compilation completes.

Each translation unit in a project produces an object file after compilation, and the linker brings them all together. Where is the main function in the application? What starts the application? One may compile a source file without an entry point, but the linker will not produce an executable program without a valid entry point.

By default, the linker looks for the `main` function as its entry point, but one may override this behavior and specify any function as the beginning of execution.

The linker brings together function *declarations*, which might appear in multiple files to introduce a function or variable, with their singular *definition*. Chapter 3, which discusses functions in greater detail, addresses this distinction. Significantly, should the linker fail to locate a required definition, it produces a linker error and fails to produce an updated executable. The *unresolved external symbol* error typically appears when first working



---

```
#include <iostream>

int main(){
    std::cout << "" << std::endl;
}
```

---

Figure 1.13: A basic C++ application with a program entry point.

in a project with multiple files.

Including a header file allows the current translation unit to prepare for and respond to a function call, but it lacks the actual code that makes the function work. The linker must resolve all the declarations with their definitions when it produces a functional executable. One typically clears these type of linking errors by supplying the required libraries in the build path through a configuration setting or optional parameter.

**Static and Dynamic Libraries** When including code defined in the project's library or through an external source (e.g., The OpenGL Extension Wrangler Library [5]) developers face a choice: include the library source directly in the final executable, or partially construct the executable with dynamic links to shared, common libraries. The two choices each present benefits and drawbacks, so one must decide based upon the project's ultimate needs. How does one plan to deploy the software? Are the machines regularly maintained? Are the versions of the external libraries critical, for a required system update may break third-party applications using the updated libraries.

Statically linking in the external libraries effectively copies their code into the project and integrates it with the final executable. This reduces the number of additional installation dependencies the user needs during installation, for everything comes statically packaged together. If a vendor updates a third-party library the application uses, the application experiences no changes, for it uses its own, internal copy. Obviously, carrying around this additional library causes the resultant program to grow in size, but it also speeds up performance, for the application need not consult the system to locate the external libraries.

Dynamically linking a library, however, offers its own set of benefits. When multiple applications use the same dynamically linked library (DLL), the operating system only loads the shared source in memory once. Furthermore, when one discovers a security flaw with a DLL, a single update to the

library potentially solves the error for every application using the library. Although third-party updates may break an application, they also provide support. Additionally, one may use a DLL across several programming languages.

☞ When in doubt, statically link external libraries, for the performance improvement generally makes up for the larger program size.

## 1.4 Exercises

DRAFT

## Chapter 2

# Basic Language Syntax

The C++ language builds upon *C with Classes* [6], and, in fact, one may view C++ as a super-set of the C language, for, with very few exceptions, all legacy C code compiles with C++. In fact, one may still use the older C-style header files, with their associated functions, in C++. Consequently, students familiar with the C language should find the transition to C++ manageable. Java emerged, in part, as an object-oriented response to C++, and the language developers sought to make the Java programming language as accessible to existing C++ programmers. Thus, much of Java's syntax looks familiar in C++. Significant differences exist between the two languages, but each is closer to the other than, for example, something like Python.

### 2.1 Data Types

The C++ language specification defines several primitive data types. When performing computations, computers must store information. The primitive data types represent the most basic storage abstraction. Ultimately, all data on a computer exists in memory and hardware as either high or low voltage signals. The data type instruct the compiler about how it should interpret an arbitrary but specific memory location. Like Java, C++ uses static, manifest type system.

Due to *manifest* typing, as shown in Figure 2.1, the source code must declare every variable's type within the current scope before it uses the storage location. Additionally, after specifying that a given variable name holds a given type, one cannot change this storage type within the current scope. By contrast, the Python language uses a *dynamic* type system which

---

```
int main(){
    int first = 1;
    char second = 'b';
    float value = 3.50;
}
```

---

Figure 2.1: Manifest typing in the C++ language. The source code must specify the distinct types for each memory location.

---

```
a = 1
a = "lucky charm"

def tikki():
    print("spots on")

a = tikki()
a()
```

---

Figure 2.2: With dynamic typing in Python, a variable may change its fundamental type within the current scope. In this example, `a` assumes the form of an integer, string, and function within the same scope.

permits this behavior as Figure 2.3 illustrates.

One does not need to supply initial for variables in C++ or Java. Java initializes primitives, but C++ remains performance focused, and initialization requires additional work. Consequently, C++ does not generate code to initialize these variables before using them. Although C++ permits this style of code, secure coding guidelines <sup>1</sup> strongly advise against it, for one cannot anticipate the initial values. [10]

### 2.1.1 Variable Names and Standards

Like any programming language, C++ compilers require the source files meet a specific syntax in order to convert them into an executable. Additionally, developers must name variables in accordance with C++ lexical requirements so the compiler can differentiate between variables and operations. That is, what are the keywords, permitted characters in a name, and any other restrictions on the names of variables in the language? Much

---

<sup>1</sup>EXP53-CPP. Do not read uninitialized memory.

---

```
#include<iostream>

int main(){
    int a;
    int *ptr;
    std::cout << a << ':' << ptr << std::endl;
}
```

---

Figure 2.3: Never in the way of possibilities, C++ permits developers to freely use uninitialized values.

like its predecessor C, C++ imposes the following restrictions on all variable names:

- Must begin with a letter in the alphabet or the underscore character.
- Spaces forbidden.
- After the first character, the variable name may include any count of additional underscore or alpha-numeric characters.
- Variable names are case-sensitive.
- The language's keywords may *not* act as variable names.

Provided with this flexibility, one might begin introducing variables to suit one's personal writing style. This degree of customization created source code with wildly varying appearance. Moreover, this made maintenance substantially more difficult, for variable names help the reader understand code. When every programmer arbitrarily names variables, or uses poor names, the entire project suffers. Consequently, most major software development firms use a *coding standard* or *style guide* [7][8][9][10].

Some style guides specify *naming conventions* which identify how one should construct a variable's name. That is, when creating a private pointer to a constant, unsigned character, a naming convention might demand programmers prefix the name with `_cptruc` so as to make clear<sup>2</sup> the variable's type and intended use. *Hungarian notation*, an early coding style, sought to include information about the variable's type and scope in its name so when developers encounter it in source, they immediately understood its use.

---

<sup>2</sup>Crystal.

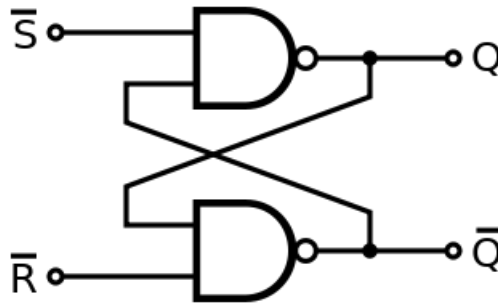


Figure 2.4: The SR flip-flop, a sequential logic circuit, stores a single, boolean value of 1 or 0 in a basic unit of memory.

The motivating philosophy remains error reduction. Theoretically, Hungarian notation catches potential errors like `_cSize = _iCurrentSize` where the value on the right hand side might overflow the allocated storage type. In practice, almost all projects of sufficient size to benefit from a strict naming convention also use an IDE. Issues with scope become quickly apparent using these tools, so the need to include an additional prefix simply introduces additional characters into the file. This will make the source file larger, but it will not improve the code.

### 2.1.2 Boolean Values

A single bit of data represents the smallest possible abstraction on a computer, for one cannot further subdivide a bit. Boolean values serve as the smallest logical value a computer holds. Typically, one uses boolean values to hold the contents of logical operations. The original C language from which C++ derives lacked direct support for boolean values as a distinct type. Instead, one simply interpreted other data types, frequently a character, as true and false equivalents.

Unlike its predecessor, the C++ language expanded to include boolean values as types. To declare a variable as a boolean type, one identifies it with the keyword `bool`. After declaring a variable of type `bool`, it may accept the values `true` or `false`. One may convert, or *cast*, between data types in C++, so the machine will treat all `true` values as equivalent to 1 and all `false` values as 0. Additionally, the special `nullptr` value, which represents an invalid pointer, also converts to 0.

---

```
// An assignment statement-expression
bool a = true;

// Using a C++11 initializer list
bool b{true};

// true (1) minus true (1) is false (0)
bool c = a - b;

// true (1) plus true (1) is true (!=0)
bool d = a + b;

// true (1) OR false (0) is true (1)
bool e = a || c;
```

---

Figure 2.5: Various declarations of boolean types in C++.

---

```
bool a{true};

if( a != false ) a = false;

if( !a ) a = true;

while( a = true ) a = false;
```

---

Figure 2.6: Three conditional uses of boolean types, but the `while` loop may not behave as expected.

Type	Range (typical)	Notes
char	-127 to 128	The default type typically includes a sign. May include a character in the implementation's character set.
signed char	-127 to 128	Use of signed guarantees the variable will include negative values.
unsigned char	0 to 255	Guaranteed to be unsigned.
wchar_t	1 wide character	Holds a single Unicode character (2 or 4 bytes).

Figure 2.7: The different character types available in C++.

### 2.1.3 Characters

We can extend single-bit, boolean values into an arbitrarily large binary number by simply combining them and correctly interpreting their new values. The number of bits we use to represent the data establishes the maximum value we may store. For example, an eight bit character type may hold 256 distinct values. We may choose to interpret these data in any number of ways. For example, the binary value `0b11111111` *could* represent 255 if one wished to ignore negative values, but it might also act as -0 using one's-compliment.

Although the character type in C++ *may* hold literal, ASCII character values, it also frequently appears when the software design needs a small number. Systems programmers optimize code for time and space, and they typically select data types based upon an optimization strategy. Thus, both C++ and C include extensive data type ranges to accommodate this level of optimization as illustrated in Figure 2.7.

The ASCII character set associates character symbols with a numerical value. Thus, one may represent the letter 'S' as either the *character literal* or its associated numerical value (`0x53`). Consequently, developers may use math operations on the data type. When one decides to increment a character value, it simply takes on the next sequential value in the alphabet. This idea extends to conditional statements as well. The statement `curChar > 's'` will return `false` if `curChar` holds 'a' and `true` if it contains 'x'.



Type	Notes
<b>short</b>	At least 16-bits
<b>signed</b>	Accepts positive or negative values. Default if omitted.
<b>unsigned short</b>	At least 16-bits
<b>int</b>	16-bits minimum, but 32-bits typically on 32 or 64 bit machines
<b>long</b>	32-bits minimum
<b>long long</b>	64-bits minimum (since C++11)

Figure 2.8: The different integer types available in C++.

### 2.1.4 Integers

By increasing the number of bits used to store a number, we dramatically expand the range of potential values for that data type. The integer data type represents the next step after a simple character. Much like the `char` type, the `int` type offers multiple flavors. One might use signed or unsigned varieties. Figure 2.8 details some of the possibilities. Unlike the `char` type, which uses eight bits, the C++ compiler implementation establishes the size of an `int`.

The multitude of options C++ offers for data types stems from its use in embedded systems. Each of these types facilitate optimizing the hardware usage at a precise level. Thus, one might legally declare an `unsigned long long int` should one need such precision.

Much like string-literals, which store values in the current character set, one may also use *integer-literals* when assigning values to a variable. On a computer, we frequently need numbers in different bases based upon its use. Obviously, base-ten integers remain the most common, so when one sets a variable to an integer-literal without adding any prefix or suffix, the compiler defaults to this numbering system.

### 2.1.5 Floating-point Numbers

Computers store voltage values in binary signal gates, but floating point numbers represent an analog abstraction. One may easily convert any unsigned integer between its hexadecimal, octal, and binary forms. The signal levels directly correlate with their mathematical representations, but how does one represent a decimal number in binary? How much precision does

---

```
int main(){

    // Assignment statement
    int a = 1;

    // Initialized using C++11 notation
    int b{310};

    // Valid multi-character constant, but obtuse
    int pleaseNo = 'no';

    // Hexidecimal literals simply use the 0x prefix
    int bitFlags = 0xFEFE;

    return 0;
}
```

---

Figure 2.9: Some of the various integer-literal representations in a basic C++ program.

one require in general? When we use decimal numbers, do we pull equally from the entire number line, or do we tend to use the values less than one, say when generating percentages, more frequently? The IEEE-754 standard defines the behavior for floating-point numbers, and most programming languages adhere to its specifications [11].

The decimal system used actually implements a common standard, and this increases the software unit's portability. That is, it compiles and operates as expected on a variety of hardware implementations. One may think of the floating point types as built-in *wrappers* around the binary representation on the computer. Many processors implement the standard in hardware, so the compiler need only implement the proper machine instructions when generating the executable code.

In the mid 1990's, Thomas R. Nicely discovered an error with the Floating Point Unit in Intel's Pentium processors [12]. The error in the implementation led to a minor error which most users never experienced, but Intel's dismissive response damaged its reputation within the tech community none-the-less. Certainly, the impact appeared minor to *most* users, but were one managing a bank of computer-controlled slot-machines or designing a flight control system, these errors may become catastrophic.

4195835.0/3145727.0	1.333
820 449 136 241 002 5	Correct value
739 068 902 037 589 4	Flawed Pentium

Figure 2.10: The Pentium FDIV Error incorrectly computes some floating point calculations.

### 2.1.6 Auto

As a manifestly typed language, C++ source requires developers specify each and every variable type in the file, for this allows the compiler to allocate the proper number of bits to hold the data. In some cases, however, one might correctly deduce the type from its context. For example, given the statement: `x = true`, one might naturally suspect the variable `x` stores `bool` type data. Certainly, `x` might represent a `long`, but the smallest, natural fit remains a boolean value.

The C++11 update introduced *placeholder* type values to help simplify the source code in these situations, and each update seems to introduce increased functionality to this keyword. For example, C++14 allows `auto` to act as the return type for lambda expressions, and C++17 introduced *structured-bindings*.

### 2.1.7 Type Casting

In a strongly typed language, each memory cell holds a distinct type. In C and C++, when we declare a variable, we must also specify its type. After introducing the variable `curCount` as a `char`, we could not then assign it the value `3.5`, for the compiler will prevent this type of implicit *down casting*, for the `char` type cannot hold the number without losing data.

On the other hand, both C++ and Java support implicit *up casting*. In this type conversion, we attempt to store something small in a larger container. Many programming languages automatically perform this operation quietly behind the scenes during compilation. One might easily store something ranging from 0-255 in a container capable of storing values ranging from 0-65535, for we are *widening* the value.

The C++ language supports two ways for performing explicit type-casting for its primitive data types. To maintain backward compatibility, developers may choose to use the old *c-style* casting method. Alternatively, one may elect to cast to a target type in something resembling a function

---

```
#include <iostream>

int main()
{
    // accept becomes type char
    auto accept = 'y';

    // ness becomes a type double
    auto ness = 3.50;

    std::set<std::string> courses;
    courses.insert({ "CS310", "CS320", "CS440", "CS496" });
    for (auto it = courses.begin(); it != courses.end(); it++)
        std::cout << *it << std::endl;

    return 0;
}
```

---

Figure 2.11: Examples of placeholder types where the compiler deduces the data type.

call. The code in Figure 2.12 illustrates use of the two styles.

## 2.2 Arrays and Pointers

### 2.2.1 Arrays

Arrays, in Java, C, and C++, all hold a collection of elements of the same type in *contiguous memory locations*. The array promises the data will exist in memory sequentially, and this promise proves essential for its performance. Due to *locality of reference*, the processor tends to access memory locations repeatedly relatively near one another. That is, when working with `curLine[4]`, one likely passed through `curLine[3]` and may very well be on the way to `curLine[5]`. Because the data exist close together in memory, the processor can directly access them. In fact, due to the way the processor fetches data from memory, it may already have those values in a fast cache.

One declares an array using the general format: `T[size]`, where `T` represents its type (e.g., `char`, `long`, `void*`, `Pokemon`) and `size` indicates how many elements exist in the array. Figure 2.14 illustrates declaring multiple arrays using C++. After creating an array, one naturally needs to access

---

```
char cCount = 127;
float sRequires = 3.50;

short data;

// A c-style narrowing (down-cast)
data = (short) sRequires;
// A narrowing (down-cast)
data = short( sRequires );

// A widening (up-cast)
data = cCount;
sRequires = cCount;
```

---

Figure 2.12: Explicitly type-casting between different variable types.

---

```
int* ptrX, ptrY;
```

---

Figure 2.13: A single line type declaration introducing two variables. The variable names suggest the developer wants two pointers, but the syntax produces two different types.

---

```
char* buffer = new char[8];
memset(buffer,0,8);
delete[] buffer;

char stackBuff[8];
memset(stackBuff,0,8);

int data[] = {1,2,3,4,5};
float gpa[50] = {4.0};
```

---

Figure 2.14: Multiple of declaring and initializing arrays in C++ illustrate the shared syntax.

---

```
void bracketed(int data[]){
    for( int i = 0; i < 10; i++ )
        std::cout << data[i] << std::endl;
}

void throughPointer(char name[]){
    for( char *ptr = name; *ptr; ptr++ )
        std::cout << *ptr;
    std::cout << std::endl;
}
```

---

Figure 2.15: Accessing the elements of an array.

the members. One may do so through the `[]` notation or via pointer manipulation as shown in Figure 2.15.

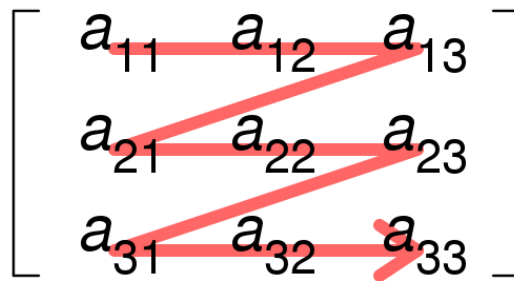
**Random Access** Because the compiler understands that the elements in an array of integers are adjacent in memory, it can quickly compute the correct address to jump to when accessing elements, randomly, from within the array. Certainly, progressing through the array sequentially from front to back occurs quickly, but it also supports rapidly accessing any member within the structure.

To do so, the compiler uses the array's base address as a starting point. The address of the first element in an array, (the item at index zero), corresponds to the array's name. So given an array `lines[16]`, the value of `lines` corresponds to the address of the item at `lines[0]`. If we increment this address by the width of its underlying type, we point to `lines[1]`.

$$\text{array\_base\_address} + (\text{index} * \text{sizeof}(\text{type})) = \&\text{array}[\text{index}]$$

**Multi-Dimensional Arrays** The C++ language stores multi-dimensional data in *row major order*. Using this format, all the elements in the array's row appear adjacent to one another. That is, what does the element at the next address hold, the row's next value, or the next *column's* data. The distinction is subtle and significant, for different architectures incorporate different strategies. Graphics cards, for example, may use *column major order*, so converting between the two may prove necessary in certain low-level applications.

## Row-major order



## Column-major order

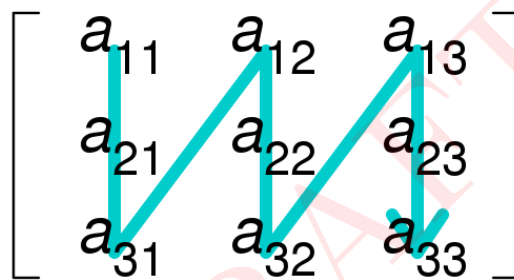


Figure 2.16: Row and Column major order as illustrated by the Wikimedia Commons.

---

```

int data[6] = {1,2,3};
double bw_image[32][32];
double rgb_image[512][512][3];

int main ()
{
    for (auto rowCounter=0; rowCounter<32; rowCounter++)
        for (auto colCounter=0; colCounter<32; colCounter++)
        {
            bw_image[rowCounter][colCounter]=(rowCounter+1)*(colCounter+1);
        }
}

```

---

Figure 2.17: Multi-dimensional arrays in C++.



Figure 2.18: Pointers hold the value of an address where the actual data rest. One accesses the data indirectly by first looking at the address held in the pointer and then visiting that address. Image from *Dinosaur Comics* ©2005 Ryan North.

### 2.2.2 Pointers

All memory locations on a computer possess a unique identifying number so the machine may access and mutate the data stored therein. The address may exist on a local hard disk or in RAM. Imagine memory as a continual list of cells one may access by number. Regardless of the underlying data type, the address of every location in memory is simply an integer.

---

```
void *current = nullptr;
char** ptr = &buffer;
```

---

**Void Pointer** Systems level programmers frequently pass around memory locations without knowing the type of its contents. In C++, the `void*` fulfills this purpose. One may think of it as a generic address to an object of irrelevant type. Should the developer wish to access its contents, simply casting the variable will convert it to the appropriate type.



---

```
void *current = nullptr;
void *old = 0;

if(current == nullptr) std::cout << "Null present" << std::endl;
if(!old) std::cout << "Null Present" << std::endl;
```

---

Figure 2.19: Use of `nullptr` and the older C-style evaluation to test for pointer initialization.

**Null Pointer** The Java programming includes the special keyword `null` to denote un-instantiated objects. C++ provides a similar feature with the `nullptr`, and one may set a pointer of any type to the `nullptr`. Historically, the C language uses 0 as an indication of a null pointer, for it represents an invalid memory location. To support readability, however, `nullptr` makes clear the context.

### Pointing into structures

Structures allow developers to cleanly group together related fields. One can create a structure for the player's ship, and in this structure it might hold several related fields. What if one wished to implement a star-ship dealership where the user scrolled through the dealership browsing for available craft? How might one store the currently selected ship?

Moreover, how might we assign the selected ship to a player? Store an index to a greater array? After selecting the ship, the user might wish to customize it with additional cargo holds or superior engines. Consequently, creating a data type with a link to another data type might be in order.

In any event, pointers to structures appear frequently within C code. Fortunately, C includes special syntax for accessing data members inside a structure pointed to by a variable. The following example illustrates both ways.

#### 2.2.3 Dereferencing

Pointers represent memory addresses, but this presents little utility in and of itself. That is, although knowing the address where data reside may serve a use in a particular algorithm, the data themselves possess meaning in the application. Sometimes one needs to know something's name and not simply where the name is on the computer. We *dereference* an address by visiting the location pointed to by the address. This process requires several steps in

---

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

struct entry {
    char *key;
    int count;
};

struct node {
    struct entry *data;
    struct node *next;
};

int main(){

    struct entry *player = (struct entry*)malloc(1);
    player->key="Mr. Jones";
    player->count=1;

    struct node *head = (struct node*)malloc(1);
    head->data = player;

    printf("First player %s\n", head->data->key);

    /* return the heap memory to the OS */
    free(player);
    free(head);

    return 0;
}
```

---

Figure 2.20: The C language introduces new, special notation for pointing into structures. This same style applies to how C++ addresses pointers into structures and classes.

---

```
#include <iostream>

void swap( int *first, int *second ){
    int tmp = *first;
    *first = *second;
    *second = tmp;
}

int main(){

    int a = 1;
    int b = 2;

    std::cout << "Before: a(" << a << "), b(" << b << std::endl;
    swap(&a, &b);
    std::cout << "After: a(" << a << "), b(" << b << std::endl;

    return 0;
}
```

---

Figure 2.21: Passing by reference in C++

machine code, for the system must first load the address where the memory lives, but it must then visit the address to retrieve the data.

### 2.2.4 References

How does one swap the value of two variables in Java? One may pass in an address of an object, so, in effect, one transfers in a *copy* of the address the method manipulates during its execution, but when it terminates, the address returns to its original value. Certainly, one may visit the locations pointed to by the parameter copies and make persistent changes, but one may not actually swap the address A holds and the address B holds in a Java method, for *Java passes references by value*.

The C++ language, however, supports creating *aliases*<sup>3</sup> to variables as Figure 2.21 illustrates. Unlike a copy, an alias references the actual variable directly. Consequently, a function may create changes impacting the outer scope, and this makes aliasing a powerful feature in C++<sup>4</sup>.

---

<sup>3</sup>Sydney Bristow.

<sup>4</sup>And a significant security risk.

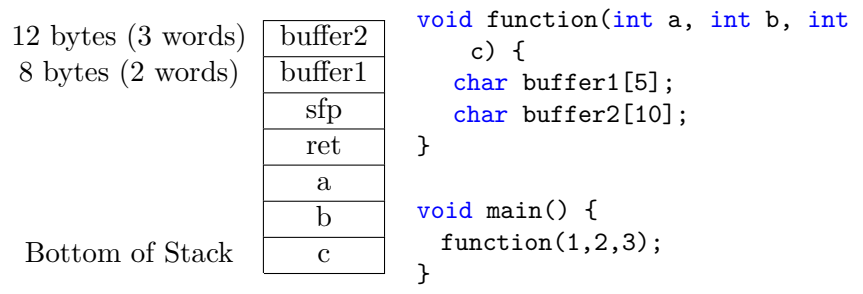


Figure 2.22: Function and its stack frame.

### 2.2.5 A Stack Frame

The stack consists of logical stack frames that are pushed when calling a function and popped when returning. A stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the instruction pointer at the time of the function call.[?]

### 2.2.6 Buffer Overflow

This attack attempts to overwhelm the *stack* where the computer stores return addresses. If one can overwrite a return address with the address of malicious code<sup>5</sup>, one may take control of the computer. Perhaps the buffer overflow attack includes a simple function to access the shell?

Because the program permits unconstrained buffer writes, for it uses `strcpy`, it will continually read in data even if the container it attempts to store them in overfills.

---

```
void function(char *str) {
    char buffer[16];

    strcpy(buffer, str);
}

void main() {
    char large_string[256];
    int i;
```

---

<sup>5</sup>Friendly to you though.

---

```
const char* name = "leroy";  
const wchar_t* name4 = L"leroy";  
const char16_t* name3 = u"Leroy";  
const char32_t* name3 = U"Leroy";  
  
const char* ignore_escape = R"(raw text ignore escape char)";
```

---

Figure 2.23: String literals.

```
for( i = 0; i < 255; i++)  
    large_string[i] = 'A';  
  
function(large_string);  
}
```

---

## 2.3 Strings

The C++ language supports characters, integers, floats, and Boolean values as primitive, fundamental types. For string types, however, one must use an array of characters or a class provided in the standard library. The standard library `string` includes methods one may use when working with them. Java also implements strings through a class<sup>6</sup>.

**String Literals** appear in read only memory. One may inspect the executable file and read the string literals contained in the program. Consequently, string literals are *immutable*. Figure 2.23 illustrates several methods for declaring string literals in C++.

**Null Termination** The length of a string literal always exceeds the number of ASCII characters stored in the string by one. The C language, and C++ by default, terminates string literals with the null character. That is, rather than storing the size of the string like an object, C uses the literal character code of 0 to indicate the end of the string. This differs than the character code for the number '0' which is 48. This fact presents interesting problems for some later functions, like `cin`, when reading a string from input into a string variable.

---

<sup>6</sup>As it does pretty much everything else.

---

```
#include <iostream>
#include <cstring>

int main(){
    const char* name = "how\0long am I?";
    std::cout << strlen(name) << std::endl;
    std::cout << sizeof(name) << std::endl;
}
```

---

Figure 2.24: Calculating string length when the string includes the null termination symbol. The main method fails to return a value.

**Computing Length** When working with string literals, which appear as `char*` or `char[]`, C includes a function for calculating the number of characters composing the string.

## 2.4 Statements and Expressions

**Expression** Constants, operators, fundamental types, and functions all represent values. When increasing a timer, one must add one to a variable. One might view it as anything that acts as a meaningful result to a function call. We call something usable in this way an *expression*. The number 3, as well as the character 'T', both represent expressions. One may use them on the right hand side of an equal sign when making an assignment.

**Statement** An executable program includes a sequence of instructions for the machine to act upon. A statement is one of these instructions. The computer executes each statement in a repeatable sequence. Statements may not hold any meaningful value, but they alter the program state in some way. After executing a statement, the program state changes. It points to the next instruction, and some of the data values in its memory may have changed. The C++ language uses semicolons to terminate statements.

**Expression Statement** In the C++ language, any expression followed by a semicolon represents an expression statement. Thus, `3;` serves as an expression statement, for it combines the expression 3 with the statement terminator.

---

```
int x;  
int y;  
{  
    x = 3;  
    y = 4;  
}
```

---


Figure 2.25: Compound statements use the curly-brace notation.

---

```
{  
    int x; // declaration statement  
    int y;  
  
    x = 3; // expression statement  
    y = 4;  
}
```

---

Figure 2.26: Blocks of code include variable declarations with their associated stack implications.

 Expression statements in C++ evaluate to a result.

**Compound Statement** To group multiple statements together, one may enclose them in curly brackets so they become a *compound statement*. Figure 2.25 shows a fairly minimal compound statement. Compound statements may, themselves, include nested compound statements.

**Block** We make the distinction that any *compound statement* that also includes a variable declaration is called a *block* of code. Variables declared within a block of code, as shown in Figure 2.26, only have visibility within the block, or any nested blocks, and fall out of context when the block ends.

### 2.4.1 Conditional Statements

if else  
    dangling else

**Switch Statement** When a conditional variable may only take an limited number of constant, integer arguments, the **switch** statement provides an efficient mechanism for multiple-path decision making. For example, a streetlight typically takes Red, Green, and Yellow values. Rather than structuring a series of *if-else* statements, the code might benefit from a **switch**.

The case statement imposes a few restrictions while offering offers unique behaviors:

1. Every case condition must be unique – no identical values
2. C only executes the **default** statement if nothing else fired
3. Cases, including the **default**, can appear in any order
4. The **break** statement terminates the switch evaluation
5. The **switch** will *fall-through* to the **next** statement unless it encounters a **break**

### 2.4.2 Iteration Statements

while for do while

Computers efficiently process bulk data. It might be a list of people who liked an article about a pizza establishment [13],

### 2.4.3 Evaluating Expressions

In C and C++, assignment statements produce a value, so one may use them as an expression. Doing so sometimes leads to code unclear to infrequent developers<sup>7</sup> as Figure 2.27 illustrates. This ambiguity presents a security risk [10], so secure standards advise against using syntax this creatively<sup>8</sup>. Alternatively, one might decide to create code depending on the order of evaluation of its components. This approach produces results dependent upon the compiler used during the compilation process and **not** on the algorithm itself. Figure 2.28 includes several examples of this behavior.

## 2.5 Exercises

1. Using your knowledge of data types, what value should the integer **res** contain after executing the following statement:

---

<sup>7</sup>Job Security

<sup>8</sup>EXP50-CPP. Do not depend on the order of evaluation for side effects



---

```
while (*s++ = *t++);
```

---

Figure 2.27: Assignment statements produce a result in C++, and this performs meaningful work when provided a null-terminated string.

---

```
j = ++j + 1;

arr[i++] = i;

void fun(int n){
    otherFun(n++,n)
}
```

---

Figure 2.28: Depending on the order of evaluation for expressions produces unreliable results, for the results may vary based on the compiler used and not the algorithm.

---

```
char curChar = 255;
int res = curChar;
```

---

Prepare experimental code which tests the result and compile it using two different C++ compilers. Do the results differ? Describe the nature of the problem.

2. What does the following code output:

---

```
void pointyStack(int *loc)
{
    int data = 310;
    loc = &data;
}

int main()
{
    int course = 320;
    int *ptr = &course;
    pointyStack(ptr);

    std::cout << *ptr << std::endl;
```

```
    return -1;
}
```

---

3. The function `strlen` reports the length of a string in bytes. Using this function, how many bytes does it return when provided the `char*` string literal: `"kiki\0soso"`

Using a `std::string`, what size do the methods `.size()` or `.length()` report? If there a difference between the two values, why?

4. Given the following code:

---

```
int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                  11, 12, 13, 14, 15, 16, 17, 18, 19,
                  20, 21, 22, 23};
```

---

What is the value of `x[1][0][2]`?

5. Given the following code: `int x[2] = 310;`

What is the value of `x[1]`?

## Chapter 3

# Functions

Independent of class. Declared before used. Activation records and stack frames. No guarantee of evaluation order.

Prevent code duplication. Allow abstraction.

declarations in header files

Functions help developers abstract processes out of the software. Rather than cutting and pasting code, one may abstract out the fundamental concept, generalize the parameters, and implement a common solution. Additionally, functions can help improve code readability and demonstrate intent, for sometimes a well-named function documents itself.

**Function Declarations** In C++, like its predecessor C, software must *declare* a function before using it in the file, for the compiler must understand how to prepare for the function's execution and what to expect when it completes. The compiler must know the return type and expected parameters every function takes before using it in a translation unit.

Generally, a function declaration takes the form:

```
return_type function_name( parameter list );
```

Certain keywords in the language modify the function declaration subtly. Figure 3.1 identifies the major keywords associated with functions independent of a class. Additional keywords modify the behavior of *methods* in classes, and we shall discuss those when we reach classes.

**Function Definition** At some point, the linker must connect the declared function to actual working code. The function definition, the general form of which appears in Figure 3.2, provides the actionable software for the

<b>auto</b>	return type placed after argument list preceded by an arrow operator
<b>inline</b>	Informs the compiler that the developer would prefer to inline the function body.
<b>constexpr</b>	The compiler can evaluate the function at compile time if provided with constant arguments.
<b>static</b>	Linkage specifier
<b>[[noreturn]]</b>	Function shall not use the normal return method

Figure 3.1: Keywords modifying a function declaration.

---

```

return_type function_name( parameter list ) {
    body of the function
}

```

---

Figure 3.2: The general form of a function definition. It provides the source code for the linker to use when creating the executable file.

function. Because it includes working code, we consider this a translation unit and not simply a header file.

**Forward Declaration** Instead of including a header file, one might instead choose to simply declare the function one intends to use in the source file and trust the linker to find the appropriate translation unit during compilation. This way, if an external header changes in some minor way, it does not impact the current source file. That is, when one changes a header file, every file including the header must also undergo recompilation. By including a function declaration for the relevant functions in the current scope, one breaks this dependency.

Unfortunately, doing so obscures the dependency and impacts code readability. Header files express preconditions for the compilation to succeed, and by inspecting them a developer may discern how the file fits in the overall project. Moreover, if the method signature *does* change in the library and its header file, those changes will not appear in the current translation unit. That is, when one includes `gps.hpp` in a file, any change to a function declaration impacts every source file using the header file, but with a forward declaration, this no longer holds true, for the translation unit need not consult the header when preparing for the function call. For these reasons, some firms strongly discourage the use of forward declarations [7].

---

```
void reserve(int guests);  
void reserve(int guests, float statusLevel);  
void reserve(int guests, float statusLevel, bool informedIce);
```

---

Figure 3.3: Changing the number or type of a function’s input argument requires a new declaration. These three function declarations yield unique signatures and require their own definitions.

## 3.1 Parameters

Every function and method definition in C++ must include a list of its *formal parameters*. The compiler uses this information when it prepares for and returns from the function call. In order to use a function, one must understand what *arguments* it requires. Recall that the C++ compilation process passes through several phases. It does not compile the function call when it appears in code. Instead, it inserts a placeholder reference for the linker to replace when it builds the final executable.

Every declaration of a function produces a unique signature, so two functions may use the same name if they accept a different number or type of formal parameters.

Questions arise, however, when one questions how those arguments entered the function. Did the compiler include code to copy versions of them behind the scenes, or did it include the actual variable? This becomes critical when one considers the state of the input arguments after the function call returns. If a function accepted the current number of users as its input argument, and it modified the count during its execution, should the count in the *caller* reflect the new value? Should functions work with the actual data, or should they each receive their own copy?

These issues become critical when one begins developing in a concurrent application, for what happens if one process modifies data used by another? If two threads launched with copies of the same variable at the same time, but they return at different times, which result should the caller assume?

### 3.1.1 Passing by Value

Working with a unique, scratch-pad copy of a variable allows a function to freely modify the input parameter without introducing side-effects in the larger program. By default C++ passes primitives and pointers by value. That is, it makes a copy of the variable on the stack and the function

---

```
#include <iostream>

void mutate(int* arr, int num){
    while(num-- > 0){
        *arr = arr[0] + 1;
        arr++;
    }
}

void printInfo(int* arr){
    std::cout << "Address: " << arr << std::endl;
    std::cout << "Contents[0]: " << arr[0] << std::endl;
}

int main(){
    int myData[4] = {0,2,4,6};
    printInfo(myData);
    mutate(myData, 4);
    printInfo(myData);
}
```

---

Figure 3.4: Array names, without an index, act as an address to another location in memory. We can change our copy of the address without side effects, but touching the array contents leaves lasting effects.

uses that version. Of note, for classes and structures, this results in the copy-constructor activating. Pointers, which represent addresses, enter the function as copies of the address.

Thus, a function may freely modify the *value* of a pointer without introducing unexpected side-effects. That said, although the address the pointer holds may change without impacting the caller, if the function de-references the address and modifies the data it points to, then those changes *will* persist after the function call.

### 3.1.2 Passing by Reference

At times, however, the code performs more optimally if it omits the copy step. If a function guarantees to make no changes to the input parameter, why does it need its own copy? Would it not be more efficient to simply include the actual variable as the parameter? What about situations when one intends to modify the parameter values? Consider a **swap** routine. When

---

```
void swap(int lhs, int rhs)
{
    int temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

---

Figure 3.5: A version of swap which passes by value. The swap successfully consumes processor resources, but it fails to perform any meaningful work.

---

```
void swap(int *lhs, int *rhs)
{
    int temp;
    temp = *lhs;
    *lhs = *rhs;
    *rhs = temp;
}
```

---

Figure 3.6: A version of swap using raw pointers. Instead of accepting the actual values to modify, this version takes their address and de-references it to change the variables' values in memory.

passing by value, as shown in Figure 3.5, the modified copies of the original variables disappear. Consequently, it fails to actually swap the values.

Alternatively, one might consider passing in a pointer to the variable. Instead of working with a copy, one works with an address and then de-references all subsequent calls. The problem with passing by value was that the modified values disappeared from the stack along with the changes. If, instead, one works with an address in memory and modifies the contents there, as Figure 3.6, it doesn't matter if the copy of the address disappears.

On the downside, one must constantly de-reference the values when working with them. This introduces additional notation requirements and potentially obscures the work the function performs. Thankfully, C++ offers an additional method for passing in parameters that omits the pointer syntax. When one passes a parameter by *reference*, as shown in Figure 3.7, the compiler creates an *alias*<sup>1</sup> to the variables. This allows the code to skip the pointer notation and use the variables directly. Any changes to the

---

<sup>1</sup>I can be whoever I want to be.

---

```
void swap(int &lhs, int &rhs)
{
    int temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

---

Figure 3.7: A version of swap using pass by reference. Here the function works with aliases to the integer variables.

---

```
void reserve(int guests, float statusLevel = 0,
             bool informedIce = false);
```

---

Figure 3.8: Modifying the example in Figure 3.3 to accept default arguments.

variable automatically take place in the outer program scope.

### 3.1.3 Default Arguments

Frequently, we call functions with the same input arguments. When computing a logarithm in a program, for example, one typically uses the same base. Although one may compute a logarithm using any base, some appear more frequently than others. Rather than providing unique functions for these common operations (e.g. `b2log` and `b10log`) one might instead code a single common function that accepts a parameter for the base to use in the calculation.

Although this approach creates a common function, it forces every call to the `log` function to also include information about the base to use. The additional parameter requires the caller to enter more text every time the caller uses the function. Instead, one may provide *default arguments* for the compiler to use should the source code omit the parameter.

## 3.2 Return Values

`main`, and only `main`, does not need to explicitly return a number, but all other functions that return values must return a value.



---

```
int gcd(int a, int b)
{
    if (a == 0 || b == 0)
        return 0;
    if (a == b)
        return a;
    if (a > b)
        return gcd(a-b, b);

    return gcd(a, b-a);
}
```

---

Figure 3.9: An example of a recursive algorithm to compute the greatest common divisor of two integers.

### 3.3 Recursion

### 3.4 Function Pointers

Some languages, like C++ and Python, allow source code to use a function in a parameter list or as a function's return value. This is not simply using the result of calling the function, as one would do in recursion, but this involves actually storing a function in a variable and then calling the variable using the function call mechanic.

### 3.5 Lambdas, Anonymous Functions, and Closures

Although not a functional programming language, C++11 and later include support for some popular functional features. With each subsequent revision, the language evolves to expand its functional capabilities. The `functional` header file includes several useful programming abstractions so one need not develop these basic ideas from scratch.

A closure is an instance of a function, a value, whose non-local variables have been bound either to values or to storage locations.

–Wikipedia

Sometimes, when working with higher-order functions, one needs a small, special-purpose function. When infrequently used, creating a named function might introduce more coding and complexity than necessary. In these

---

```
int binarySearch(int arr[], int left, int right, int target)
{
    if (right >= length) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) return mid;

        if (arr[mid] > target)
            return binarySearch(arr, left, mid - 1, target);

        return binarySearch(arr, mid + 1, right, target);
    }
    return -1;
}
```

---

Figure 3.10: An example of a recursive algorithm. This version of binary search returns the index of the matching element, but at what cost?

---

```
#include <iostream>

int compress(int hashCode, int numbins)
{
    return hashCode % numbins;
}

int operation(int first, int second, int (*functocall)(int, int))
{
    return (*functocall)(first, second);
}

int main()
{
    int (*stars)(int, int) = compress;

    std::cout << operation(15, 10, stars) << std::endl;

    return 0;
}
```

---

Figure 3.11: An example of first-class and higher-order functions.

---

```
std::vector<int> some_list{ 1, 2, 3, 4, 5 };
int total = 0;
std::for_each(begin(some_list), end(some_list), [&total](int x) {
    total += x;
});
```

---

Figure 3.12: This computes the total of all elements in the list. The variable `total` is stored as a part of the lambda function's closure. Since it is a reference to the stack variable `total`, it can change its value.

---

```
[](int x, int y) -> int { return x + y; }
```

---

Figure 3.13: A simple lambda function illustrating the general syntax.

situations, one might use an *anonymous function*. Anonymous functions in C++ take the general form:

```
[capture](parameters) -> return_type function_body
```

One may *capture* variables in the greater program scope by including them in the square brackets.

Capturing by reference creates potentially tricky expressions when the lambda expression outlives its captures as shown in Figure 3.15. Consequently, secure coding guidelines<sup>2</sup> address this issue by prohibiting its use [10]. Instead, one might capture the variable by value. This results in the function receiving its own copy of the variable, so it does not depend on the stack's contents.

---

<sup>2</sup>EXP61-CPP. A lambda object must not outlive any of its reference captured objects.

<code>[]</code>	Nothing used in outer scope.
<code>[<i>identifier</i>]</code>	Copy <i>identifier</i> by value
<code>[=<i>identifier</i>]</code>	Copy <i>identifier</i> by value
<code>[&amp;<i>identifier</i>]</code>	Copy <i>identifier</i> by reference
<code>[&amp;<i>first</i>,=<i>second</i>]</code>	Copy <i>first</i> by reference <i>second</i> by value

Figure 3.14: Lambda capture specifiers if the object should be copied by value or reference.

---

```
auto badlamb() {  
    int n = 108;  
    return [&] {  
        n = 310;  
        return n;  
    };  
}  
  
void awesometown() {  
    int n = badlamb();  
}
```

---

Figure 3.15: This lambda indicates captures `n` by reference with the `&` symbol. What happens when `badlamb` returns and `n` falls out of scope?

### 3.6 Exercises

1. Given an unsorted array of integers, its length, and a target sum, write a function that returns `true` if any two items in the input array combine to equal the target value.
2. The 0-1 Knapsack: Encumbrance limits restrict the number of items games allow players to carry. Consequently, while exploring and collecting inventory items, players must manually calculate which items to keep and which to discard. Write a function that performs this calculation auto-mechanically. It shall take three input parameters by constant reference: an array of item weights, and array of item values, and an integer weight limit. It shall return an array of boolean values where a `true` in the array indicates that the corresponding index item belongs in the array to maximize its value.

Weights: {1, 2, 3, 4, 5}

Values: {15,20,30,40,50}

Limit: 50

Return: {true, false, false, true, false}

3. Software Decode: Write a function that accepts an in-order array of `unsigned short` values. The function shall then scan the array for a specific pattern: Three values contained within the array equally spaced 20 units apart. The function shall return the index position

within the original array where the pattern begins or -1 if not present.

Given the input array: `data[] = {10,20,31,40,55,60,65525}`

The function shall return: 1

4. 1-1-0 Detection: Modify Problem 3 such that it returns the index of the beginning of a sequence where the first two elements are 20 units apart and there is NO element 40 units away from the first. Two elements present, one absent.
5. Modify Problem 3 to generalize the gap spacing. Instead of hard-coding the difference of 20 into the function, change the function signature to include it as a parameter.
6. Modify Problem 3 such that the input array may wrap-around or overflow.

Given the input array: `data[] = {65515,65525,9,20,31,40,55}`

The function shall return: 0

7. Using the following code fragment, what does `nigma` return when *value* = 5? *value* = 25? Symbolic answers suffice.

---

```
int nigma(int ed){  
    return (ed <= 0) ? 1 : ed * nigma(ed-1);  
}
```

---

8. Write a function that accepts four parameters: two *sorted* integer arrays and each of their associated lengths. This function shall return nothing. Instead, it shall modify the contents of the arrays passed into the function *in-place*. It shall merge the contents of the two arrays such that the smallest elements in the combined set shall appear in the first array and the largest elements shall be in the second array. That is, at the conclusion of the function `first[flength - 1] <= second[0]`.
9. The binary search example provided in Figure 3.10 uses recursion. Provide the code for an alternate version of the same algorithm in an iterative form (i.e., non-recursive).
10. Using the function defined in Figure 3.9, write a function that returns the *least common multiple* of two numbers. The solution may use `std::abs`.

$$lcm(a, b) = \frac{|a * b|}{gcd(a, b)}$$

11. Describe what happens when the caller invokes the `gcd` function defined in Figure 3.9 with `a=20` and `b=-20`. Does the computer produce a result? How might one modify the code to prevent poor behavior when provided ill-formed input?
12. The Josephus Problem: A group of  $n$  people form a circle and begin a counting out game. Counting begins at an initial starting point, and it continues around the circle along the specified direction skipping  $k$  people. Players exit the game when the counting finishes on their position, so the circle shrinks with each iteration. After removing a player, counting resumes from the next player in the circle. The final player wins the game. Given the total number of players  $n$  and  $k$  which skips  $k - 1$  players, write a function that returns the safe position within the circle. For example, if provided 5 players and with a skip value of 3, the safe position returned is 4.

Players: 5 Width: 3					
1	2	3	4	5	Initial
1	2	<b>X</b>	4	5	Round 1
<b>X</b>	2	X	4	5	Round 2
X	2	X	4	<b>X</b>	Round 3
X	<b>X</b>	X	4	X	Winner

## Chapter 4

# Input and Output

When working with values and data, a typical development task includes extracting the data from an existing file or producing some sort of modified output file. An application might prompt a user for input or display ASCII text as output. In C++, the language accomplishes these input and output tasks through *I/O Streams*.

Input streams use the `<iostream>` header file while output streams use the `<ostream>` library header file. If using both, one may simply include `<iostream>` for access to both.

**String Streams** By default, the `cin` operator stops extracting values when it reaches a space – be it whitespace, a tab, or a newline. This proves problematic when using `cin` to set the value of a `string` as shown in Figure 4.4. Instead, developers must employ the `getline` function, as shown in Figure 4.5.

Standard I/O Streams	
<code>cout</code>	System output (typically screen)
<code>cin</code>	System input (typically the keyboard)
<code>cerr</code>	Unbuffered error output
<code>clog</code>	Buffered error output

Figure 4.1: The standard I/O streams in C++. Developers may override the default destinations.

---

```
#include <ostream>

std::cout << "All those moments will be lost in time," << std::endl;
std::cout << "like tears in rain.\nTime to die.";
```

---

Figure 4.2: Writing to the screen. This code uses two different methods for inserting newline characters at the end of the text.

---

```
#include <iostream>

int main( )
{
    int value;
    cout << "What do you want: ";
    cin >> value;
    cout << "I'm not going to give you " << value;
    return 0;
}
```

---

Figure 4.3: Prompting and reading an integer from the user in C++. What happens when a giant crustacean from the paleolithic era inputs 3.50?

---

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

int main( )
{
    std::string fullName;
    std::cout << "Enter your full name (First M. Last): ";
    std::cin >> fullName;
    std::cout << fullName << " is only your first name" << std::endl;
    return 0;
}
```

---

Figure 4.4: Although the `cin` operation accepts the user's full name, when it extracts the value from the input, it terminates when it reaches the first space. Consequently, only the first name appears in the string variable `fullName`.



---

```

#include <iostream>
#include <string>

int main ()
{
    std::string fullName;
    std::cout << "Enter your full name (First M. Last): ";
    getline (std::cin, fullName);
    std::cout << fullName << " is an awesome full name." << std::endl;
    return 0;
}

```

---

Figure 4.5: Although the `cin` operation accepts the user's full name, when it extracts the value from the input, it terminates when it reaches the first space. Consequently, only the first name appears in the string variable `fullName`.

Format Manipulators	
<code>endl</code>	End of the line.
<code>setw</code>	Changes the width of the next input or output field
<code>setprecision(<i>num</i>)</code>	Display <i>num</i> floating point digits.
<code>showpoint</code>	Always display the decimal point.
<code>showbase</code>	Include the numeric base prefix.
<code>dec</code>	Display integers using decimal format.
<code>hex</code>	Display integers using hexadecimal format.
<code>oct</code>	Display integers in octal format.
<code>ws</code>	Discard leading white-space in an input stream.

Figure 4.6: Format manipulators, in the `<ios>` header file, allow one to change the default behavior of input and output streams.

---

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

int main( int argc, char* argv[] )
{
    std::ofstream dataFile;
    dataFile.open("weaponCodes.csv");

    dataFile << "Weapon Code,Description" << std::endl;
    dataFile << "1,8 in" << std::endl;
    dataFile << "2,175mm" << std::endl;
    dataFile << "3,155mm" << std::endl;

    dataFile.close();

    return 0;
}
```

---

Figure 4.7: Producing a generic CSV file in the current directory. The interface closely matches producing input and output to the screen.

# Chapter 5

## Classes

Classes represent a style of data abstraction inherent in Object Oriented programming. They allow us to group together data and behavior in a type. We communicate with objects and classes by sending them *messages* through their *methods*. Each class offers a set of methods the caller may use to communicate with the class and change its behavior.

The legacy C-style `struct` grouped together multiple variables into a single entity, but these basic abstractions lacked any methods. With the evolution to C++11, however, the `struct` type gained increased functionality, and the language introduced the concept of a class.

### 5.1 Defining a Class

Prior to using a class, the source code must describe its methods and fields. Class definitions must include the method declarations.

---

```
class Student
{
    std::string name;
    long redID;
};
```

---

Figure 5.1: A basic class definition holding two member variables with the default visibility.

---

```
class Vector3 : public Vector
{
public:

    virtual ~Vector3(){};

    Vector3();
    Vector3(double, double, double);
    Vector3(double*);
    virtual ~Vector3(){};

    Vector3 cross(const Vector3& rhs) const;
    double dot(Vector3&);
    void scale(double, Vector3&);
};
```

---

Figure 5.2: A class definition for a basic three-dimensional mathematical vector. It inherits from the general Vector type.

☞ In C++, a **struct** differs from a **class** in the visibility it assigns members without an access specifier. Structures treat members as **public** by default; Classes treat them as **private**.

### 5.1.1 Constructors

A *default constructor* accepts no parameters. By default, C++ provides a trivial default constructor for every class if the source provides none. That said, upon creating a class constructor, no matter how many parameters that constructor accepts, C++ no longer introduces an automatic default constructor. If one takes the time to specify how to construct something, the compiler assumes those parameters are essential for its construction.

Secure coding guidelines<sup>1</sup> advise developers to take extra care with an object's construction and destruction, for throwing an unresolved exception from a constructor produces inconsistent results.

---

<sup>1</sup>DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions

---

```
ClassName( const ClassName& other );  
  
ClassName( ClassName& other );
```

---

Figure 5.3: The basic format for a copy constructor in C++.

### 5.1.2 Uniform Initialization

In C++11 and later, developers may initialize the public members of a class using uniform initialization.

---

```
Pokemon(int hp) : currHp(hp), maxHp(hp) {};
```

---

### 5.1.3 Copy Constructor

C++ calls a special constructor each time a copy happens. If one uses an object as a parameter to a function, and it passes the object by value, C++ will implicitly call the copy constructor behind the scenes. It does the same thing when returning an object from a function as it copies it onto the stack. This provides the function with its own copy of the class to use, and this is also why passing in objects by constant reference may be a good idea.

### 5.1.4 Destructors

In Java, the JVM calls the `finalize()` method attached to every object. For almost all applications, the default method provided in the `Object` class remains sufficient, but for objects with complicated resource allocations, users may wish to override this behavior. One might wish to disconnect from an external server, shutdown resources, or notify an `Observer`. To do so, one need only override the method in the class definition.

The C++ language also provides a default destructor, and for many applications the default suffices. If an object allocates heap memory, however, one must explicitly remove the reference and free the memory. Failing to do so might lead to a memory leak. Destructors are noted by a tilde appearing in front of the class name.

### 5.1.5 Access Modifiers and Specifiers

Supporting the object oriented concepts of encapsulation and information hiding, programming languages like Java and C++ include special keywords

---

```
#include<cstdlib>
using namespace std;

class Label{
private:
    int serno;
public:
    Label(int number){serno = number;}
};

class Orange{
private:
    Label *farmer;
public:
    Orange(){farmer = new Label(1);}
    ~Orange(){free(farmer);}
};
```

---

Figure 5.4: Example of a class with a destructor and constructor in C++. The allocation of Heap memory requires a **free** to avoid memory leaks.

to indicate the visibility of a given variable within a scope. The goal remains simplifying the interface to other components so that only the essential is exposed.

The C programming language provides no mechanism for access modification, for it does not include classes and the structures remain data focused. That is, structures in C hold data, but they do not include methods or the notion of operator overloading. Python fully supports classes, but it does not provide a method for access modification.

To indicate a variable or method is internal in python, one simply prefixes its name with an underscore character. This naming convention helps developers achieve much of the same encapsulation without forcing it in software.

**In Java** Access modifiers are the keywords which are used with classes, variables, methods and constructors to control their level of access. Java has four access modifiers: **public**, **private**, **protected**, and **package** (or *default*).

<b>package</b>	When no access modifier is specified, its visibility is limited within the package.
<b>public</b>	It has visibility everywhere.
<b>private</b>	It has visibility only within its own class.
<b>protected</b>	It has scope within all inheriting classes and objects in the same package.

Figure 5.5: Access specifiers in Java.

<b>public</b>	It has visibility everywhere.
<b>private</b>	It has visibility only within its own class.
<b>protected</b>	It has scope within all inheriting classes.

Figure 5.6: Access specifiers in C++.

**In C++** In classes and structures, three keywords impact a variable's visibility within the current scope: **public**, **private**, and **protected**.

### Friendly Modifiers

A friend function is a function outside the class with special permission to access the internal state of the class. Because it may access the internal workings of a class, the target class must specify that the friend function has access with the **friend** keyword before use. That is, a class must explicitly identify all external functions it considers friendly.

## 5.2 Overloading Operators

### 5.3 Class Hierarchy

Sometimes, one needs to build a program up from the ground by its bootstraps, but other times it's nice to suddenly gain powers due to your parent's status. The inheritance or an estate goes to the next-of-kin in a simple model<sup>2</sup>. Object-oriented languages support this abstraction in some way.

Using inheritance, the new types defined by the software become a specialized *type* of the parent class. They possess an *IS-A* relationship. Using inheritance, we may establish a model for a simple Duck class. In this model,

<sup>2</sup>Genetic testing may complicate this in the future.

---

```
// classes_as_friends1.cpp
// compile with: /c
class B;

class A {
public:
    int Func1( B& b );

private:
    int Func2( B& b );
};

class B {
private:
    int _b;

    // A::Func1 is a friend function to class B
    // so A::Func1 has access to all members of B
    friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; } // OK
```

---

Figure 5.7: Friendly example from Microsoft.



we want to simulate multiple duck types: Wood Ducks, Mallards, and Canvasback. All three of these animals possess the abilities to quack and fly.

For ducks, apparently, each type produces a slightly different call. Consequently, in this application, we want to specify that all ducks possess this ability, but we want to require every duck type to implement their own duck sound. For our purposes, the birds in this model all fly the same way, so they do not need unique behavior for that method.

Our goal is to:

- Create a simple, universal `fly` method all ducks can share.
- Require each implementing type to define their own quack noises.

We could implement this functionality directly in each specific class. Instead, we will add a level of abstraction to the problem. Although we want to actually model three specific duck types, we will begin by introducing a higher-level abstraction: the `Duck`. We do not intend for implementations to actually instantiate bland duck objects, but every mallard and wood duck share functions, and one might combine these functions into a parent class object so one only needs to write the code in one place.

One major advantage of Inheritance emerges in its ability to introduce new functionality into the inheriting classes without changing their source code. So far, we have discussed inheriting functions in advance and by design, but we may also use inheritance during the maintenance cycle. If one were to modify the `Duck` parent class by adding new functions and their associated code, then every existing class that inherits from `Duck` also gains that ability.

- Pro: Model relationships between objects
- Pro: Code-reuse
- Pro: Add new features
- Con: Introduces dependencies which may be difficult to unravel
- Con: Tight coupling

## 5.4 Exercises

1. Using the code in Figure 5.11, code a *copy constructor* that sends the text "copied." to the standard output each time it executes.

---

```
class name : access_mode parent_class_name
{
    // Define the subclass here . . .
};
```

---

Figure 5.8: Basic structure for inheritance in C++.

---

```
#include <iostream>

namespace Targets
{
    class Duck
    {
    public:
        virtual void quack(void){};
        void fly(){std::cout << "Flap Flap" << std::endl;};
    };

    class WoodDuck : public Duck
    {
    public:
        void quack(void);
    };

    class Mallard : public Duck
    {
    public:
        void quack(void);
    };

    class Canvasback : public Duck
    {
    public:
        void quack(void);
        void fly(void) { std::cout << "Canvas Flap" <<
            std::endl;};
    };
};
```

---

Figure 5.9: An example of Inheritance in C++ using ducks.

---

```
using namespace Targets;

void WoodDuck::quack(void){
    std::cout << "Woody Quack!" << std::endl;
};

void Mallard::quack(void){
    std::cout << "Malignant Quack!" << std::endl;
};

void Canvasback::quack(void){
    std::cout << "Burlap Quack!" << std::endl;
}

int main(){

    Targets::WoodDuck fred;
    fred.quack();
    fred.fly();

    Mallard mary;
    mary.quack();
    mary.fly();

    Canvasback carl;
    carl.quack();
    carl.fly();

    std::cout << "Complete!" << std::endl;

}
```

---

Figure 5.10: Using the earlier duck example defining class methods outside their definition.

---

```
#include<iostream>
#include<cmath>
using namespace std;

class Vec3 {
private:
    double values[3];

public:
    Vec3( double v0, double v1, double v2 ){
        values[0] = v0; values[1] = v1; values[2] = v2;
    }
}
```

---

Figure 5.11: A code fragment for a simple **Vec3** mathematical vector to use for exercises.

- Building on the code in Figure 5.11, create a two parameter *mutator* method that accepts an index and updated value to use.
- Overload the *\** operator in the code in Figure 5.11 such that when the **Vec3** is multiplied by a **float** it returns a scaled **Vec3** object (e.g. **Vec3(1,2,3) \* 2 == Vec3(2,4,6)**).
- Create a *normalize* method which modifies the vector's current values to unit length. That is, the normalize function adjusts the existing values in the **Vec3** object such that the length from the origin at 0,0,0 to the point defined in **Vec3** is one unit. *Hint: Pythagoras*
- The three-dimensional cross product of two vectors produces a new vector orthogonal to the two original vectors. Produce the code necessary to create a class method for the **Vec3**, defined in Figure 5.11, which computes a cross product between its saved values and another **Vec3** vectors. This method shall take a single parameter and return a **Vec3** object with the orthogonal values.

Given two vectors  $a$  and  $b$ , the scalar components for the cross product ( $s = s_0i + s_1j + s_2k = a \times b$ ) follow:

$$s_0 = a_1b_2 - a_2b_1$$

$$s_1 = a_2b_0 - a_0b_2$$

$$s_3 = a_0b_1 - a_1b_0$$

6. Primitive arrays, using the standard bracket notation, provide powerful functionality in both C and C++, but they lack many of the methods and features object oriented programmers expect. Create a class, called `DynamicCharArray`, which wraps around the standard `char` array and offers the following features:
  - Default constructor: initializes an internal array of 8 elements
  - Copy constructor: called implicitly when making a copy
  - `int size()`: returns the length of the array
  - `void expand(int amount)`: increases the capacity of the array by the specified amount. It will need to create a new internal array and copy the elements over to accomplish this correctly, but it need not initialize the increased space.
  - Destructor: if the solution allocates heap memory, the destructor shall free it.
7. Override the `=` operator in Problem 6 such that it copies the elements entirely from the other array into this array.
8. Building on the class defined in Problem 6, override the `[]` operator to provide element access to the internal data.

DRAFT

## Chapter 6

# Scope and Namespaces

### 6.1 Scope

Programs *bind* the names of variables in source to particular memory locations during their execution. A variable is in scope in a program all the places where one may legally reference that variable name. We introduce a new scope in C++ when we begin a function call, iteration, or even when we simply introduce empty curly braces. Figure 6.1 illustrates using curly braces to begin a new scope, but it contains an error that prevents compilation. The inner-block of code declares a variable, but the outer-block attempts to assign to it, and the compiler prevents the assignment.

Pointers represent a challenge to scope, for one may hold a lingering pointer to a variable no longer in scope. This may lead to unexpected, unrecoverable behavior, or it may represent a potential security risk. Figure 6.2 demonstrates an example of this error.

---

```
{
    {
        int block = 2;
    }

    block = -1;
}
```

---

Figure 6.1: This source contains errors and will not compile due to variable scope issues. The variable `block` is not visible in the outer compound statement.

---

```
include <iostream>

int main(){

    unsigned short *ptr;

    {
        unsigned short data[65535] = {310};
        ptr = data;
    }

    std::cout << "Reading: " << *ptr << std::endl;
    *ptr = 320;
    std::cout << "Writing: " << *ptr << std::endl;

    // data[0] = 1; NO data is out of scope
}
```

---

Figure 6.2: Although free of syntax errors, this code includes a pointer to a variable out of scope. This represents a significant logic error.

Typically, students experience difficulty with scope when they attempt to access a loop counter variable inappropriately.

## 6.2 Namespaces

Secure coding guidelines<sup>1</sup> advise against modifying the default namespace [10]. That is, the `std` namespace includes those tools available to everyone by downloading and installing the standard libraries. By introducing custom code in this namespace, it no longer clearly delineates the custom code from that which arrives stock.

If one does not include a namespace identifier in a source unit, then the compiler automatically places it in the *global namespace*. Keeping items in this namespace serves useful purposes in an educational environment, but in any meaningful program good form dictates that developers place their classes in an appropriate namespace. C++ places some of the most universally useful tools in the `std` namespace. These subroutines and objects create the *C++ Standard Library*.

---

<sup>1</sup>DCL58-CPP. Do not modify the standard namespaces



---

```
for( int i = 0; i < 100; i++){  
    // visible inside the loop  
    i += 3;  
}  
// std::cout << i; NOT here
```

---

Figure 6.3: Variable visibility in a simple iterative for loop. The loop control variable falls out of scope when the loop ends, so the compiler does not understand the binding.

---

```
#include <iostream>  
  
int x = 1;  
int y = 2;  
  
void clash(int x, int y){  
    x = 3;  
    y = 4;  
}  
  
void bang(){  
    x++;  
    y++;  
}  
  
int main(){  
    int x = 50;  
    int y = 60;  
  
    clash(x,y);  
    bang();  
  
    std::cout << x << ',' << y << std::endl;  
    return 0;  
}
```

---

Figure 6.4: The compiler attempts to resolve the binding as close to the use as possible, so the inner declaration shadows the outer name. This really feels like an exam question, doesn't it?

---

```
namespace Fleet
{
    // a class object in the namespace
    class SuperDimensionalFortress
    {
    public:
        void Transform(){}
        bool isSelfAware(){}
    }

    // This is C++, so a function without a class in the namespace
    void CheckStatus(SuperDimensionalFortress){}
};
```

---

Figure 6.5: An example of namespace in C++

---

```
Fleet::SuperDimensionalFortress sdf;
sdf.Transform();
Fleet::CheckStatus(sdf);
```

---

Figure 6.6: Working with the fully-qualified name.

---

```
using Fleet::SuperDimensionalFortress;
SuperDimensionalFortress sdf;
```

---

Figure 6.7: Bringing one identifier into *scope* with the **using** directive.

---

```
using namespace Fleet;
SuperDimensionalFortress sdf;
sdf.Transform();
CheckStatus(sdf);
```

---

Figure 6.8: Grabbing everything in the namespace and bringing it into scope.

---

```
import java.util.ArrayList;
import java.util.List;

...

List<Integer> myList = new ArrayList();
```

---

Figure 6.9: Pulling in subroutines defined in other modules in Java through the `import` statement.

In addition to defining a namespace at the global level, one might wish to break sub-sections of code into their own, distinct namespaces. That is, one may wish to *nest* namespaces. Entities within a nested namespace have access to all other entities within the same, nested-namespace, but they may also access entities within their parent namespace.

On the other hand, entities in the outer namespace may not access the members of the inner namespace without qualification.

### 6.2.1 Java Packages

By reference, in Java one may group together related classes in a common **package**. Were one developing custom data structures, for example, one might elect to place them all in the package `datastructures`<sup>2</sup>. This way, when you reference your custom **Vector** object, it does not conflict with a similar structure in the Java Collections Framework.

All entities within the same package in Java have access to the **public**, **protected**, and unspecified access permission level (package-scope) of all other types within the package. That is, everything in the same package may peer into all but the inner-most, **private** workings of the objects.

---

<sup>2</sup>Or `edu.sdsu.cs.datastructures` if you have an awesome 310 instructor

DRAFT

## Chapter 7

# Templates and Genericism

**Class Templates** provide a way to specify that some of the class' members come from templates.

**Function Templates** identify that a function, independent of a class, accepts generic parameters.

**Non-type Parameters** extend beyond simply specifying that a type in the code is generic, one may even supply generic values. This functionality extends C++ templates into a new level of potential complexity.

---

```
template <class T, int N>
void DynamicArray<T,N>::set(int x, T value) {
    storage[x]=value;
}
```

---

Generic programming facilitates designing algorithms and data structures in terms of their operation and use and not the specifics of what they hold. It attempts to abstract out the key concepts by deferring many of the implementation specifics until the point of instantiation. A queue for coffee, for example, provides the same operations as a queue for cattle on a dairy. The items in line, be they coffee or people, sequentially exit in the order they arrived. Thus, one can program all the operations on the queue independent of what goes inside.

Languages provide differing levels of support for Generic programming, and they tend to refer to it in slightly different terms. Be it *generics* in Java and Python or *templates* in C++, generic programming ultimately seeks to reduce the amount of code developers need to write and maintain.

---

```
template <class Receiver>
class SimpleCommand : public Command {
    Action _act;
    Receiver *_rec;

public:
    typedef void (Receiver::* Action)();

    SimpleCommand(Receiver* r, Action a) :
        _rec(r), _act(a) {}

    virtual void Execute(){_act->Execute();}
}

```

---

```
m1a1 = Tank();
SimpleCommand<Tank> tnkOff = SimpleCommand(m1a1, &Tank::powerOff())
tnkOff.Execute();

```

---

Figure 7.1: An example of a C++ template for the Command design pattern in *Design Patterns*.

The Python programming does not require any formal syntax with generics. Instead, it uses a strategy called *duck typing*. The idea follows the saying, "if it looks like a duck and quacks like a duck, it's a duck." Instead of trying to verify the operation is permitted at compile time, Python assumes it is and acts accordingly.

Templates support generic programming and design abstraction. They represent a general method for creating source code to accomplish a task. In C++, the process of generating the source code from the template is called *template instantiation*. The resulting template code is a *template specialization*. Multiple specializations may result from a single template.

One cannot explicitly state a template's requirements in code, and the user must glean this from the design. It behaves somewhat like a macro substitution. One can easily create a syntax error by calling a macro with invalid input. Likewise, one might introduce an error by instantiating a template in C++ with an object that doesn't support the required types or includes nonsensical values.

## Part II

# Analysis





## Chapter 8

# Complexity

How rapidly does something grow? For example, if a linear algorithm takes an hour to complete with the current number of users, how long will it require with twice the users? Growth might occur in the amount of time it takes something to complete, or it could manifest in the additional amount of storage required as the problem size increases.

When measuring complexity growth, one needs at least two data points. For example, knowing a car passed a highway patrol officer at exactly noon provides little actionable information about its speed. If, however, we include an observation from a second highway patrol officer precisely one mile away who notices the exact vehicle passing that officer's position thirty seconds later. Given these two data points, reckless driving seems like a likely outcome <sup>1</sup>.

### 8.1 Empirical Analysis

When experimenting with working code, measuring the amount of time an algorithm or section of code takes to complete provides insight into its operation. By systematically timing and increasing the input size, one may simply measure how much longer it took to perform the operation. Based upon the results, and assuming sufficient input size, one may approximate the growth curve.

Of course, this requires working code, and for problems beyond trivial educational projects this may present a programming paradox, for cannot

---

<sup>1</sup>Although it might be Highway Patrol night at The Monster Mile and the officer one mile away is also six inches away.

use the analysis to justify implementing the design if one needs the implemented design to perform the analysis. In order to justify changes, however, one may use empirical timing results to illustrate a system's current architectural bottlenecks.

Additionally, an analysis of working code remains strictly tied to the machine, and operating conditions, under which the testers performed the analysis. That is, timing results on one computer carry no significance on any other machine. Algorithms running on a machine with a modern graphics processing unit, for example, mean nothing when contrasted against the same algorithm implemented on a TRS-80 color computer. Even on machines running identical hardware, background processes may impact the measured time.

Both Java and C++ provide convenient library and system functions that accomplish timings at various precision (e.g., `System.nanoTime()`, `System.currentTimeMillis()`, `std::clock()`). The code in Figure 8.1 illustrates the C++ mechanism for accomplishing timings.

## 8.2 Theoretical Analysis

Machine independent. May work from design concept.

### 8.2.1 Complexity Classes

## 8.3 Exercises

For each code fragment below, give the complexity class of the algorithm using  $O$  or  $\Theta$  notation as appropriate. Give the tightest possible upper bound. The input size variable is  $n$ .

1. \_\_\_\_\_  

```
for(int i=0; i < n/2; i+=2)
  for(int j=0; j < n/4; j+=4)
    for(int k=0; k < n/8; k+=8)
      for(int m=0; m < n/16; m+=16)
        x++;
```

\_\_\_\_\_

2. \_\_\_\_\_  

```
for(int i=1; i <= n; i*=2)
  for(int j=1; j <= i; j+=2)
    x++;
```

\_\_\_\_\_

---

```
#include <iostream>
#include <iomanip>
#include <chrono>
#include <ctime>

void consumeTime()
{
    double d = 0;
    for(int n=0; n<10000; ++n)
        for(int m=0; m<10000; ++m)
            d += d*n*m;
}

int main()
{
    std::clock_t clockStart = std::clock();
    consumeTime();
    std::clock_t clockEnd = std::clock();

    std::cout << std::fixed << std::setprecision(2) << "CPU time
        used: "
        << 1000.0 * (clockEnd - clockStart) / CLOCKS_PER_SEC <<
        " ms\n"
        << std::endl;
}
```

---

Figure 8.1: Measuring elapsed time on a particular machine.

---

3.

```

for(int x = 0; x < n; x++)
{
    for(int i=0; i < n; i++)
        std::cout << "Single";
    for(int j=0; j < n; j++)
        std::cout << "Letter";
    for(int k=0; k < n; k++)
        std::cout << "Variables";
}

```

---

4.

```

int x = n;
for(int i=0; i < 10000; i++)
    while((x-=2) > 0)
        std::cout << "The value of x is: " << x << std::endl;

```

---

5. Given the empirical times below for the algorithm X, what is the likely  $O$  complexity of this algorithm?

$n$	Time in nS
100K	1206
200K	2707
300K	3496
400K	5310
500K	7774
600K	8922

6. If a certain function  $g(n) = \Omega(n^2)$ , could it also be  $\Theta(n^2)$  and/or  $O(n^2)$ ? Explain.
7. An application with 1200 clients gained increased popularity after a development conference where it acquired another 5000. Currently, the back-end processing software the server executes every night takes 30 minutes to run. Given that the complexity of the processing software is  $O(n^2)$ . How long will the program take to run with the increased load?
8. After developing a specialized search algorithm, young Anakin rushed into empirical testing. With a limited sample size of  $n = 100$  people, the search took *1 hour*. After a quick check of the results, however,

he noticed the test only included the men in the original data. After he corrected this mistake and included the women and children in the test, the sample size increased to  $n = 1000$ . Predict, as accurately as possible, the maximum duration the algorithm will take with the new sample size if the algorithm's complexity is  $O(n^2)$ .

9. How long will the algorithm in question 8 take if the algorithm is  $O(n!)$  instead of  $O(n^2)$ ?
10. Given a list of integers, return the sub-sequence that produces the maximum sum. For example, the input list 2, 3, 5, -33, 5, 6, 1, 3, 3 produces: 5, 6, 1, 3, 3

DRAFT

DRAFT

## Chapter 9

# Sorting and Searching

When discussing different sort algorithms, one must consider the sort's behavior with respect to the items it sorts and any additional resources the sort requires. Two terms help characterize an algorithm: *in-place* and *stable*.

An otherwise speedy algorithm may become unusable in an application if the algorithm's storage requirements grow beyond the machine's capacity. The term *in-place* references algorithms whose memory requirements remain constant no matter how large the problem grows. These procedures may use scratch variables as necessary, but the amount of scratch memory remains totally independent of the input size. Algorithms cannot allocate a duplicate array or data structure in any way tied to the input size and still claim *in-place* performance.

How should a sorting algorithm handle duplicate entries? For simply sorting numbers, without any additional context, this characteristic may seem meaningless. When the numbers the sort works with reference customer priority levels, however, and each number corresponds to a unique person, the order suddenly matters<sup>1</sup>. A *stable* sorting algorithm guarantees that the order the individual duplicates appeared in the original list appears in the sorted list.

### 9.1 Binary Search

Case	Complexity
Best	$O(\log(n))$
Worse	$O(n^2)$
Actual	$O(kn)$

---

<sup>1</sup>At least for the people inside.

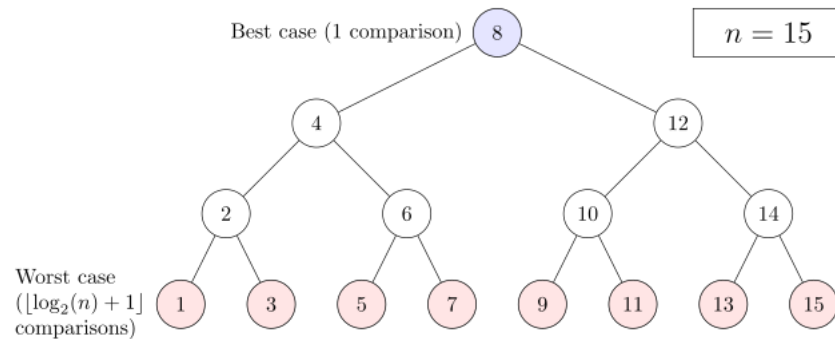


Figure 9.1: An visualization of binary search's complexity. This image courtesy the Wikimedia Commons.

Scanning through an unsorted array to detect the presence of a particular value requires linear time, for one must inspect every item in the array to establish a missing value. For sorted arrays, however, this no longer holds true, for the data themselves inform the algorithm about where it should look for the target value. Rather than simply starting from the front of the array and repetitively inspecting each element, one could easily eliminate half the values by simply starting the search at the middle element.

## 9.2 Selection Sort

Case	Complexity
Best	$O(n^2)$
Worse	$O(n^2)$

Selection sort works *in-place* by partitioning the array to sorted and unsorted portions. Initially, the sorted portion holds a size of zero and the unsorted portion takes up the entire array space, but after the first sort iteration, the sorted size will equal one and the unsorted portion will contain one less item. The algorithm works by scanning the entire unsorted portion of the array for the smallest item it contains. After it finds the item with the smallest value, it swaps it with the item at the front of the unsorted array. This effectively grows the sorted portion by one and shrinks the unsorted region by one.

Anytime an algorithm blindly swaps into an array, as the selection sort algorithm does when it throws the item at the front of the unsorted portion into the spot formally associated with the smallest value item, it scrambles



---

```

int binarySearch(int arr[], int left, int right, int target)
{
    if (right >= left) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) return mid;

        return (arr[mid] > target) ?
            binarySearch(arr, left, mid - 1, target) :
            binarySearch(arr, mid + 1, right, target);
    }

    return -1;
}

```

---

Figure 9.2: The source code for a recursive version of the binary search algorithm.

the algorithm's stability. Due to this behavior, selection sort lacks stability.

### 9.3 Insertion Sort

Case	Complexity
Best	$O(n)$
Worse	$O(n^2)$
Actual	$O(kn)$

Much like selection sort, insertion sort works *in-place* by partitioning the input array into two parts: sorted and unsorted. Unlike the earlier sort, however, insertion sort only works with one new item at a time. Rather than scan the entire unsorted array for the smallest item, simply *insert* the item at the front of the unsorted portion where it belongs with respect to the sorted array.

The initial array:

dataToSort =	10	9	2	8	7	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

1. Pass One:

dataToSort =	10	9	2	8	7	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

---

```
void selectionSort(int arr[], int n)
{
    int min_idx;

    for (int i = 0; i < n-1; i++)
    {
        min_idx = i;
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

---

Figure 9.3: Selection sort's source code.

---

```
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; i++)
    {
        int j = i - 1;
        int cur = arr[i];

        while (j >= 0 && arr[j] > cur)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = cur;
    }
}
```

---

Figure 9.4: Insertion sort's in-place source code requires a constant number of scratch variables.

## 2. Pass Two:

dataToSort =	10	9	2	8	7	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
dataToSort =	9	10	2	8	7	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
dataToSort =	9	10	2	8	7	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

## 3. Pass Three:

dataToSort =	9	10	2	8	7	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
dataToSort =	9	2	10	8	7	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
dataToSort =	2	9	10	8	7	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
dataToSort =	2	9	10	8	7	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

At this point, we begin to witness the complexity buildup. As we add 2 to the sorted list, it must shuffle through every position currently in the sorted portion. This swap requires processor time. If every new item entering the sorted portion of the array requires shifting the existing contents, which requires linear time, then sorting  $n$  items requires:

$$O(n) * O(n) = O(n^2)$$

## 4. Pass Four:

dataToSort =	2	9	10	8	7	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
dataToSort =	2	9	8	10	7	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
dataToSort =	2	8	9	10	7	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
dataToSort =	2	8	9	10	7	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

5. Pass Five:

dataToSort =	2	8	9	10	7	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
dataToSort =	2	8	9	7	10	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
dataToSort =	2	8	7	9	10	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
dataToSort =	2	7	8	9	10	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
dataToSort =	2	7	8	9	10	3	6	5	4	0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

## 9.4 Merge Sort

Case	Complexity
Best	$O(n \log(n))$
Worse	$O(n \log(n))$

Rather than comparing and swapping entries throughout the array, the merge sort algorithm breaks the input array into sub-arrays and then combines them back together in order in a new array. Unlike both insertion and selection sort, merge sort uses another working array during its operation. Consequently, this algorithm fails to meet the requirements for in-place.

With merge sort, one might actually store more items than one can store in memory or on a single storage media. For example, one could combine a massive customer list in Asia with one from Europe onto two new storage disks in sorted order. The first would contain an alphabetical list of customers, starting from the first and ending when the disk lacks additional capacity. Then, on a second new disk, the algorithm would begin storing the remaining customers in alphabetical order where it left off on the first list.

Although clearly not an in-place algorithm, it does preserve the order of duplicates so long as the code always takes from the left array first in the event of duplicate entries at the front of the left and right lists.

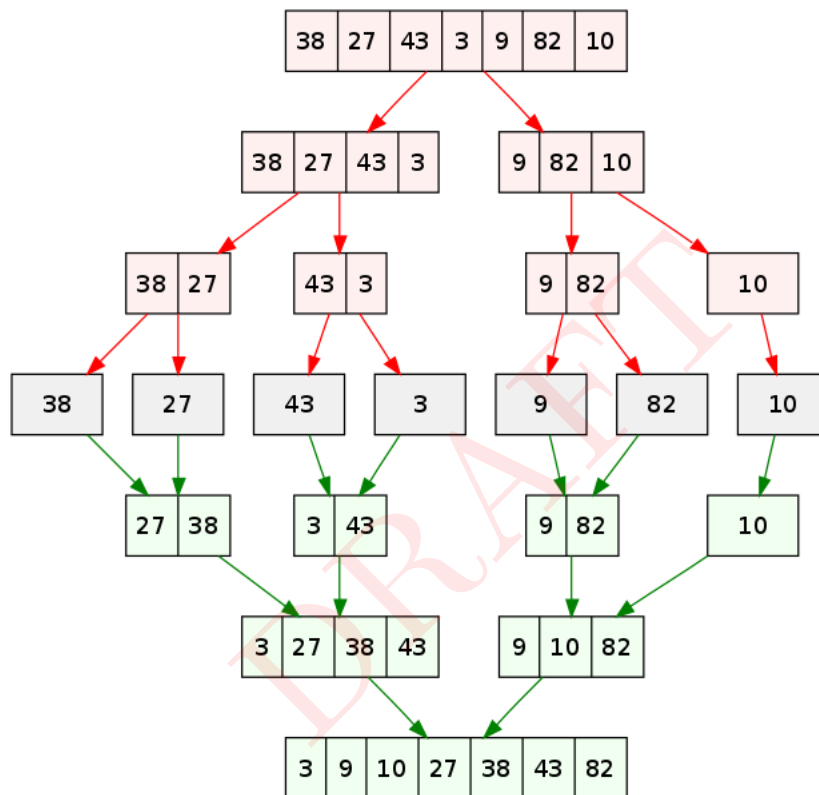


Figure 9.5: A breakdown of merge sort's steps working from top down. This image courtesy Wikimedia and the Public Domain.



Figure 9.6: Radix sort assisted Herman Hollerith during the tabulation of the United States census.

## 9.5 Radix Sort

Case	Complexity
Best	$O(wn)$
Worse	$O(wn)$

Although we typically discard coefficients when discussing the complexity of an algorithm, with Radix Sort, the coefficient grows large enough to impact performance. Moreover, even though radix sort uses linear complexity, another sort belonging to  $O(n \log(n))$  will scale the linear factor less than radix sort's coefficient. Additionally, unlike insertion sort, radix sort requires additional working memory proportional to the input size.

The algorithm works by shuffling items from the input array into different buckets. In a base-10 system, unlike what a computer uses, one requires ten different buckets for each of our digits 0-9. For each digit, the algorithm allocates an array capable of holding new items inside. As the algorithm works from least significant digit toward most significant digit, it places the item in the appropriate bucket. After processing the least significant digit for every number in the input array, it pulls everything out of the buckets it sorted them into and starts again using the next digit.

dataToSort =									
02	17	28	90	10	03	06	15	24	00
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Digit	
0	90, 10, 00
1	
2	02
3	03
4	24
5	15
6	06
7	17
8	28
9	

After placing each value into the appropriate bin, we may now return these items to the original array. We do so by emptying out each bucket, in-order.

dataToSort =	90	10	00	02	03	24	15	06	17	28
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Then we may again place the items from the original array into the appropriate, corresponding bucket.

Digit	
0	00, 02, 03, 06
1	10, 15, 17
2	24, 28
3	
4	
5	
6	
7	
8	
9	90

Finally, we may empty out each bucket and return the data to the original array. Having processed each digit, the algorithm completes and leaves us with a sorted array.

dataToSort =	00	02	03	06	10	15	17	24	28	90
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Thankfully, computers use binary representations of numbers, not their base-10 form, so we require only two buckets within which we sort items. Unfortunately, we must process every item in the list to sort at least once

for each digit. Binary numbers contain as many digits as bits required to represent the underlying type, so a two-byte integer requires *thirty-two* passes.

## 9.6 Quick Sort

Case	Complexity
Best	$O(n \log(n))$
Worse	$O(n^2)$

A divide-and-conquer algorithm, quick sort implements a comparison sort. The algorithm works by repeatedly partitioning the array into two halves based upon a chosen pivot value selected from within the list. With each pass of quick sort, one more element finds the correct final position in the array. Significantly, the partitioning ensures that everything to the left of the pivot's final position in the array contains smaller values, and everything to the right of the pivot contains larger values. Ideally, the pivot cleanly divides the array into two halves, but only the arrangement of the data themselves determine this behavior.

The algorithm follows these basic steps:

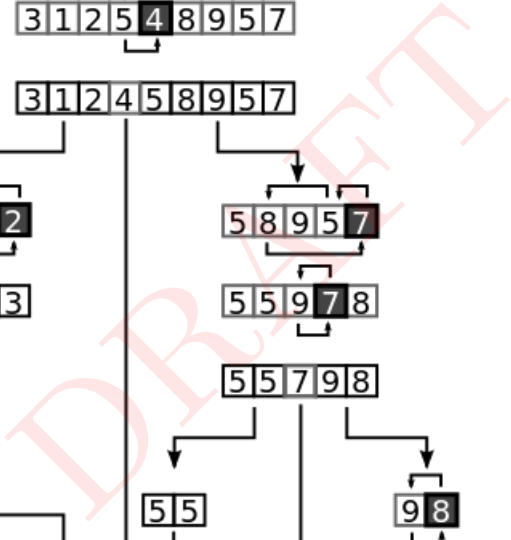
1. Select a pivot value from within the list.
2. Rearrange the elements in the array such that everything that appears to the left of the pivot contains smaller values and everything to the right holds larger values.
3. Recursively call this procedure on each sub-divided half of the array.

Performance degrades when the pivot selection fails to divide the array. An in-order series of numbers (either descending or ascending) places the pivot value solidly at one extreme<sup>2</sup>. Consequently, quick sort's worst case performance reflects this imperfection. When quick sort cannot divide-and-conquer, it must linearly scan the list a linear number of times. This produces  $O(n^2)$  performance.

---

<sup>2</sup>Assuming one simply selects the last item in the list as the pivot. One cannot escape this behavior by permuting it and choosing an alternate element as the index.





WikiMedia and the public domain.

## 9.7 Exercises

1. What is the complexity, in  $O()$  notation, for performing each of the following sorts on an array of size  $n$ :
  - (a) Using *selection* sort when the input is in its natural order (ascending)?
  - (b) Using *insertion* sort when the input is in its natural order (ascending)?
  - (c) Performing *merge* sort on an array filled entirely with duplicate items?
  - (d) Using *quick* sort on an array in apparently random order?
  - (e) Using *quick* sort on an array filled entirely with  $n$  copies of the same item?
2. Using the insertion sort algorithm, starting from index position 0, show the contents of the input array after each indicated position swap. The unsorted portion should shrink and the sorted portion should grow with every pass.

dataToSort =

10	-7	4	2	-10	22	8	7	9	0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

After completing the *third* pass:

dataToSort =

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

3. The code for binary search shown in Figure 9.2 uses recursion to perform the search. Rewrite the algorithm to eliminate the recursion (e.g., using a **while** loop).

# **Part III**

## **Data Structures: Linear**



Abstract Data Type	Data Structure
List	Array Tuple Dynamic Array Linked List
Queue (priority)	Linked List Heap Ordered Array
Stack	Linked List Dynamic Array
Dictionary / Map	Hash Table 2-3 Tree Splay Tree AVL Tree Red-Black Tree
Graph	Adjacency List Adjacency Matrix

Figure 9.8: Abstract data types and data structures implementing the interface.

DRAFT

# Chapter 10

## Lists

### 10.1 Stack

Possible to create with an array.

### 10.2 Queue

Single linked list configured the correct direction

#### 10.2.1 Priority Queue

### 10.3 Random Access Lists

#### 10.3.1 Array

#### 10.3.2 Dynamic Array

#### 10.3.3 Circular Array

### 10.4 Sequential Access Lists

The array based lists offer rapid access to any element within the structure in  $\Theta(1)$  due to random access, but one cannot easily append elements to the front or rear of the structure, for doing so requires one shuffle the data inside accordingly. Moreover, because the system guarantees it returns space for the array in contiguous memory locations, increasing the array size requires a second allocation for even more space. What happens if the data are too large for a contiguous memory block?

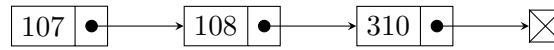


Figure 10.1: The basic structure of a sequential access list. Each Node connects to the next through a saved address.

---

```

struct Node
{
    Node(int value) : data(value){next = NULL;};

    int data;
    Node *next;
}
  
```

---

Figure 10.2: A basic node for a singly linked list. It contains a place to store the data value as well as a link to the next Node in the chain.

Linked lists provide all the same **List** functions as their array-based counterparts, but they implement the functions in different ways. Rather than keeping the data grouped together, a linked list distributes it throughout the memory space. As an advantage, one no longer needs to move items in memory when inserting in the front, rear, or middle of the list. For this functionality, however, the linked list sacrifices random access. Instead, link lists use *sequential access*. This means that in order to get to position  $i + 1$  in the list, one must first navigate to  $i$ . Thus, to get to the fourth item in the list, one must first visit the first, second, and third elements.

Each position in the list maintains both a value (the data relevant to the application), as well as the address to the next item in the list. This requires more overhead than a traditional array, for the index positions naturally inform one where to look in memory for the data.

#### 10.4.1 Singly Linked List

The basic, the singly linked list includes the address of the next item in the list. One may think of this as a one-way street, for going forward through the list provides the data for the next item in the sequence, but it remains silent about its origin. That is, one may understand where next to go to find an item, but one cannot traverse the list in reverse. From index position four, the singly-linked list cannot return to index position three, but it can traverse to index position five.



---

```
void addLast(Node *start, int value){
    if( start == null ) return;
    if( start->next == nullptr ){
        start->next = new Node(value);
    }
    else addLast(start->next, value);
}
```

---

Figure 10.3: A tail-recursive `addLast` method for a basic singly-linked list.

---

```
int removeFirst(Node *&head){
    if( head == nullptr ) return nullptr;
    int toReturn = head->data;
    head = head->next;
    return toReturn;
}
```

---

Figure 10.4: This function accepts an address to the start of the list as an aliased variable. To remove the first item, simply point the head to the second item. What is this C++ code missing?

In an object-oriented programming model, one might include node objects as shown in Figure 10.2. Linked lists predate object oriented programming, however, so one need not introduce classes and methods. Rather than implementing the `add` and `remove` methods as part of a class, one simply creates functions that take the node as a parameter. The code in Figure 10.3 shows one method for achieving this by passing in the working node.

Removing items from the front of a singly-linked list, unlike a standard array, requires constant time, for one need only update the pointer to the start of the list. Instead of pointing `start` to the current front item, move it so it points to `front->next`. No matter how long the list spans, this operation requires the same number of steps.

#### 10.4.2 Doubly Linked List

In many applications, one only needs to traverse a list in one direction, but a task might require the user to jump backwards in the list. For example, in a two-button menu system, users frequently move up and down through the displays. In a word processor, one may undo and redo commands. For

---

```
struct DNode
{
    DNode(int value) : data(value){next=NULL; prev=NULL;};

    int data;
    DNode *next;
    DNode *prev;
}
```

---

Figure 10.5: An updated node for a doubly linked list. With the addition of the back-pointer, most of the space to store the node goes to keeping track of list maintenance data.

these situations, a doubly-linked list may work wonderfully.

Unfortunately, the additional link in the structure requires substantial code, for the linked list node must now maintain both a `next` and `previous` pointer. Adding to the list no longer impacts a single node, for now one must update both what came before and what comes next. This back-pointer to the node's predecessor also requires more storage space.

### 10.4.3 Linked List with a Tracer

By working with pointers to pointers and addresses of addresses, a function in C++ might actually modify an alias to the input node. Rather than passing in simply a pointer to the start of the list, pass in the address where *that* pointer lives in memory. Then, when one changes it, one changes its value in the outer scope. In C and C++, a function can set the linked list's starting node to an entirely new node. Java developers cannot achieve this precise functionality due to the language's design.

Significantly, the use of aliased variables in C++ falls well within the language's intent and style. This functionality allows the language to support these types of data structures without incorporating a strictly object-oriented design. One need not preserve or maintain the number of items inside the list if one never intends to check it. C++ developers need not include this inefficiency. At the same time, one might alternatively construct a class in C++ which accomplishes all the list requirements while preserving its state in some internal variables.

To achieve specifically *tracer* functionality, as shown in Figure 10.7, one must use triple reference pointers, and the Java language took steps in its design to strictly avoid this level of referencing. The power it offers in-

---

```
void insertAfter(struct DNode* previous, int value)
{
    if (previous == nullptr) return;

    DNode* toAdd = new DNode(value);

    toAdd->next = previous->next;
    previous->next = toAdd;
    toAdd->prev = previous;

    if (toAdd->next != nullptr)
        toAdd->next->prev = toAdd;
}
```

---

Figure 10.6: An example of the complications introduced by the second pointer. Adding another node to the list requires substantial updating.

---

```
void insertAtPosition(DNode **head, int value, int position)
{
    DNode *toAdd = new DNode(value);

    DNode **tracer = head;

    int cursor = 0;
    while((*tracer) && (++cursor < position))
        tracer = &(*tracer)->next;

    toAdd->next = *tracer;
    *tracer = toAdd;
}
```

---

Figure 10.7: Triple reference pointers in C and C++ allow one to use casting to access the data contents, yet it makes it easy to work directly with the head reference if necessary.

troduces potential security vulnerabilities as makes the code less readable to inexperienced developers<sup>1</sup>. When working with other C++ developers, however, this code style quickly becomes second nature.

#### 10.4.4 Circular Linked List

The list examples given so far use a `nullptr` value to establish the list's end. Although this variable holds no meaningful data, it requires sixty-four bits on a modern computer to hold this empty data value. Instead, we shall point this otherwise wasted space toward the list's head. Even the last item in the list holds information about a next node using this approach, and one may safely traverse past the list's end without creating a segmentation fault. By continuing to navigate the `next` links contained in a circular linked list, one may endlessly scroll through the list.

Using this approach, one only needs to store a pointer to the list's *end*. Rather than keeping track of the start or head, simply maintain a single reference to the list's end. To get to the front, one need only refer to it as `tail->next`.

---

<sup>1</sup>Job security . . . or how to get kicked off a team?

## 10.5 Exercises

1. What is the complexity, in  $O()$  notation, for:
  - (a) Retrieving the value for a single item located at an arbitrary position within any *random access* data structure?
  - (b) Using binary search to find a single item in a doubly linked list containing  $n$  items inside?
  - (c) Removing a single item from the end of a doubly-linked list of size  $n$  with both a head and tail pointer reference?
2. Rewrite the code in Figure 10.3 by replacing the recursive call with a `while` loop.
3. How does one correctly implement a queue, within the expected complexity for each major operation, using only a singly-linked list? What additional variables must one track? From what end does the dequeue operation remove items in the queue?
4. Doubly linked lists require additional overhead, but, they add functionality not available with singly linked lists. Describe two benefits provided by doubly linked lists that are not provided by singly linked lists. The lists in this question are unordered.
5. Given an array of *unsorted* integers and a target value, write a function that scans the input array and displays two values, and their index positions, within the array with a sum totaling the target value. What is the complexity of your solution?
6. Write a function called `isCircular` that accepts a parameter of type `Node`, as defined in Figure 10.2, and returns `true` if the node is part of a circular linked list.
7. Using a singly linked list of characters, write a function called `isPalindrome` which accepts a `std::string` and returns `true` if the string is a palindrome.
8. Write a function called `rotateRight` which, when provided a singly linked list and an integer value, rotates the elements the specified number of positions to the right. That is, rotating a list one position to the right causes everything inside to move to the right while the element at the list's end moves to index zero.

DRAFT

## Part IV

# Data Structures: Non-linear

DRAFT





## Chapter 11

# The Binary Heap

A non-linear, array-based data structure.

The heap:

1. Is *either* a minimum or a maximum heap.
2. Uses *left-justification*
3. Is a *complete* binary tree.

### 11.1 Array-based Trees

Software developers must constantly maintain the links contained in a node-based data structure. These extra links require additional storage space, for one must store the address to the next datum. With an array, however, all values are adjacent with one another in the computer's memory. Due to the heap's rules, which carefully limit its shape,

left	$2i + 1$
right	$2i + 2$
parent	$\frac{i-1}{2}$

### 11.2 Insertion

Due to the heap's strict structural rules, adding one item to the heap changes its shape in a deterministic way. Without question, and regardless of the data set, a heap with six items will take the shape as shown in Figure 11.4. This fact holds true no matter how many values the heap contains. The

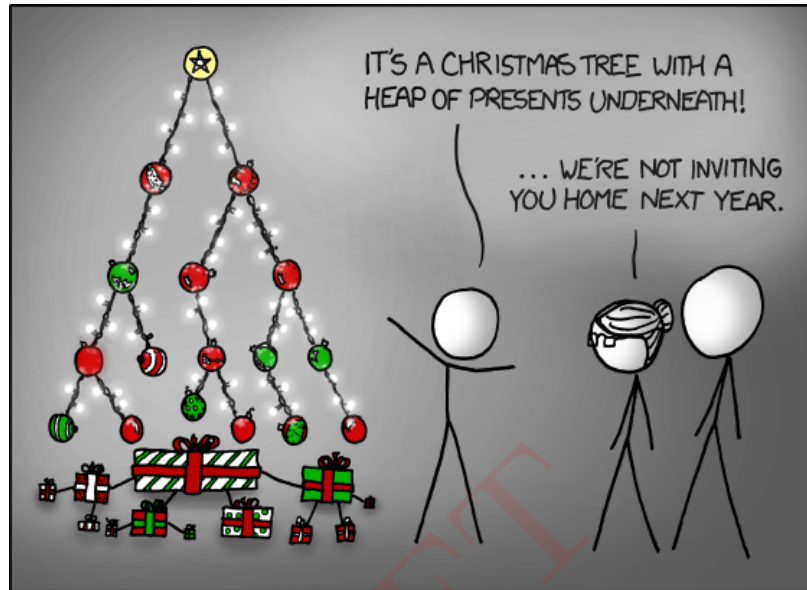


Figure 11.1: Trees and Heaps courtesy of XKCD and the Creative Commons.

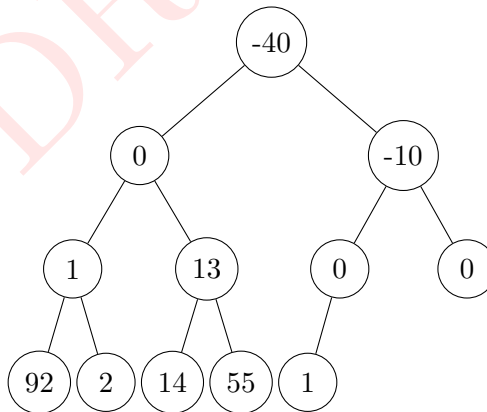


Figure 11.2: The minimum heap property holds for every node within the heap. All heaps, regardless of the input values, are complete, left-justified trees.

-40	0	-10	1	13	0	0	92	2	55	14	1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Figure 11.3: The array backing the abstract tree shown in Figure 11.2. All heaps use arrays, so this data structure represents the entire limits of the heap.

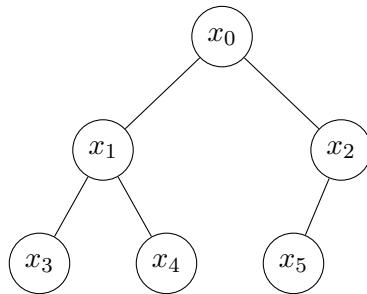


Figure 11.4: A left-justified, complete binary tree with six items will always take this shape.

heap rules force it to assume one, and only one, shape for every heap of a given size. Consequently, when the heap grows to include seven values, the heap will take the next well-defined.

The heap rules make implementing the data structure on an array not only possible, but the ideal method. With each addition, the backing array simply grows its size by one<sup>1</sup>. We lack all choice in the heap's shape, so, instead, we must swap values within the nodes themselves during run-time. The insertion operation simply involves placing the new value in the next available array space. From there, it then *bubbles-up* to its correct position through a series of swaps with its parent.

The insertion procedure:

1. Drop the new value into array in the first position "outside" the heap.
2. Compare the recently inserted value with its parent:

<sup>1</sup>A data structure with a buffer, which reduces array re-allocations, will improve performance for a queue application.

- (a) if the parent's value is lower (in a min heap), stop, for the item is in its correct position.
- (b) else swap the new value with it's parent's value.
- (c) continue these steps until parent is lower or item is now root (no parent).

### 11.3 Deletion

When deleting from a heap, we always remove the root value. As the heap represents a queue, this appears natural, for the first item in a line stands at its exit. The next customer is the one at the front of the line. Thankfully, one may quickly remove an item from a heap.

Removal Process:

1. Store the old root value in a variable to return.
2. Overwrite the contents of the root with the item at the end of the heap.
3. Bubble down from the root by:
  - (a) identify the minimum child (in a min heap)
  - (b) compare the minimum child with the root and swap if necessary
  - (c) continue these steps until the item falls into its correct position.

### 11.4 Heapification

By slightly modifying the algorithm's approach to building a heap, one may reduce the complexity associated with building a heap. The performance improvement impacts wall-clock time, for it fails to reduce the algorithm's complexity for any meaningful task. Certainly, one may build a queue quickly with this method, but one must still remove items from the heap, and that time dominates the analysis. This is analogous to speeding up into slower moving traffic.

### 11.5 Performance

Optimized for a priority queue, heaps quickly fulfill that ADT's major operations.

---

```
void heapify(int arr[], int size, int cur)
{
    int maxIdx = cur;
    const int left = getLeftIndex(cur);
    const int right = getRightIndex(cur);

    if (left < size && arr[left] > arr[maxIdx])
        maxIdx = left;

    if (right < size && arr[right] > arr[maxIdx])
        maxIdx = right;

    if (maxIdx != cur) {
        temp = arr[cur];
        arr[cur] = arr[maxIdx];
        arr[maxIdx] = temp;

        heapify(arr, size, maxIdx);
    }
}
```

---

Figure 11.5: Source code for a recursive heapification. This approach only works when one knows all the data structure's contents in advance like when sorting an array.

Operation	Best Case	Worst Case
insertion	$O(1)$	$O(\log(n))$
deletion	$O(\log(n))$	$O(\log(n))$
peek	$O(1)$	$O(1)$
search	$O(1)$	$O(n)$

Working on an array of  $n$  items:

Operation	Worst Case
heapsort	$\theta(n \log(n))$
heapify	$O(n)$

DRAFT

## 11.6 Exercises

1. After inserting  $n$  items into a maximum heap in sequential order, what is the worst case performance for finding the single largest item in the tree?
2. After inserting  $n$  items into a minimum heap in *random* order, what is the worst case performance for finding the single largest item in the tree?
3. In a minimum heap of  $n$  items, what is the worst case performance for finding the most frequently occurring item inside?
4. Given an array of  $n$  items, what is the worst case performance for putting the array in heap-order through heapification?
5. What is the worst case performance for removing the smallest item from a minimum heap of size  $n$ ?
6. The heapsort algorithm is not stable, but you can make it appear so by using a **Wrapper** on items inserted into the heap. Describe this process. What modification must one make to the heap, if any? What modification does one make to the items it contains?
7. On a separate sheet of paper, create a heap by inserting each of the items in the following list into the heap, one item at a time, as if the numbers represented **priorities** for a queue of customers arriving in real time.  
Priorities: 55, 0, 0, 92, 1, -82, -8, 0, 2, 11, 13, 37
8. A heap containing 1,048,576 items must have, at most, how many edges from its root to its farthest leaf?
9. What is the array index for the parent of an item located at index 12 in a heap? The heap's root uses index 0.

DRAFT



## Chapter 12

# Maps and Sets

### 12.1 Hash Tables

Amortized costs.

#### 12.1.1 Open Addressing

#### 12.1.2 Separate Chaining

#### 12.1.3 Collisions and Clustering

#### 12.1.4 Performance

### 12.2 Binary Search Trees

The binary search algorithm, covered in Section 9.2, locates a single item in a sorted array in  $O(\log(n))$  time. If we build a structure to mimic the algorithm's behavior, we will create a map implementation with fast key lookup times. The algorithm requires a sorted list, but what if we built a singly-linked list, and instead of simply having a `next` field, the data structure's nodes included *two* links. Rather than sort the data as they arrive by rearranging the list, simply add the new items to one of the two, internal lists. The code in Figure 12.1 shows a node in one such structure.

The binary search algorithm calls itself on either the left or right half of the array depending on the value of the currently inspected item. Lower targets must appear in the lower portion, or left, side of the array, and larger values appear to the right of the current value.

- Lower values appear in the tree connected to the *left* node.

---

```
template<typename K, class V>
struct MapNode{
    K key;
    V value;

    MapNode *left;
    MapNode *right;
} root;
```

---

Figure 12.1: A node in the binary search tree associates the *key* and *value* using a linked group of binary nodes.

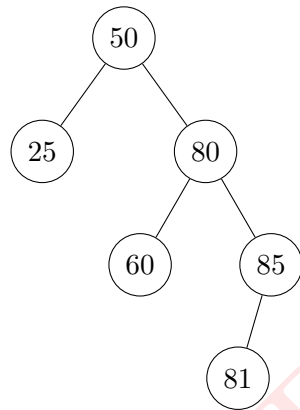


Figure 12.2: The unbalanced binary search tree resulting from entering in the keys in the order: 50, 80, 60, 25, 85, and 81. A different sequence may produce a different structure.

- Greater values appear in the tree connected to the *right* node.
- Links with `nullptr` values indicate empty nodes.

### 12.2.1 Insertion

Recall that maps must contain unique keys. The values may appear multiple times, but the search keys themselves must appear unique. Adding a new key and value pair to the data structure requires one to instantiate a new node if the key is missing. All additions, or insertions, in this style of data structure occur at a *null* child. For new keys, the tree neither changes nor

overwrites nodes in the tree's interior.<sup>1</sup>

The steps for insertion in a basic binary search tree:

1. Traverse the tree, starting at the root node.
2. Jump either left or right based upon if the new key is greater or smaller than the current node's key.
3. Continue this process until the jump leads to a `nullptr` child.
4. Instantiate a new node in the map with the desired key and value provided.

### 12.2.2 Removal

Removing from a singly-linked list required one to stop at the link *before* the one to remove. In a binary tree, we may view this as deleting from the *parent*. Deletion always happens at the parent's level, but the strategy changes based upon the number of children to which the node to delete connects.

Typically, the remove or delete method returns the *value* previously stored inside. This tends to simplify the caller's software in many situations. If the value proves unnecessary to the implementation, developers may simply ignore it in code.

**Zero Children** In the easiest case, where the desired node to remove from the tree is a *leaf* value, one need only set the parent's corresponding `left` or `right` links to `nullptr`<sup>2</sup>.

**One Child** For parent nodes with an only-child, removal closely mimics removal from the middle of a singly-linked list. After locating the key one wishes to delete and determining it has only one child, the algorithm simply needs to redirect the parent's `left` or `right` links (as applicable) such that they now point to the node-to-delete's only child. The parent will point to its former grandchild.

---

<sup>1</sup>This differs slightly from the `put` method which *either* inserts a new node in the tree, if the key is absent, or sets the value associated with the existing key in the structure to the new value specified in the latest `put` call.

<sup>2</sup>Obviously, one must take special consideration with the `root` node. That is, when deleting the only key in a tree.

---

```
template<typename K, class V>
class V delete( struct MapNode* start, K key )
{
    if (key < start->key && start->left != nullptr)
        return delete(start->left, key);

    if (key > start->key && start->right != nullptr)
        return delete(start->right, key);

    if( key == start->key){
        V val = start->value;

        if (start->left == NULL)
        {
            struct node *temp = start->right;
            free(start);
            return val;
        }
        else if (start->right == NULL)
        {
            struct node *temp = start->left;
            free(start);
            return val;
        }

        K inOrderKey = inOrderPredecessor(start->left);

        start->key = inOrderKey;
        start->value = delete( start->left, key);

        return val;
    }
    return nullptr;
}
```

---

Figure 12.3: Rough code for a removal method in a binary search tree. This fragment misses several error conditions and significant use cases.

---

```
template<typename K>
K inOrderPredecessor(MapNode *root){
    if( root == nullptr || root->left == nullptr )
        return nullptr;
    return maxKey( root->left );
}

template<typename K>
K inOrderSuccessor(MapNode *root){
    if( root == nullptr || root->right == nullptr )
        return nullptr;
    return minKey( root->right );
}
```

---

Figure 12.4: Locating the in-order predecessor is simpler with helper functions.

**Two Children** When deleting nodes with two children from the binary search tree, one must replace the root key and value with something meaningful. That is, it must preserve the binary search tree ordering so that subsequent key searches correctly locate the values. Using the in-order predecessor or in-order successor will supply a value that keeps the other connections between nodes valid.

**In-Order Predecessor** Deleting any node with two children from the tree requires an algorithm to efficiently locate a replacement. One may do so using recursive code in a few simple lines.

### 12.2.3 Performance

Maps and dictionaries focus on quickly locating a record within the structure, so the binary search tree favors these operations. Every function associated with finding a *key* must operate quickly. These functions include: **put**, **get**, and **contains**. Assuming a relatively complete tree, these lookup times occur in  $O(\log(n))$  time, for the tree eliminates half the potential values with every step from parent to child.

Unfortunately, this reflects the *best* performance, for a different entry order may produce a different tree. For example, inserting values in a tree that are already in sorted order will produce an unbalanced binary search tree. These trees more closely resemble linked lists, and, consequently, their

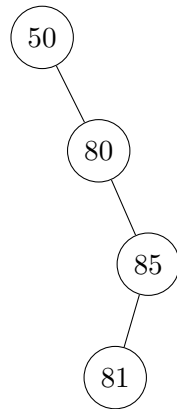


Figure 12.5: This tree more closely resembles a linked list, and it performs as poorly as well. Searching for the single key 82 requires linear time.

worst case performance mirrors the worst-case performance of a linked list. A severely unbalanced list may require  $O(n)$  when locating a *single* value in the tree. Figure 12.5 illustrates one such tree.

### 12.3 Balanced Trees

The unbalanced binary search tree introduced in Section 12.2 demonstrates the basic binary search ordering, but its performance depends upon the insertion order. Randomly distributed numbers should create a fairly complete tree, but an in-order sequence builds something more like a linked-list. Consequently, we now consider a few advanced map structures that produce balanced trees.

In the unbalanced, basic binary search tree, its height grows with  $n$  in an in-order sequence. Every addition to this tree causes the structure to drop down another level. Observe that the height of this unbalanced tree grows at  $O(n)$ . With trees built this way, we cannot access or find the items inside any faster than we could using a linked-list. This breaks the basic, fundamental role of the map: finding keys quickly. Balanced trees perform maintenance with each addition and removal to help guarantee fast lookup times. A balanced tree grows at  $O(\log(n))$ .

Figure 12.6 illustrates a few of the possible configurations one may achieve with using only three nodes. For trees this small, the performance difference is negligible, but as we add more than three items to the map our lookup times become quadratic.

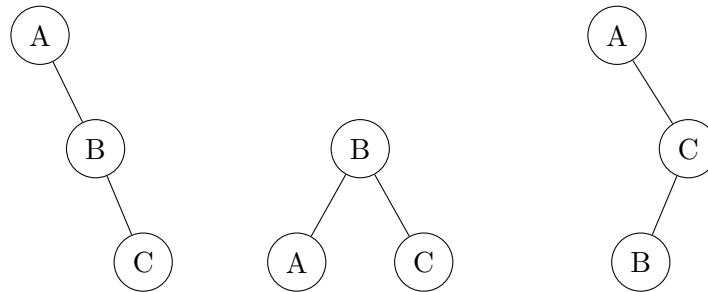


Figure 12.6: Several configurations possible using the same input data but in a different order. Only the middle configuration, however, offers balanced performance.

### 12.3.1 Rotating Nodes

Many tree structures use rotation as a method for correcting an unbalanced tree. In this process, the tree moves nodes around to achieve a balanced structure. Both the AVL tree and the Red-Black tree use rotation to reduce a tree's height. The process of rotating, however, remains consistent between the two trees. They each use different rules, however, when establishing when to rotate the nodes.

The rotation process for a series of three nodes inserted in sequential order requires a rearrangement of parents and children. For these nodes, we need to either rotate them right or rotate them left to move the middle value to the root position. Imagine a roller-coaster approaching the apex of its climb and starting to roll over the top. The unbalanced three nodes look like a line of cars going up the hill. As the first car rotates over the top, the middle car moves up. We want to mimic this behavior using three nodes and source code.

Rotating a series of three nodes to the left requires the previous grandparent node (i.e. the root in the sub-tree) to become its child's *left* child. Effectively, the grandparent's right child becomes the new root node, and the grandparent moves down into its child's position. This requires the rotation code to simply move the node links around. The data values and keys in each node, however, remain unchanged – it is only their *left* or *right* links that accept new values.

The Java source in Figure 12.7 illustrates this on a right rotation. Here,

---

```

public Node<E> rotateRight(Node<E> gp){
    Node<E> temp = gp.left;
    gp.left = temp.right;
    temp.right = gp;
    return temp;
}

```

---

Figure 12.7: Rotating `Node` objects in the Java programming language.

we want the previous grandparent to become its child's *right* child. To do so, however, we must update and track several links.

### 12.3.2 AVL Trees

The first popular balanced-tree, the AVL<sup>3</sup> tree, manages its height by keeping track of the heights of each of its sub-trees. If one child's size grows too high, we trigger a tree rotation. For AVL trees, the rule remains simple:

For each node in the AVL tree, the difference in height between the left and right sub-trees shall differ by no more than one.

Mistakenly, on the internet, many casual sources suggest this is *the* condition required for a balanced tree. This assertion remains false. Although a tree exhibiting these properties will perform in a balanced fashion, it is not an absolute requirement. Moreover, the AVL condition *in no way* guarantees a *complete* tree.

The total number of nodes in a complete tree structure, like the heap discussed in Section 11, must contain a specific number of nodes.

$$T(n) = 2^{\text{height}+1} - 1 \quad (12.1)$$

The minimum number of nodes in an AVL tree of height  $h$ , however, is:

$$S(h) = \begin{cases} 1, & \text{if } h = 0. \\ 2, & \text{if } h = 1. \\ S(h-1) + S(h-2) + 1, & \text{otherwise} \end{cases} \quad (12.2)$$

---

<sup>3</sup>It is named after the men who first discovered it, so its name, although a testament to their majesty and greatness, provides *zero* insight into how it performs. Vanity over clarity.



Height	Complete	AVL
0	1	1
1	3	2
2	7	4
3	15	7
4	31	12
5	63	20
6	127	33
7	255	54
8	511	88

Figure 12.8: The minimum number of nodes a complete tree vs an AVL tree. Observe that the AVL condition fails to produce a truly complete tree, but it performs well enough for a map.

Using these two equations, we can establish the minimum number of nodes for each tree of a given height and contrast the differences between an AVL tree and a truly complete tree. As shown in Figure 12.8, an AVL tree may reach a height of *eight* with as few as eighty-eight nodes. A complete tree will contain an equal amount of data with only a height of *six*.

**Insertion** Adding key-value pairs to the AVL tree follows the standard insertion procedure for a binary search tree. One begins at the root node and then traverses either left or right based upon the value of the key in the root. After we add the new node to the tree, however, we must evaluate the impact of the new node on the tree's balance. All new nodes enter the tree with a height of zero, for they are leaves with neither a left nor a right child. Their *parent* nodes, however, must verify the addition did not violate the AVL condition and rotate if necessary. An addition may require the parent node to update its own height, and this could create a significant impact on the tree above it.

Let us walk through the procedure by inserting each of the following keys into the tree in sequential order from left to right<sup>4</sup>.

{ 18, 78, 4, 11, 22, 42 }

After inserting the first five keys, the AVL tree and unbalanced binary search tree look identical.

<sup>4</sup>Exactly like you will do on the exam

---

```

template <class K, class V>
struct AvlNode {
    K key;
    V value;

    unsigned char myHeight;

    AvlNode *left;
    AvlNode *right;

    AvlNode(K k, V v) : key(k), value(v), myHeight(0),
                      left(nullptr), right(nullptr) {}

    ~AvlNode() {
        delete left; left = nullptr;
        delete right; right = nullptr;
    }
};

```

---

Figure 12.9: An example of the source required for a node in an AVL tree. It tracks the left and right children just as a standard binary search tree, but it now includes the additional space required to track its height.

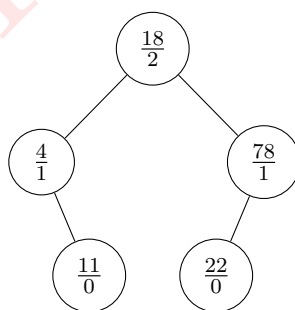


Figure 12.10: An AVL tree with the  $\frac{\text{keys}}{\text{heights}}$  of each node. After inserting five keys in the AVL tree, it still resembles its basic, unbalanced counterpart.

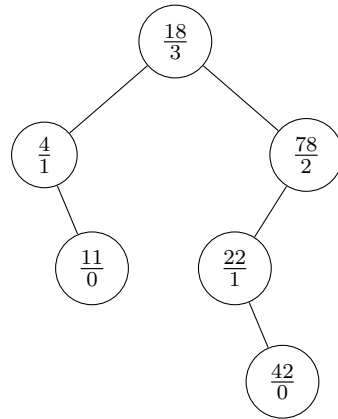


Figure 12.11: The addition of the 42 key introduces an AVL tree violation. The 78 node has a left tree height of 2 and nothing on the right.

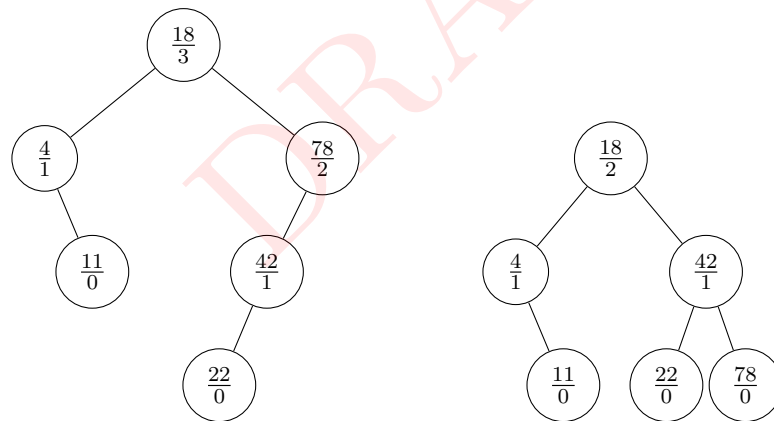


Figure 12.12: The fix-up for the violation introduced with the 42 key involves first rotating the 22 and 42 nodes left and then rotating to the right through 78.

With AVL trees, a single insertion may unbalance multiple nodes higher up in the tree. Adding one to one side of the tree may cause several internal nodes in the sub-tree to detect an AVL violation. After the insertion, the tree moves back along the insertion path all the way to the root. As it moves back through the tree, it verifies that each node satisfies the AVL condition. The first node it encounters on its trip from the new leaf back to the root that has sub-trees that violate the condition must rotate.

By correcting the error at its lowest level in the tree, this automatically reduces the height of any violating nodes *above* it in the tree. The only reason the nodes above it have unbalanced subtrees is because the latest insertion violated the AVL condition. A single insertion can only increase the height of a sub-tree by no more than *one*, so all nodes above it in the tree violating the AVL condition may only be off by one. Thus, reducing the height at the lowest level also corrects those nodes.

In an AVL tree, one only ever needs to perform a single rotation fix-up step. This fact does *not* hold true for the red-black trees discussed in Section 12.3.3, for they may require multiple adjustments to correct the structure. Things in the AVL tree appear slightly simpler in that regard.

### 12.3.3 Red Black Trees

Rather than tracking the height at each node, which requires substantial additional storage, the red-black tree achieves balanced performance using only a single additional bit<sup>5</sup> For this tree to work, however, it requires additional rules which define the tree's properties and what to do when an insertion violates one of its rules.

- The root node for the tree is always *black*.
- New nodes enter the tree *red*.
- Treat all null (missing) children as *black*.
- No two red nodes may touch one another in the tree.
- Check: the path from every leaf to the tree's root passes through the same number of *black* nodes.

---

<sup>5</sup>We cannot allocate a single bit in Java or C++. Instead, an 8-bit or boolean value stores the color.

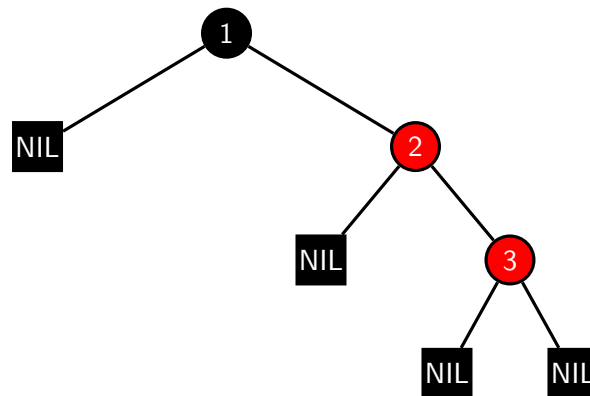


Figure 12.13: The addition of the key 3 causes a red-violation in this red-black tree. This configuration triggers a rotation due to the NIL attached to node 1.

**Insertion** As in the unbalanced tree, all additions to the structure occur at null children. That is, we do not overwrite existing data, nor do we place the node inside the middle of the structure and manage those links. Insertion always increases the map's size, so it makes sense that new values go where nothing existed before.

When inserting the first node in an empty red-black tree, the node enters the tree as a red leaf, for the tree rules demand all new nodes arrive *red*, but the tree then immediately changes its color to *black*. Let us run through the process with an in-order series of numbers:

{ 1, 2, 3, 4, 5, 6 }

As Figure 12.13 shows, our first real red-violation happens when we attempt to insert the key 3 in the tree. Here, it conflicts with the matching red color of its parent node. Because the root has no left child, the tree treats it as a black node. In this situation, the red-black tree demands we perform a rotation. Figure 12.14 shows the reduced height after this happens.

Moving along, when we insert the next number in the tree, we introduce another red-violation. In Figure 12.15, both the 4 and 3 are red. Looking at 3's parent node, we observe that 2's other child, the 1 key, is currently red. In this situation, we perform a *color-flip* on nodes 1, 2, and 3. Note that the node 4 enters the tree as red here, and it remains that color after the fix-up. The nodes involved in the change are its parent, grandparent, and aunt. Figure 12.16 shows what the tree looks like after this step concludes.

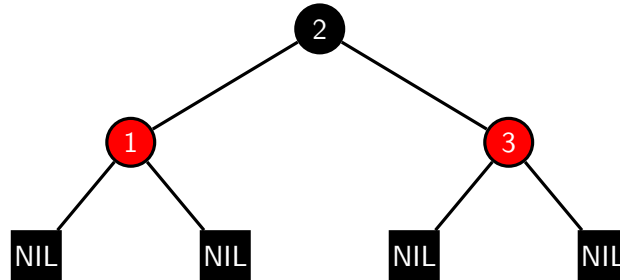


Figure 12.14: The reduced height in the red-black tree after the rotation triggered by 3's addition to the map.

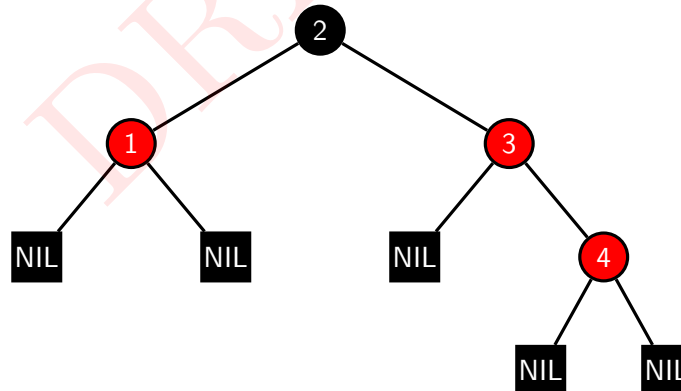


Figure 12.15: Yet another red-violation when we insert the next number in the in-order sequence.

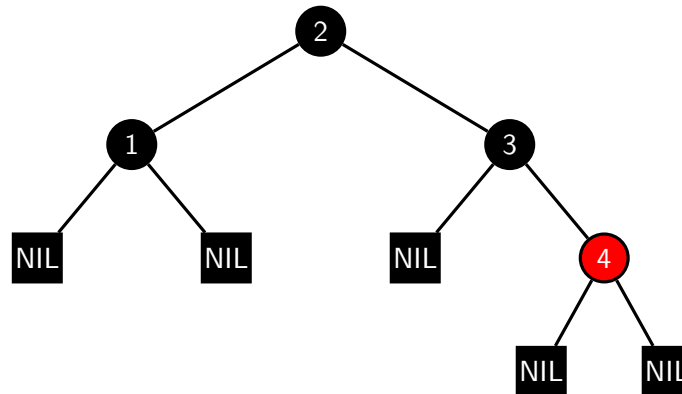


Figure 12.16: Changing the color of the new node's parent, grandparent, and aunt resolves this problem. Because this color flip includes the root, it actually causes two fix-up steps: color flip, root change to black.

As Figure 12.17 shows, yet another red-violation appears with the next number in the in-order sequence. In this case, because node 3's other child is `null`, we treat it as black per the red-black tree rules. As previously established, we rotate the nodes when we detect this condition. Figure 12.18 shows what the final tree looks like after this concludes with the key 5 successfully inserted in the map.

## 12.4 B-Trees

The binary search tree structure expands to more than one link. B-Trees generalize the idea. The binary tree acts as a simple 2-tree, for each node contains two links. What if we could expand this idea to include three links? In this way, we might slightly optimize our tree's structure and search times. We could use a lower link and greater link, just like in the binary search tree, but what if there were also some concept of a *middle* value?

The 2-3 tree, one form of B-tree, captures this idea. In order for there to be a middle tree to traverse, then the nodes inside the tree must contain two keys: a lower and upper key. Items falling between the two keys will enter the middle sub-tree. Additionally, items lower than the smaller key will fall in the left sub-tree while items greater than the larger key will fall into the right sub-tree.

Figure 12.25 illustrates this idea with a 2-3 tree. Unlike the binary tree, or a simple 3-tree, the 2-3 tree uses both traditional binary search

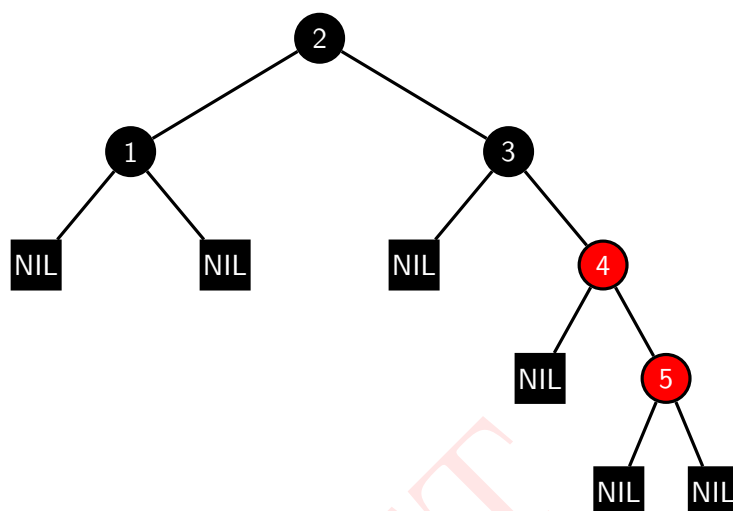


Figure 12.17: Adding key 5 to the map triggers a rotation to the left for nodes 3, 4, and 5.

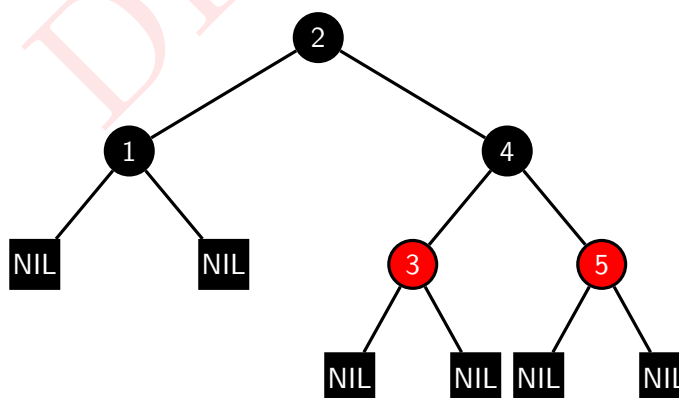


Figure 12.18: After the rotation triggered by 5's addition to the map.



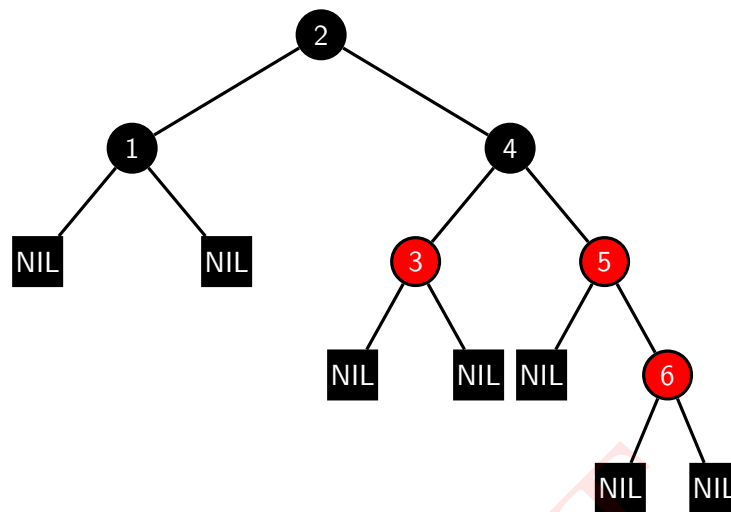


Figure 12.19: With node 6, another red-violation appears.

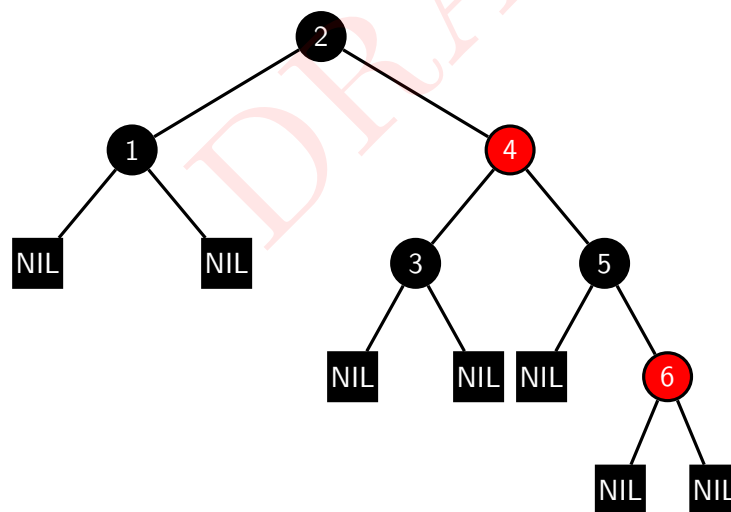


Figure 12.20: A color flip clears the problem. Observe there are the same number of black nodes between every leaf and the root.

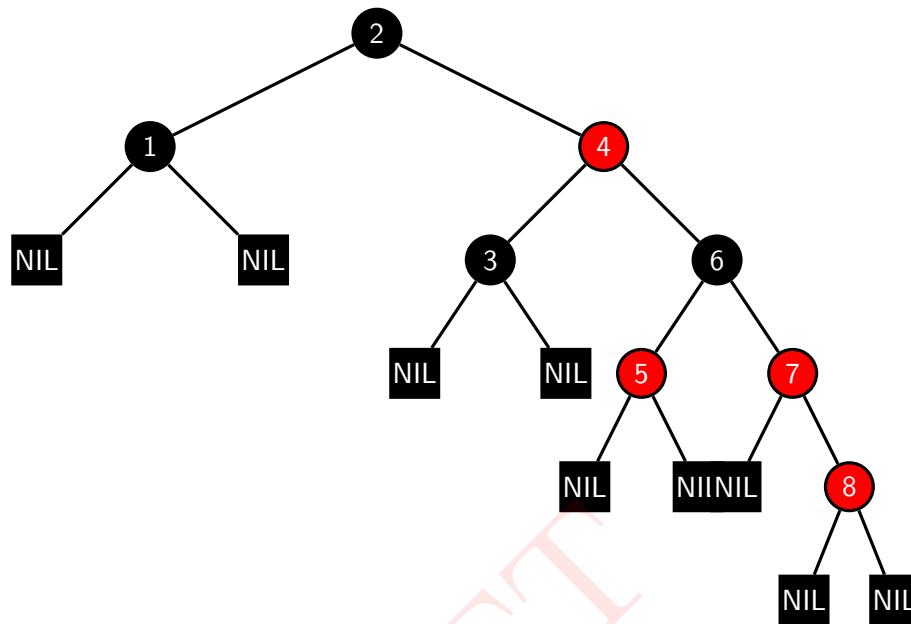


Figure 12.21: A simple color resolves the initial red-violation here, but it introduces side-effects above it in the tree.

tree nodes as well as 3-nodes. When items enter the map via an insertion method, the tree decides if it should enter the map as a new binary node or, alternatively, swell an existing binary node into a 3-node. The 2-3 tree's insertion procedure chooses between adding a new binary node or expanding an existing binary leaf into a 3-node.

As a result, the tree self-balances and maintains a truly complete structure at every level. This provides obvious<sup>6</sup> performance benefits as the tree's shape helps guarantee logarithmic lookups. Another, subtler benefit also arrives with a 3-node: we fetch multiple keys with each node access. The nodes in a binary search tree, unlike the values in an array, appear throughout memory without a predictable pattern. Each node appears as an island somewhere in the heap, and it uses its links to point to the other node islands. Thus, accessing the keys to perform the search requires a de-referenced memory access.

This procedure takes longer than simply looking up each item in an array. Fetching nodes from memory is a slow operation. The 3-node helps reduce the number of data fetches one needs from memory, for a 3-node

<sup>6</sup>Super.

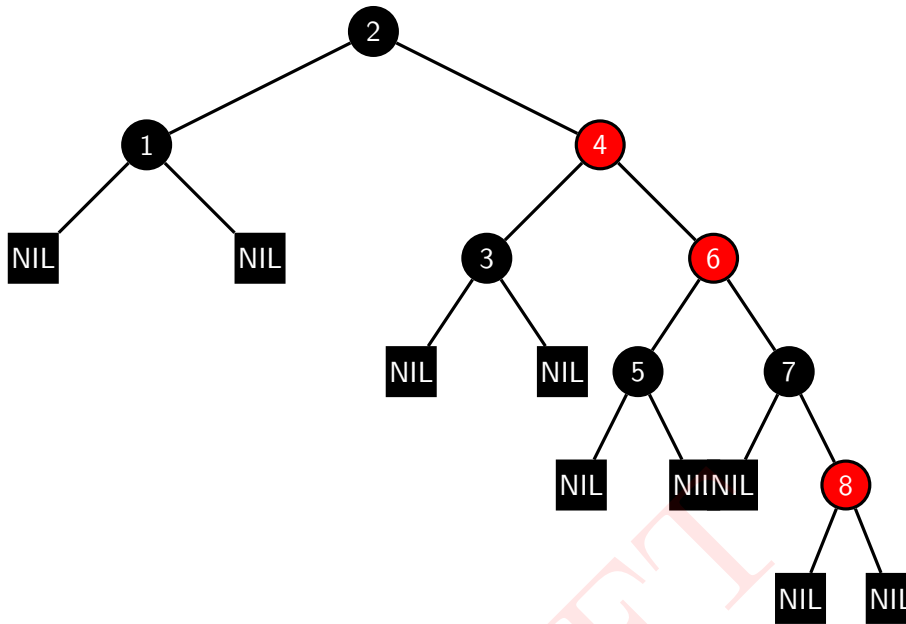


Figure 12.22: Now nodes 4 and 6 share red as their color, but they are adjacent, so this creates a violation.

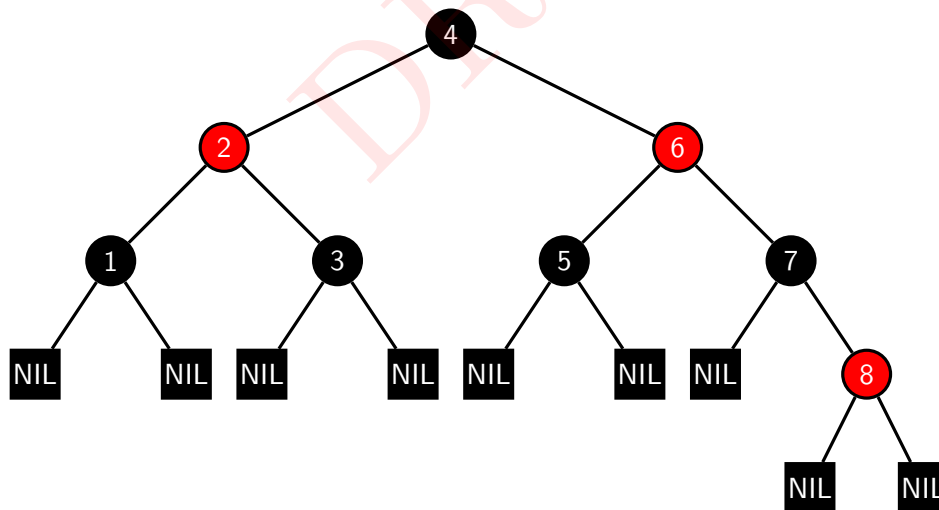


Figure 12.23: Rotating nodes 2, 4, and 6 solves the problem.

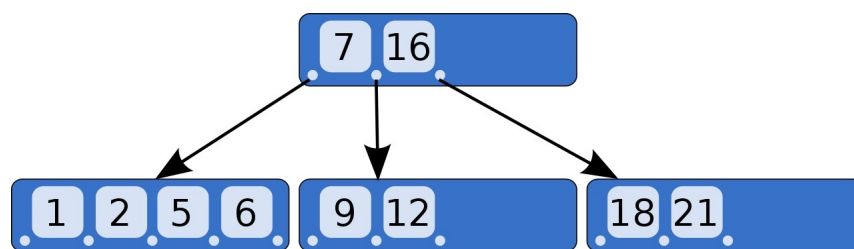


Figure 12.24: A node in a 5-tree tracks four keys and includes links to five other 5-nodes. This illustration courtesy the WikiMedia Commons

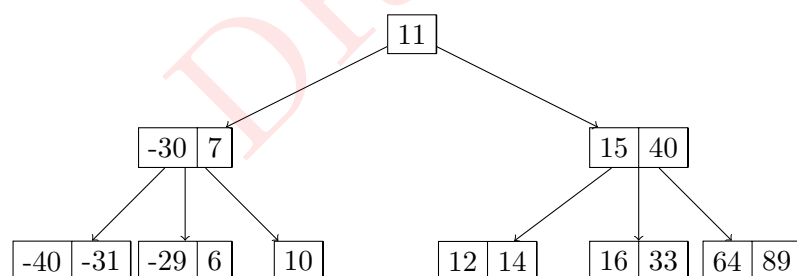


Figure 12.25: The 2-3 tree changes the number of keys and links each node in the tree uses so it is always complete. The name references the number of outbound links each node will possess.

includes two keys. What if our storage media were terribly slow, like a tape-drive or external memory array? Using the 3-node would slightly speed up our overall performance if we planned on doing substantial processing or searching in the tree.

What if, however, rather than limiting ourselves to simply two keys, what if we expanded it to 128, or 256, or 2048? Searching through the keys will become a chore itself, but fetching the 2048-node from memory supplies 2047 keys in a single read. When reading from a slow drive, this style of tree may offer empirical performance improvements.

DRAFT

## 12.5 Exercises

1. The code-fragment for the `inOrderPredecessor` in Figure 12.4 calls the functions `minKey` and `maxKey`. Supply the code for each of these functions.
2. A hash table using probing currently has a total of 128 bins inside to hold items. Due to the nature of this specific data set, all items produce a `hashCode` with the common factor of 8. If the hash table simply performs a modulus of the hash code by 128 to determine the proper item placement, what is the maximum number of items the table can hold without rehashing?
3. Identify the *worst* case performance for inserting a single item into a hash table using *open-addressing* given the table currently holds  $n$  items.
4. What is the worst case *amortized* cost of inserting  $n$  items into an initially empty hash table?
5. What is the average complexity one should expect from a Hash Table using probing when inserting a single item into a table with  $n$  items?
6. An incorrect chained hash table implementation uses only a single bucket, for this avoids the need to rehash. In this Map implementation, what is the complexity for searching for  $n$  keys?
7. Draw the final AVL tree resulting from inserting the following number sequence in order from left to right. Show each major insertion (e.g., one where the tree rotates) for partial credit: 88, 45, 3, 12, 11, 43, 22, 10, 0
8. Draw the final Red-Black tree, noting each node's color, resulting from inserting the following number sequence in order from left to right. Show each major insertion for partial credit: 79, 5, 19, 20, 39, 40, 41, 6, 30
9. Create the code to build and demonstrate an *unbalanced* binary search tree from scratch. This class will include the private, internal `Node` class the data structure as well as any helper classes necessary. The binary search tree created for this problem must adhere to the binary search tree convention of placing small items in the left-subtree and larger items in the right-subtree. The solution shall include a means

to put keys into the map as well as to retrieve the corresponding value.

Program Name:	bst
Program Input:	cin
Program Output:	cout
Return	0 on completion
Allowed Imports:	iostream string vector list

10. Modify the tree created in Problem 9 such that it supports producing an in-order *keyset* iterator. The driver shall demonstrate this functionality.
11. Modify the tree created in Problem 9 such that it supports producing a *value* iterator and demonstrate its performance. The iterator shall produce values corresponding to the **keyset** order. That is, the second value iterated shall correspond to the second key iterated.
12. Include and demonstrate the deletion method in your solution to Problem 9. The solution may use the in-order predecessor or successor.
13. Write a function, **isAnagram** which accepts two **std::string** parameters and returns **true** if the strings are anagrams of one another.
14. Create a *program* that randomly combines words contained in its input stream and sends them to the output stream. At program execution, the application shall read in the entire set of words to use during the combination process. After reaching the EOF symbol, the program shall then begin randomly selecting words and combining them into a concatenated output word. The program produces only one combined output word for each execution.

By default, the program randomly squeezes four words in its dictionary together and omits any spacing between them (e.g., somewhatlikethis-butrandom). When unable to construct strings meeting the minimum length from the input stream, the program shall alert the user to an error and suggest corrective action. There shall be no duplicate words in any single output combination.

Program Name:	crunch
Program Input:	cin
Program Output:	cout
Return	0 on completion
Allowed Imports:	ctime cstdlib iostream map string

Example Use:

```
user@computer: $ ./crunch <dict.txt
reducefishersplendidprogram
user@computer: $ ./crunch -s <dict.txt
windows omlette lethargic amount
user@computer: $ ./crunch -n 2 -s<dict.txt
hospitable chicken repair overflow
scrawny belief remain actually
```

15. Modify the program described in Problem 14 to accept a command line argument which specifies the number of strings the program outputs in a given execution. The program shall crunch together four words *n* times for the current execution.
16. Students shall create a program which reads ASCII strings from the standard input until it reaches the EOF symbol. From the accumulated input, the program shall extract only its unique tokens, separated by the space character, and send these to the standard output. For example, if provided with a file containing a text document, it shall send to the standard output every *unique* word contained therein. The order the words appear in the output does not need to be sorted in any particular order.

The program shall insert a space character between each output word and ignore line endings.

Program Name:	unique
Command Line Args:	none
Program Input:	cin
Program Output:	cout
Return	0 on completion
Allowed Imports:	iostream array map string



Example Use:

```
user@computer: $ ./unique<iamsam.txt
```

DRAFT

DRAFT

**Part V**

**Algorithms**

DRAFT



12.5.1 Performance

12.6 Graph Algorithms

The Graph ADT establishes a method for storing the relationship between vertices  $V$  and their edges  $E$ . It builds upon the idea of a linked list or binary search tree, but rather than limit the children or next items in the sequence to one or two, fixed pointers, the Graph ADT allows for a dynamic structure. No longer limited to a single sequence, where the next item in the chain is the next list item, vertices in a graph may circle back on themselves, connect back to multiple other nodes in the graph, or even remain unreachable islands.

Neural networks serve as one form of a graph where each node in a layer represents a vertex, and the connections between layers serve as edges. On-line game worlds may connect different regions of the map through a graph. The MMORPG EverQuest clearly connected regions of the map in different 'zones.' To move from one region to another, players ran to a specified area of the current map (vertex) until they encountered a loading screen. The loading screen served as an edge, and after the process completed, the player appeared in the adjacent zone at the edge's endpoint. One might further interpret the loading time players experienced during this transition as a 'weight' or 'cost' for traversing the edge.

12.6.1 Depth-First Search

12.6.2 Breadth-First Search

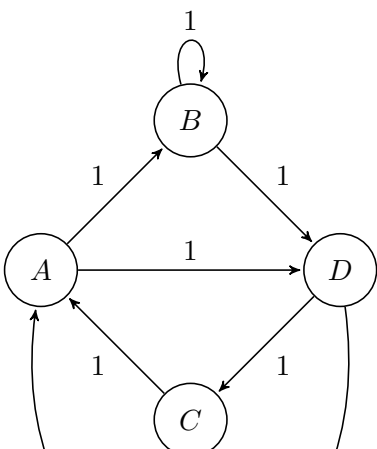
12.6.3 Dijkstra's Shortest Path

12.6.4 A\* Search

12.6.5 Performance

12.7 Exercises

1. Given the following graph, draw the adjacency matrix in the space provided. For any directed edges, include the appropriate arrow indicating the direction of travel.



151

	Destination				
	A	B	C	D	E
A					
B					
C					
D					
E					

2. Given a comma separated value file (CSV) as a command-line argument, construct a graph from the specified vertex information and display information about its connectivity. For each vertex in the graph, display a list of directly adjacent vertex names.

**Input File** The CSV file provided as the program's input argument adheres to the following formatting specification.

- Only one entry (i.e. Vertex or Edge) appears on each line in the file
- Entry may represent either a vertex name or an edge connection
- Vertex names appear as single items on the line
- Edge connections include three parts: Source, Destination
- Edge and Vertex names appear as strings
- Edges and Vertex Names may appear interlaced
- A new vertex may appear as a vertex entry or when included in an edge connection.

Program Name:	buildGraph
Command Line Args:	path to CSV to use
Program Input:	cin
Program Output:	cout
Return	0 on completion
Allowed Imports:	iostream array map string vector

3. Modify your solution to Problem 2 such that the edge information in the file may also include a weight parameter for the connection. For, one would specify an edge between vertex A and vertex B with a cost of 10 units as: A,B,10. The graph shall display the connection cost for each adjacent vertex when it performs the printing function.
4. Incorporate Dijkstra's shortest path algorithm in your solution to 2. Prompt the user to select from any two vertex labels in the graph and compute a shortest path between. The solution shall display each vertex name, in its proper order, in the path.

# List of Figures

1.1	Some of the major differences between C++ and Java . . . .	5
1.2	<i>Real Programmers</i> , courtesy of XKCD and the Creative Commons. . . . .	6
1.3	Establishing an encrypted connection to the university server.	8
1.4	<i>Password Strength</i> , courtesy of XKCD and the Creative Commons. . . . .	10
1.5	An example of a recursive directory copy of a local folder to the university server through SCP. . . . .	12
1.6	A rough overview of the C++ compilation process. . . . .	13
1.7	An example of the pre-processor expanded output from a minimal C program. . . . .	14
1.8	A basic C macro for finding the maximal value. The pre-processor replaces the macro variable <code>a</code> with <code>x++</code> , so what values do <code>x</code> and <code>y</code> have after executing this expression? . . . .	15
1.9	An example of <i>variable capture</i> . What values for <code>a</code> and <code>temp</code> does the program print? . . . . .	15
1.10	Examples of pre-processor directives included with C++. . .	16
1.11	An example of a header guard which only includes the contents of a header file the first time it encounters it in code. This style works with both C and C++. . . . .	17
1.12	An example of a header guard using the pre-processor's <code>#pragma once</code> directive. Not all C++ compilers support this feature. .	17
1.13	A basic C++ application with a program entry point. . . .	19
2.1	Manifest typing in the C++ language. The source code must specify the distinct types for each memory location. . . . .	22

2.2	With dynamic typing in Python, a variable may change its fundamental type within the current scope. In this example, <code>a</code> assumes the form of an integer, string, and function within the same scope. . . . .	22
2.3	Never in the way of possibilities, C++ permits developers to freely use uninitialized values. . . . .	23
2.4	The SR flip-flop, a sequential logic circuit, stores a single, boolean value of 1 or 0 in a basic unit of memory. . . . .	24
2.5	Various declarations of boolean types in C++. . . . .	25
2.6	Three conditional uses of boolean types, but the <code>while</code> loop may not behave as expected. . . . .	25
2.7	The different character types available in C++. . . . .	26
2.8	The different integer types available in C++. . . . .	27
2.9	Some of the various integer-literal representations in a basic C++ program. . . . .	28
2.10	The Pentium FDIV Error incorrectly computes some floating point calculations. . . . .	29
2.11	Examples of placeholder types where the compiler deduces the data type. . . . .	30
2.12	Explicitly type-casting between different variable types. . . .	31
2.13	A single line type declaration introducing two variables. The variable names suggest the developer wants two pointers, but the syntax produces two different types. . . . .	31
2.14	Multiple of declaring and initializing arrays in C++ illustrate the shared syntax. . . . .	31
2.15	Accessing the elements of an array. . . . .	32
2.16	Row and Column major order as illustrated by the Wikimedia Commons. . . . .	33
2.17	Multi-dimensional arrays in C++. . . . .	33
2.18	Pointers hold the value of an address where the actual data rest. One accesses the data indirectly by first looking at the address held in the pointer and then visiting that address. Image from <i>Dinosaur Comics</i> ©2005 Ryan North. . . . .	34
2.19	Use of <code>nullptr</code> and the older C-style evaluation to test for pointer initialization. . . . .	35
2.20	The C language introduces new, special notation for pointing into structures. This same style applies to how C++ addresses pointers into structures and classes. . . . .	36
2.21	Passing by reference in C++ . . . . .	37
2.22	Function and its stack frame. . . . .	38



2.23	String literals. . . . .	39
2.24	Calculating string length when the string includes the null termination symbol. The main method fails to return a value. . . . .	40
2.25	Compound statements use the curly-brace notation. . . . .	41
2.26	Blocks of code include variable declarations with their associated stack implications. . . . .	41
2.27	Assignment statements produce a result in C++, and this performs meaningful work when provided a null-terminated string. . . . .	43
2.28	Depending on the order of evaluation for expressions produces unreliable results, for the results may vary based on the compiler used and not the algorithm. . . . .	43
3.1	Keywords modifying a function declaration. . . . .	46
3.2	The general form of a function definition. It provides the source code for the linker to use when creating the executable file. . . . .	46
3.3	Changing the number or type of a function's input argument requires a new declaration. These three function declarations yield unique signatures and require their own definitions. . . .	47
3.4	Array names, without an index, act as an address to another location in memory. We can change our copy of the address without side effects, but touching the array contents leaves lasting effects. . . . .	48
3.5	A version of swap which passes by value. The swap successfully consumes processor resources, but it fails to perform any meaningful work. . . . .	49
3.6	A version of swap using raw pointers. Instead of accepting the actual values to modify, this version takes their address and de-references it to change the variables' values in memory. . . . .	49
3.7	A version of swap using pass by reference. Here the function works with aliases to the integer variables. . . . .	50
3.8	Modifying the example in Figure 3.3 to accept default arguments. . . . .	50
3.9	An example of a recursive algorithm to compute the greatest common divisor of two integers. . . . .	51
3.10	An example of a recursive algorithm. This version of binary search returns the index of the matching element, but at what cost? . . . . .	52
3.11	An example of first-class and higher-order functions. . . . .	52

3.12	This computes the total of all elements in the list. The variable <code>total</code> is stored as a part of the lambda function's closure. Since it is a reference to the stack variable <code>total</code> , it can change its value. . . . .	53
3.13	A simple lambda function illustrating the general syntax. . .	53
3.14	Lambda capture specifiers if the object should be copied by value or reference. . . . .	53
3.15	This lambda indicates captures <code>n</code> by reference with the <code>&amp;</code> symbol. What happens when <code>badlamb</code> returns and <code>n</code> falls out of scope? . . . . .	54
4.1	The standard I/O streams in C++. Developers may override the default destinations. . . . .	57
4.2	Writing to the screen. This code uses two different methods for inserting newline characters at the end of the text. . . .	58
4.3	Prompting and reading an integer from the user in C++. What happens when a giant crustacean from the paleolithic era inputs 3.50? . . . . .	58
4.4	Although the <code>cin</code> operation accepts the user's full name, when it extracts the value from the input, it terminates when it reaches the first space. Consequently, only the first name appears in the string variable <code>fullName</code> . . . . .	58
4.5	Although the <code>cin</code> operation accepts the user's full name, when it extracts the value from the input, it terminates when it reaches the first space. Consequently, only the first name appears in the string variable <code>fullName</code> . . . . .	59
4.6	Format manipulators, in the <code>&lt;ios&gt;</code> header file, allow one to change the default behavior of input and output streams. . .	59
4.7	Producing a generic CSV file in the current directory. The interface closely matches producing input and output to the screen. . . . .	60
5.1	A basic class definition holding two member variables with the default visibility. . . . .	61
5.2	A class definition for a basic three-dimensional mathematical vector. It inherits from the general <code>Vector</code> type. . . . .	62
5.3	The basic format for a copy constructor in C++. . . . .	63
5.4	Example of a class with a destructor and constructor in C++. The allocation of Heap memory requires a <code>free</code> to avoid memory leaks. . . . .	64

5.5	Access specifiers in Java. . . . .	65
5.6	Access specifiers in C++. . . . .	65
5.7	Friendly example from Microsoft. . . . .	66
5.8	Basic structure for inheritance in C++. . . . .	68
5.9	An example of Inheritance in C++ using ducks. . . . .	68
5.10	Using the earlier duck example defining class methods outside their definition. . . . .	69
5.11	A code fragment for a simple <code>Vec3</code> mathematical vector to use for exercises. . . . .	70
6.1	This source contains errors and will not compile due to variable scope issues. The variable <code>block</code> is not visible in the outer compound statement. . . . .	73
6.2	Although free of syntax errors, this code includes a pointer to a variable out of scope. This represents a significant logic error. . . . .	74
6.3	Variable visibility in a simple iterative for loop. The loop control variable falls out of scope when the loop ends, so the compiler does not understand the binding. . . . .	75
6.4	The compiler attempts to resolve the binding as close to the use as possible, so the inner declaration shadows the outer name. This really feels like an exam question, doesn't it? . . .	75
6.5	An example of namespace in C++ . . . . .	76
6.6	Working with the fully-qualified name. . . . .	76
6.7	Bringing one identifier into <i>scope</i> with the <code>using</code> directive. . .	76
6.8	Grabbing everything in the namespace and bringing it into scope. . . . .	76
6.9	Pulling in subroutines defined in other modules in Java through the <code>import</code> statement. . . . .	77
7.1	An example of a C++ template for the Command design pattern in <i>Design Patterns</i> . . . . .	80
8.1	Measuring elapsed time on a particular machine. . . . .	85
9.1	An visualization of binary search's complexity. This image courtesy the Wikimedia Commons. . . . .	90
9.2	The source code for a recursive version of the binary search algorithm. . . . .	91
9.3	Selection sort's source code. . . . .	92

9.4	Insertion sort's in-place source code requires a constant number of scratch variables. . . . .	92
9.5	A breakdown of merge sort's steps working from top down. This image courtesy Wikimedia and the Public Domain. . . .	95
9.6	Radix sort assisted Herman Hollerith during the tabulation of the United States census. . . . .	96
9.7	An example of quick sort on an input array. This image courtesy WikiMedia and the public domain. . . . .	99
9.8	Abstract data types and data structures implementing the interface. . . . .	103
10.1	The basic structure of a sequential access list. Each Node connects to the next through a saved address. . . . .	106
10.2	A basic node for a singly linked list. It contains a place to store the data value as well as a link to the next Node in the chain. . . . .	106
10.3	A tail-recursive <b>addLast</b> method for a basic singly-linked list.	107
10.4	This function accepts an address to the start of the list as an aliased variable. To remove the first item, simply point the head to the second item. What is this C++ code missing? . .	107
10.5	An updated node for a doubly linked list. With the addition of the back-pointer, most of the space to store the node goes to keeping track of list maintenance data. . . . .	108
10.6	An example of the complications introduced by the second pointer. Adding another node to the list requires substantial updating. . . . .	109
10.7	Triple reference pointers in C and C++ allow one to use casting to access the data contents, yet it makes it easy to work directly with the head reference if necessary. . . . .	109
11.1	Trees and Heaps courtesy of XKCD and the Creative Commons.	116
11.2	The minimum heap property holds for every node within the heap. All heaps, regardless of the input values, are complete, left-justified trees. . . . .	116
11.3	The array backing the abstract tree shown in Figure 11.2. All heaps use arrays, so this data structure represents the entire limits of the heap. . . . .	117
11.4	A left-justified, complete binary tree with six items will always take this shape. . . . .	117

- 11.5 Source code for a recursive heapification. This approach only works when one knows all the data structure's contents in advance like when sorting an array. . . . . 119
- 12.1 A node in the binary search tree associates the *key* and *value* using a linked group of binary nodes. . . . . 124
- 12.2 The unbalanced binary search tree resulting from entering in the keys in the order: 50, 80, 60, 25, 85, and 81. A different sequence may produce a different structure. . . . . 124
- 12.3 Rough code for a removal method in a binary search tree. This fragment misses several error conditions and significant use cases. . . . . 126
- 12.4 Locating the in-order predecessor is simpler with helper functions. . . . . 127
- 12.5 This tree more closely resembles a linked list, and it performs as poorly as well. Searching for the single key 82 requires linear time. . . . . 128
- 12.6 Several configurations possible using the same input data but in a different order. Only the middle configuration, however, offers balanced performance. . . . . 129
- 12.7 Rotating Node objects in the Java programming language. . . 130
- 12.8 The minimum number of nodes a complete tree vs an AVL tree. Observe that the AVL condition fails to produce a truly complete tree, but it performs well enough for a map. . . . 131
- 12.9 An example of the source required for a node in an AVL tree. It tracks the left and right children just as a standard binary search tree, but it now includes the additional space required to track its height. . . . . 132
- 12.10 An AVL tree with the  $\frac{\text{keys}}{\text{heights}}$  of each node. After inserting five keys in the AVL tree, it still resembles its basic, unbalanced counterpart. . . . . 132
- 12.11 The addition of the 42 key introduces an AVL tree violation. The 78 node has a left tree height of 2 and nothing on the right. 133
- 12.12 The fix-up for the violation introduced with the 42 key involves first rotating the 22 and 42 nodes left and then rotating to the right through 78. . . . . 133
- 12.13 The addition of the key 3 causes a red-violation in this red-black tree. This configuration triggers a rotation due to the NIL attached to node 1. . . . . 135

12.14	The reduced height in the red-black tree after the rotation triggered by 3's addition to the map. . . . .	136
12.15	Yet another red-violation when we insert the next number in the in-order sequence. . . . .	136
12.16	Changing the color of the new node's parent, grandparent, and aunt resolves this problem. Because this color flip includes the root, it actually causes two fix-up steps: color flip, root change to black. . . . .	137
12.17	Adding key 5 to the map triggers a rotation to the left for nodes 3, 4, and 5. . . . .	138
12.18	After the rotation triggered by 5's addition to the map. . . .	138
12.19	With node 6, another red-violation appears. . . . .	139
12.20	A color flip clears the problem. Observe there are the same number of black nodes between every leaf and the root. . . .	139
12.21	A simple color resolves the initial red-violation here, but it introduces side-effects above it in the tree. . . . .	140
12.22	Now nodes 4 and 6 share red as their color, but they are adjacent, so this creates a violation. . . . .	141
12.23	Rotating nodes 2, 4, and 6 solves the problem. . . . .	141
12.24	A node in a 5-tree tracks four keys and includes links to five other 5-nodes. This illustration courtesy the Wikimedia Commons . . . . .	142
12.25	The 2-3 tree changes the number of keys and links each node in the tree uses so it is always complete. The name references the number of outbound links each node will possess. . . .	142

# Bibliography

- [1] P. J. Landin, “The Mechanical Evaluation of Expressions,” *The Computer Journal*, vol. 6, pp. 308–320, 01 1964.
- [2] D. Jackson, “The google technical interview: How to get your dream job,” *XRDS*, vol. 20, pp. 12–14, Dec. 2013.
- [3] NIST, *NIST Special Publication 800-63B: Digital Identity Guidelines*. National Institute of Standards and Technology (NIST), June 2017.
- [4] L. Lasker and W. F. Parkes, “War games,” 1983.
- [5] GLEW, “The opengl extension wrangler library.” <http://glew.sourceforge.net/install.html>.
- [6] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley Professional, 4th ed., 2013.
- [7] Google, *Google C++ Style Guide*. Google, Inc., 2019.
- [8] Isocpp, “The c++ core guidelines.” <https://github.com/isocpp/CppCoreGuidelines>, May 2019.
- [9] Epic, “Coding standard.” <https://docs.unrealengine.com/en-us/Programming/Development/CodingStandard>, 2019.
- [10] S. E. Institute, *SEI CERT C++ Coding Standard*. Carnegie Mellon University, 2016.
- [11] D. Goldberg, “What every computer scientist should know about floating-point arithmetic,” *ACM Comput. Surv.*, vol. 23, pp. 5–48, Mar. 1991.
- [12] T. R. Nicely, “Pentium fdiv flaw faq.” <http://www.trnicely.net/pentbug/pentbug.html>.

- [13] N. Vo and K. Lee, “The rise of guardians: Fact-checking URL recommendation to combat fake news,” *CoRR*, vol. abs/1806.07516, 2018.

DRAFT