Think C++, a game perspective

Lisa Patacchiola, Allen B. Downey (original author)

Version 2.0 alpha 0.7

Think C++, a game perspective

The LaTeX source and code for this version of the book are available from

`https://github.com/lpatacch/thinkCPPGames`

The LaTeX source and code for the original book is available from

`https://github.com/AllenDowney/ThinkCPP`

Preface

0.1 Notes from Lisa

Hi! If you are reading this note, you are reading an alpha version of this book. That means that I have not completely converted the book over the the format I eventually want to have. Double check my github account that you have the most recent version.

If this is the most recent version, here is the information so far.

Chapter 1 and 2 has some work done. I have changed some of the examples to have more of a game flavor. I have moved Functions a bit further back in the book and have moved If statements up to Chapter 3 and loops have been moved up to Chapter 4. I have also brought switch statements and for loops up as well. Chapter 3 has been started, but has not been converted over yet. You will still find "FIXME"s in the chapter, and I have not switched the examples to something more game-like. I have put stubs for flowcharts and pseudo code, but I have not put anything in there yet. Chapter 4 has been started as well. There have been some flowcharts added. Other chapters have had small changes, mostly so the previous links are no longer broken.

I have added an appendix to talk about how to turn on warnings on Replit and how to setup Visual Studio Code on a Windows PC. There is also a new chapter (17) on advanced features like lambda expressions.

More notes:

I was previously using a different book for my classes. It did have lots of information, but it seemed like it had too much information. Students couldn't get through the reading because they found it too long and complex.

I started to use ThinkCPP, but I noticed that the book was using various coding practices that were no longer acceptable. (They were perfect when it was published, but C++ has changed a bit since then.) I first was only going to change a few examples, but after a few student questions, I decided to do a larger change to this book.

0.2 Acknowledgments

0.3 Contributions

The first edition was called "Think like a computer scientist" and was written by Allen B. Downey.

Lisa Patacchiola wrote the new additions and created Replit projects for the examples.

Contents

Preface **i**
0.1 Notes from Lisa . . . i
0.2 Acknowledgments . . . ii
0.3 Contributions . . . ii

1 The way of the program **1**
1.1 What is a programming language? . . . 1
1.2 What is a program? . . . 3
1.3 What is debugging? . . . 4
1.3.1 Compile-time errors . . . 4
1.3.2 Run-time errors . . . 4
1.3.3 Logic errors and semantics . . . 4
1.3.4 Experimental debugging . . . 5
1.4 Formal and natural languages . . . 6
1.5 The first program . . . 7
1.6 Exercises . . . 9
1.6.1 My world . . . 9
1.7 Glossary . . . 10

2 Variables and types **11**
2.1 More output . . . 11
2.1.1 Escape Sequences . . . 12
2.2 Values . . . 13
2.3 Variables . . . 14
2.4 Assignment . . . 14
2.5 Initialization . . . 15
2.5.1 Legacy . . . 15
2.5.2 Braced . . . 16
2.6 Outputting variables . . . 16
2.6.1 Format . . . 17
2.7 Variable Naming Rules . . . 18
2.7.1 Keywords . . . 18
2.8 Operators . . . 19
2.8.1 Not all the math operators . . . 19

2.8.2 Integer Division 20
2.8.3 Modulus operator 21
2.9 Order of operations 21
2.10 Operators for characters 22
2.11 Composition 22
2.12 Multiple assignment 23
2.13 Exercises 24
2.13.1 Hello World 24
2.13.2 Modulus Practice 24
2.13.3 Validation 25
2.14 Glossary 26

3 Conditionals **29**
3.1 Floating-point 29
3.1.1 Converting from `double` to `int` 30
3.2 Conditional execution 31
3.3 Alternative execution 33
3.4 Chained conditionals 34
3.5 Nested conditionals 34
3.6 Logical operators 36
3.7 Short circuiting the logic 36
3.8 Conditional (ternary) Operator 37
3.9 `switch` statement 38
3.9.1 Fallthrough 39
3.10 Pseudocode 40
3.11 Flowcharts 41
3.11.1 Oval 41
3.11.2 Diamond 41
3.11.3 Rectangle 41
3.11.4 Parallelogram 41
3.11.5 How they connect 42
3.12 Testing Techniques 43
3.13 Exercises 44
3.13.1 Rock Paper Scissors 44
3.13.2 Magic 8 Ball 44
3.13.3 Decision Maker 45
3.14 Glossary 45

4 Iteration **47**
4.1 The `while` statement 47
4.2 New Operators 50
4.3 Increment and decrement operators 50
4.4 Parts of a loop 52
4.5 Definite loop 52
4.6 Indefinite loop 53
4.7 `do-while` statement 53

4.8 `for` statement . . . 55
4.9 Break and Continue . . . 56
4.10 Glossary . . . 56
4.11 Game Loops . . . 57

5 Function **59**
5.1 Math functions . . . 59
5.2 Math constants . . . 61
5.3 Composition . . . 61
5.4 Adding new functions . . . 61
5.5 Definitions and uses . . . 64
5.6 Function Prototypes . . . 65
5.7 Programs with multiple functions . . . 66
5.8 Parameters and arguments . . . 66
5.9 Parameters and variables are local . . . 67
5.10 Functions with multiple parameters . . . 68
5.11 Functions with results . . . 69
5.12 The `return` statement . . . 70
5.13 Recursion . . . 70
5.14 Infinite recursion . . . 72
5.15 Stack diagrams for recursive functions . . . 73
5.16 Glossary . . . 73

6 Fruitful functions **75**
6.1 Return values . . . 75
6.2 Program development . . . 77
6.3 Composition . . . 79
6.4 Overloading . . . 80
6.5 Boolean values . . . 81
6.6 Boolean variables . . . 81
6.7 Logical operators . . . 82
6.8 Bool functions . . . 82
6.9 Returning from `main` . . . 83
6.10 More recursion . . . 83
6.11 Leap of faith . . . 86
6.12 One more example . . . 87
6.13 Encapsulation and generalization . . . 87
6.14 Functions . . . 89
6.15 More encapsulation . . . 89
6.16 Local variables . . . 89
6.17 More generalization . . . 90
6.18 Glossary . . . 92
6.19 Traditional array . . . 94
6.20 std::array . . . 95
6.20.1 Conversions to and from traditional arrays . . . 96
6.21 A new way to send an array into a function . . . 97

6.22 And wait, there is more! . . . 98

7 Strings and things **99**
7.1 Different Strings available . . . 99
7.2 C-Strings . . . 99
7.3 C++ strings . . . 100
7.4 `string` variables . . . 100
7.5 Extracting characters from a string . . . 100
7.6 Length . . . 101
7.7 A run-time error . . . 102
7.8 Traversal . . . 102
7.9 Range based for loop . . . 103
7.10 The `find` function . . . 104
7.11 Our own version of `find` . . . 104
7.12 Looping and counting . . . 105
7.13 String concatenation . . . 105
7.14 `string`s are mutable . . . 106
7.15 `string`s are comparable . . . 107
7.16 Character classification . . . 107
7.17 Other `string` functions . . . 108
7.18 string_view . . . 108
7.19 Glossary . . . 109

8 Structures **111**
8.1 Compound values . . . 111
8.2 `Point` objects . . . 111
8.3 Accessing instance variables . . . 112
8.4 Operations on structures . . . 113
8.5 Structures as parameters . . . 114
8.6 Call by value . . . 114
8.7 Call by reference . . . 115
8.8 Rectangles . . . 116
8.9 Structures as return types . . . 118
8.10 Passing other types by reference . . . 118
8.11 Getting user input . . . 119
8.12 Glossary . . . 121

9 More structures **123**
9.1 Time . . . 123
9.2 `printTime` . . . 124
9.3 Functions for objects . . . 124
9.4 Pure functions . . . 125
9.5 `const` parameters . . . 126
9.6 Modifiers . . . 127
9.7 Fill-in functions . . . 128
9.8 Which is best? . . . 129

9.9 Incremental development versus planning . . . 129
9.10 Generalization . . . 130
9.11 Algorithms . . . 130
9.12 Glossary . . . 131

10 Vectors **133**
10.1 Accessing elements . . . 134
10.2 Copying vectors . . . 135
10.3 `for each` loops . . . 135
10.4 Vector size . . . 136
10.5 Vector functions . . . 136
10.6 Random numbers . . . 137
10.7 Statistics . . . 138
10.8 Vector of random numbers . . . 139
10.9 Counting . . . 140
10.10Checking the other values . . . 140
10.11A histogram . . . 142
10.12A single-pass solution . . . 142
10.13Random seeds . . . 143
10.14Glossary . . . 143

11 Member functions **145**
11.1 Objects and functions . . . 145
11.2 `print` . . . 146
11.3 Implicit variable access . . . 147
11.4 Another example . . . 148
11.5 Yet another example . . . 149
11.6 A more complicated example . . . 149
11.7 Constructors . . . 150
11.8 Initialize or construct? . . . 151
11.9 One last example . . . 152
11.10Header files . . . 152
11.11Glossary . . . 155

12 Vectors of Objects **157**
12.1 Composition . . . 157
12.2 `Card` objects . . . 157
12.3 The `printCard` function . . . 159
12.4 The `equals` function . . . 161
12.5 The `isGreater` function . . . 162
12.6 Vectors of cards . . . 163
12.7 The `printDeck` function . . . 165
12.8 Searching . . . 165
12.9 Bisection search . . . 166
12.10Decks and subdecks . . . 169
12.11Glossary . . . 169

13 Objects of Vectors **171**
13.1 Enumerated types . . . 171
13.2 `switch` statement . . . 172
13.3 Decks . . . 173
13.4 Another constructor . . . 174
13.5 `Deck` member functions . . . 175
13.6 Shuffling . . . 176
13.7 Sorting . . . 177
13.8 Subdecks . . . 177
13.9 Shuffling and dealing . . . 178
13.10 Mergesort . . . 179
13.11 Glossary . . . 181

14 Classes and invariants **183**
14.1 Private data and classes . . . 183
14.2 What is a class? . . . 184
14.3 Complex numbers . . . 185
14.4 Accessor functions . . . 187
14.5 Output . . . 188
14.6 A function on `Complex` numbers . . . 189
14.7 Another function on `Complex` numbers . . . 189
14.8 Invariants . . . 190
14.9 Preconditions . . . 191
14.10 Private functions . . . 193
14.11 Glossary . . . 194

15 File Input/Output and `apmatrixes` **195**
15.1 Streams . . . 195
15.2 File input . . . 196
15.3 File output . . . 197
15.4 Parsing input . . . 197
15.5 Parsing numbers . . . 199
15.6 The `Set` data structure . . . 200
15.7 `apmatrix` . . . 203
15.8 A distance matrix . . . 204
15.9 A proper distance matrix . . . 205
15.10 Glossary . . . 207

16 Advanced Classes **209**
16.1 Inheritance . . . 209
16.2 Polymorphism . . . 209
16.3 Overloading Operators . . . 209
16.3.1 Overloading Parentheses . . . 209
16.4 Copy/Move . . . 209
16.5 Smart Pointers . . . 209

17 Advanced Topics **211**
17.1 Function Pointer 211
17.2 Function Objects 212
17.3 Lambda Expressions 213
17.3.1 Generic Lambda 214
17.3.2 Constant expressions 214
17.3.3 Return Values 214
17.3.4 Captures 215
17.3.5 Mutable 216
17.4 std::function 216
17.4.1 Problems with returned reference 217
17.4.2 Predefined functions 217
17.5 Glossary 217

A Integrated Development Environments **219**
A.1 Installing 219
A.1.1 Replit 219
A.1.2 Visual Studio Code 219
A.2 Setting up a project 220
A.2.1 Replit 220
A.2.2 Visual Studio Code 221
A.3 Settings 221
A.3.1 Replit 222
A.3.2 Visual Studio Code 223

Chapter 1

The way of the program

The goal of this book is to teach you to think like a computer scientist. I like the way computer scientists think because they combine some of the best features of Mathematics, Engineering, and Natural Science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating trade-offs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem-solving**. By that I mean the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called "The way of the program."

1.1 What is a programming language?

The programming language you will be learning is C++, because it is used in many video games and in the Unreal Engine. Both C++ and C# are **high-level languages**; other high-level languages you might have heard of are Java, C and Python.

As you might infer from the name "high-level language," there are also **low-level languages**, sometimes referred to as machine language or assembly language. Loosely-speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be translated before they can run. This translation takes some time, which is a small disadvantage of high-level languages.

But the advantages are enormous. First, it is *much* easier to program in a high-level language; by "easier" I mean that the program takes less time to write, it's shorter and easier to read, and it's more likely to be correct. Secondly, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can only run on

one kind of computer, and have to be rewritten to run on another.

Due to these advantages, almost all programs are written in high-level languages. Low-level languages are only used for a few special applications.

There are two ways to translate a program; **interpreting** or **compiling**. An interpreter is a program that reads a high-level program and does what it says. In effect, it translates the program line-by-line, alternately reading lines and carrying out commands.

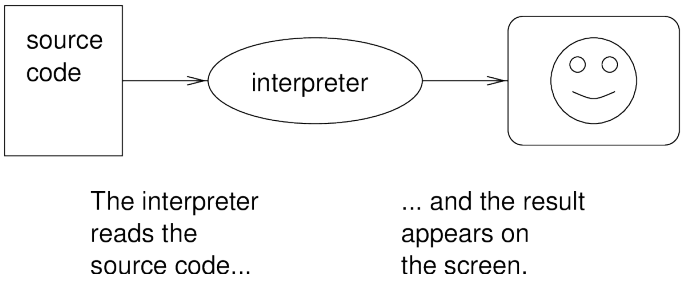
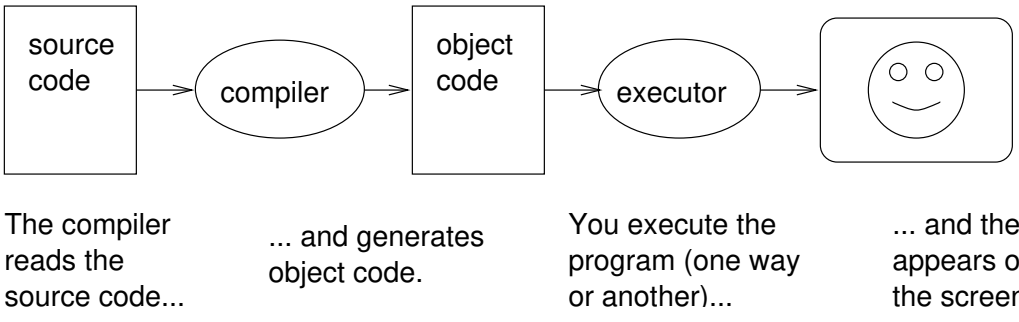


Figure 1.1: From source code to Result

A compiler is a program that reads a high-level program and translates it all at once, before executing any of the commands. Often you compile the program as a separate step, and then execute the compiled code later. In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**.

As an example, suppose you write a program in C++. You might use a text editor to write the program (a text editor is a simple word processor). When the program is finished, you might save it in a file named `program.cpp`, where "program" is an arbitrary name you make up, and the suffix `.cpp` is a convention that indicates that the file contains C++ source code.

Then, depending on what your programming environment is like, you might leave the text editor and run the compiler. The compiler would read your source code, translate it, and create a new file named `program.o` or `program.obj` to contain the object code, and `program.out` or `program.exe` to contain the executable.



The next step is to run the program, which requires some kind of executor. The role of the executor is to load the program (copy it from disk into memory) and make the computer start executing the program.

Although this process may seem complicated, the good news is that in most programming environments (sometimes called integrated development environments(IDE)), these steps are automated for you. Usually you will only have to write a program and type a single command to compile and run it. On the other hand, it is useful to know what the steps are that are happening in the background so that if something goes wrong, you can figure out what it is.

The longer code examples in this book will use Replit.com, an online programming platform/IDE, and community. Code listings include links to the "repl" that you can copy ("fork") and experiment with.

1.2 What is a program?

A program is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, like solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, like searching and replacing text in a document or (strangely enough) compiling a program. Or, it could be something fun, like a character trying to walk around and find a dragon. All of these things can be done with a program.

The instructions (or commands, or statements) look different in different programming languages, but there are a few basic functions that appear in just about every language:

input: Get data from the keyboard, or a file, or some other device.

output: Display data on the screen or send data to a file or other device.

math: Perform basic mathematical operations like addition and multiplication. (Sometimes called "Processing")

testing: Check for certain conditions and execute the appropriate sequence of statements. (Sometimes called "Selection")

repetition: Perform some action repeatedly, usually with some variation. (Sometimes called "Iteration" or "looping")

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of functions that look more or less like these. Thus, one way to describe programming is the process of breaking a large, complex task up into smaller and smaller subtasks until eventually the subtasks are simple enough to be performed with one of these simple functions.

1.3 What is debugging?

Programming is a complex process, and since it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

There are a few different kinds of errors that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

1.3.1 Compile-time errors

The compiler can only translate a program if the program is syntactically correct; otherwise, the compilation fails and you will not be able to run your program. **Syntax** refers to the structure of your program and the rules about that structure. A mistake with the syntax is often called a **syntax error**.

For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a syntax error. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e e cummings without spewing error messages.

Compilers are not so forgiving. If there is a single syntax error anywhere in your program, the compiler will print an error message and quit, and you will not be able to run your program.

To make matters worse, there are more syntax rules in C++ than there are in English, and the error messages you get from the compiler are often not very helpful. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

When you get a syntax error, don't think that the compiler is yelling at you for getting something wrong. Think of it ask asking you for help. "Help" it is saying "I have no idea what this means, please help me". Honestly, I like syntax errors more than any of the other types of errors. Syntax errors are one of the few errors where the computer tells you it found a problem and suggests what it thinks is wrong.

1.3.2 Run-time errors

The second type of error is a run-time error, so-called because the error does not appear until you run the program.

For the simple sorts of programs we will be writing for the next few weeks, run-time errors are rare, so it might be a little while before you encounter one.

1.3.3 Logic errors and semantics

The third type of error is the **logical** or **semantic** error. If there is a logical error in your program, it will compile and run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong.

Let's say you have a program that tells the computer to do the following steps:

1. Get the pitcher of water
2. Pour the water
3. Get the glass

A person would see that list and would assume you meant to get the glass first. The computer would do exactly as you asked and spill water on the table and then put a glass on top. There was an obvious logical error in your program, but it ran without a compile error. As far as the computer was concerned, it ran perfectly. But, you know that was not your original intent.

Identifying logical errors can be tricky, since it requires you to work backwards by looking at the output of the program and trying to figure out what it is doing.

1.3.4 Experimental debugging

One of the most important skills you should acquire from working with this book is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways debugging is like detective work. You are confronted with clues and you have to infer the processes and events that lead to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (from A. Conan Doyle's *The Sign of Four*).

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should always start with a working program that does *something*, and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux" (from *The Linux Users' Guide* Beta Version 1).

In later chapters I will make more suggestions about debugging and other programming practices.

1.4 Formal and natural languages

Natural languages are the languages that people speak, like English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

> **Programming languages are formal languages that have been designed to express computations.**

As I mentioned before, formal languages tend to have strict rules about syntax. For example, 3+3 = 6 is a syntactically correct mathematical statement, but 3 = +6$ is not. Also, H_2O is a syntactically correct chemical name, but ${}_2Zz$ is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, like words and numbers and chemical elements. One of the problems with `3=+6$` is that `$` is not a legal token in mathematics (at least as far as I know). Similarly, ${}_2Zz$ is not legal because there is no element with the abbreviation Zz.

The second type of syntax error pertains to the structure of a statement; that is, the way the tokens are arranged. The statement `3=+6$` is structurally illegal, because you can't have a plus sign immediately after an equals sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this unconsciously). This process is called **parsing**.

For example, when you hear the sentence, "The other shoe fell," you understand that "the other shoe" is the subject and "fell" is the verb. Once you have parsed a sentence, you can figure out what it means, that is, the semantics of the sentence. Assuming that you know what a shoe is, and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common—tokens, structure, syntax and semantics—there are many differences.

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness: Natural languages are full of idiom and metaphor. If I say, "The other shoe fell," there is probably no shoe and nothing falling. Formal languages mean exactly what they say.

People who grow up speaking a natural language (everyone) often have a hard time adjusting to formal languages. In some ways the difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose: The literal meaning of words is more important and the structure contributes more meaning. Prose is more amenable to analysis than poetry, but still often ambiguous.

Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, remember that the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.5 The first program

Traditionally the first program people write in a new language is called "Hello, World." because all it does is print the words "Hello, World." In C++, this program looks like this:

```
#include <iostream>

// main: generate some simple output

int main ()
{
  std::cout << "Hello, world." << std::endl;
  return 0;
}
```

Some people judge the quality of a programming language by the simplicity of the "Hello, World." program. By this standard, C++ does reasonably well. Even so, this simple program contains several features that are hard to explain

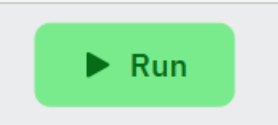


Figure 1.2: The Run button on Replit

to beginning programmers. For now, we will ignore some of them, like the first line.

The third line begins with `//`, which indicates that it is a **comment**. A comment is a bit of English text that you can put in the middle of a program, usually to explain what the program does. When the compiler sees a `//`, it ignores everything from there until the end of the line.

In the fourth line, you can ignore the word `int` for now, but notice the word `main`. `main` is a special name that indicates the place in the program where execution begins. When the program runs, it starts by executing the first statement in `main` and it continues, in order, until it gets to the last statement, and then it quits.

There is no limit to the number of statements that can be in `main`, but the example contains only one. It is a basic **output** statement, meaning that it outputs or displays a message on the screen.

`std::cout` is a special object provided by the system to allow you to send output to the screen. The symbol `<<` is an **operator** that you apply to `std::cout` and a string, and that causes the string to be displayed.

`std::endl` is a special symbol that represents the end of a line. When you send an `std::endl` to `std::cout`, it causes the cursor to move to the next line of the display. The next time you output something, the new text appears on the next line. (Note: the word is endl ends with an L, not a 1.)

Like all statements, the output statement ends with a semi-colon (`;`).

There are a few other things you should notice about the syntax of this program. First, C++ uses curly-braces ({ and }) to group things together. In this case, the output statement is enclosed in curly-braces, indicating that it is *inside* the definition of `main`. Also, notice that the statement is indented, which helps to show visually which lines are inside the definition.

At this point it would be a good idea to sit down in front of a computer and compile and run this program.

To try program out, go to: Hello World Project.

When the project is loaded, you should see a button that looks like Figure 1.2.

If you click that "Run" button, it will compile the code and run it. You should see the words "Hello World!" on the right side of the environment. Congratulations! You just created your first program!

As I mentioned, the C++ compiler is a real stickler for syntax. If you make any errors when you type in the program, chances are that it will not compile

successfully. For example, if you misspell `iostream`, you might get an error message like the following:

```
hello.cpp:1: oistream.h: No such file or directory
```

There is a lot of information on this line, but it is presented in a dense format that is not easy to interpret. A more friendly compiler might say something like:

> "On line 1 of the source code file named hello.cpp, you tried to include a header file named oistream.h. I didn't find anything with that name, but I did find something named iostream. Is that what you meant, by any chance?"

Unfortunately, few compilers are so accommodating. The compiler is not really very smart, and in most cases the error message you get will be only a hint about what is wrong. It will take some time to gain facility at interpreting compiler messages.

Nevertheless, the compiler can be a useful tool for learning the syntax rules of a language. Starting with a working program (like hello.cpp), modify it in various ways and see what happens. If you get an error message, try to remember what the message says and what caused it, so if you see it again in the future you will know what it means.

1.6 Exercises

1.6.1 My world

This would be a good time to get your development environment working. Although I have many of the examples on Replit, you can use any development environment you wish, as long as it supports C++. I have some information about how to install some of the more popular environments in Appendix A.1.

After you have installed your development environment, you will need to create a C++ project. Follow either the instructions from the appendix (A.2), or what comes with your development environment to make a new C++ project. Some automatically give you a hello world project. If you are one of the lucky ones with the pre-made c++ file, you will be using that for this exercise.. Otherwise, copy the hello world program from 1.5. First, before you change any of the hello world code, click "run". For most development environments, this will compile and execute the program, so you can see what it will output. You should see something similar to "Hello world".

Once you can see the "Hello world". Try changing the code to say, "Hello! (Put your name here) got this running" and run the new code. Yay! You made your first project!

1.7 Glossary

problem-solving: The process of formulating a problem, finding a solution, and expressing the solution.

high-level language: A programming language like C++ that is designed to be easy for humans to read and write.

low-level language: A programming language that is designed to be easy for a computer to execute. Also called "machine language" or "assembly language."

portability: A property of a program that can run on more than one kind of computer.

formal language: Any of the languages people have designed for specific purposes, like representing mathematical ideas or computer programs. All programming languages are formal languages.

natural language: Any of the languages people speak that have evolved naturally.

interpret: To execute a program in a high-level language by translating it one line at a time.

compile: To translate a program in a high-level language into a low-level language, all at once, in preparation for later execution.

source code: A program in a high-level language, before being compiled.

object code: The output of the compiler, after translating the program.

executable: Another name for object code that is ready to be executed.

algorithm: A general process for solving a category of problems.

bug: An error in a program.

syntax: The structure of a program.

semantics: The meaning of a program.

parse: To examine a program and analyze the syntactic structure.

syntax error: An error in a program that makes it impossible to parse (and therefore impossible to compile).

run-time error: An error in a program that makes it fail at run-time.

logical error: An error in a program that makes it do something other than what the programmer intended.

debugging: The process of finding and removing any of the three kinds of errors.

Chapter 2

Variables and types

2.1 More output

As I mentioned in the last chapter, you can put as many statements as you want in `main`. For example, to output more than one line:

```
#include <iostream>

// main: generate some simple output

int main ()
{
  // output 1 line
  std::cout << "Hello, world." << std::endl;

  // output 1 more
  std::cout << "How are you?" << std::endl; //great!
  return 0;
}
```

As you can see, it is legal to put comments at the end of a line, as well as on a line by themselves.

If you want to try that with the replit code, go back to the project link `https://replit.com/@lpatacch/helloWorld#hello.cpp`. Then, look for a button on the right hand of the screen with the words "Fork Repl". It should look like Figure 2.1 You now should have a copy of the code that you can change any way you want.

In the program. the phrases that appear in quotation marks are called **strings**, because they are made up of a sequence (string) of letters. Actually, strings can contain any combination of letters, numbers, punctuation marks, and other special characters.

Often it is useful to display the output from multiple output statements all on one line. You can do this by leaving out the first `endl`:

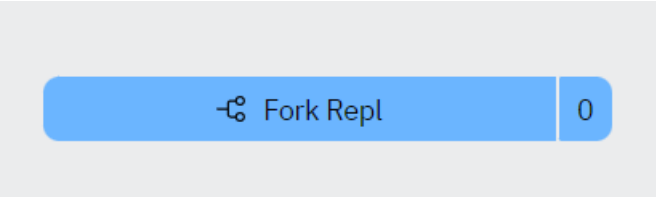


Figure 2.1: The Fork button on Replit

```
int main ()
{
  std::cout << "The game ";
  std::cout << "is about to start!" << std::endl;
  return 0
}
```

In this case the output appears on a single line as `The game is about to start!`. Notice that there is a space between the word "game" and the second quotation mark. This space appears in the output, so it affects the behavior of the program.

Spaces that appear outside of quotation marks generally do not affect the behavior of the program. For example, I could have written:

```
int main ()
{
  std::cout<<"The game";
  std::cout<<" is about to start!"<<     std::endl;
  return 0;
}
```

This program would compile and run just as well as the original. The breaks at the ends of lines (newlines) do not affect the program's behavior either, so I could have written:

```
int main(){std::cout<<"The game ";std::cout<<
    "is about to start!"<< std::endl;return 0;}
```

That would work, too, although you have probably noticed that the program is getting harder and harder to read. Newlines and spaces are useful for organizing your program visually, making it easier to read the program and locate syntax errors.

2.1.1 Escape Sequences

There are special character sequences that can be put in a string that will act differently than other characters. They are called escape sequences. One popular escape sequence is '\n'. This character is very similar to std::endl. When printed, it goes to the next line. For example, this code:

```
#include <iostream>

// main: generate some simple output

int main ()
{
  // This prints in 2 lines
  std::cout << "Hello, world. \n How are you?" << std::endl;
  return 0;
}
```

Displays the following:

```
Hello, world.
 How are you?
```

There are other characters that can be helpful. I put some of the most popular in the table below.

Escape Character	Behavior
\n	Goes to the next line
\t	Creates a horizontal tab
\\	Displays the slash character
\"	Displays the double quote character

Table 2.1: Escape characters and behavior

2.2 Values

A value is one of the fundamental things—like a letter or a number—that a program manipulates. The only values we have manipulated so far are the string values we have been outputting, like `"Hello, world."`. You (and the compiler) can identify string values because they are enclosed in quotation marks.

There are other kinds of values, including integers and characters. An integer is a whole number like 1 or 17. You can output integer values the same way you output strings:

```
  std::cout << 17 << std::endl;
```

A character value is a letter or digit or punctuation mark enclosed in single quotes, like `'a'` or `'5'`. You can output character values the same way:

```
  std::cout << '}' << std::endl;
```

This example outputs a single close curly-brace on a line by itself.

It is easy to confuse different types of values, like `"5"`, `'5'` and `5`, but if you pay attention to the punctuation, it should be clear that the first is a string, the second is a character and the third is an integer. The reason this distinction is important should become clear soon.

2.3 Variables

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a named location that stores a value.

Just as there are different types of values (integer, character, etc.), there are different types of variables. When you create a new variable, you have to declare what type it is. For example, the character type in C++ is called `char`. The following statement creates a new variable named `sarah` that has type `char`.

```
char sarah;
```

This kind of statement is called a **declaration**.

The type of a variable determines what kind of values it can store. A `char` variable can contain characters, and it should come as no surprise that `int` variables can store integers.

There are several types in C++ that can store string values, but we are going to skip that for now (see Chapter 7).

To create an integer variable, the syntax is

```
int bob;
```

where `bob` is the arbitrary name you made up for the variable. In general, you will want to make up variable names that indicate what you plan to do with the variable. For example, if you saw these variable declarations:

```
char firstLetter;
char lastLetter;
int hour, minute;
```

you could probably make a good guess at what values would be stored in them. This example also demonstrates the syntax for declaring multiple variables with the same type: `hour` and `minute` are both integers (`int` type).

2.4 Assignment

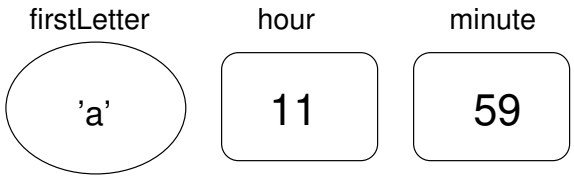
Now that we have created some variables, we would like to store values in them. We do that with an **assignment statement**.

```
firstLetter = 'a';   // give firstLetter the
                     // value 'a'
hour = 11;           // assign the value 11 to hour
minute = 59;         // set minute to 59
```

This example shows three assignments, and the comments show three different ways people sometimes talk about assignment statements. The vocabulary can be confusing here, but the idea is straightforward:

- When you declare a variable, you create a named storage location.
- When you make an assignment to a variable, you give it a value.

A common way to represent variables on paper is to draw a box with the name of the variable on the outside and the value of the variable on the inside. This kind of figure is called a **state diagram** because is shows what state each of the variables is in (you can think of it as the variable's "state of mind"). This diagram shows the effect of the three assignment statements:



I sometimes use different shapes to indicate different variable types. These shapes should help remind you that one of the rules in C++ is that a variable has to have the same type as the value you assign it. For example, you cannot store a string in an `int` variable. The following statement generates a compiler error.

```
int hour;
hour = "Hello.";        // WRONG !!
```

This rule is sometimes a source of confusion, because there are many ways that you can convert values from one type to another, and C++ sometimes converts things automatically. But for now you should remember that as a general rule variables and values have the same type, and we'll talk about special cases later.

Another source of confusion is that some strings *look* like integers, but they are not. For example, the string `"123"`, which is made up of the characters `1`, `2` and `3`, is not the same thing as the *number* `123`. This assignment is illegal:

```
minute = "59";          // WRONG!
```

2.5 Initialization

When variables are created, they can be set to a value immediately instead of waiting until later. This can shorten your code.

2.5.1 Legacy

Pre-C++ 11, it was quite common to use an equation to initialize your variables. Here is an example:

```
int hitPoints = 10;
```

Most of our examples use this format, because people are used to this style when using other computer languages. But, C++11 and later have a different style that can be used instead.

2.5.2 Braced

This new style is called braced initialization. Instead of having an equal sign, the line is even shorter. Although it does the same thing as the equal sign, it can help with some common programming errors. We will be talking about this in later chapters.

```
int hitPoints{10};
```

2.6 Outputting variables

You can output the value of a variable using the same commands we used to output simple values.

```
int hour, minute;
char colon;

hour = 11;
minute = 59;
colon = ':';

std::cout << "The battle will start at  ";
std::cout << hour;
std::cout << colon;
std::cout << minute;
std::cout << std::endl;
```

Try this project here:`https://replit.com/@lpatacch/OutputVariables#outvar.cpp`. This program creates two integer variables named `hour` and `minute`, and a character variable named `colon`. It assigns appropriate values to each of the variables and then uses a series of output statements to generate the following:

```
The battle will start at 11:59
```

When we talk about "outputting a variable," we mean outputting the *value* of the variable. To output the *name* of a variable, you have to put it in quotes. For example: `std::cout << "hour";`

As we have seen before, you can include more than one value in a single output statement, which can make the previous program more concise:

```
int hour, minute;
char colon;

hour = 11;
minute = 59;
colon = ':';

std::cout << "The battle will start at " << hour
              << colon << minute << std::endl;
```

On one line, this program outputs a string, two integers, a character, and the special value `endl`. Very impressive!

2.6.1 Format

There is a new way to output values in C++20[1] called Format. It is a more recent method (so new that at the time of this writing, it was not supported by some compilers, including Replit.) If you try this code and get a compiler error that "error: no member named 'format' in namespace 'std'," it may not be supported in your development environment yet.

If this is supported [2], this is a different way for you to be able to output variables. To get the following code to work, you will need to have the line:

```
#include <format>
```

up where your "#include <iostream>" line is.

To use format, you will be putting {} where you want your values to be placed. The above example could be changed to this:

```
int hour, minute;
char colon;

hour = 11;
minute = 59;
colon = ':';

std::cout << std::format(
 "The battle will start at {}{}{}.\n",
    hour, colon, minute);
```

The first variable will be put where the first {} is, the second in the second {} and so on. If you want, you can even number the braces, to make it easier to see what variable will be output where.

```
std::cout << std::format(
  "The battle will start at {0}{2}{1}.\n",
    hour, minute, colon);
```

The first variable will be at the 0 slot, the second will be at the 1 slot, and the third with be at the 2 slot.

This method is an excellent way to output values, but because it is yet to be available for all environments, the following examples in the book will not use it. If you prefer this method, it is perfectly fine to use this instead of the one in the other examples.

[1] There are different versions of C++, named for the year that they were approved. C++20 was approved in 2020.

[2] If you are looking for a development environment that supports format, the following code was tested on Visual Studio Code using the Microsoft compiler.

2.7 Variable Naming Rules

A few sections ago, I said that you can make up any name you want for your variables, but that's not quite true. There are certain rules that you need to follow.

1. The name can not start with a number. (It is ok for a number to be in the name, it just can't be the first letter.)
2. No spaces in the name
3. Capitalization matters (num is different than NUM)
4. No special characters can be in the name other than underscore.

Here are some examples of variables that do not follow the naming rules, and what you can use instead

```
int 3player;     \\ WRONG, can not start with a
                 \\ number
int player3;     \\ This would work
int my3player    \\ This will also work

int hit points; \\ WRONG, spaces not allowed in the
                 \\ name
int hitPoints;   \\ This would work
int hit_points; \\ This also would work

int cash$;       \\ This would not work because of
                 \\ $sign
int cashMoney;   \\ changed to use words instead
```

2.7.1 Keywords

In addition, there are certain words that are reserved in C++ because they are used by the compiler to parse the structure of your program, and if you use them as variable names, it will get confused. These words, called **keywords**, include `int`, `char`, `void`, `endl` and many more.

The complete list of keywords is included in the C++ Standard, which is the official language definition adopted by the the International Organization for Standardization (ISO) [3]. You can download a copy electronically from

`http://www.ansi.org/`

Rather than memorize the list, I would suggest that you take advantage of a feature provided in many development environments: code highlighting. As

[3] There are many versions of this standard. The language has been adding more and more features over time. There is information in the Appendix on how to switch the version of the standard you are using.

you type, different parts of your program should appear in different colors. For example, keywords might be blue, strings red, and other code black. If you type a variable name and it turns blue, watch out! You might get some strange behavior from the compiler.

2.8 Operators

Operators are special symbols that are used to represent simple computations like addition and multiplication. Most of the operators in C++ do exactly what you would expect them to do, because they are common mathematical symbols. For example, the operator for adding two integers is `+`.

The following are all legal C++ expressions whose meaning is more or less obvious:

```
1+1        hour-1       hour*60 + minute     minute/60
```

Expressions can contain both variables names and integer values. In each case the name of the variable is replaced with its value before the computation is performed.

2.8.1 Not all the math operators

Although some of the arithmetic operators that you would expect are there, there are some that do not work in C++. Here are a few examples of what does not work, and how to write it in a different way.

```
int powerUpLevel = 2;
int damageLevel = 3;

int damageGiven = powerUpLeveldamageLevel;
    // WRONG
    // You can't just put variables next to each
    // other to have them multiply

int damageGiven = powerUpLevel x damageLevel;
    // WRONG
    // You can't use x as a multiplication sign. It
    // is often used as a variable

int damageGiven = powerUpLevel * damageLevel;
    // Correct!
    // When you multiply, you should use the * symbol

int valToPower = powerUpLevel^2;        // WRONG
    // Although you can use the ^ symbol for a
    // particular math operation, it is not to
    // calculate the square (or any other power)
    // We will see this later in the book
```

```
    int valToPower = powerUpLevel * powerUpLevel;
        // Correct!
        // To square a number, you can multiply it with
        // itself
        // There is also another way to do this which we
        // will see later

    int percentHurt = damageGiven/100;      // Correct
        // To divide, we should use the / symbol
        // There is something about this that is an
        // issue, that we will talk about right now.
```

Also, this symbol should not be used in code – ÷. Code does not know what to do with it, so you should use the other method for division.

2.8.2 Integer Division

Addition, subtraction and multiplication all do what you expect, but you might be surprised by division. For example, the following program:

```
int hour, minute;
int hitpoints = 49;
hour = 11;
minute = 59;

std::cout << "Number of minutes since the midnight";
std::cout<<" battle: "<< hour*60 + minute << std::endl;

// This is integer division, so the result will be zero
std::cout << "Fraction of hitpoints left ";
std::cout << hitpoints/50 << std::endl;
```

would generate the following output:

```
Number of minutes since the midnight battle: 719
Fraction of the hitpoints left: 0
```

Try it yourself here: `https://replit.com/@lpatacch/IntegerDivision#intdiv.cpp`. The first line is what we expected, but the second line is odd. The value of the variable `hitpoints` is 49, and 49 divided by 50 is 0.98, not 0. The reason for the discrepancy is that C++ is performing **integer division**.

When both of the **operands** are integers (operands are the things operators operate on), the result must also be an integer, and by definition integer division always rounds *down*, even in cases like this where the next integer is so close.

A possible alternative in this case is to calculate a percentage rather than a fraction:

```
std::cout << "Percentage of the hitpoints left:";
std::cout << " " << hitpoints*100/50 << std::endl;
```

The result is:

```
Percentage of the hitpoints left: 98
```

Again the result is rounded down, but at least now the answer is approximately correct. In order to get an even more accurate answer, we could use a different type of variable, called floating-point, that is capable of storing fractional values. We'll get to that soon in Chapter 3.1.

2.8.3 Modulus operator

There is another way to get the information that the integer division is not using. That is a new operator called the **modulus**, or **remainder** operator.

The modulus operator works on integers (and integer expressions) and yields the *remainder* when the first operand is divided by the second. In C++, the modulus operator is a percent sign, `%`. The syntax is exactly the same as for other operators:

```
int quotient = 7 / 3;
int remainder = 7 % 3;
```

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

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

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

2.9 Order of operations

When more than one operator appears in an expression the order of evaluation depends on the rules of **precedence**. A complete explanation of precedence can get complicated, but just to get you started:

- Multiplication and division (and modulus) happen before addition and subtraction. So `2*3-1` yields 5, not 4, and `2/3-1` yields -1, not 1 (remember that in integer division `2/3` is 0).

- If the operators have the same precedence they are evaluated from left to right. So in the expression `minute*100/60`, the multiplication happens first, yielding `5900/60`, which in turn yields `98`. If the operations had gone from right to left, the result would be `59*1` which is `59`, which is wrong.

- Any time you want to override the rules of precedence (or you are not sure what they are) you can use parentheses. Expressions in parentheses are evaluated first, so `2 * (3-1)` is 4. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even though it doesn't change the result.

2.10 Operators for characters

Interestingly, the same mathematical operations that work on integers also work on characters. For example,

```
char letter;
letter = 'a' + 1;
std::cout << letter << std::endl;
```

outputs the letter `b`. Although it is syntactically legal to multiply characters, it is almost never useful to do it.

Earlier I said that you can only assign integer values to integer variables and character values to character variables, but that is not completely true. In some cases, C++ converts automatically between types. For example, the following is legal.

```
int number;
number = 'a';
std::cout << number << std::endl;
```

The result is 97, which is the number that is used internally by C++ to represent the letter `'a'`. However, it is generally a good idea to treat characters as characters, and integers as integers, and only convert from one to the other if there is a good reason.

Automatic type conversion is an example of a common problem in designing a programming language, which is that there is a conflict between **formalism**, which is the requirement that formal languages should have simple rules with few exceptions, and **convenience**, which is the requirement that programming languages be easy to use in practice.

More often than not, convenience wins, which is usually good for expert programmers, who are spared from rigorous but unwieldy formalism, but bad for beginning programmers, who are often baffled by the complexity of the rules and the number of exceptions. In this book I have tried to simplify things by emphasizing the rules and omitting many of the exceptions.

2.11 Composition

So far we have looked at the elements of a programming language—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, we know how to multiply integers and we know how to output values; it turns out we can do both at the same time:

```
std::cout << 17 * 3;
```

Actually, I shouldn't say "at the same time," since in reality the multiplication has to happen before the output, but the point is that any expression, involving numbers, characters, and variables, can be used inside an output statement. We've already seen one example:

```
std::cout << hour*60 + minute << std::endl;
```

You can also put arbitrary expressions on the right-hand side of an assignment statement:

```
int percentage;
percentage = (hitpoints * 100) / 50;
```

This ability may not seem so impressive now, but we will see other examples where composition makes it possible to express complex computations neatly and concisely.

WARNING: There are limits on where you can use certain expressions; most notably, the left-hand side of an assignment statement has to be a *variable* name, not an expression. That's because the left side indicates the storage location where the result will go. Expressions do not represent storage locations, only values. So the following is illegal: `minute+1 = hour;`.

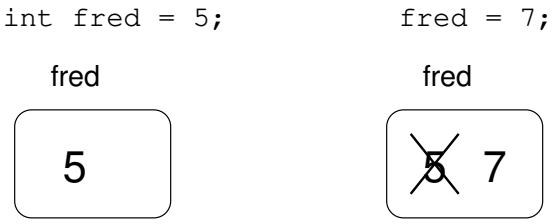
2.12 Multiple assignment

I haven't said much about it, but it is legal in C++ to make more than one assignment to the same variable. The effect of the second assignment is to replace the old value of the variable with a new value.

```
int fred = 5;
cout << fred;
fred = 7;
cout << fred;
```

The output of this program is `57`, because the first time we print `fred` his value is 5, and the second time his value is 7.

This kind of **multiple assignment** is the reason I described variables as a *container* for values. When you assign a value to a variable, you change the contents of the container, as shown in the figure:



When there are multiple assignments to a variable, it is especially important to distinguish between an assignment statement and a statement of equality. Because C++ uses the `=` symbol for assignment, it is tempting to interpret a statement like `a = b` as a statement of equality. It is not!

First of all, equality is commutative, and assignment is not. For example, in mathematics if $a = 7$ then $7 = a$. But in C++ the statement `a = 7;` is legal, and `7 = a;` is not.

Furthermore, in mathematics, a statement of equality is true for all time. If $a = b$ now, then a will always equal b. In C++, an assignment statement can make two variables equal, but they don't have to stay that way!

```
int a = 5;
int b = a;     // a and b are now equal
a = 3;         // a and b are no longer equal
```

The third line changes the value of `a` but it does not change the value of `b`, and so they are no longer equal. In many programming languages an alternate symbol is used for assignment, such as `<-` or `:=`, in order to avoid confusion.

Although multiple assignment is frequently useful, you should use it with caution. If the values of variables are changing constantly in different parts of the program, it can make the code difficult to read and debug.

2.13 Exercises

FIXME get a value print something cool and gamelike

2.13.1 Hello World

You have seen hello world a few times by now. Can you recreate it in this Parson Problem? `https://lpatacch.github.io/thinkCPPGamesEx/HelloParsonsPractice.html`

2.13.2 Modulus Practice

If you have problems understanding how the modulus operator works, try this small web application. `https://lpatacch.github.io/thinkCPPGamesEx/ModPractice.html`

2.13.3 Validation

Computers have to deal with numbers a lot. There are messages that are sent to and from the internet. In order to make sure the numbers were entered or received properly, programs usually run the values through a "validator". This makes sure that the number that the program has makes sense. One method to do validation is called a "check digit". This is an extra number put at the end of an ID number to see if the number could be valid. These digits were originally made to check if a person typed in an ID correctly, but has been used by programs to see if a data file or message was corrupted. The check digit method we are going to use has to do with UPC's. UPC numbers are IDs that are connected to products. It is a way to uniquely identify an item that is sold.

Your friend has a website where people can list the games that they are willing to let other people borrow or purchase. He originally let people type in the UPC numbers, but a few of them do not seem to be correct. He has asked you to make a UPC validator program so he does not have to look up as many numbers.

You found out the following information. UPC's have 12 digits. The first 11 digits are the product number and the number to the far right is the "check" digit. To calculate the digit, it does the following:

1. Take the odd number digits (first, third, fifth, etc.), add them together and multiply them by three.
2. Take the even number digits and add them together
3. Add those two numbers together
4. Mod that number by 10. (Use the remainder operator)
5. Take 10 and subtract the number you have and mod it by 10.

The number that results is the check digit.

Here is an example: A game (Teenage Mutant Ninja Turtles: The Cowabunga Collection - PlayStation 4) has the UPC number of: 083717203469

1. The odd digits are: 0+3+1+2+3+6. The sum is 15. 3*15 is 45
2. The even digits are: 8+7+7+0+4. (We do not use the check digit in the calculation). The sum is 26.
3. 45+26 = 71
4. 71 % 10 = 1
5. (10-1) % 10 = 9

9 was the digit that was the last number, so it IS a valid UPC number.

For your program, the first version will ask for each digit of the UPC and put it in a different variable. (For example, the first number can be placed in

an int variable named dig1, the second in dig2). Using those variables, create the code to do the math described above and have it print the check digit you calculated. Try putting the example UPC and see if it also comes up with the number 9. Then, try a UPC number that you have around the house. Did it work?

For an extra challenge, have the user enter the number as an integer (instead of 12 integers) and from that number calculate the digits and put the results in your variables. The program can then use the equation you have already made. HINT: You can use division and modulus operators to get each digit.

This program can be made even better if you could tell the user if the value that was entered as the check digit was correct. This type of branching code will be introduced in the next chapter (3).

2.14 Glossary

variable: A named storage location for values. All variables have a type, which determines which values it can store.

value: A letter, or number, or other thing that can be stored in a variable.

type: A set of values. The types we have seen are integers (`int` in C++) and characters (`char` in C++).

keyword: A reserved word that is used by the compiler to parse programs. Examples we have seen include `int`, `void` and `endl`.

statement: A line of code that represents a command or action. So far, the statements we have seen are declarations, assignments, and output statements.

declaration: A statement that creates a new variable and determines its type.

assignment: A statement that assigns a value to a variable.

expression: A combination of variables, operators and values that represents a single result value. Expressions also have types, as determined by their operators and operands.

operator: A special symbol that represents a simple computation like addition or multiplication.

modulus: An operator that works on integers and yields the remainder when one number is divided by another. In C++ it is denoted with a percent sign (`%`).

operand: One of the values on which an operator operates.

precedence: The order in which operations are evaluated.

composition: The ability to combine simple expressions and statements into compound statements and expressions in order to represent complex computations concisely.

tab: A special character, written as `\t` in C++, that causes the cursor to move to the next tab stop on the current line.

Chapter 3

Conditionals

3.1 Floating-point

In the last chapter we had some problems dealing with numbers that were not integers. We worked around the problem by measuring percentages instead of fractions, but a more general solution is to use floating-point numbers, which can represent fractions as well as integers. In C++, there are two floating-point types, called `float` and `double`. In this book we will use `double`s exclusively.

You can create floating-point variables and assign values to them using the same syntax we used for the other types. For example:

```
double pi;
pi = 3.14159;
```

Remember, it is also legal to declare a variable and assign a value to it at the same time:

```
int x = 1;
string empty = "";
double pi = 3.14159;
```

In fact, this syntax is quite common. A combined declaration and assignment is sometimes called an **initialization**.

(FYI, if you are using C++20, you can use the mathematical constant of PI, like the following:

```
double mypi = std::numbers::pi;
```

There are lots of constants available. We will be talking more about them when we talk about the Math library in Section 5.1.)

Although floating-point numbers are useful, they are often a source of confusion because there seems to be an overlap between integers and floating-point numbers. For example, if you have the value `1`, is that an integer, a floating-point number, or both?

Strictly speaking, C++ distinguishes the integer value `1` from the floating-point value `1.0`, even though they seem to be the same number. They belong to

different types, and strictly speaking, you are not allowed to make assignments between types. For example, the following is illegal

```
int checkMe = 1.1;  // WARNING!
```

because the variable on the left is an `int` and the value on the right is a `double`. If you are using the legacy initialization (see above), the compiler will give you a warning and let you know that checkMe will be set to 1, and not 1.1. But, your code will continue to run. If you want an error to happen instead, you can use the braced initialization:

```
int checkMe{1.1};   // ERROR!
```

Instead of giving a warning, the compiler will create an an error, complaining "type 'double' cannot be narrowed to 'int' in initializer list".

But it is easy to forget this "no assignment between types" rule, especially because there are places where C++ automatically converts from one type to another. For example,

```
double y = 1;
```

should technically not be legal, but C++ allows it by converting the `int` to a `double` automatically. This leniency is convenient, but it can cause problems; for example:

```
double y = 1 / 3;
```

You might expect the variable `y` to be given the value `0.333333`, which is a legal floating-point value, but in fact it will get the value `0.0`. The reason is that the expression on the right appears to be the ratio of two integers, so C++ does *integer* division, which yields the integer value `0`. Converted to floating-point, the result is `0.0`.

One way to solve this problem (once you figure out what it is) is to make the right-hand side a floating-point expression:

```
double y = 1.0 / 3.0;
```

This sets `y` to `0.333333`, as expected.

All the operations we have seen—addition, subtraction, multiplication, and division—work on floating-point values, although you might be interested to know that the underlying mechanism is completely different. In fact, most processors have special hardware just for performing floating-point operations.

3.1.1 Converting from `double` to `int`

As I mentioned, C++ converts `int`s to `double`s automatically if necessary, because no information is lost in the translation. On the other hand, going from a `double` to an `int` requires rounding off. C++ doesn't perform this operation automatically, in order to make sure that you, as the programmer, are aware of the loss of the fractional part of the number.

The simplest way to convert a floating-point value to an integer is to use a **typecast**. Typecasting is so called because it allows you to take a value that

belongs to one type and "cast" it into another type (in the sense of molding or reforming, not throwing).

The syntax for typecasting is different than code we have seen so far. For example:

```
double pi = 3.14159;
int x = static_cast< int >(pi);
```

The `static_cast< int >` casting operator returns an integer, so `x` gets the value 3. Converting to an integer always rounds down, even if the fraction part is 0.99999999.

For every type in C++, there is a corresponding function that typecasts its argument to the appropriate type. The format follows the following pattern:

```
T newVariable = static_cast< T >(oldVariable);
```

To use this, change T to be the type you are converting into. Put the variable name you are converting where "oldVariable" is. Last, change "newVariable" to the name of the variable you want the new value to be placed.

Here are some examples so you can try them out: `https://replit.com/@lpatacch/DivisionDoubleIntExamples#divdoubles.cpp`

Older versions of C++ (and C) used to do casting like the following:

```
double pi = 3.14159;
int x = (int)pi;      // don't do, old style
```

If you do this type of casting in newer versions of C++, you will get warnings saying you are using an older style of casting.

3.2 Conditional execution

In order to write useful programs, we almost always need the ability to check certain conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the `if` statement:

```
if (x > 0) {
  std::cout << "x is positive" << std::endl;
}
```

That code is doing the following:

```
IF x is greater than 0 THEN
    Output "x is positive" and then go to the next line
ENDIF
```

That also means that if x is zero or less, it will not output anything and just go to the following line of code.

The expression in parentheses is called the condition. If it is true, then the statements in brackets get executed. If the condition is not true, nothing happens.

The condition can contain any of the `comparison operators`:

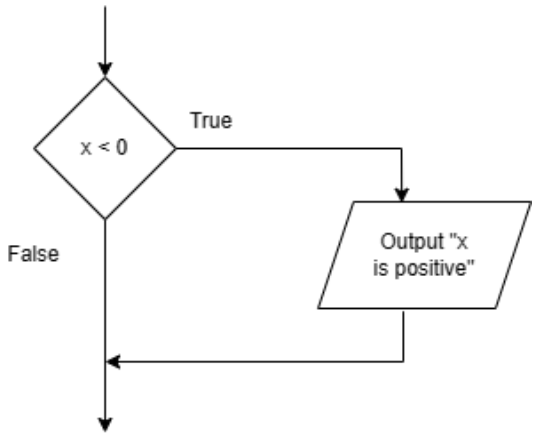


Figure 3.1: If flowchart

```
    x == y          // x equals y
    x != y          // x is not equal to y
    x > y           // x is greater than y
    x < y           // x is less than y
    x >= y          // x is greater than or equal to y
    x <= y          // x is less than or equal to y
```

Although these operations are probably familiar to you, the syntax C++ uses is a little different from mathematical symbols like $=$, $\neq$ and $\leq$. A common error is to use a single `=` instead of a double `==`. Remember that `=` is the assignment operator, and `==` is a comparison operator. Also, there is no such thing as `=<` or `=>`.

Unfortunately, the default behavior in many compilers is to not display warnings when you use one equal instead of two in an if statements. (Check out Chapter A.3.1 if you want to know how to turn on the warnings.) Be sure to be careful when you are using equal signs and if statements.

The two sides of a condition operator have to be the same type. You can only compare `ints` to `ints` and `doubles` to `doubles`.

All the examples of if statements that we have seen so far is using curly braces. But, if statements do work without the curly braces. For example, the following if statement would work:

```
int x = -1;
if (x < 0)
    std::cout << "negative";
```

The above if statement would print "negative". It does lead to some confusion though. If you have the following if statement, it may not print what you expect.

```
int x = 1;
if (x < 0)
```

```
    std::cout << "negative. ";
    std::cout << "this is below zero"; // WRONG!
```

This would print "this is below zero" even though the second line is indented. The second line is not part of the if statement. It was finished after the line that printed "negative".

```
int x = 1;
if (x < 0)
{
    std::cout << "negative. ";
    std::cout << "this is below zero"; // Fixed!
}
```

By adding the curly braces, the second std::cout is now part of the if statement. The curly braces and the cout statements inside of them is a **code block**. A code block (also called a compound statement) is one or statements surrounded by curly braces. They act as a logical unit. An if statement can run one statement conditionally, or one code block. This allows you to do one or more things if a condition is true.

Even though it is possible for it to work without curly braces, it is suggested that you always use them. Most programmers will expect them to be there, and it is easy for bugs to be introduced because of the confusion.

3.3 Alternative execution

A second form of conditional execution is alternative execution, in which there are two possibilities, and the condition determines which one gets executed. This is sometimes called the `if else` statement. The syntax looks like:

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

(If you do not remember what the % sign is doing, check out the Modulus section in Chapter 2.8.3) That code is doing the following:

```
IF x mod 2 is equal to 0 THEN
    Output "x is even" and then go to the next line
ELSE
    Output "x is odd" and then go to the next line
ENDIF
```

If the remainder when `x` is divided by 2 is zero, then we know that `x` is even, and this code displays a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed.

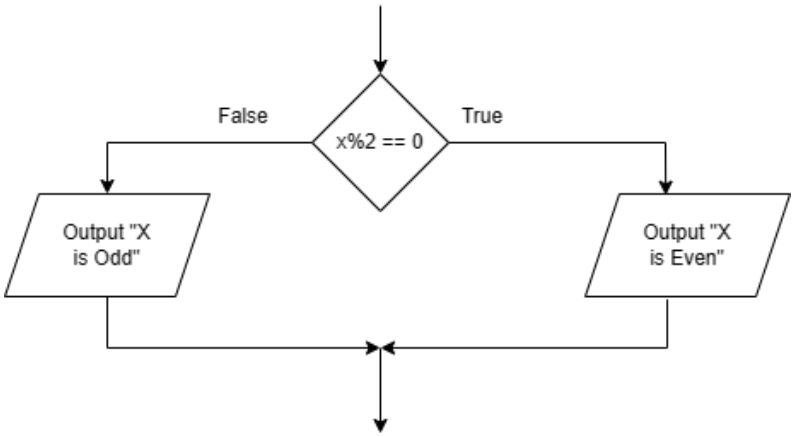


Figure 3.2: If else flowchart

3.4 Chained conditionals

Sometimes you want to check for a number of related conditions and choose one of several actions. One way to do this is by **chaining** a series of `if`s and `else`s:

```
if (x > 0) {
  std::cout << "x is positive" << std::endl;
} else if (x < 0) {
  std::cout << "x is negative" << std::endl;
} else {
  std::cout << "x is zero" << std::endl;
}
```

These chains can be as long as you want, although they can be difficult to read if they get out of hand. One way to make them easier to read is to use standard indentation, as demonstrated in these examples. If you keep all the statements and curly-braces lined up, you are less likely to make syntax errors and you can find them more quickly if you do.

3.5 Nested conditionals

In addition to chaining, you can also nest one conditional within another. We could have written the previous example as:

```
if (x == 0) {
  std::cout << "x is zero" << std::endl;
} else {
  if (x > 0) {
    std::cout << "x is positive" << std::endl;
```

```
    } else {
      std::cout << "x is negative" << std::endl;
    }
  }
```

There is now an outer conditional that contains two branches. The first branch contains a simple output statement, but the second branch contains another `if` statement, which has two branches of its own. Fortunately, those two branches are both output statements, although they could have been conditional statements as well.

```
IF x is equal to zero THEN
    Output "X is zero"
ELSE
    IF x is greater than zero THEN
        Output "x is positive"
    ELSE
        Output "x is negative"
    ENDIF
ENDIF
```

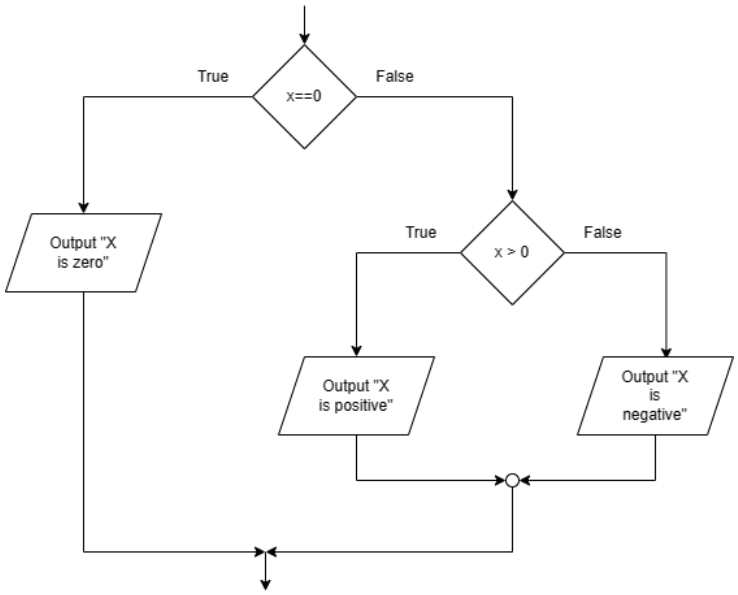


Figure 3.3: Nested if/else statement

Notice again that indentation helps make the structure apparent, but nevertheless, nested conditionals get difficult to read very quickly. In general, it is a good idea to avoid them when you can.

On the other hand, this kind of **nested structure** is common, and we will see it again, so you better get used to it.

3.6 Logical operators

Nested if statements can get tricky to look at, so there are a couple of different ways to make it look less complex. The first we will look at is "logical operators". There are three **logical operators** in C++: AND, OR and NOT, which are denoted by the symbols `&&`, `||` and `!`. The semantics (meaning) of these operators is similar to their meaning in English. For example `x > 0 && x < 10` is true only if `x` is greater than zero AND less than 10.

`n%2==0 || n%3 == 0` is true if *either* of the conditions is true, that is, if the number is divisible by 2 OR the number is divisible by 3.

Finally, the NOT operator has the effect of negating or inverting a bool expression, so `!(n%2 == 0)` is true if `(n%2 == 0)` is false; that is, if the number is odd.

Logical operators often provide a way to simplify nested conditional statements. For example, how would you write the following code using a single conditional?

```
if (x > 0) {
  if (x < 10) {
    std::cout << "x is a positive single digit.";
    std::cout << std::endl;
  }
}
```

But, you need to be careful when you are making a comparison. What makes sense to a human does not make sense to a computer. If you want to see if x is less than y and z, the following are what many students think is a solution (it is not.)

```
if ((x < y) && (z))   // WRONG!
{
}
```

Although it is comparing x to y, it is not comparing z. It is just checking if Z is zero or not. Instead, this is what the if statement should be:

```
if ((x < y) && (x < z))  // this is the way!
```

3.7 Short circuiting the logic

If you are not exactly sure how AND works, here is a table that explains the specifics. If you have one comparison (first) and it is ANDed to the second comparison (second), the third column is the result.

first	second	first && second
true	true	true
true	false	false
false	true	false
false	false	false

Table 3.1: Truth table for AND

If you take a look at the table, you may notice that the only way the final result is true is that both comparisons need to be true. Because of that, when C++ code is running, it has a shortcut. If it notices the first item of an AND comparison is false, it does not bother checking the second comparison. If you happen to know the comparison that is false more often, you can have your code run faster by having it be the first of the comparisons.

Next is how the OR comparison works. If you look at the OR truth table, you will notice that you only need one item to be true for the result to be true. Just like the AND comparison, there is a shortcut. As soon as it sees a "true",

first	second	first OR second
true	true	true
true	false	true
false	true	true
false	false	false

Table 3.2: Truth table for OR

the program goes straight to the resulting code. If you know the comparison that happens to be true more often, you can have your code run quicker by having it be the first comparison.

3.8 Conditional (ternary) Operator

There is another operator that people can use instead of the if/else statements. It is a more compact way to check a condition, and then set the appropriate value. For example:

```
y = ( x < 0 ) ? -1 : 1;
```

That line is equivalent to:

```
if ( x < 0 ) {
  y = -1;
}
else {
  y = 1;
}
```

It can also be used to decide what to print:

```
std::cout << ((w < 0)? "negative":"positive or zero");
```

The above code checks what w is set to. If it is less than 0, it will print "negative". Otherwise, it prints "positive or zero".

It can also be used to help use the appropriate grammar:

```
int thing = 2;
std::cout << "There " << (thing == 1 ? "is " : "are ");
std::cout << thing << " toy" << (thing == 1 ? "" : "s");
std::cout << ".\n";
```

If there is one "thing", it says "There is 1 toy.". If there are other numbers, like 2, it says, "There are 2 toys."

Here are many examples of conditional code: `https://replit.com/@lpatacch/conditional#conditional.cpp`

3.9 `switch` statement

Another method that we can use instead of a complex if statement is `switch` statements. A `switch` statement is an alternative to a chained conditional that is syntactically prettier and often more efficient. It looks like this:

```
switch (symbol) {
 case '+':
   z = x + y;
   break;
 case '*':
   z = x * y;
   break;
 default:
   std::cout << "I only know how to perform addition";
   std::cout <<" and multiplication" << std::endl;
   break;
}
```

This `switch` statement is equivalent to the following chained conditional:

```
if (symbol == '+') {
  z = x + y;
} else if (symbol == '*') {
  z = x * y;
} else {
  std::cout << "I only know how to perform addition";
  std::cout << "and multiplication" << std::endl;
}
```

In general it is good style to include a `default` case in every `switch` statement, to handle errors or unexpected values.

The `break` statements are necessary in each branch in a `switch` statement because otherwise the flow of execution "falls through" to the next case. Without the `break` statements, the symbol `+` would make the program perform addition, and then perform multiplication, and then print the error message. Occasionally this feature is useful, but most of the time it is a source of errors when people forget the `break` statements.

3.9.1 Fallthrough

Here is an example of code using switch that is using the fall through feature. This method of stacking cases is common, so no error message is created if the compiler sees this:

```
/* In this code, the user types a character to decide
   which way to go. If they type 'L' or 'l', they will
   go left. If they type 'R' or 'r', they will go
   right. If they type anything else, there is an
   error message
 */

switch (direction) {
  case 'L':
  case 'l':
      std::cout << "Going left" << std::endl;
      break;
  case 'R':
  case 'r':
      std::cout << "Going Right" << std::endl;
      break;
  default:
      std::cout << "Please choose R or L" << std::endl;
}
```

The above code allows the programmer to have the same result if a person typed either R or r.

The next example is something more uncommon. It is using the fallthrough feature without stacking cases. Each case will be printing something connected to its case, and then an printing the items for the lower cases.

```
std::cin >> dayOfChristmas;
std::cout << "On the " << dayOfChristmas;
std::cout << " day of Christmas";
std::cout << " my true love gave to me";
switch (dayOfChristmas) {
  case 3:
      std::cout << " 3 french hens,";
  case 2:
      std::cout << " 2 calling birds, and";
  case 1:
      std::cout << " a partridge in a pear tree";
```

```
        std::cout << std::endl;
}
```

There is more of a chance that this type of code is a mistake, so the C++ 2017 standard (and the later standards) now has a warning if you write a switch statement with this feature. If you want to avoid this warning, you can add the following:

```
[[fallthrough]];
```

where a "break" statement would normally be. So, if you take the previous example, it would look like this:

```
std::cout << "On the " << dayOfChristmas;
std::cout << " day of Christmas";
std::cout << " my true love gave to me";
switch (dayOfChristmas) {
  case 3:
      std::cout << " 3 french hens,";
      [[fallthrough]];
  case 2:
      std::cout << " 2 calling birds, and";
      [[fallthrough]];
  case 1:
      std::cout << " a partridge in a pear tree";
      std::cout << std::endl;
  }
```

Remember, this warning removal feature is ONLY available if you are using the 2017 standard or later. (For information on how to change the standard on Reddit, check out Section A.3.1.)

Here are some examples of switch statements: `https://replit.com/@lpatacch/SwitchStatements#switchstatement.cpp`

3.10 Pseudocode

You may have noticed several examples of code logic written in this chapter. Those are examples of Pseudocode. Pseudocode is a way to explain your code in whatever language you speak. This can allow you to work through the logic before having to deal with the syntax of the coding language. I have found over the years that I tend to work out my logic first in pseudocode. Languages come and go, but logic is here to stay.

In order to make things easier to follow, there are several rules that people use when writing pseudocode. First, keywords are capitalized. Some keywords are IF, THEN, ELSE and ENDIF. The ENDIF keyword is used to show when an if statement is over. If there is no else statement, the ENDIF is placed after the statement is finished.

```
IF x is greater than 0 THEN
```

```
   Output "x is positive" and then go to the next line
ENDIF
```

Another thing that should be noticed is the indentation. Items inside an IF or ELSE should be indented to make it easier to see when the statement is over.

One important point is to not make your pseudocode too technical. You are not trying to write the code itself, just a plan on how the code could be written. The pseudocode should be more understandable for those who do not know code.

3.11 Flowcharts

Just like pseudocode, I have put a few figures of flowcharts around the if and if else statements. Flowcharts are another way to show how code is running. Instead of using just words, flowcharts add shapes and arrows to help people understand the logic of the code. There are some specifics of how flowcharts generally work. Each shape has a certain meaning.

3.11.1 Oval

The oval shape is used for the beginning and ending of the program. Inside the oval is the word "start" or the word "end". Most full flowcharts will have two ovals.

There is an exception to this. If the flowchart is a small part of a larger algorithm, the ovals could be replaced with small circles. These small circles have information about where they connect to the larger drawing.

3.11.2 Diamond

A diamond is used for selection statements (if statements). It shows where a decision is made. Inside the diamond shape is the condition that is being checked. The flowlines off the decision should have a label to explain if they are the path for true or false.

3.11.3 Rectangle

Rectangle shapes are used for any processing statements (math). The equations are placed inside the rectangle.

3.11.4 Parallelogram

If there is any input or output, these statements are placed in a parallelogram. The item that will be printed or requested is placed in the shape.

3.11.5 How they connect

When you are putting together a flowchart, you will be connecting the different sections. Each section is a component and it should not jump from one section to another. The figure at 3.4 shows the locations where a if statement can connect to another section. You can have if statements inside if statements (as

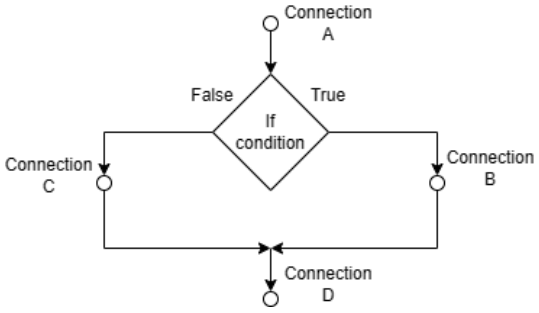


Figure 3.4: The spots where items can be placed with an if statement

we saw when we reviewed "Nested conditionals"). But, you should never link from inside an if statement to inside another. It does not matter if the inside of both if statements are doing the same thing. For example, the chart at Figure 3.5 is a incorrect flowchart:

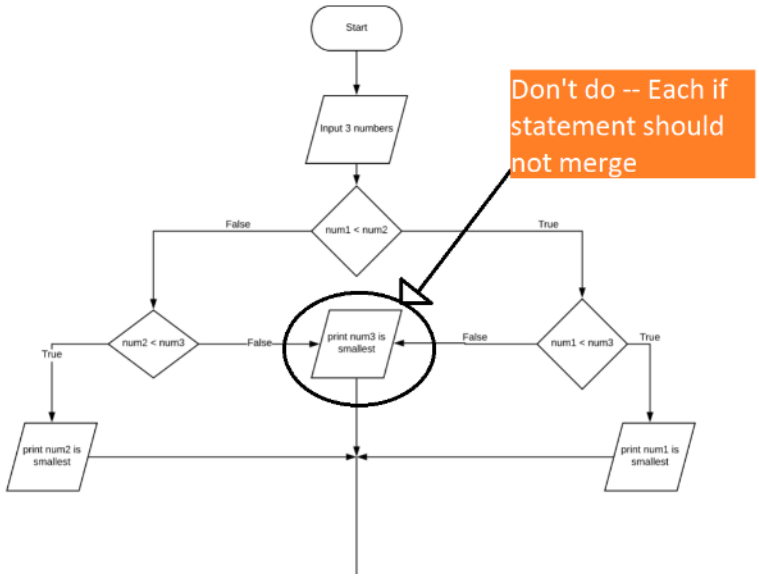


Figure 3.5: Incorrect If Flowchart

There a couple of reasons why this flowchart is incorrect. First, that is not how the code will be flowing. It will not jump to the other if statement, it is staying in the original if statement. Second, each part of the flowchart tends to be a line of code. If you created code following the incorrect flowchart, you would expect there to only be one line with that print statement. But, the code would actually have two print statements. For both those reasons, the chart at Figure 3.6 is what the flowchart should look like:

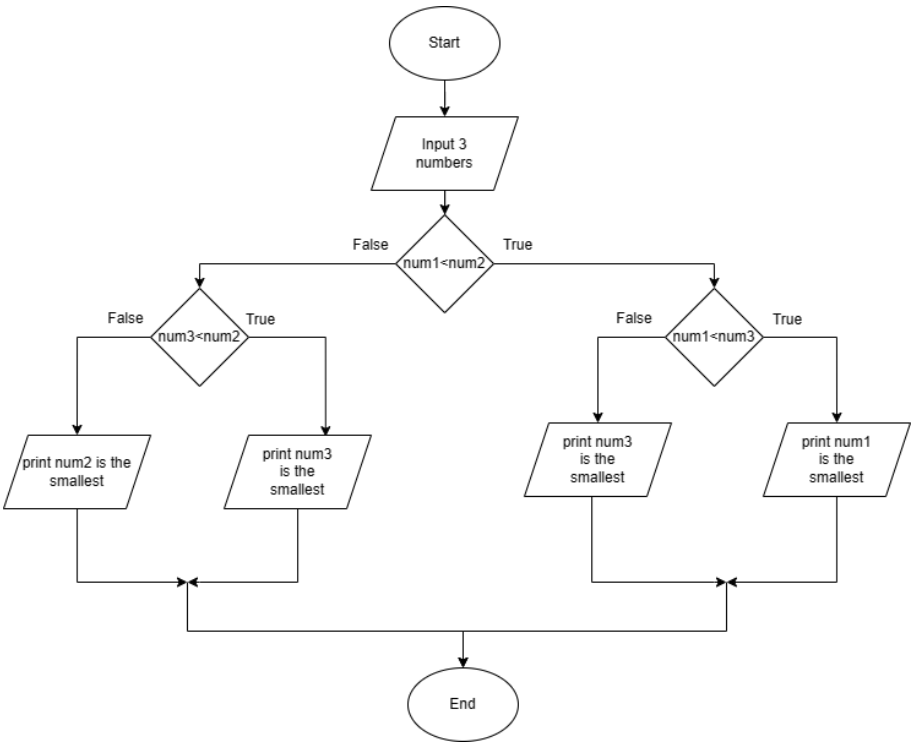


Figure 3.6: What the if statement should look like

3.12 Testing Techniques

Now that we have code with if statements, we can't just run the code once to make sure it works. To be truly sure that the entire program works, it is important to test each possible path of the program. For example, if we have a program that checks to see if a number is less than zero, we should check a negative number, a positive number, and zero to make sure the program works.

```
int main() {
  int x = 1; // Later, set to 0, and then -1.
  if (x == 0) {
```

```
    std::cout << "x is zero" << std::endl;
  } else {
    if (x > 0) {
      std::cout << "x is positive" << std::endl;
    } else {
      std::cout << "x is negative" << std::endl;
    }
  }
}
```

If I put those values in, I should see the x is zero for zero, x is positive for the 1, and x is negative for the -1. If one of those does not print the correct item, I would know there is a bug, and would know where to check. For example, if the "x is zero" did not print, I would be looking at the first if statement condition. Check the paths in Figure 3.7.

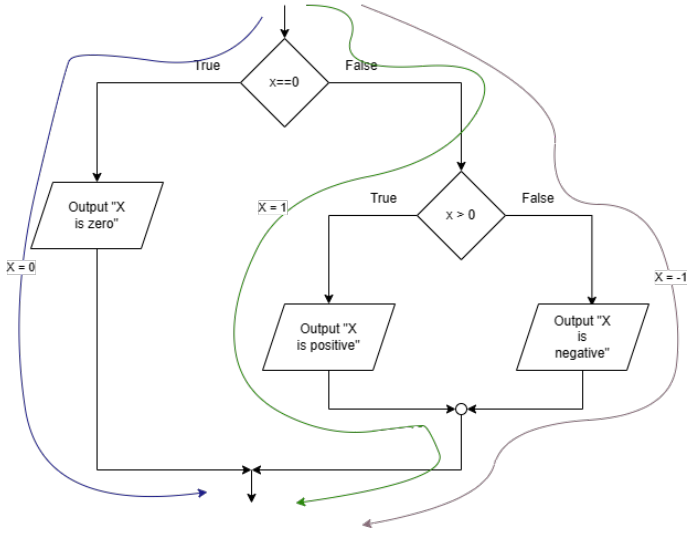


Figure 3.7: Nested if else flowchart with pathlines

It is good to test each section of code before adding more code to the program. This will allow you to have a better idea where the bug could be.

3.13 Exercises

3.13.1 Rock Paper Scissors

FIXME

3.13.2 Magic 8 Ball

FIXME

3.13.3 Decision Maker

FIXME This assignment will be based on the assignment below. Students are asked to create a formula that makes a judgement But, should they make a judgement without talking to others?

3.14 Glossary

floating-point: A type of variable (or value) that can contain fractions as well as integers. There are a few floating-point types in C++; the one we use in this book is `double`.

conditional: A block of statements that may or may not be executed depending on some condition.

chaining: A way of joining several conditional statements in sequence.

nesting: Putting a conditional statement inside one or both branches of another conditional statement.

switch statement: A different way to write conditional code. The switch selects the case to run depending on the variable it is checking.

pseudocode: A way of designing programs by writing rough drafts in a combination of keywords and English.

flowchart: A way of designing programs with shapes and arrows used to show the flow of the program.

code block: One or more statements surrounded by curly braces. This group of statements can act as a logical unit for conditionals and loops.

Chapter 4

Iteration

One of the things computers are often used for is the automation of repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

The type of repetition we will be looking at now is called **iteration**, and C++ provides several language features that make it easier to write iterative programs.

The three features we are going to look at are the `while` statement, the `do-while` statement, and the `for` statement. There are another features, like the for-each and ranged based for loop, that we will cover in another chapter.

4.1 The `while` statement

Using a `while` statement, we can write a countdown. What we would want to do is start with a certain number, then have the number go down (and print it out) while it is above zero. Then, print Blastoff!.

Here is pseudocode for that:

```
Set n to 3
WHILE n is more than zero
    print n
    subtract 1 from n
ENDWHILE
print "Blastoff!"
```

Next, we can convert this to C++ code. It is not that much different than the pseudocode. Just like if statements, while statements use conditions to know when to continue and when to end. As long as the condition after the `while` is true, the loop will continue to run. Once it is false, the loop will stop.

```
n = 3;
while (n > 0) {
  std::cout << n << std::endl;
```

```
    n = n-1;
  }
  std::cout << "Blastoff!" << std::endl;
```

You can almost read a `while` statement as if it were English. What this means is, "While `n` is greater than zero, continue displaying the value of `n` and then reducing the value of `n` by 1. When you get to zero, output the word 'Blastoff!"'

This can be drawn as the following flowchart. It is a little easier to understand how the code is looping by looking at the flowchart. Notice how after the n has one subtracted from it, the condition is checked again. That is the spot where it decides if the program will continue with the loop or end the loop.

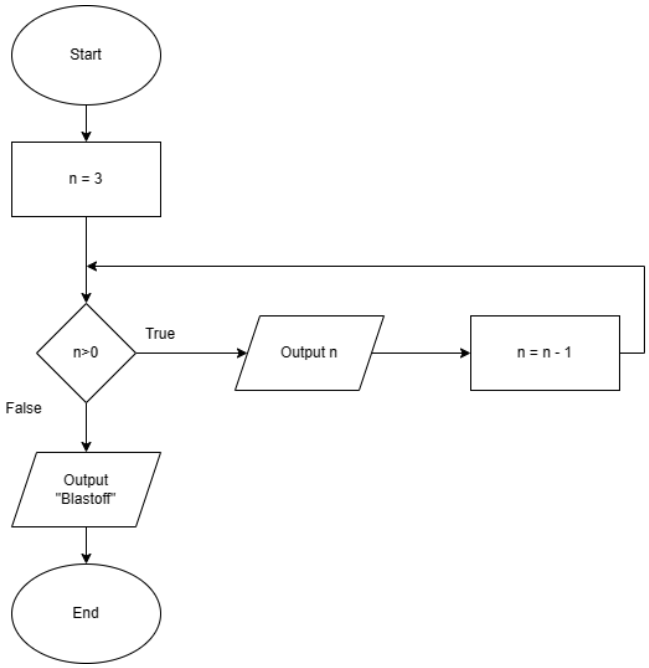


Figure 4.1: Flowchart for the countdown program

More formally, the flow of execution for a `while` statement is as follows:

1. Evaluate the condition in parentheses, yielding `true` or `false`.
2. If the condition is false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, execute each of the statements between the curly-braces, and then go back to step 1.

This type of flow is called a **loop** because the third step loops back around to the top. Notice that if the condition is false the first time through the loop,

the statements inside the loop are never executed. The statements inside the loop are called the **body** of the loop.

The body of the loop should change the value of one or more variables so that, eventually, the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop.

In the case of `countdown`, we can prove that the loop will terminate because we know that the value of `n` is finite, and we can see that the value of `n` gets smaller each time through the loop (each **iteration**), so eventually we have to get to zero. In other cases it is not so easy to tell:

```
while (n != 1) {
  std::cout << n << std::endl;
  if (n%2 == 0) {           // n is even
    n = n / 2;
  } else {                  // n is odd
    n = n*3 + 1;
  }
}
```

To run the countdown code or the flipping around code, see While loop examples The condition for this loop is `n != 1`, so the loop will continue until `n` is 1, which will make the condition false.

At each iteration, the program outputs the value of `n` and then checks whether it is even or odd. If it is even, the value of `n` is divided by two. If it is odd, the value is replaced by $3n + 1$. For example, if the starting value (the argument passed to `sequence`) is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program will terminate. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, then the value of `n` will be even every time through the loop, until we get to 1. The previous example ends with such a sequence, starting with 16.

Particular values aside, the interesting question is whether we can prove that this program terminates for *all* values of n. So far, no one has been able to prove it *or* disprove it!

This book has a while loop simulator in order to help you understand while loops, `https://lpatacch.github.io/thinkCPPGamesEx/WhilePractice.html`. With this, you can change the initial value, the condition it is checking (<, >, ==, !=, >=, <=), the value checked, and how the variable will change. Use this as many times as you wish to understand how while loops work.

4.2 New Operators

Before we get more into loops, I want to cover some more math operators that we have not seen yet. In the countdown program, we had the counter change values with this equation:

```
n = n-1;
```

That equation took whatever was in n, subtracted 1, and placed it back in n. That behavior is very typical with loops. One variable is changed until it reaches the end value. It is so common that there is a short cut for this behavior. The above equation can be rewritten to this:

```
n -= 1;
```

This equation means the same as above, the value n has 1 subtracted from it, and then the result is put in n. Similar equations are available for many other math operators:

Math Operator	What it does	sample equation
+=	addition	`n += 3;`
-=	subtraction	`n -= 5;`
*=	multiplication	`n *= 4;`
\=	division	`n \= 2;`
%=	modulus	`n %= 7;`

Table 4.1: New operators

4.3 Increment and decrement operators

Incrementing and decrementing by one are such common operations that C++ provides special operators for them. The `++` operator adds one to the current value of an `int`, `char` or `double`, and `--` subtracts one. Neither operator works on `string`s, and neither *should* be used on `bool`s.

The equation we modified in the last section:

```
n -= 1;
```

If you notice, the n was decremented by one. So, we can use the even shorter form here:

```
--n;
```

This subtracts 1 from n. This makes a very small line of code. All you need is the increment or the decrement operator, and the variable you will be changing.

You may have noticed that on some lines I put the ++ before the variable, and some after the variable. Both are legal. When it is on it's own line, it seems like it is doing the same thing. But, it is doing something slightly different.

The ++ before the variable is called a pre-incrementor. It means that it will be doing the adding before anything else in the line. The ++ after the variable is called a post-incrementor. It means that it will be doing the adding after most of the things on the line.

Technically, it is legal to increment a variable and use it in an expression at the same time. For example, you might see something like:

```
std::cout << i++ << std::endl;
```

Looking at this, it is not clear whether the increment will take effect before or after the value is displayed. Because expressions like this tend to be confusing, I would discourage you from using them. In fact, to discourage you even more, I'm not going to tell you what the result is, other than letting you know it is a post-incrementor. If you really want to know, you can try it.

Using the increment operators, we can calculate the average weight of a person's inventory:

```
int num = 0;
int weight = 0;
int total = 0;
std::cout << "Enter the weight of the item.\n";
std::cout << "When done, enter -1\n";
std::cin >> weight;
while (weight >= 0) {
  //if it is a valid weight, add it to the total
  total += weight;

  //add one to the number of items
  ++num;

  //ask for the next weight
  std::cout << "Enter the weight of the item.\n";
  std::cout << "When done, enter -1\n";
  std::cin >> weight;
}

if (num != 0)
{
  std::cout << "The average is " << total/num <<".\n";
}
else {
  std::cout << "No items to average.\n";
}
```

To run this code, try this Inventory Project.

It is a common error to write something like

```
num = num++;              // WRONG!!
```

Unfortunately, this is syntactically legal, so the compiler will not warn you. The effect of this statement is to leave the value of `num` unchanged. This is often a difficult bug to track down.

Remember, you can write `num = num +1;`, or you can write `num++;`, but you shouldn't mix them.

The ++ and – does not work on a complex expression either. For example,

```
( n + 7 )++;
```

would not compile. The ++ and – should be used only on a variable, not an expression.

4.4 Parts of a loop

The loops we have written so far have a number of elements in common. All of them start by initializing a variable; they have a test, or condition, that depends on that variable; and inside the loop they do something to that variable, like increment it. This can be described by the following pseudocode:

```
INITIALIZER;
while (CONDITION) {
  BODY
  MODIFICATION
}
```

Loops, in order to work properly, need to have all three parts. If there is no modification section, the loop would never be able to leave the loop. If the initialization is wrong (or missing), the loop may loop the wrong amount of times (or never loop). And last, if the condition is wrong, the loop will never end properly. Whenever you loop is not working properly, if you look at those three parts, you will find the problem.

4.5 Definite loop

There are two different ways that loops can usually be identified. The first type is one where you know how many times the loop will repeat. These type of loops tend to have a counter that is either counting down or up, and the counter is modified by a constant value each time through the loop. These loops are sometimes called definite loops, or counting loops. We have seen example with the countdown loop. The count down started at three and the counter went down by one each time through the loop. Here is another example:

```
int val = 7;
std::cout << "Here is a multiplication table for ";
std::cout << val <<"\n";

int x = 1;
```

```
while (x <= 10)
{
   std::cout << x << "\t" << x * val << "\n";
   x++;
}
```

To run this code, see the Definite loop project This loop has the condition as x¡=10. If you look in the loop, you will notice that x is being modified by 1 each time through the loop. Also, x is initialized to 1 when the loop starts. That means that the loop will repeat 10 times. Every time you run this code, it will repeat 10 times.

In case you are interested, the \t is one of the escape sequences that we mentioned in 2.1. This spaces out each column by a tab stop. There are better ways to format tables, but we will not cover those methods for a while.

4.6 Indefinite loop

There is a different type of loop where you will not know how many times the loop will run. The loop is checking a condition that will change, but it is different every time we run the loop. We showed an example in the weight of the inventory code. This loop may have not repeated at all, or it could have repeated a thousand times. Since you can not tell how many times it will repeat, it can be called an "indefinite loop". The number of repeats depended on what the user was entering. That code was waiting on a particular value (a negative number). When a loop is waiting for a particular value, that value is called a flag, or a sentinel, which is why this loop may be called a sentinel loop.

Just like if statements do not require curly braces to run, while loops don't either. A while loop without curly braces will only repeat the line right after the while condition. That is because the definition of a while loop is:

```
while (condition)
   statement or code block
```

It is suggested that you always use code blocks with while loops to prevent confusion and bugs.

4.7 `do-while` statement

The second type of loop that C++ supports is called a do-while loop. It works very similarly to the while loop. It should be initialized before the loop starts, it should be modified inside the loop, and it has a condition it checks. The difference is where it checks the condition. The do-while loop checks the condition at the end of the loop, guaranteeing the loop always runs at least once. The while loop checks the condition at the beginning, which could mean the loop might not run at all. The general syntax looks like the following:

```
INITIALIZER;
 do {
  BODY
  MODIFICATION
} while (CONDITION);
```

For example, the multiplication table could be changed to a do-while:

```
int val = 7;
std::cout << "Here is a multiplication table for ";
std::cout << val <<"\n";

int x = 1;
do {
   std::cout << x << "\t" << x * val << "\n";
   x++;
} while (x <= 10);
```

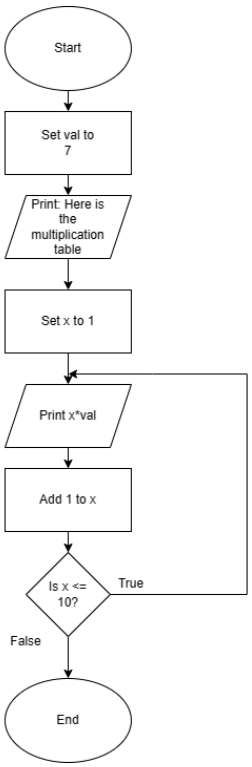


Figure 4.2: Flowchart for the do-while multiplication table

By now, you may be wondering, "Why should I use this approach instead of a while loop?" The main advantage is that the code within the loop is guaranteed

to execute at least once. I often use this type of loop when obtaining input from the user. In many programs, we ask a question. Unfortunately, the user may not always provide a valid response. While we know that we need to ask the question once, we cannot be certain if we will need to repeat it if the user provides an incorrect response. For instance, let's consider an example where we ask the user whether they want to turn left or right. The only correct answer is 'l' or 'r'. [1]

```
char direction = '\0';
do {
   std::cout << "Do you want to go (l)eft or (r)ight?\n";
   std::cin >> direction;
   if ((direction != 'l')&&(direction != 'r'))
   {
      std::cout << "Please enter l or r\n";
   }
} while ((direction != 'l')&&(direction != 'r'));
```

If we decided to do this with a while, we would have needed to ask the question both outside and inside the loop. The do-while reduces the amount of times the same code is duplicated in the program.

To try either of the do-while examples, try the Do-while project.

4.8 for statement

We have had many examples of definite or counting loops. This type of loop is so common that there is an alternate loop statement, called `for`, that expresses it more concisely. The general syntax looks like this:

```
for (INITIALIZER; CONDITION; INCREMENTOR) {
  BODY
}
```

This statement is exactly equivalent to

```
INITIALIZER;
while (CONDITION) {
  BODY
  INCREMENTOR
}
```

except that it is more concise and, since it puts all the loop-related statements in one place, it is easier to read. For example:

```
int i;
for (i = 0; i < 4; i++) {
```

[1] Technically, we should have changed the answer to lowercase to allow for L and R, but I wanted this example to be simpler than that.

```
    cout << i << endl;
  }
```

is equivalent to

```
  int i = 0;
  while (i < 4) {
    cout << i << endl;
    i++;
  }
```

Although using for loops when incrementing is common, it can also be used when decrementing. Here is the countdown program using a for loop instead:

```
  for (n = 3; n > 0, n--) {
    std::cout << n << std::endl;
  }
  std::cout << "Blastoff!" << std::endl;
```

Also, the counter does not need to only change by one. You can use any math operation that you want. Here is an example of a for loop that modifies the counter by five every time through the loop.

```
std::cout << "Counting to 100 by 5's \n";
for (n = 0; n < 100, n += 5 ) {
    std::cout << n << " ";
  }
std::cout << std::endl
```

4.9 Break and Continue

FIXME add examples of how this works

4.10 Glossary

loop: A statement that executes repeatedly while a condition is true or until some condition is satisfied.

infinite loop: A loop whose condition is always true.

body: The statements inside the loop.

iteration: One pass through (execution of) the body of the loop, including the evaluation of the condition.

definite loop: A loop where you know how many times it will run before the loop starts.

indefinite loop: A loop where you do not know how many times it will run before the loop starts.

4.11 Game Loops

FIXME - put the game loop information here

Chapter 5

Function

5.1 Math functions

In mathematics, you have probably seen functions like sin and log, and you have learned to evaluate expressions like $\sin(\pi/2)$ and $\log(1/x)$. First, you evaluate the expression in parentheses, which is called the **argument** of the function. For example, $\pi/2$ is approximately 1.571, and $1/x$ is 0.1 (if x happens to be 10).

Then you can evaluate the function itself, either by looking it up in a table or by performing various computations. The sin of 1.571 is 1, and the log of 0.1 is -1 (assuming that log indicates the logarithm base 10).

This process can be applied repeatedly to evaluate more complicated expressions like $\log(1/\sin(\pi/2))$. First we evaluate the argument of the innermost function, then evaluate the function, and so on.

C++ provides a set of built-in functions that includes most of the mathematical operations you can think of. The math functions are invoked using a syntax that is similar to mathematical notation:

```
double log = std::log (17.0);
double angle = 1.5;
double height = std::sin (angle);
```

The first example sets `log` to the logarithm of 17, base e. There is also a function called `log10` that takes logarithms base 10.

The second example finds the sine of the value of the variable `angle`. C++ assumes that the values you use with `sin` and the other trigonometric functions (`cos`, `tan`) are in *radians*. To convert from degrees to radians, you can divide by 360 and multiply by 2π.

If you don't happen to know π to 15 digits, you can calculate it using the `acos` function. The arccosine (or inverse cosine) of -1 is π, because the cosine of π is -1.

```
double pi = std::acos(-1.0);
```

```
  double degrees = 90;
  double angle = degrees * 2 * pi / 360.0;
```

Before you can use any of the math functions, you have to include the cmath **header file**. Header files contain information the compiler needs about functions that are defined outside your program. For example, in the "Hello, world!" program we included a header file named `iostream` using an **include** statement:

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

`iostream` contains information about input and output (I/O) streams, including the object named `cout`. C++ has a powerful feature called namespaces, that allow you to write your own implementation of `cout`. But in most cases, we would need to use the standard implementation. To convey this to the compiler, we use the line

```
using namespace std;
```

To save yourself some typing, you can write `using namespace std;` whenever you use `iostream`. This allows a shorthand where you can skip typing the std:: in your code. For example, the hello world program could be shortened to the following:

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

// main: generate some simple output
int main ()
{
  cout << "Hello, world." << endl;
  return 0;
}
```

However, in general it is not good practice, since it imports the entirety of std into your program. This can become a problem when your programs get more complicated, and it takes over some of the namespace you would want to use. For this course it is an acceptable time-saving device, but I would suggest you remove it for any portfolio projects.

Just like iostream, the math header file contains information about the math functions. You can include it at the beginning of your program along with `iostream`:

```
#include <cmath>
```

Such header files have an initial 'c' to signify that these header files have been derived from the **C** language.

5.2 Math constants

Although you can calculate PI using acos, C++ has done the calculation ahead of time for you. If you are using C++20 or later, you can use the values in std::numbers.

```
double mypi = std::numbers::pi;
```

There are many other values available too, like e and phi.

If you are using an older version of the compilers, you may need to use a different technique to get PI. If you are including ¡cmath¿, you should also have M_PI defined.

```
double mypi = M_PI;
```

If you have a very old version of C++, it may need a special line above the include cmath in order for M_PI to work. But, I suggest you try the above versions first before using this:

```
#define _USE_MATH_DEFINES
#include <cmath>
```

5.3 Composition

Just as with mathematical functions, C++ functions can be **composed**, meaning that you use one expression as part of another. For example, you can use any expression as an argument to a function:

```
double x = cos (angle + pi/2);
```

This statement takes the value of `pi`, divides it by two and adds the result to the value of `angle`. The sum is then passed as an argument to the `cos` function.

You can also take the result of one function and pass it as an argument to another:

```
double x = exp (log (10.0));
```

This statement finds the log base e of 10 and then raises e to that power. The result gets assigned to `x`; I hope you know what it is.

5.4 Adding new functions

So far we have only been using the functions that are built into C++, but it is also possible to add new functions. Actually, we have already seen one function definition: `main`. The function named `main` is special because it indicates where the execution of the program begins, but the syntax for `main` is the same as for any other function definition:

```
void NAME ( LIST OF PARAMETERS ) {
  STATEMENTS
}
```

You can make up any name you want for your function, except that you can't call it `main` or any other C++ keyword. The list of parameters specifies what information, if any, you have to provide in order to use (or **call**) the new function.

`main` doesn't take any parameters, as indicated by the empty parentheses `()` in it's definition[1]. The first couple of functions we are going to write also have no parameters, so the syntax looks like this:

```
void newLine () {
  std::cout << std::endl;
}
```

This function is named `newLine`; it contains only a single statement, which outputs a newline character, represented by the special value `endl`.

In `main` we can call this new function using syntax that is similar to the way we call the built-in C++ commands:

```
int main ()
{
  std::cout << "First Line." << std::endl;
  newLine ();
  std::cout << "Second Line." << std::endl;
  return 0;
}
```

The output of this program is

```
First line.

Second line.
```

Notice the extra space between the two lines. What if we wanted more space between the lines? We could call the same function repeatedly:

```
int main ()
{
  std::cout << "First Line." << std::endl;
  newLine ();
  newLine ();
  newLine ();
  std::cout << "Second Line." << std::endl;
  return 0;
}
```

[1] You can add parameters in main to get access to the command line parameters. We will show how to use them much later in the book. For now, it is OK to use main without parameters.

We could use a for loop to call it multiple times:

```
int main ()
{
  std::cout << "First Line." << std::endl;
  for (int x = 0; x < 3; x++) {
     newLine ();
  }
  std::cout << "Second Line." << std::endl;
  return 0;
}
```

It is acceptable to have a function inside a loop. Or we could write a new function, named `threeLine`, that prints three new lines:

```
void threeLine ()
{
  newLine ();  newLine ();  newLine ();
}

int main ()
{
  std::cout << "First Line." << std::endl;
  threeLine ();
  std::cout << "Second Line." << std::endl;
  return 0;
}
```

You should notice a few things about this program:

- You can call the same procedure repeatedly. In fact, it is quite common and useful to do so.
- You can have one function call another function. In this case, `main` calls `threeLine` and `threeLine` calls `newLine`. Again, this is common and useful.
- In `threeLine` I wrote three statements all on the same line, which is syntactically legal (remember that spaces and new lines usually don't change the meaning of a program). On the other hand, it is usually a better idea to put each statement on a line by itself, to make your program easy to read. I sometimes break that rule in this book to save space.

So far, it may not be clear why it is worth the trouble to create all these new functions. Actually, there are a lot of reasons, but this example only demonstrates two:

1. Creating a new function gives you an opportunity to give a name to a group of statements. Functions can simplify a program by hiding a complex computation behind a single command, and by using English words in place of arcane code. Which is clearer, `newLine` or `std::cout << std::endl`?

2. Creating a new function can make a program smaller by eliminating repetitive code. For example, a short way to print nine consecutive new lines is to call `threeLine` three times. How would you print 27 new lines?

5.5 Definitions and uses

Pulling together all the code fragments from the previous section, the whole program looks like this:

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

void newLine ()
{
  cout << endl;
}

void threeLine ()
{
  newLine ();  newLine ();  newLine ();
}

int main ()
{
  cout << "First Line." << endl;
  threeLine ();
  cout << "Second Line." << endl;
  return 0;
}
```

This program contains three function definitions: `newLine`, `threeLine`, and `main`.

Inside the definition of `main`, there is a statement that uses or calls `threeLine`. Similarly, `threeLine` calls `newLine` three times. Notice that the definition of each function appears above the place where it is used.

This is necessary in C++; the definition of a function must appear before (above) the first use of the function. You should try compiling this program with the functions in a different order and see what error messages you get.

5.6 Function Prototypes

You may have realized that it is possible for code flow to be written in a way that there is no possible way to have the function that you call be defined above the one that called it. For example:

```
void functionA(char letter)
{
    if (letter == 'A')
    {
        std::cout << "A";
    }
    else {
        if (letter == 'B') {
            functionB(letter);
        }
    }
}
void functionB(char letter)
{
    if (letter == 'B')
    {
        std::cout << "B";
    }
    else {
        if (letter == 'A') {
            functionA(letter);
        }
    }
}
```

That (mostly pointless) bit of code can never be ordered in a way for it to work. So, there is a different feature – function prototypes. It is the first line of the function, with a semicolon instead of the body after it. To get the above code to compile, you just need this line at the top of the file:

```
void functionB(char letter);
```

When C++ compiles, it reads from the top of the file to the bottom. When it gets to a function that is being called, it needs to have seen it before to know how to call it. The function prototype is a way to let it know "They will be a function, this is it's name and these are the parameters". It will use that information if it sees a call to that function before it sees the definition.

Many people find function prototyping confusing. It is OK to not make a function prototype if you do not need it. But, it is good to know that this feature is available if you ever do need it.

5.7 Programs with multiple functions

When you look at program code listing that contains several functions, it is tempting to read it from top to bottom, but that is likely to be confusing, because that is not the **order of execution** of the program.

Execution always begins at the first statement of `main`, regardless of where it is in the program (often it is at the bottom). Statements are executed one at a time, in order, until you reach a function call. Function calls are like a detour in the flow of execution. Instead of going to the next statement, you go to the first line of the called function, execute all the statements there, and then come back and pick up again where you left off.

That sounds simple enough, except that you have to remember that one function can call another. Thus, while we are in the middle of `main`, we might have to go off and execute the statements in `threeLine`. But while we are executing `threeLine`, we get interrupted three times to go off and execute `newLine`.

Fortunately, C++ is adept at keeping track of where it is, so each time `newLine` completes, the program picks up where it left off in `threeLine`, and eventually gets back to `main` so the program can terminate.

What's the moral of this sordid tale? When you read a program, don't read from top to bottom. Instead, follow the flow of execution.

5.8 Parameters and arguments

Some of the built-in functions we have used have **parameters**, which are values that you provide to let the function do its job. For example, if you want to find the sine of a number, you have to indicate what the number is. Thus, `sin` takes a `double` value as a parameter.

Some functions take more than one parameter, like `pow`, which takes two `doubles`, the base and the exponent.

Notice that in each of these cases we have to specify not only how many parameters there are, but also what type they are. So it shouldn't surprise you that when you write a function definition, the parameter list indicates the type of each parameter. For example:

```
void printTwice (char phil) {
  std::cout << phil << phil << std::endl;
}
```

This function takes a single parameter, named `phil`, that has type `char`. Whatever that parameter is (and at this point we have no idea what it is), it gets printed twice, followed by a newline. I chose the name `phil` to suggest that the name you give a parameter is up to you, but in general you want to choose something more illustrative than `phil`.

In order to call this function, we have to provide a `char`. For example, we might have a `main` function like this:

```
int main () {
  printTwice ('a');
  return 0;
}
```

The `char` value you provide is called an **argument**, and we say that the argument is **passed** to the function. In this case the value `'a'` is passed as an argument to `printTwice` where it will get printed twice.

Alternatively, if we had a `char` variable, we could use it as an argument instead:

```
int main () {
  char argument = 'b';
  printTwice (argument);
  return 0;
}
```

Notice something very important here: the name of the variable we pass as an argument (`argument`) has nothing to do with the name of the parameter (`phil`). Let me say that again:

> **The name of the variable we pass as an argument has nothing to do with the name of the parameter.**

They can be the same or they can be different, but it is important to realize that they are not the same thing, except that they happen to have the same value (in this case the character `'b'`).

The value you provide as an argument must have the same type as the parameter of the function you call. This rule is important, but it is sometimes confusing because C++ sometimes converts arguments from one type to another automatically. For now you should learn the general rule, and we will deal with exceptions later.

5.9 Parameters and variables are local

Parameters and variables only exist inside their own functions. Within the confines of `main`, there is no such thing as `phil`. If you try to use it, the compiler will complain. Similarly, inside `printTwice` there is no such thing as `argument`.

Variables like this are said to be **local**. In order to keep track of parameters and local variables, it is useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but the variables are contained in larger boxes that indicate which function they belong to.

For example, the stack diagram for `printTwice` looks like this in figure 5.1:

Whenever a function is called, it creates a new **instance** of that function. Each instance of a function contains the parameters and local variables for that

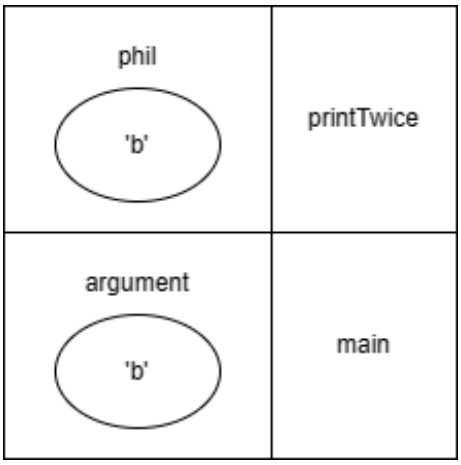


Figure 5.1: Stack diagram from when printTwice is called

function. In the diagram an instance of a function is represented by a box with the name of the function on the outside and the variables and parameters inside.

In the example, `main` has one local variable, `argument`, and no parameters. `printTwice` has no local variables and one parameter, named `phil`.

5.10 Functions with multiple parameters

The syntax for declaring and invoking functions with multiple parameters is a common source of errors. First, remember that you have to declare the type of every parameter. For example

```
void printTime (int hour, int minute) {
  std::cout << hour;
  std::cout << ":";
  std::cout << minute;
}
```

It might be tempting to write `(int hour, minute)`, but that format is only legal for variable declarations, not for parameters.

Another common source of confusion is that you do not have to declare the types of arguments. The following is wrong!

```
int hour = 11;
int minute = 59;
printTime (int hour, int minute);   // WRONG!
```

In this case, the compiler can tell the type of `hour` and `minute` by looking at their declarations. It is unnecessary and illegal to include the type when you pass them as arguments. The correct syntax is `printTime (hour, minute)`.

Do you remember the if/else code we did in the Chapter 3.3 to check if something was even or odd? If you think you might want to check the parity (evenness or oddness) of numbers often, you might want to "wrap" this code up in a function, as follows:

```
void printParity (int x) {
  if (x%2 == 0) {
    std::cout << "x is even" << std::endl;
  } else {
    std::cout << "x is odd" << std::endl;
  }
}
```

Now you have a function named `printParity` that will display an appropriate message for any integer you care to provide. It is perfectly acceptable to have an if statement in a function. In `main` you would call this function as follows:

```
  printParity (17);
```

Always remember that when you *call* a function, you do not have to declare the types of the arguments you provide. C++ can figure out what type they are. You should resist the temptation to write things like:

```
int number = 17;
printParity (int number);          // WRONG!!!
```

5.11 Functions with results

You might have noticed by now that some of the functions we are using, like the math functions, yield results. Other functions, like `newLine`, perform an action but don't return a value. That raises some questions:

- What happens if you call a function and you don't do anything with the result (i.e. you don't assign it to a variable or use it as part of a larger expression)?
- What happens if you use a function without a result as part of an expression, like `newLine() + 7`?
- Can we write functions that yield results, or are we stuck with things like `newLine` and `printTwice`?

The answer to the third question is "yes, you can write functions that return values," and we'll do it in Chapter 6.1. I will leave it up to you to answer the other two questions by trying them out. Any time you have a question about what is legal or illegal in C++, a good way to find out is to ask the compiler.

5.12 The `return` statement

The `return` statement allows you to terminate the execution of a function before you reach the end. One reason to use it is if you detect an error condition:

```
#include <cmath>

void printLogarithm (double x) {
  if (x <= 0.0) {
    std::cout << "Positive numbers only, please." << std::endl;
    return;
  }

  double result = log (x);
  std::cout << "The log of x is " << std::result;
}
```

This defines a function named `printLogarithm` that takes a `double` named `x` as a parameter. The first thing it does is check whether `x` is less than or equal to zero, in which case it displays an error message and then uses `return` to exit the function. The flow of execution immediately returns to the caller and the remaining lines of the function are not executed.

I used a floating-point value on the right side of the condition because there is a floating-point variable on the left.

Remember that any time you want to use one a function from the math library, you have to include the header file `cmath`.

5.13 Recursion

I mentioned earlier that it is legal for one function to call another, and we have seen several examples of that. I neglected to mention that it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical and interesting things a program can do.

For example, we can take our countdown while loop and change it to be recursive:

```
void countdown (int n) {
  if (n == 0) {
    std::cout << "Blastoff!" << std::endl;
  } else {
    std::cout << n << std::endl;
    countdown (n-1);
  }
}
```

The name of the function is `countdown` and it takes a single integer as a parameter. If the parameter is zero, it outputs the word "Blastoff." Otherwise, it outputs the parameter and then calls a function named `countdown`—itself—passing `n-1` as an argument.

What happens if we call this function like this:

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

void countdown (int n) {
  if (n == 0) {
    cout << "Blastoff!" << endl;
  } else {
    cout << n << endl;
    countdown (n-1);
  }
}

int main ()
{
  countdown (3);
  return 0;
}
```

The execution of `countdown` begins with `n=3`, and since `n` is not zero, it outputs the value 3, and then calls itself...

> The execution of `countdown` begins with `n=2`, and since `n` is not zero, it outputs the value 2, and then calls itself...
>
> > The execution of `countdown` begins with `n=1`, and since `n` is not zero, it outputs the value 1, and then calls itself...
> >
> > > The execution of `countdown` begins with `n=0`, and since `n` is zero, it outputs the word "Blastoff!" and then returns.
> >
> > The countdown that got `n=1` returns.
>
> The countdown that got `n=2` returns.

The countdown that got `n=3` returns.

And then you're back in `main` (what a trip). So the total output looks like:

```
3
2
1
Blastoff!
```

As a second example, let's look again at the functions `newLine` and `threeLine`.

```
void newLine () {
  std::cout << std::endl;
}

void threeLine () {
  newLine ();  newLine ();  newLine ();
}
```

Although these work, they would not be much help if I wanted to output 2 newlines, or 106. We could solve it with a loop, or we could try a recursive solution.

```
void nLines (int n) {
  if (n > 0) {
    std::cout << std::endl;
    nLines (n-1);
  }
}
```

This program is similar to `countdown`; as long as `n` is greater than zero, it outputs one newline, and then calls itself to output `n-1` additional newlines. Thus, the total number of newlines is `1 + (n-1)`, which usually comes out to roughly `n`.

The process of a function calling itself is called **recursion**, and such functions are said to be **recursive**.

5.14 Infinite recursion

In the examples in the previous section, notice that each time the functions get called recursively, the argument gets smaller by one, so eventually it gets to zero. When the argument is zero, the function returns immediately, *without making any recursive calls*. This case—when the function completes without making a recursive call—is called the **base case**.

If a recursion never reaches a base case, it will go on making recursive calls forever and the program will never terminate. This is known as **infinite recursion**, and it is generally not considered a good idea.

In most programming environments, a program with an infinite recursion will not really run forever. Eventually, something will break and the program will report an error. This is the first example we have seen of a run-time error (an error that does not appear until you run the program). The error that would normally happen is called a "stack overflow".

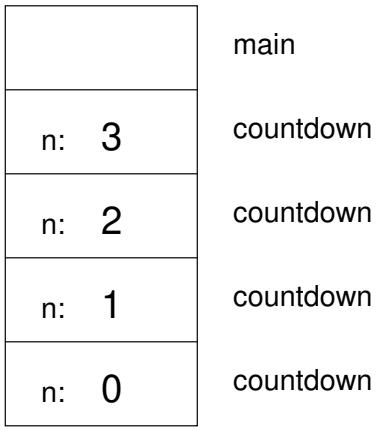
You should write a small program that recurses forever and run it to see what happens.

5.15 Stack diagrams for recursive functions

In a previous section we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can make it easier to interpret a recursive function.

Remember that every time a function gets called it creates a new instance that contains the function's local variables and parameters.

This figure shows a stack diagram for countdown, called with `n = 3`:



There is one instance of `main` and four instances of `countdown`, each with a different value for the parameter `n`. The bottom of the stack, `countdown` with `n=0` is the base case. It does not make a recursive call, so there are no more instances of `countdown`.

The instance of `main` is empty because `main` does not have any parameters or local variables. As an exercise, draw a stack diagram for `nLines`, invoked with the parameter `n=4`.

5.16 Glossary

initialization: A statement that declares a new variable and assigns a value to it at the same time.

function: A named sequence of statements that performs some useful function. Functions may or may not take parameters, and may or may not produce a result.

parameter: A piece of information you provide in order to call a function. Parameters are like variables in the sense that they contain values and have types.

argument: A value that you provide when you call a function. This value must have the same type as the corresponding parameter.

call: Cause a function to be executed.

recursion: The process of calling the same function you are currently executing.

infinite recursion: A function that calls itself recursively without every reaching the base case. Eventually an infinite recursion will cause a run-time error.

Chapter 6

Fruitful functions

6.1 Return values

Some of the built-in functions we have used, like the math functions, have produced results. That is, the effect of calling the function is to generate a new value, which we usually assign to a variable or use as part of an expression. For example:

```
double e = exp (1.0);
double height = radius * sin (angle);
```

But so far all the functions we have written have been **void** functions; that is, functions that return no value. When you call a void function, it is typically on a line by itself, with no assignment:

```
nLines (3);
countdown (n-1);
```

In this chapter, we are going to write functions that return things, which I will refer to as **fruitful** functions, for want of a better name. The first example is `area`, which takes a `double` as a parameter, and returns the area of a circle with the given radius:

```
double area (double radius) {
  double pi = acos (-1.0);
  double area = pi * radius * radius;
  return area;
}
```

The first thing you should notice is that the beginning of the function definition is different. Instead of `void`, which indicates a void function, we see `double`, which indicates that the return value from this function will have type `double`.

Also, notice that the last line is an alternate form of the `return` statement that includes a return value. This statement means, "return immediately from this function and use the following expression as a return value." The expression you provide can be arbitrarily complicated, so we could have written this function more concisely:

```
double area (double radius) {
  return acos(-1.0) * radius * radius;
}
```

On the other hand, **temporary** variables like `area` often make debugging easier. In either case, the type of the expression in the `return` statement must match the return type of the function. In other words, when you declare that the return type is `double`, you are making a promise that this function will eventually produce a `double`. If you try to `return` with no expression, or an expression with the wrong type, the compiler will take you to task.

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
double absoluteValue (double x) {
  if (x < 0) {
    return -x;
  } else {
    return x;
  }
}
```

Since these returns statements are in an alternative conditional, only one will be executed. Although it is legal to have more than one `return` statement in a function, you should keep in mind that as soon as one is executed, the function terminates without executing any subsequent statements.

Code that appears after a `return` statement, or any place else where it can never be executed, is called **dead code**. Some compilers warn you if part of your code is dead.

If you put return statements inside a conditional, then you have to guarantee that *every possible path* through the program hits a return statement. For example:

```
double absoluteValue (double x) {
  if (x < 0) {
    return -x;
  } else if (x > 0) {
    return x;
  }                          // WRONG!!
}
```

This program is not correct because if `x` happens to be 0, then neither condition will be true and the function will end without hitting a return statement. Unfortunately, many C++ compilers do not catch this error. As a result, the program may compile and run, but the return value when `x==0` could be anything, and will probably be different in different environments.

By now you are probably sick of seeing compiler errors, but as you gain more experience, you will realize that the only thing worse than getting a compiler error is *not* getting a compiler error when your program is wrong.

Here's the kind of thing that's likely to happen: you test `absoluteValue` with several values of `x` and it seems to work correctly. Then you give your program to someone else and they run it in another environment. It fails in some mysterious way, and it takes days of debugging to discover that the problem is an incorrect implementation of `absoluteValue`. If only the compiler had warned you!

From now on, if the compiler points out an error in your program, you should not blame the compiler. Rather, you should thank the compiler for finding your error and sparing you days of debugging. Some compilers have an option that tells them to be extra strict and report all the errors they can find. You should turn this option on all the time.

As an aside, you should know that there is a function in the math library called `fabs` that calculates the absolute value of a `double`—correctly.

6.2 Program development

At this point you should be able to look at complete C++ functions and tell what they do. But it may not be clear yet how to go about writing them. I am going to suggest one technique that I call **incremental development**.

As an example, imagine you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2). By the usual definition,

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{6.1}$$

The first step is to consider what a `distance` function should look like in C++. In other words, what are the inputs (parameters) and what is the output (return value).

In this case, the two points are the parameters, and it is natural to represent them using four `double`s. The return value is the distance, which will have type `double`.

Already we can write an outline of the function:

```
double distance (double x1, double y1, double x2, double y2) {
  return 0.0;
}
```

The `return` statement is a placekeeper so that the function will compile and return something, even though it is not the right answer. At this stage the

function doesn't do anything useful, but it is worthwhile to try compiling it so we can identify any syntax errors before we make it more complicated.

In order to test the new function, we have to call it with sample values. Somewhere in `main` I would add:

```
  double dist = distance (1.0, 2.0, 4.0, 6.0);
  cout << dist << endl;
```

I chose these values so that the horizontal distance is 3 and the vertical distance is 4; that way, the result will be 5 (the hypotenuse of a 3-4-5 triangle). When you are testing a function, it is useful to know the right answer.

Once we have checked the syntax of the function definition, we can start adding lines of code one at a time. After each incremental change, we recompile and run the program. That way, at any point we know exactly where the error must be—in the last line we added.

The next step in the computation is to find the differences $x_2 - x_1$ and $y_2 - y_1$. I will store those values in temporary variables named `dx` and `dy`.

```
double distance (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  cout << "dx is " << dx << endl;
  cout << "dy is " << dy << endl;
  return 0.0;
}
```

I added output statements that will let me check the intermediate values before proceeding. As I mentioned, I already know that they should be 3.0 and 4.0.

When the function is finished I will remove the output statements. Code like that is called **scaffolding**, because it is helpful for building the program, but it is not part of the final product. Sometimes it is a good idea to keep the scaffolding around, but comment it out, just in case you need it later.

The next step in the development is to square `dx` and `dy`. We could use the `pow` function, but it is simpler and faster to just multiply each term by itself.

```
double distance (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  double dsquared = dx*dx + dy*dy;
  cout << "dsquared is " << dsquared;
  return 0.0;
}
```

Again, I would compile and run the program at this stage and check the intermediate value (which should be 25.0).

Finally, we can use the `sqrt` function to compute and return the result.

```
double distance (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  double dsquared = dx*dx + dy*dy;
  double result = sqrt (dsquared);
  return result;
}
```

Then in `main`, we should output and check the value of the result.

As you gain more experience programming, you might find yourself writing and debugging more than one line at a time. Nevertheless, this incremental development process can save you a lot of debugging time.

The key aspects of the process are:

- Start with a working program and make small, incremental changes. At any point, if there is an error, you will know exactly where it is.
- Use temporary variables to hold intermediate values so you can output and check them.
- Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

6.3 Composition

As you should expect by now, once you define a new function, you can use it as part of an expression, and you can build new functions using existing functions. For example, what if someone gave you two points, the center of the circle and a point on the perimeter, and asked for the area of the circle?

Let's say the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we have a function, `distance`, that does that.

```
  double radius = distance (xc, yc, xp, yp);
```

The second step is to find the area of a circle with that radius, and return it.

```
  double result = area (radius);
  return result;
```

Wrapping that all up in a function, we get:

```
double fred (double xc, double yc, double xp, double yp) {
  double radius = distance (xc, yc, xp, yp);
  double result = area (radius);
  return result;
}
```

The name of this function is `fred`, which may seem odd. I will explain why in the next section.

The temporary variables `radius` and `area` are useful for development and debugging, but once the program is working we can make it more concise by composing the function calls:

```
double fred (double xc, double yc, double xp, double yp) {
  return area (distance (xc, yc, xp, yp));
}
```

6.4 Overloading

In the previous section you might have noticed that `fred` and `area` perform similar functions—finding the area of a circle—but take different parameters. For `area`, we have to provide the radius; for `fred` we provide two points.

If two functions do the same thing, it is natural to give them the same name. In other words, it would make more sense if `fred` were called `area`.

Having more than one function with the same name, which is called **overloading**, is legal in C++ *as long as each version takes different parameters.* So we can go ahead and rename `fred`:

```
double area (double xc, double yc, double xp, double yp) {
  return area (distance (xc, yc, xp, yp));
}
```

This looks like a recursive function, but it is not. Actually, this version of `area` is calling the other version. When you call an overloaded function, C++ knows which version you want by looking at the arguments that you provide. If you write:

```
double x = area (3.0);
```

C++ goes looking for a function named `area` that takes a `double` as an argument, and so it uses the first version. If you write

```
double x = area (1.0, 2.0, 4.0, 6.0);
```

C++ uses the second version of `area`.

Many of the built-in C++ commands are overloaded, meaning that there are different versions that accept different numbers or types of parameters.

Although overloading is a useful feature, it should be used with caution. You might get yourself nicely confused if you are trying to debug one version of a function while accidentally calling a different one.

Actually, that reminds me of one of the cardinal rules of debugging: **make sure that the version of the program you are looking at is the version of the program that is running!** Some time you may find yourself making one change after another in your program, and seeing the same thing every time

you run it. This is a warning sign that for one reason or another you are not running the version of the program you think you are. To check, stick in an output statement (it doesn't matter what it says) and make sure the behavior of the program changes accordingly.

6.5 Boolean values

The types we have seen so far are pretty big. There are a lot of integers in the world, and even more floating-point numbers. By comparison, the set of characters is pretty small. Well, there is another type in C++ that is even smaller. It is called **boolean**, and the only values in it are `true` and `false`.

Without thinking about it, we have been using boolean values for the last couple of chapters. The condition inside an `if` statement or a `while` statement is a boolean expression. Also, the result of a comparison operator is a boolean value. For example:

```
if (x == 5) {
  // do something
}
```

The operator `==` compares two integers and produces a boolean value.

The values `true` and `false` are keywords in C++, and can be used anywhere a boolean expression is called for. For example,

```
while (true) {
  // loop forever
}
```

is a standard idiom for a loop that should run forever (or until it reaches a `return` or `break` statement).

6.6 Boolean variables

As usual, for every type of value, there is a corresponding type of variable. In C++ the boolean type is called **bool**. Boolean variables work just like the other types:

```
bool fred;
fred = true;
bool testResult = false;
```

The first line is a simple variable declaration; the second line is an assignment, and the third line is a combination of a declaration and as assignment, called an initialization.

As I mentioned, the result of a comparison operator is a boolean, so you can store it in a `bool` variable

```
  bool evenFlag = (n%2 == 0);     // true if n is even
  bool positiveFlag = (x > 0);    // true if x is positive
```

and then use it as part of a conditional statement later

```
  if (evenFlag) {
    cout << "n was even when I checked it" << endl;
  }
```

A variable used in this way is called a **flag**, since it flags the presence or absence of some condition.

6.7 Logical operators

We have looked at logical operators FIXME – show how the and/or and NOT were using booleans

6.8 Bool functions

Functions can return `bool` values just like any other type, which is often convenient for hiding complicated tests inside functions. For example:

```
bool isSingleDigit (int x)
{
  if (x >= 0 && x < 10) {
    return true;
  } else {
    return false;
  }
}
```

The name of this function is `isSingleDigit`. It is common to give boolean functions names that sound like yes/no questions. The return type is `bool`, which means that every return statement has to provide a `bool` expression.

The code itself is straightforward, although it is a bit longer than it needs to be. Remember that the expression `x >= 0 && x < 10` has type `bool`, so there is nothing wrong with returning it directly, and avoiding the `if` statement altogether:

```
bool isSingleDigit (int x)
{
  return (x >= 0 && x < 10);
}
```

In `main` you can call this function in the usual ways:

```
cout << isSingleDigit (2) << endl;
bool bigFlag = !isSingleDigit (17);
```

The first line outputs the value `true` because 2 is a single-digit number. Unfortunately, when C++ outputs `bool`s, it does not display the words `true` and `false`, but rather the integers `1` and `0`.[1]

The second line assigns the value `true` to `bigFlag` only if 17 is *not* a single-digit number.

The most common use of `bool` functions is inside conditional statements

```
if (isSingleDigit (x)) {
  cout << "x is little" << endl;
} else {
  cout << "x is big" << endl;
}
```

6.9 Returning from `main`

Now that we have functions that return values, I can let you in on a secret. `main` is not really supposed to be a `void` function. It's supposed to return an integer:

```
int main ()
{
  return 0;
}
```

The usual return value from `main` is 0, which indicates that the program succeeded at whatever it was supposed to to. If something goes wrong, it is common to return -1, or some other value that indicates what kind of error occurred.

Of course, you might wonder who this value gets returned to, since we never call `main` ourselves. It turns out that when the system executes a program, it starts by calling `main` in pretty much the same way it calls all the other functions.

There are even some parameters that are passed to `main` by the system, but we are not going to deal with them for a little while.

6.10 More recursion

So far we have only learned a small subset of C++, but you might be interested to know that this subset is now a **complete** programming language, by which I mean that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features

[1] There is a way to fix that using the `boolalpha` flag, but it is too hideous to mention.

we have used so far (actually, we would need a few commands to control devices like the keyboard, mouse, disks, etc., but that's all).

Proving that claim is a non-trivial exercise first accomplished by Alan Turing, one of the first computer scientists (well, some would argue that he was a mathematician, but a lot of the early computer scientists started as mathematicians). Accordingly, it is known as the Turing thesis. If you take a course on the Theory of Computation, you will have a chance to see the proof.

To give you an idea of what you can do with the tools we have learned so far, we'll evaluate a few recursively-defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is typically not very useful:

frabjuous: an adjective used to describe something that is frabjuous.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the mathematical function **factorial**, you might get something like:

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

(Factorial is usually denoted with the symbol !, which is not to be confused with the C++ logical operator `!` which means NOT.) This definition says that the factorial of 0 is 1, and the factorial of any other value, n, is n multiplied by the factorial of $n-1$. So 3! is 3 times 2!, which is 2 times 1!, which is 1 times 0!. Putting it all together, we get 3! equal to 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can usually write a C++ program to evaluate it. The first step is to decide what the parameters are for this function, and what the return type is. With a little thought, you should conclude that factorial takes an integer as a parameter and returns an integer:

```
int factorial (int n)
{
}
```

If the argument happens to be zero, all we have to do is return 1:

```
int factorial (int n)
{
  if (n == 0) {
    return 1;
  }
}
```

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of $n-1$, and then multiply it by n.

```
int factorial (int n)
{
  if (n == 0) {
    return 1;
  } else {
    int recurse = factorial (n-1);
    int result = n * recurse;
    return result;
  }
}
```

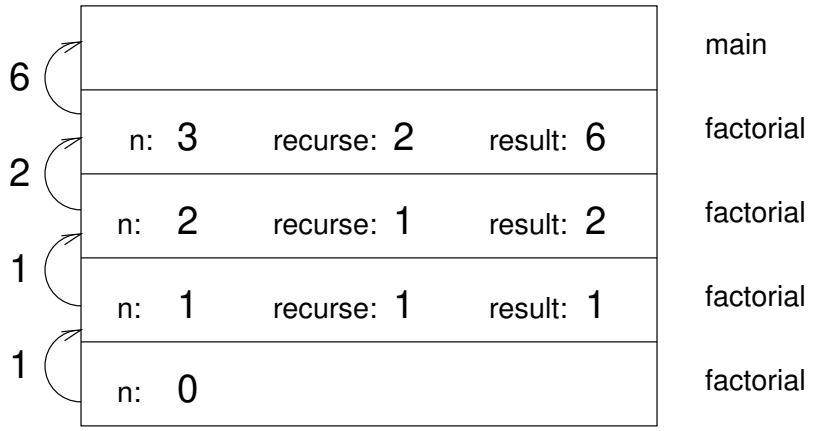
If we look at the flow of execution for this program, it is similar to `nLines` from the previous chapter. If we call `factorial` with the value 3:

Since 3 is not zero, we take the second branch and calculate the factorial of $n-1$...

> Since 2 is not zero, we take the second branch and calculate the factorial of $n-1$...
>
> > Since 1 is not zero, we take the second branch and calculate the factorial of $n-1$...
> >
> > > Since 0 *is* zero, we take the first branch and return the value 1 immediately without making any more recursive calls.
> >
> > The return value (1) gets multiplied by `n`, which is 1, and the result is returned.
>
> The return value (1) gets multiplied by `n`, which is 2, and the result is returned.

The return value (2) gets multiplied by `n`, which is 3, and the result, 6, is returned to `main`, or whoever called `factorial (3)`.

Here is what the stack diagram looks like for this sequence of function calls:



The return values are shown being passed back up the stack.

Notice that in the last instance of `factorial`, the local variables `recurse` and `result` do not exist because when `n=0` the branch that creates them does not execute.

6.11 Leap of faith

Following the flow of execution is one way to read programs, but as you saw in the previous section, it can quickly become labarynthine. An alternative is what I call the "leap of faith." When you come to a function call, instead of following the flow of execution, you *assume* that the function works correctly and returns the appropriate value.

In fact, you are already practicing this leap of faith when you use built-in functions. When you call `cos` or `exp`, you don't examine the implementations of those functions. You just assume that they work, because the people who wrote the built-in libraries were good programmers.

Well, the same is true when you call one of your own functions. For example, in Section 6.8 we wrote a function called `isSingleDigit` that determines whether a number is between 0 and 9. Once we have convinced ourselves that this function is correct—by testing and examination of the code—we can use the function without ever looking at the code again.

The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should *assume* that the recursive call works (yields the correct result), and then ask yourself, "Assuming that I can find the factorial of $n-1$, can I compute the factorial of n?" In this case, it is clear that you can, by multiplying by n.

Of course, it is a bit strange to assume that the function works correctly when you have not even finished writing it, but that's why it's called a leap of faith!

6.12 One more example

In the previous example I used temporary variables to spell out the steps, and to make the code easier to debug, but I could have saved a few lines:

```
int factorial (int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial (n-1);
  }
}
```

From now on I will tend to use the more concise version, but I recommend that you use the more explicit version while you are developing code. When you have it working, you can tighten it up, if you are feeling inspired.

After `factorial`, the classic example of a recursively-defined mathematical function is `fibonacci`, which has the following definition:

$$fibonacci(0) = 1$$
$$fibonacci(1) = 1$$
$$fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);$$

Translated into C++, this is

```
int fibonacci (int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return fibonacci (n-1) + fibonacci (n-2);
  }
}
```

If you try to follow the flow of execution here, even for fairly small values of `n`, your head explodes. But according to the leap of faith, if we assume that the two recursive calls (yes, you can make two recursive calls) work correctly, then it is clear that we get the right result by adding them together.

6.13 Encapsulation and generalization

Encapsulation usually means taking a piece of code and wrapping it up in a function, allowing you to take advantage of all the things functions are good for. We have seen two examples of encapsulation, when we wrote `printParity` in Section 5.10 and `isSingleDigit` in Section 6.8.

Generalization means taking something specific, like printing multiples of 2, and making it more general, like printing the multiples of any integer.

Here's a function that encapsulates the loop from the previous section and generalizes it to print multiples of `n`.

```
void printMultiples (int n)
{
  int i = 1;
  while (i <= 6) {
    cout << n*i << "   ";
    i = i + 1;
  }
  cout << endl;
}
```

To encapsulate, all I had to do was add the first line, which declares the name, parameter, and return type. To generalize, all I had to do was replace the value 2 with the parameter `n`.

If we call this function with the argument 2, we get the same output as before. With argument 3, the output is:

```
3   6   9   12   15   18
```

and with argument 4, the output is

```
4   8   12   16   20   24
```

By now you can probably guess how we are going to print a multiplication table: we'll call `printMultiples` repeatedly with different arguments. In fact, we are going to use another loop to iterate through the rows.

```
  int i = 1;
  while (i <= 6) {
    printMultiples (i);
    i = i + 1;
  }
```

First of all, notice how similar this loop is to the one inside `printMultiples`. All I did was replace the print statement with a function call.

The output of this program is

```
1   2   3   4   5   6
2   4   6   8   10   12
3   6   9   12   15   18
4   8   12   16   20   24
5   10   15   20   25   30
6   12   18   24   30   36
```

which is a (slightly sloppy) multiplication table. If the sloppiness bothers you, try replacing the spaces between columns with tab characters and see what you get.

6.14 Functions

In the last section I mentioned "all the things functions are good for." About this time, you might be wondering what exactly those things are. Here are some of the reasons functions are useful:

- By giving a name to a sequence of statements, you make your program easier to read and debug.
- Dividing a long program into functions allows you to separate parts of the program, debug them in isolation, and then compose them into a whole.
- Functions facilitate both recursion and iteration.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

6.15 More encapsulation

To demonstrate encapsulation again, I'll take the code from the previous section and wrap it up in a function:

```
void printMultTable () {
  int i = 1;
  while (i <= 6) {
    printMultiples (i);
    i = i + 1;
  }
}
```

The process I am demonstrating is a common development plan. You develop code gradually by adding lines to `main` or someplace else, and then when you get it working, you extract it and wrap it up in a function.

The reason this is useful is that you sometimes don't know when you start writing exactly how to divide the program into functions. This approach lets you design as you go along.

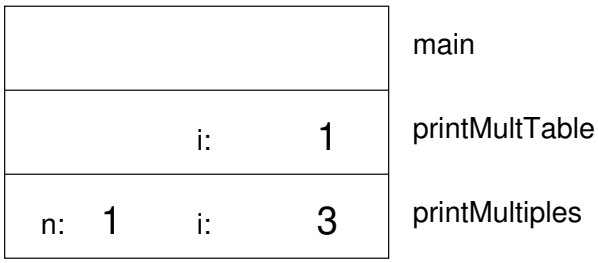
6.16 Local variables

About this time, you might be wondering how we can use the same variable `i` in both `printMultiples` and `printMultTable`. Didn't I say that you can only declare a variable once? And doesn't it cause problems when one of the functions changes the value of the variable?

The answer to both questions is "no," because the `i` in `printMultiples` and the `i` in `printMultTable` are *not the same variable*. They have the same name, but they do not refer to the same storage location, and changing the value of one of them has no effect on the other.

Remember that variables that are declared inside a function definition are local. You cannot access a local variable from outside its "home" function, and you are free to have multiple variables with the same name, as long as they are not in the same function.

The stack diagram for this program shows clearly that the two variables named `i` are not in the same storage location. They can have different values, and changing one does not affect the other.



Notice that the value of the parameter `n` in `printMultiples` has to be the same as the value of `i` in `printMultTable`. On the other hand, the value of `i` in `printMultiple` goes from 1 up to `n`. In the diagram, it happens to be 3. The next time through the loop it will be 4.

It is often a good idea to use different variable names in different functions, to avoid confusion, but there are good reasons to reuse names. For example, it is common to use the names `i`, `j` and `k` as loop variables. If you avoid using them in one function just because you used them somewhere else, you will probably make the program harder to read.

6.17 More generalization

As another example of generalization, imagine you wanted a program that would print a multiplication table of any size, not just the 6x6 table. You could add a parameter to `printMultTable`:

```
void printMultTable (int high) {
  int i = 1;
  while (i <= high) {
    printMultiples (i);
    i = i + 1;
  }
}
```

I replaced the value 6 with the parameter `high`. If I call `printMultTable` with the argument 7, I get

```
1   2   3   4   5   6
2   4   6   8   10   12
```

```
3   6   9   12   15   18
4   8   12   16   20   24
5   10   15   20   25   30
6   12   18   24   30   36
7   14   21   28   35   42
```

which is fine, except that I probably want the table to be square (same number of rows and columns), which means I have to add another parameter to `printMultiples`, to specify how many columns the table should have.

Just to be annoying, I will also call this parameter `high`, demonstrating that different functions can have parameters with the same name (just like local variables):

```
void printMultiples (int n, int high) {
  int i = 1;
  while (i <= high) {
    cout << n*i << "   ";
    i = i + 1;
  }
  cout << endl;
}

void printMultTable (int high) {
  int i = 1;
  while (i <= high) {
    printMultiples (i, high);
    i = i + 1;
  }
}
```

Notice that when I added a new parameter, I had to change the first line of the function (the interface or prototype), and I also had to change the place where the function is called in `printMultTable`. As expected, this program generates a square 7x7 table:

```
1   2   3   4   5   6   7
2   4   6   8   10   12   14
3   6   9   12   15   18   21
4   8   12   16   20   24   28
5   10   15   20   25   30   35
6   12   18   24   30   36   42
7   14   21   28   35   42   49
```

When you generalize a function appropriately, you often find that the resulting program has capabilities you did not intend. For example, you might notice that the multiplication table is symmetric, because $ab = ba$, so all the entries in the table appear twice. You could save ink by printing only half the table. To do that, you only have to change one line of `printMultTable`. Change

```
printMultiples (i, high);
```

to

```
printMultiples (i, i);
```

and you get

```
1
2   4
3   6   9
4   8   12   16
5   10   15   20   25
6   12   18   24   30   36
7   14   21   28   35   42   49
```

I'll leave it up to you to figure out how it works.

6.18 Glossary

return type: The type of value a function returns.

return value: The value provided as the result of a function call.

dead code: Part of a program that can never be executed, often because it appears after a `return` statement.

scaffolding: Code that is used during program development but is not part of the final version.

void: A special return type that indicates a void function; that is, one that does not return a value.

overloading: Having more than one function with the same name but different parameters. When you call an overloaded function, C++ knows which version to use by looking at the arguments you provide.

boolean: A value or variable that can take on one of two states, often called *true* and *false*. In C++, boolean values can be stored in a variable type called `bool`.

flag: A variable (usually type `bool`) that records a condition or status information.

comparison operator: An operator that compares two values and produces a boolean that indicates the relationship between the operands.

logical operator: An operator that combines boolean values in order to test compound conditions.

encapsulate: To divide a large complex program into components (like functions) and isolate the components from each other (for example, by using local variables).

local variable: A variable that is declared inside a function and that exists only within that function. Local variables cannot be accessed from outside their home function, and do not interfere with any other functions.

generalize: To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

development plan: A process for developing a program. In this chapter, I demonstrated a style of development based on developing code to do simple, specific things, and then encapsulating and generalizing.

We have seen many types of variables so far. We have seen integers, characters and booleans. The next type of variable is a bit different than the others. It allows you to have many variables of the same type grouped together under the same variable name. This type of variable is called an array.

6.19 Traditional array

First, we will be looking at the type of array that came from the language "C". Traditional arrays can be found in some legacy programs, or code that has to interface with modules written in C.

This type is defined with square brackets ([]). The definition explains what type of array it is, and that it is an array. For example, here is an array of integers:

```
int weights[5];
```

The above code created an array of type integer. It can hold 5 values, and the variable is named "weights". When the array is created, it does not set values in the array, so the numbers could be anything. To initialize the values to 0, you can do the following:

```
int weights[5]{};
```

That would set all 5 slots to 0. If you have certain values that you want to be initialized, you can do something similar:

```
int weights[5]{1,2,3,4,5};
```

That will set the first element to 1, the second to 2 and so on. The great thing about this method is that it catches if you put too many numbers:

```
int weights[5]{1,2,3,4,5,6}; // WRONG!
```

This will cause an error and not compile because this code is attempting to place 6 items in 5 slots.

Next step is how to get to the values, and change the values. These 5 values are indexed. You can access the variables by using an index number. Let's say I want to know what the first weight is. I can print it out like:

```
std::cout << "Weight 1 is "<< weight[0];
```

You may have noticed that this first index is 0. When computers index arrays, the first item is at zero. Also, the second is at 1 index, and the last item is at index size-1. For example, in the weights array, which has a size of 5, the last item is at index 4. I can print it like this:

```
std::cout << "Weight 5 is "<< weight[4];
```

If I want to change the value, I can use the index. Here is an example of me setting the first element of weights to 15.

```
weight[0] = 15;
std::cout << "Weight 1 is "<< weight[0];
```

It is possible to send this type of variable into a function, but I am not going to cover how to do that in this book. (When a traditional array is sent to a function, it loses information about it's size and is converted into a pointer.) Instead, I suggest you use some of the other helper functions to convert an array to and from one of the other types available.

Although this type of variable does work, it can be hard to understand how to use. If you are not careful, it is possible to access memory beyond where you are supposed to and could cause a memory corruption The Guidelines show many ways that this can cause a problem. . For this reason, Modern C++ suggests avoiding the traditional array if you can and use some of the newer array types[2]. We will cover the first one now.

6.20 std::array

In C++11, `std::array` was made to have an array that works like some of the other C++ containers do. They keep the information about their size when sent into a function, and copying code works the way that you would expect. There is also some safety code that checks to make sure you are not accessing memory that you shouldn't be. std::array is a templated class that is part of the C++ Standard Library. [3] In order to use this type of array, you need to #include¡array¿. That command can be up the top of your file, with your other include statements. Here is an example of making a similar weights array, but with std::array instead:

```
std::array<int, 5> weight; //create an array of size 5
```

This code used the less than and greater than as part of the definition. The first thing that is listed is the type of std::array we are making. This is making an integer array. There is a comma, then we list the number of items in the std::array. The name of the variable is listed after the greater than sign. We can initialize in similar ways to the traditional array:

```
std::array<int, 5> weight{}; //This sets all the values to 0
// weights2 has index 1 to be 0 and so on.
std::array<int, 5> weight2{1, 2, 3, 4, 5};
```

Note, in some older versions of the compiler (older than C++14), you would need the following syntax:

```
// weights2 has index 0 to be 1 and so on.
std::array<int, 5> weight2{ {1, 2, 3, 4, 5} };
```

[2]As of this books writing, the current standard is to avoid using the "pointer" types. So,items to avoid – traditional arrays, c-strings, and pointers.

[3]Templates and classes are both keywords we didn't talk about yet. I will be talking about both later on in the book. For now, know that templates are a way that you can get different types to work with the same code. Classes are a way to organize functions and data together.

Also, if you are using C++17 or above, std::array can figure out the size and type from the initialization. So this could work instead:

```
// C++17 and later can figure out the size and type
// by the values that are initialized
std::array weight3{ 1, 2, 3, 4, 5 };
```

The code that we used for the traditional arrays to get or set values will work for std::arrays too. But, there are even better calls that checks the index and creates an error (called an exception) if we try to access something beyond where we should. This code uses a function named "at".

```
std::array<int, 5> weight; //create an array of size 5

// The new commands to set and view
weight.at(0) = 7;  // sets the 0 index to 7
std::cout << weight.at(0); //prints the 0 index

std::cout << weight.at(6); // WRONG -- Too far!
```

Each time you use the "at" command, it checks to make sure the index you are using is valid. If it is, the code allows you to see or set what is at the index. If it is an invalid index, it will cause an error. In the above code, trying to view the value at the seventh item would cause an error. This is another run time error.

6.20.1 Conversions to and from traditional arrays

There are occasional times where you will need to deal with traditional arrays. One way to handle this is to convert the traditional array to a std::array. You could copy each item with a for loop, but there are is an even easier method.

std::to_array

std::to_array was added to the language in C++20. By using this command, you can use a traditional array to initialize a std::array. After it is created, you can use the safer array for the rest of your code:

```
// create a traditional array
int gems[]{75, 67, 50};

//convert it over to a std::array
std::array gemarray{std::to_array(gems)};

// look! It worked!
std::cout << gemarray.size();
```

.data()

If you need to use a C library that expects a parameter to be a traditional array, you can get the pointer to the std::array data with the data() method. Here is an example:

```
    // This will send an array, and the array size to the
    // function that requires a traditional array
    pointerFunction(gemarray.data(), gemarray.size());
```

6.21 A new way to send an array into a function

C++20 added a different way to send an array into a function. This particular method works for traditional arrays, std::arrays, and vectors. It is called a span. It is also a templated class. A span does not own the memory, it just has access to it. That means that it can not delete the memory, but it can read and write to it. So, if you are using code that has a traditional array, you can use the array through span. It will not create more memory, just allow you to use it differently. Here is an example of code that takes some type of int array and calculates the average of what was sent.

```
int average(std::span<int> items){
  int avg;
  int total{0};
  // if there is nothing in the span, the avg is 0
  if (items.size() == 0)
  {
    return 0;
  }
  // loop through the items, add them together
  for (const auto &item:items){
      total += item;
    }
    // calculate the average
  int valsize = static_cast< int >(items.size());
  avg = total/valsize;
  return avg;
}
```

The average code above can be used with a traditional array, std::array or even a std::vector. The following code shows how to call average.

```
  std::array weight{1, 2, 3, 4, 5}; // a std::array
  int gems[]{75, 67, 50}; // a traditional array

  std::cout << "Avg Gems " << average(gems) << std::endl;
  std::cout << "Avg weight "<< average(weight) << std::endl;
```

Note: this only works with traditional arrays if it hasn't decayed into a pointer. (This can happen if you send it into another function.) If that happened, you will need to create a span with the size information.

```
    average(std::span{pointerArray,10});
```

Spans have a lot more features. You can select just part of a span to create another smaller span, and much more. But, we will not be covering that material right now.

6.22 And wait, there is more!

The new array, std::array solved many problems, but it still has some of the drawbacks. A std::array has a certain size, and there is no way to change it while it is running. There is an even better version of arrays that is flexible about it's size called "Vectors". We will cover vectors in Chapter 10.

Chapter 7

Strings and things

7.1 Different Strings available

We have seen five types of values—booleans, characters, integers, floating-point numbers and strings—but only four types of variables—`bool`, `char`, `int` and `double`. So far we have no way to store a string in a variable or perform operations on strings.

In fact, there are several kinds of variables in C++ that can store strings. One is a basic type that is part of the C++ language, sometimes called "a native C string." The syntax for C strings is a bit ugly, and using them requires using similar methods as traditional arrays, so for the most part we are going to avoid them. But, here are the basics:

7.2 C-Strings

What people call c-strings are just character arrays:

```
char myString[]{ "Hi, I am a string!" };
```

The array is large enough to hold all the characters in the string and an extra character at the end. Every native c-string ends with the character '
0'. This is why these strings are sometimes called "null-terminated strings". For example,

```
char hellostr[6]{'h', 'e', 'l', 'l', 'o', '\0'};
```

Although the word "hello" only has 5 letters, the array needs to be at least 6 characters to hold the extra $'\backslash 0'$.

You can use all of the methods you learned with built in arrays with the native c-strings.

7.3 C++ strings

The string type we are going to use is called `string`, which is one of the classes that belong to the C++ Standard Library. [1]

Unfortunately, it is not possible to avoid C strings altogether. In a few places in this chapter I will warn you about some problems you might run into using `string`s instead of C strings.

7.4 `string` variables

You can create a variable with type `string` in the usual ways:

```
std::string first;
first = "Hello, ";
std::string second = "world.";
```

The first line creates a `string` without giving it a value. The second line assigns it the string value `"Hello."` The third line is a combined declaration and assignment, also called an initialization.

Normally when string values like `"Hello, "` or `"world."` appear, they are treated as C strings. In this case, when we assign them to a `string` variable, they are converted automatically to `string` values.

We can output strings in the usual way:

```
std::cout << first << second << std::endl;
```

In order to compile this code, you will have to include the header file for the `string` class, which means you will need to add the line #include¡string¿ to your file.

Before proceeding, you should type in the program above and make sure you can compile and run it.

7.5 Extracting characters from a string

Strings are called "strings" because they are made up of a sequence, or string, of characters. The first operation we are going to perform on a string is to extract one of the characters. C++ can use square brackets (`[` and `]`) for this operation:

```
string fruit = "banana";
char letter = fruit[1];
cout << letter << endl;
```

[1] You might be wondering what I mean by **class**.It will be a few more chapters before I can give a complete definition, but for now a class is a collection of functions that defines the operations we can perform on some type. The `string` class contains all the functions that apply to `string`s.

You can also use the "at" command that we learned about when we were learning about std::arrays in Section 6.20.

```
std::string fruit = "banana";
char letter = fruit.at(1);
std::cout << letter << std::endl;
```

The expression `fruit[1]` and `fruit.at(1)` both indicate that I want character number 1 from the string named `fruit`. The result is stored in a `char` named `letter`. When I output the value of `letter`, I get a surprise:

```
a
```

`a` is not the first letter of `"banana"`. Unless you are a computer scientist. For perverse reasons, computer scientists always start counting from zero. The 0th letter ("zeroeth") of `"banana"` is `b`. The 1th letter ("oneth") is `a` and the 2th ("twoeth") letter is `n`.

If you want the the zereoth letter of a string, you have to put zero in the square brackets:

```
char letter = fruit[0];
```

or

```
char letter = fruit.at(0);
```

7.6 Length

To find the length of a string (number of characters), we can use the `length` function. The syntax for calling this function is a little different from what we've seen before:

```
int length;
length = fruit.length();
```

To describe this function call, we would say that we are **invoking** the length function on the string named `fruit`. This vocabulary may seem strange, but we will see many more examples where we invoke a function on an object. The syntax for function invocation is called "dot notation," because the dot (period) separates the name of the object, `fruit`, from the name of the function, `length`.

`length` takes no arguments, as indicated by the empty parentheses `()`. The return value is an integer, in this case 6. Notice that it is legal to have a variable with the same name as a function.

To find the last letter of a string, you might be tempted to try something like

```
int length = fruit.length();
char last = fruit[length];       // WRONG!!
```

That won't work. The reason is that there is no 6th letter in `"banana"`. Since we started counting at 0, the 6 letters are numbered from 0 to 5. To get the last character, you have to subtract 1 from `length`.

```
int length = fruit.length();
char last = fruit[length-1];
```

Also, if you did this code with the at method, there would have been an error when you ran the code:

```
int length = fruit.length();
char last = fruit.at(length);       // WRONG!!
```

7.7 A run-time error

Way back in Section 1.3.2 I talked about run-time errors, which are errors that don't appear until a program has started running.

So far, you probably haven't seen many run-time errors, because we haven't been doing many things that can cause one. Well, now we are. If you use the `at` method and you provide an index that is negative or greater than `length-1`, you will get a run-time error and a message something like this:

```
index out of range: 6, string: banana
```

Try it in your development environment and see how it looks.

7.8 Traversal

A common thing to do with a string is start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. A natural way to encode a traversal is with a `while` statement:

```
int index = 0;
while (index < fruit.length()) {
  char letter = fruit[index];
  cout << letter << endl;
  index = index + 1;
}
```

This loop traverses the string and outputs each letter on a line by itself. Notice that the condition is `index < fruit.length()`, which means that when `index` is equal to the length of the string, the condition is false and the body of the loop is not executed. The last character we access is the one with the index `fruit.length()-1`.

The name of the loop variable is `index`. An **index** is a variable or value used to specify one member of an ordered set, in this case the set of characters

in the string. The index indicates (hence the name) which one you want. The set has to be ordered so that each letter has an index and each index refers to a single character.

As an exercise, write a function that takes a `string` as an argument and that outputs the letters backwards, all on one line.

7.9 Range based for loop

A C++ string is considered a container class, so you can use a different type of loop - a ranged based for loop. We showed an example when we covered std::arrays and std::spans, but we didn't explain what it was doing. I am a fan of these types of loops because it reduces the amount of things we need to type.

Here is an example of a ranged based loop that does the same thing as the while loop we saw earlier.

```
for (const auto &letter:fruit){
  std::cout << letter << std::endl;
}
```

Let's go into each part of this loop.

1. The "for" lets the code know there will be a for loop
2. Parentheses surround the information about the loop
3. The const is letting the code know that the information should not change
4. auto is an easy way to create a variable. The compile sees what value it will be set to, and makes sure it is that type. In this case, std::strings are made of characters, so it makes letter be a character.
5. The & makes the variable a reference. [2]
6. letter - This is the variable that it is using in the loop. Each time the loop runs, it takes the next character from fruit and places it in letter
7. : fruit - This is showing where it should be getting the values to put into letter.

This code works because the code knows so much about the "fruit" object. It knows how many letters are in it, and it knows how to take each character in turn and place it in letter. No need to keep track of the index. It is doing it for you.

[2] We have not covered references yet. The short version of the explanation is that it is a way to set a value without creating more memory. We will cover this in more depth later in Section 8.7

7.10 The `find` function

The `string` class provides several other functions that you can invoke on strings. The `find` function is like the opposite the `[]` operator. Instead of taking an index and extracting the character at that index, `find` takes a character and finds the index where that character appears.

```
string fruit = "banana";
int index = fruit.find('a');
```

This example finds the index of the letter `'a'` in the string. In this case, the letter appears three times, so it is not obvious what `find` should do. According to the documentation, it returns the index of the *first* appearance, so the result is 1. If the given letter does not appear in the string, `find` returns "std::string::npos".

In addition, there is a version of `find` that takes another `string` as an argument and that finds the index where the substring appears in the string. For example,

```
string fruit = "banana";
int index = fruit.find("nan");
```

This example returns the value 2.

You should remember from Section 6.4 that there can be more than one function with the same name, as long as they take a different number of parameters or different types. In this case, C++ knows which version of `find` to invoke by looking at the type of the argument we provide.

7.11 Our own version of `find`

If we are looking for a letter in a `string`, we may not want to start at the beginning of the string. One way to generalize the `find` function is to write a version that takes an additional parameter—the index where we should start looking. Here is an implementation of this function.

```
int find (string s, char c, int i)
{
  while (i<s.length()) {
    if (s[i] == c) return i;
    i = i + 1;
  }
  return string::npos;
}
```

Instead of invoking this function on a `string`, like the first version of `find`, we have to pass the `string` as the first argument. The other arguments are the character we are looking for and the index where we should start.

7.12 Looping and counting

The following program counts the number of times the letter `'a'` appears in a string:

```
string fruit = "banana";
int length = fruit.length();
int count = 0;

int index = 0;
while (index < length) {
  if (fruit[index] == 'a') {
    count = count + 1;
  }
  index = index + 1;
}
cout << count << endl;
```

This program demonstrates a common idiom, called a **counter**. The variable `count` is initialized to zero and then incremented each time we find an `'a'`. (To **increment** is to increase by one; it is the opposite of **decrement**, and unrelated to **excrement**, which is a noun.) When we exit the loop, `count` contains the result: the total number of a's.

As an exercise, encapsulate this code in a function named `countLetters`, and generalize it so that it accepts the string and the letter as arguments.

As a second exercise, rewrite this function so that instead of traversing the string, it uses the version of `find` we wrote in the previous section.

7.13 String concatenation

Interestingly, the `+` operator can be used on strings; it performs string **concatenation**. To concatenate means to join the two operands end to end. For example:

```
string fruit = "banana";
string bakedGood = " nut bread";
string dessert = fruit + bakedGood;
cout << dessert << endl;
```

The output of this program is `banana nut bread`.

Unfortunately, the `+` operator does not work on native C strings, so you cannot write something like

```
string dessert = "banana" + " nut bread";
```

because both operands are C strings. As long as one of the operands is `string`, though, C++ will automatically convert the other.

It is also possible to concatenate a character onto the beginning or end of a `string`. In the following example, we will use concatenation and character arithmetic to output an abecedarian series.

"Abecedarian" refers to a series or list in which the elements appear in alphabetical order. For example, in Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack and Quack. Here is a loop that outputs these names in order:

```
string suffix = "ack";

char letter = 'J';
while (letter <= 'Q') {
  cout << letter + suffix << endl;
  letter++;
}
```

The output of this program is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Of course, that's not quite right because I've misspelled "Ouack" and "Quack." As an exercise, modify the program to correct this error.

Again, be careful to use string concatenation only with `string`s and not with native C strings. Unfortunately, an expression like `letter + "ack"` is syntactically legal in C++, although it produces a very strange result, at least in my development environment.

7.14 `string`s are mutable

You can change the letters in a `string` one at a time using the `[]` operator on the left side of an assignment. For example,

```
string greeting = "Hello, world!";
greeting[0] = 'J';
cout << greeting << endl;
```

produces the output `Jello, world!`.

7.15 strings are comparable

All the comparison operators that work on ints and doubles also work on strings. For example, if you want to know if two strings are equal:

```
if (word == "banana") {
  cout << "Yes, we have no bananas!" << endl;
}
```

The other comparison operations are useful for putting words in alphabetical order.

```
if (word < "banana") {
  cout << "Your word, " << word << ", comes before banana." << endl;
} else if (word > "banana") {
  cout << "Your word, " << word << ", comes after banana." << endl;
} else {
  cout << "Yes, we have no bananas!" << endl;
}
```

You should be aware, though, that the string class does not handle upper and lower case letters the same way that people do. All the upper case letters come before all the lower case letters. As a result,

```
Your word, Zebra, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, like all lower-case, before performing the comparison. The next sections explains how. I will not address the more difficult problem, which is making the program realize that zebras are not fruit.

7.16 Character classification

It is often useful to examine a character and test whether it is upper or lower case, or whether it is a character or a digit. C++ provides a library of functions that perform this kind of character classification. In order to use these functions, you have to include the header file cctype.

```
char letter = 'a';
if (isalpha(letter)) {
  cout << "The character " << letter << " is a letter." << endl;
}
```

You might expect the return value from isalpha to be a bool, but for reasons I don't even want to think about, it is actually an integer that is 0 if the argument is not a letter, and some non-zero value if it is.

This oddity is not as inconvenient as it seems, because it is legal to use this kind of integer in a conditional, as shown in the example. The value 0 is treated as `false`, and all non-zero values are treated as `true`.

Technically, this sort of thing should not be allowed—integers are not the same thing as boolean values. Nevertheless, the C++ habit of converting automatically between types can be useful.

Other character classification functions include `isdigit`, which identifies the digits 0 through 9, and `isspace`, which identifies all kinds of "white" space, including spaces, tabs, newlines, and a few others. There are also `isupper` and `islower`, which distinguish upper and lower case letters.

Finally, there are two functions that convert letters from one case to the other, called `toupper` and `tolower`. Both take a single character as a parameter and return a (possibly converted) character.

```
char letter = 'a';
letter = toupper (letter);
cout << letter << endl;
```

The output of this code is `A`.

As an exercise, use the character classification and conversion library to write functions named `stringToUpper` and `stringToLower` that take a single `string` as a parameter, and that modify the string by converting all the letters to upper or lower case. The return type should be `void`.

7.17 Other `string` functions

This chapter does not cover all the `string` functions. Two additional ones, `c_str` and `substr`, are covered in Section 15.2 and Section 15.4.

7.18 string_view

In C++17, string_view was added to the language. String views are another way that we can use strings. This version of strings are good for variables that we are not planning on changing. It is a "view" of a string in memory that is somewhere else. This type needs the code to "#include <string_view >"

If you are sending a string (or c-string) into a function and you do not want it to change in the function, using std::string_view as the parameter will be the preferred solution over the const std::string& that was required before C++17.

```
void printme(std::string_view yum)
{
  std::cout << yum << std::endl;
}
```

String views are just a view of a different variables memory. If the other variable was changed, the view variable would be changed as well.

```
    std::string hey{"hey "};
    std::string_view nothey{hey};
    std::cout << nothey;
    hey[1]='i';
    std::cout << nothey;

This has the output:
hey hiy
```

7.19 Glossary

object: A collection of related data that comes with a set of functions that operate on it. The objects we have used so far are the `cout` object provided by the system, std::arrays, spans and `string`s.

index: A variable or value used to select one of the members of an ordered set, like a character from a string.

traverse: To iterate through all the elements of a set performing a similar operation on each.

counter: A variable used to count something, usually initialized to zero and then incremented.

increment: Increase the value of a variable by one. The increment operator in C++ is `++`. In fact, that's why C++ is called C++, because it is meant to be one better than C.

decrement: Decrease the value of a variable by one. The decrement operator in C++ is `--`.

concatenate: To join two operands end-to-end.

Chapter 8

Structures

8.1 Compound values

Most of the data types we have been working with represent a single value—an integer, a floating-point number, a boolean value. `string`s are different in the sense that they are made up of smaller pieces, the characters. Thus, `string`s are an example of a **compound** type.

Depending on what we are doing, we may want to treat a compound type as a single thing (or object), or we may want to access its parts (or instance variables). This ambiguity is useful.

It is also useful to be able to create your own compound values. C++ provides two mechanisms for doing that: **structures** and **classes**. We will start out with structures and get to classes in Chapter 14 (there is not much difference between them).

8.2 `Point` objects

As a simple example of a compound structure, consider the concept of a mathematical point. At one level, a point is two numbers (coordinates) that we treat collectively as a single object. In mathematical notation, points are often written in parentheses, with a comma separating the coordinates. For example, $(0, 0)$ indicates the origin, and (x, y) indicates the point x units to the right and y units up from the origin.

A natural way to represent a point in C++ is with two `double`s. The question, then, is how to group these two values into a compound object, or structure. The answer is a `struct` definition:

```
struct Point {
  double x, y;
};
```

`struct` definitions appear outside of any function definition, usually at the beginning of the program (after the `include` statements).

This definition indicates that there are two elements in this structure, named `x` and `y`. These elements are called **instance variables**, for reasons I will explain a little later.

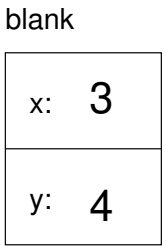
It is a common error to leave off the semi-colon at the end of a structure definition. It might seem odd to put a semi-colon after a curly-brace, but you'll get used to it.

Once you have defined the new structure, you can create variables with that type:

```
Point blank;
blank.x = 3.0;
blank.y = 4.0;
```

The first line is a conventional variable declaration: `blank` has type `Point`. The next two lines initialize the instance variables of the structure. The "dot notation" used here is similar to the syntax for invoking a function on an object, as in `fruit.length()`. Of course, one difference is that function names are always followed by an argument list, even if it is empty.

The result of these assignments is shown in the following state diagram:



As usual, the name of the variable `blank` appears outside the box and its value appears inside the box. In this case, that value is a compound object with two named instance variables.

8.3 Accessing instance variables

You can read the values of an instance variable using the same syntax we used to write them:

```
int x = blank.x;
```

The expression `blank.x` means "go to the object named `blank` and get the value of `x`." In this case we assign that value to a local variable named `x`. Notice that there is no conflict between the local variable named `x` and the instance variable named `x`. The purpose of dot notation is to identify *which* variable you are referring to unambiguously.

You can use dot notation as part of any C++ expression, so the following are legal.

```
cout << blank.x << ", " << blank.y << endl;
double distance = blank.x * blank.x + blank.y * blank.y;
```

The first line outputs `3, 4`; the second line calculates the value 25.

8.4 Operations on structures

Most of the operators we have been using on other types, like mathematical operators (`+`, `%`, etc.) and comparison operators (`==`, `>`, etc.), do not work on structures. Actually, it is possible to define the meaning of these operators for the new type, but we won't do that in this book.

On the other hand, the assignment operator *does* work for structures. It can be used in two ways: to initialize the instance variables of a structure or to copy the instance variables from one structure to another. An initialization looks like this:

```
Point blank = { 3.0, 4.0 };
```

The values in curly braces get assigned to the instance variables of the structure one by one, in order. So in this case, `x` gets the first value and `y` gets the second.

Unfortunately, this syntax can be used only in an initialization, not in an assignment statement. So the following is illegal.

```
Point blank;
blank = { 3.0, 4.0 };       // WRONG !!
```

You might wonder why this perfectly reasonable statement should be illegal; I'm not sure, but I think the problem is that the compiler doesn't know what type the right hand side should be. If you add a typecast:

```
Point blank;
blank = (Point){ 3.0, 4.0 };
```

That works.

It is legal to assign one structure to another. For example:

```
Point p1 = { 3.0, 4.0 };
Point p2 = p1;
cout << p2.x << ", " <<  p2.y << endl;
```

The output of this program is `3, 4`.

8.5 Structures as parameters

You can pass structures as parameters in the usual way. For example,

```
void printPoint (Point p) {
  cout << "(" << p.x << ", " << p.y << ")" << endl;
}
```

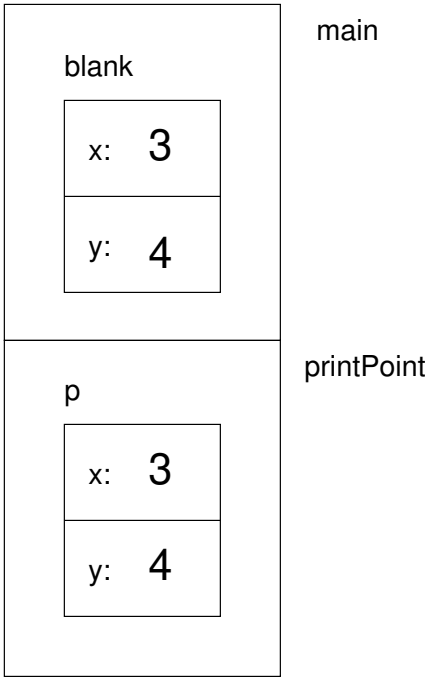
`printPoint` takes a point as an argument and outputs it in the standard format. If you call `printPoint (blank)`, it will output `(3, 4)`.

As a second example, we can rewrite the `distance` function from Section 6.2 so that it takes two `Point`s as parameters instead of four `double`s.

```
double distance (Point p1, Point p2) {
  double dx = p2.x - p1.x;
  double dy = p2.y - p1.y;
  return sqrt (dx*dx + dy*dy);
}
```

8.6 Call by value

When you pass a structure as an argument, remember that the argument and the parameter are not the same variable. Instead, there are two variables (one in the caller and one in the callee) that have the same value, at least initially. For example, when we call `printPoint`, the stack diagram looks like this:



If `printPoint` happened to change one of the instance variables of `p`, it would have no effect on `blank`. Of course, there is no reason for `printPoint` to modify its parameter, so this isolation between the two functions is appropriate.

This kind of parameter-passing is called "pass by value" because it is the value of the structure (or other type) that gets passed to the function.

8.7 Call by reference

An alternative parameter-passing mechanism that is available in C++ is called "pass by reference." This mechanism makes it possible to pass a structure to a procedure and modify it.

For example, you can reflect a point around the 45-degree line by swapping the two coordinates. The most obvious (but incorrect) way to write a `reflect` function is something like this:

```
void reflect (Point p)      // WRONG !!
{
  double temp = p.x;
  p.x = p.y;
  p.y = temp;
}
```

But this won't work, because the changes we make in `reflect` will have no effect on the caller.

Instead, we have to specify that we want to pass the parameter by reference. We do that by adding an ampersand (`&`) to the parameter declaration:

```
void reflect (Point& p)
{
  double temp = p.x;
  p.x = p.y;
  p.y = temp;
}
```

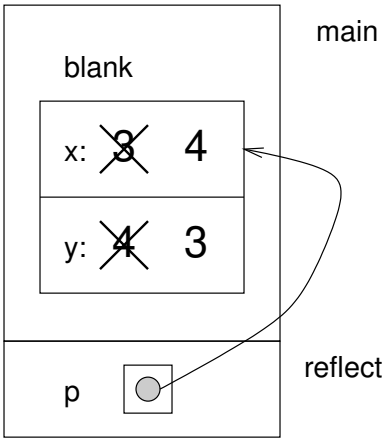
Now we can call the function in the usual way:

```
  printPoint (blank);
  reflect (blank);
  printPoint (blank);
```

The output of this program is as expected:

```
(3, 4)
(4, 3)
```

Here's how we would draw a stack diagram for this program:



The parameter `p` is a reference to the structure named `blank`. The usual representation for a reference is a dot with an arrow that points to whatever the reference refers to.

The important thing to see in this diagram is that any changes that `reflect` makes in `p` will also affect `blank`.

Passing structures by reference is more versatile than passing by value, because the callee can modify the structure. It is also faster, because the system does not have to copy the whole structure. On the other hand, it is less safe, since it is harder to keep track of what gets modified where. Nevertheless, in C++ programs, almost all structures are passed by reference almost all the time. In this book I will follow that convention.

8.8 Rectangles

Now let's say that we want to create a structure to represent a rectangle. The question is, what information do I have to provide in order to specify a rectangle? To keep things simple let's assume that the rectangle will be oriented vertically or horizontally, never at an angle.

There are a few possibilities: I could specify the center of the rectangle (two coordinates) and its size (width and height), or I could specify one of the corners and the size, or I could specify two opposing corners.

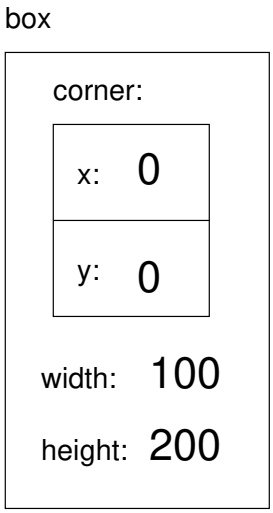
The most common choice in existing programs is to specify the upper left corner of the rectangle and the size. To do that in C++, we will define a structure that contains a `Point` and two doubles.

```
struct Rectangle {
  Point corner;
  double width, height;
};
```

Notice that one structure can contain another. In fact, this sort of thing is quite common. Of course, this means that in order to create a `Rectangle`, we have to create a `Point` first:

```
Point corner = { 0.0, 0.0 };
Rectangle box = { corner, 100.0, 200.0 };
```

This code creates a new `Rectangle` structure and initializes the instance variables. The figure shows the effect of this assignment.



We can access the `width` and `height` in the usual way:

```
box.width += 50.0;
cout << box.height << endl;
```

In order to access the instance variables of `corner`, we can use a temporary variable:

```
Point temp = box.corner;
double x = temp.x;
```

Alternatively, we can compose the two statements:

```
double x = box.corner.x;
```

It makes the most sense to read this statement from right to left: "Extract `x` from the `corner` of the `box`, and assign it to the local variable `x`."

While we are on the subject of composition, I should point out that you can, in fact, create the `Point` and the `Rectangle` at the same time:

```
Rectangle box = { { 0.0, 0.0 }, 100.0, 200.0 };
```

The innermost curly braces are the coordinates of the corner point; together they make up the first of the three values that go into the new `Rectangle`. This statement is an example of **nested structure**.

8.9 Structures as return types

You can write functions that return structures. For example, `findCenter` takes a `Rectangle` as an argument and returns a `Point` that contains the coordinates of the center of the `Rectangle`:

```
Point findCenter (Rectangle& box)
{
  double x = box.corner.x + box.width/2;
  double y = box.corner.y + box.height/2;
  Point result = {x, y};
  return result;
}
```

To call this function, we have to pass a box as an argument (notice that it is being passed by reference), and assign the return value to a `Point` variable:

```
  Rectangle box = { {0.0, 0.0}, 100, 200 };
  Point center = findCenter (box);
  printPoint (center);
```

The output of this program is `(50, 100)`.

8.10 Passing other types by reference

It's not just structures that can be passed by reference. All the other types we've seen can, too. For example, to swap two integers, we could write something like:

```
void swap (int& x, int& y)
{
  int temp = x;
  x = y;
  y = temp;
}
```

We would call this function in the usual way:

```
  int i = 7;
  int j = 9;
  swap (i, j);
  cout << i << j << endl;
```

The output of this program is `97`. Draw a stack diagram for this program to convince yourself this is true. If the parameters `x` and `y` were declared as regular parameters (without the `&`s), `swap` would not work. It would modify `x` and `y` and have no effect on `i` and `j`.

When people start passing things like integers by reference, they often try to use an expression as a reference argument. For example:

```
int i = 7;
int j = 9;
swap (i, j+1);          // WRONG!!
```

This is not legal because the expression `j+1` is not a variable—it does not occupy a location that the reference can refer to. It is a little tricky to figure out exactly what kinds of expressions can be passed by reference. For now a good rule of thumb is that reference arguments have to be variables.

8.11 Getting user input

The programs we have written so far are pretty predictable; they do the same thing every time they run. Most of the time, though, we want programs that take input from the user and respond accordingly.

There are many ways to get input, including keyboard input, mouse movements and button clicks, as well as more exotic mechanisms like voice control and retinal scanning. In this text we will consider only keyboard input.

In the header file `iostream`, C++ defines an object named `cin` that handles input in much the same way that `cout` handles output. To get an integer value from the user:

```
int x;
cin >> x;
```

The `>>` operator causes the program to stop executing and wait for the user to type something. If the user types a valid integer, the program converts it into an integer value and stores it in `x`.

If the user types something other than an integer, C++ doesn't report an error, or anything sensible like that. Instead, it puts some meaningless value in `x` and continues.

Fortunately, there is a way to check and see if an input statement succeeds. We can invoke the `good` function on `cin` to check what is called the **stream state**. `good` returns a `bool`: if true, then the last input statement succeeded. If not, we know that some previous operation failed, and also that the next operation will fail.

Thus, getting input from the user might look like this:

```
#include <iostream>

using namespace std;

int main ()
{
  int x;

  // prompt the user for input
```

```
  cout << "Enter an integer: ";

  // get input
  cin >> x;

  // check and see if the input statement succeeded
  if (cin.good() == false) {
    cout << "That was not an integer." << endl;
    return -1;
  }

  // print the value we got from the user
  cout << x << endl;
  return 0;
}
```

`cin` can also be used to input a `string`:

```
  string name;

  cout << "What is your name? ";
  cin >> name;
  cout << name << endl;
```

Unfortunately, this statement only takes the first word of input, and leaves the rest for the next input statement. So, if you run this program and type your full name, it will only output your first name.

Because of these problems (inability to handle errors and funny behavior), I avoid using the `>>` operator altogether, unless I am reading data from a source that is known to be error-free.

Instead, I use a function in the header `string` called `getline`.

```
  string name;

  cout << "What is your name? ";
  getline (cin, name);
  cout << name << endl;
```

The first argument to `getline` is `cin`, which is where the input is coming from. The second argument is the name of the `string` where you want the result to be stored.

`getline` reads the entire line until the user hits Return or Enter. This is useful for inputting strings that contain spaces.

In fact, `getline` is generally useful for getting input of any kind. For example, if you wanted the user to type an integer, you could input a string and then check to see if it is a valid integer. If so, you can convert it to an integer value. If not, you can print an error message and ask the user to try again.

To convert a string to an integer you can use the `atoi` function defined in the header file `cstdlib`. We will get to that in Section 15.4.

8.12 Glossary

structure: A collection of data grouped together and treated as a single object.

instance variable: One of the named pieces of data that make up a structure.

reference: A value that indicates or refers to a variable or structure. In a state diagram, a reference appears as an arrow.

pass by value: A method of parameter-passing in which the value provided as an argument is copied into the corresponding parameter, but the parameter and the argument occupy distinct locations.

pass by reference: A method of parameter-passing in which the parameter is a reference to the argument variable. Changes to the parameter also affect the argument variable.

Chapter 9

More structures

9.1 Time

As a second example of a user-defined structure, we will define a type called `Time`, which is used to record the time of day. The various pieces of information that form a time are the hour, minute and second, so these will be the instance variables of the structure.

The first step is to decide what type each instance variable should be. It seems clear that `hour` and `minute` should be integers. Just to keep things interesting, let's make `second` a `double`, so we can record fractions of a second.

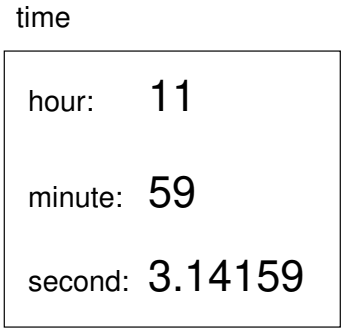
Here's what the structure definition looks like:

```
struct Time {
  int hour, minute;
  double second;
};
```

We can create a `Time` object in the usual way:

```
  Time time = { 11, 59, 3.14159 };
```

The state diagram for this object looks like this:



The word "instance" is sometimes used when we talk about objects, because every object is an instance (or example) of some type. The reason that instance variables are so-named is that every instance of a type has a copy of the instance variables for that type.

9.2 printTime

When we define a new type it is a good idea to write function that displays the instance variables in a human-readable form. For example:

```
void printTime (Time& t) {
  cout << t.hour << ":" << t.minute << ":" << t.second << endl;
}
```

The output of this function, if we pass `time` an argument, is `11:59:3.14159`.

```
#include <iostream>

using namespace std;

struct Time {
  int hour, minute;
  double second;
};


void printTime (Time& t) {
  cout << t.hour << ":" << t.minute << ":" << t.second << endl;
  cout << "Time is " << t.hour << " hour " << t.minute << " minutes " << t.second << "  se
}


int main ()
{
 Time time = { 11, 59, 3.14159 };
 printTime(time);

 return 0;
}
```

9.3 Functions for objects

In the next few sections, I will demonstrate several possible interfaces for functions that operate on objects. For some operations, you will have a choice of several possible interfaces, so you should consider the pros and cons of each of these:

pure function: Takes objects and/or basic types as arguments but does not modify the objects. The return value is either a basic type or a new object created inside the function.

modifier: Takes objects as parameters and modifies some or all of them. Often returns void.

fill-in function: One of the parameters is an "empty" object that gets filled in by the function. Technically, this is a type of modifier.

9.4 Pure functions

A function is considered a pure function if the result depends only on the arguments, and it has no side effects like modifying an argument or outputting something. The only result of calling a pure function is the return value.

One example is `after`, which compares two `Time`s and returns a `bool` that indicates whether the first operand comes after the second:

```
bool after (Time& time1, Time& time2) {
  if (time1.hour > time2.hour) return true;
  if (time1.hour < time2.hour) return false;

  if (time1.minute > time2.minute) return true;
  if (time1.minute < time2.minute) return false;

  if (time1.second > time2.second) return true;
  return false;
}
```

What is the result of this function if the two times are equal? Does that seem like the appropriate result for this function? If you were writing the documentation for this function, would you mention that case specifically?

A second example is `addTime`, which calculates the sum of two times. For example, if it is `9:14:30`, and your breadmaker takes 3 hours and 35 minutes, you could use `addTime` to figure out when the bread will be done.

Here is a rough draft of this function that is not quite right:

```
Time addTime (Time& t1, Time& t2) {
  Time sum;
  sum.hour = t1.hour + t2.hour;
  sum.minute = t1.minute + t2.minute;
  sum.second = t1.second + t2.second;
  return sum;
}
```

Here is an example of how to use this function. If `currentTime` contains the current time and `breadTime` contains the amount of time it takes for your

breadmaker to make bread, then you could use `addTime` to figure out when the bread will be done.

```
Time currentTime = { 9, 14, 30.0 };
Time breadTime = { 3, 35, 0.0 };
Time doneTime = addTime (currentTime, breadTime);
printTime (doneTime);
```

The output of this program is `12:49:30`, which is correct. On the other hand, there are cases where the result is not correct. Can you think of one?

The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than 60. When that happens we have to "carry" the extra seconds into the minutes column, or extra minutes into the hours column.

Here's a second, corrected version of this function.

```
Time addTime (Time& t1, Time& t2) {
  Time sum;
  sum.hour = t1.hour + t2.hour;
  sum.minute = t1.minute + t2.minute;
  sum.second = t1.second + t2.second;

  if (sum.second >= 60.0) {
    sum.second -= 60.0;
    sum.minute += 1;
  }
  if (sum.minute >= 60) {
    sum.minute -= 60;
    sum.hour += 1;
  }
  return sum;
}
```

Although it's correct, it's starting to get big. Later, I will suggest an alternate approach to this problem that will be much shorter.

This code demonstrates two operators we have not seen before, `+=` and `-=`. These operators provide a concise way to increment and decrement variables. For example, the statement `sum.second -= 60.0;` is equivalent to `sum.second = sum.second - 60;`

9.5 `const` parameters

You might have noticed that the parameters for `after` and `addTime` are being passed by reference. Since these are pure functions, they do not modify the parameters they receive, so I could just as well have passed them by value.

The advantage of passing by value is that the calling function and the callee are appropriately encapsulated—it is not possible for a change in one to affect the other, except by affecting the return value.

On the other hand, passing by reference usually is more efficient, because it avoids copying the argument. Furthermore, there is a nice feature in C++, called `const`, that can make reference parameters just as safe as value parameters.

If you are writing a function and you do not intend to modify a parameter, you can declare that it is a **constant reference parameter**. The syntax looks like this:

```
void printTime (const Time& time) ...
Time addTime (const Time& t1, const Time& t2) ...
```

I've included only the first line of the functions. If you tell the compiler that you don't intend to change a parameter, it can help remind you. If you try to change one, you should get a compiler error, or at least a warning.

9.6 Modifiers

Of course, sometimes you *want* to modify one of the arguments. Functions that do are called modifiers.

As an example of a modifier, consider `increment`, which adds a given number of seconds to a `Time` object. Again, a rough draft of this function looks like:

```
void increment (Time& time, double secs) {
  time.second += secs;

  if (time.second >= 60.0) {
    time.second -= 60.0;
    time.minute += 1;
  }
  if (time.minute >= 60) {
    time.minute -= 60;
    time.hour += 1;
  }
}
```

The first line performs the basic operation; the remainder deals with the special cases we saw before.

Is this function correct? What happens if the argument `secs` is much greater than 60? In that case, it is not enough to subtract 60 once; we have to keep doing it until `second` is below 60. We can do that by replacing the `if` statements with `while` statements:

```
void increment (Time& time, double secs) {
```

```
  time.second += secs;

  while (time.second >= 60.0) {
    time.second -= 60.0;
    time.minute += 1;
  }
  while (time.minute >= 60) {
    time.minute -= 60;
    time.hour += 1;
  }
}
```

This solution is correct, but not very efficient. Can you think of a solution that does not require iteration?

9.7 Fill-in functions

Occasionally you will see functions like `addTime` written with a different interface (different arguments and return values). Instead of creating a new object every time `addTime` is called, we could require the caller to provide an "empty" object where `addTime` can store the result. Compare the following with the previous version:

```
void addTimeFill (const Time& t1, const Time& t2, Time& sum) {
  sum.hour = t1.hour + t2.hour;
  sum.minute = t1.minute + t2.minute;
  sum.second = t1.second + t2.second;

  if (sum.second >= 60.0) {
    sum.second -= 60.0;
    sum.minute += 1;
  }
  if (sum.minute >= 60) {
    sum.minute -= 60;
    sum.hour += 1;
  }
}
```

One advantage of this approach is that the caller has the option of reusing the same object repeatedly to perform a series of additions. This can be slightly more efficient, although it can be confusing enough to cause subtle errors. For the vast majority of programming, it is worth a spending a little run time to avoid a lot of debugging time.

Notice that the first two parameters can be declared `const`, but the third cannot.

9.8 Which is best?

Anything that can be done with modifiers and fill-in functions can also be done with pure functions. In fact, there are programming languages, called **functional** programming languages, that only allow pure functions. Some programmers believe that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. Nevertheless, there are times when modifiers are convenient, and cases where functional programs are less efficient.

In general, I recommend that you write pure functions whenever it is reasonable to do so, and resort to modifiers only if there is a compelling advantage. This approach might be called a functional programming style.

9.9 Incremental development versus planning

In this chapter I have demonstrated an approach to program development I refer to as **rapid prototyping with iterative improvement**. In each case, I wrote a rough draft (or prototype) that performed the basic calculation, and then tested it on a few cases, correcting flaws as I found them.

Although this approach can be effective, it can lead to code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to know if you have found all the errors.

An alternative is high-level planning, in which a little insight into the problem can make the programming much easier. In this case the insight is that a `Time` is really a three-digit number in base 60! The `second` is the "ones column," the `minute` is the "60's column", and the `hour` is the "3600's column."

When we wrote `addTime` and `increment`, we were effectively doing addition in base 60, which is why we had to "carry" from one column to the next.

Thus an alternate approach to the whole problem is to convert `Time`s into `double`s and take advantage of the fact that the computer already knows how to do arithmetic with `double`s. Here is a function that converts a `Time` into a `double`:

```
double convertToSeconds (const Time& t) {
  int minutes = t.hour * 60 + t.minute;
  double seconds = minutes * 60 + t.second;
  return seconds;
}
```

Now all we need is a way to convert from a `double` to a `Time` object:

```
Time makeTime (double secs) {
  Time time;
  time.hour = int (secs / 3600.0);
  secs -= time.hour * 3600.0;
  time.minute = int (secs / 60.0);
```

```
  secs -= time.minute * 60;
  time.second = secs;
  return time;
}
```

You might have to think a bit to convince yourself that the technique I am using to convert from one base to another is correct. Assuming you are convinced, we can use these functions to rewrite `addTime`:

```
Time addTime (const Time& t1, const Time& t2) {
  double seconds = convertToSeconds (t1) + convertToSeconds (t2);
  return makeTime (seconds);
}
```

This is much shorter than the original version, and it is much easier to demonstrate that it is correct (assuming, as usual, that the functions it calls are correct). As an exercise, rewrite `increment` the same way.

9.10 Generalization

In some ways converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with times is better.

But if we have the insight to treat times as base 60 numbers, and make the investment of writing the conversion functions (`convertToSeconds` and `makeTime`), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add more features later. For example, imagine subtracting two `Time`s to find the duration between them. The naive approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct.

Ironically, sometimes making a problem harder (more general) makes is easier (fewer special cases, fewer opportunities for error).

9.11 Algorithms

When you write a general solution for a class of problems, as opposed to a specific solution to a single problem, you have written an **algorithm**. I mentioned this word in Chapter 1, but did not define it carefully. It is not easy to define, so I will try a couple of approaches.

First, consider something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not really algorithmic.

But if you were "lazy," you probably cheated by learning a few tricks. For example, to find the product of n and 9, you can write $n - 1$ as the first digit

and $10 - n$ as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules.

In my opinion, it is embarrassing that humans spend so much time in school learning to execute algorithms that, quite literally, require no intelligence.

On the other hand, the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming.

Some of the things that people do naturally, without difficulty or conscious thought, are the most difficult to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

Later in this book, you will have the opportunity to design simple algorithms for a variety of problems. If you take the next class in the Computer Science sequence, Data Structures, you will see some of the most interesting, clever, and useful algorithms computer science has produced.

9.12 Glossary

instance: An example from a category. My cat is an instance of the category "feline things." Every object is an instance of some type.

instance variable: One of the named data items that make up a structure. Each structure has its own copy of the instance variables for its type.

constant reference parameter: A parameter that is passed by reference but that cannot be modified.

pure function: A function whose result depends only on its parameters, and that has no effects other than returning a value.

functional programming style: A style of program design in which the majority of functions are pure.

modifier: A function that changes one or more of the objects it receives as parameters, and usually returns `void`.

fill-in function: A function that takes an "empty" object as a parameter and fills it its instance variables instead of generating a return value.

algorithm: A set of instructions for solving a class of problems by a mechanical, unintelligent process.

Chapter 10

Vectors

A **vector** is a set of values where each value is identified by a number (called an index). A `string` is similar to a vector, since it is made up of an indexed set of characters. The nice thing about vectors is that they can be made up of any type of element, including basic types like `int`s and `double`s, and user-defined types like `Point` and `Time`.

The `vector` type is defined in the C++ Standard Template Library (STL). In order to use it, you have to include the header file `vector`; again, the details of how to do that depend on your programming environment.

You can create a vector the same way you create other variable types:

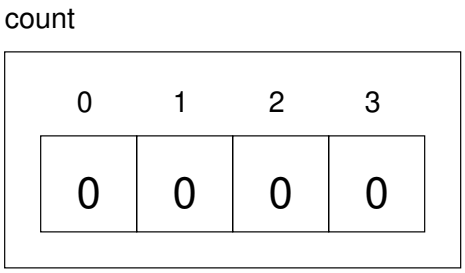
```
vector<int> count;
vector<double> doubleVector;
```

The type that makes up the vector appears in angle brackets (`<` and `>`). The first line creates a vector of integers named `count`; the second creates a vector of `double`s. Although these statements are legal, they are not very useful because they create vectors that have no elements (their size is zero). It is more common to specify the size of the vector in parentheses:

```
vector<int> count (4);
```

The syntax here is a little odd; it looks like a combination of a variable declarations and a function call. In fact, that's exactly what it is. The function we are invoking is a `vector` constructor. A **constructor** is a special function that creates new objects and initializes their instance variables. In this case, the constructor takes a single argument, which is the size of the new vector.

The following figure shows how vectors are represented in state diagrams:



The large numbers inside the boxes are the **elements** of the vector. The small numbers outside the boxes are the indices used to identify each box. When you allocate a new vector, the elements are not initialized. They could contain any values.

There is another constructor for `vector`s that takes two parameters; the second is a "fill value," the value that will be assigned to each of the elements.

```
vector<int> count (4, 0);
```

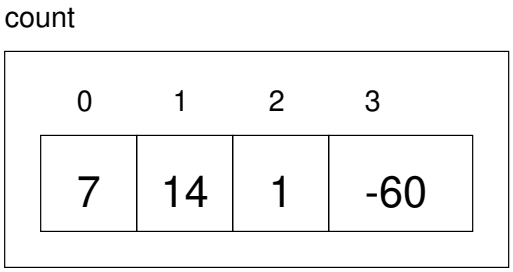
This statement creates a vector of four elements and initializes all of them to zero.

10.1 Accessing elements

The `[]` operator reads and writes the elements of a vector in much the same way it accesses the characters in an `string`. As with `string`s, the indices start at zero, so `count[0]` refers to the "zeroeth" element of the vector, and `count[1]` refers to the "oneth" element. You can use the `[]` operator anywhere in an expression:

```
count[0] = 7;
count[1] = count[0] * 2;
count[2]++;
count[3] -= 60;
```

All of these are legal assignment statements. Here is the effect of this code fragment:



Since elements of this vector are numbered from 0 to 3, there is no element with the index 4. It is a common error to go beyond the bounds of a vector, which causes a run-time error. The program outputs an error message like "Illegal vector index", and then quits.

You can use any expression as an index, as long as it has type `int`. One of the most common ways to index a vector is with a loop variable. For example:

```
int i = 0;
while (i < 4) {
  cout << count[i] << endl;
  i++;
}
```

This `while` loop counts from 0 to 4; when the loop variable `i` is 4, the condition fails and the loop terminates. Thus, the body of the loop is only executed when `i` is 0, 1, 2 and 3.

Each time through the loop we use `i` as an index into the vector, outputting the `i`th element. This type of vector traversal is very common. Vectors and loops go together like fava beans and a nice Chianti.

10.2 Copying vectors

There is one more constructor for `vector`s, which is called a copy constructor because it takes one `vector` as an argument and creates a new vector that is the same size, with the same elements.

```
vector<int> copy (count);
```

Although this syntax is legal, it is almost never used for `vector`s because there is a better alternative:

```
vector<int> copy = count;
```

The `=` operator works on `vector`s in pretty much the way you would expect.

10.3 `for each` loops

FIXME – change to be more vector specific (change to be FOR EACH loops instead)

```
int i;
for (i = 0; i < 4; i++) {
  cout << count[i] << endl;
}
```

is equivalent to

```
int i = 0;
while (i < 4) {
  cout << count[i] << endl;
  i++;
}
```

10.4 Vector size

There are a few functions you can invoke on an `vector`. One of them is very useful, though: `size()`. Not surprisingly, it returns the size of the vector (the number of elements).

It is a good idea to use this value as the upper bound of a loop, rather than a constant. That way, if the size of the vector changes, you won't have to go through the program changing all the loops; they will work correctly for any size vector.

```
int i;
for (i = 0; i < count.size(); i++) {
  cout << count[i] << endl;
}
```

The last time the body of the loop gets executed, the value of `i` is `count.size() - 1`, which is the index of the last element. When `i` is equal to `count.size()`, the condition fails and the body is not executed, which is a good thing, since it would cause a run-time error. One thing that we should notice here is that the size() function is called every time the loop is executed. Calling a function again and again reduces execution speed, so it would be better to store the size in some variable by calling the `size()` function before the loop begins, and use this variable to check for the last element. You can try this program as an excercise.

10.5 Vector functions

The best feature of a vector is its resizeability A vector, once declared, can be resized from anywhere within the program. Suppose we have a situation where we input numbers from the user and store them in a vector till he inputs `-1`, and then display them. In such a case, we do not know the size of the vector beforehand. So we need to add new values to the end of a vector as the user inputs them. We can use the vector function `push_back()` for that purpose.

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
```

```
    vector<int> values;
    int c,i,len;
    cin>>c;

    while(c != -1) {
      values.push_back(c);
      cin >> c;
    }
    len=values.size();
    for(i = 0; i < len; i++) {
      cout << values[i] << endl;
    }
  }
```

10.6 Random numbers

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

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

C++ provides a function called `random` that generates pseudorandom numbers. It is declared in the header file `cstdlib`, which contains a variety of "standard library" functions, hence the name.

The return value from `random` is an integer between 0 and `RAND_MAX`, where `RAND_MAX` is a large number (about 2 billion on my computer) also defined in the header file. Each time you call `random` you get a different randomly-generated number. To see a sample, run this loop:

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

int main ()
{
  for (int i = 0; i < 4; i++) {
    int x = random ();
    cout << x << endl;
  }
```

```
  return 0;
}
```

On my machine I got the following output:

```
1804289383
846930886
1681692777
1714636915
```

You will probably get something similar, but different, on yours.

Of course, we don't always want to work with gigantic integers. More often we want to generate integers between 0 and some upper bound. A simple way to do that is with the modulus operator. For example:

```
int x = random ();
int y = x % upperBound;
```

Since `y` is the remainder when `x` is divided by `upperBound`, the only possible values for `y` are between 0 and `upperBound - 1`, including both end points. Keep in mind, though, that `y` will never be equal to `upperBound`.

It is also frequently useful to generate random floating-point values. A common way to do that is by dividing by `RAND_MAX`. For example:

```
int x = random ();
double y = double(x) / RAND_MAX;
```

This code sets `y` to a random value between 0.0 and 1.0, including both end points. As an exercise, you might want to think about how to generate a random floating-point value in a given range; for example, between 100.0 and 200.0.

10.7 Statistics

The numbers generated by `random` are supposed to be distributed uniformly. That means that each value in the range should be equally likely. If we count the number of times each value appears, it should be roughly the same for all values, provided that we generate a large number of values.

In the next few sections, we will write programs that generate a sequence of random numbers and check whether this property holds true.

10.8 Vector of random numbers

The first step is to generate a large number of random values and store them in a vector. By "large number," of course, I mean 20. It's always a good idea to start with a manageable number, to help with debugging, and then increase it later.

The following function takes a single argument, the size of the vector. It allocates a new vector of `int`s, and fills it with random values between 0 and `upperBound-1`.

```
vector<int> randomVector (int n, int upperBound) {
  vector<int> vec (n);
  for (int i = 0; i<vec.size(); i++) {
    vec[i] = random () % upperBound;
  }
  return vec;
}
```

The return type is `vector<int>`, which means that this function returns a vector of integers. To test this function, it is convenient to have a function that outputs the contents of a vector.

```
void printVector (const vector<int>& vec) {
  for (int i = 0; i<vec.size(); i++) {
    cout << vec[i] << " ";
  }
}
```

Notice that it is legal to pass `vector`s by reference. In fact it is quite common, since it makes it unnecessary to copy the vector. Since `printVector` does not modify the vector, we declare the parameter `const`.

The following code generates a vector and outputs it:

```
  int numValues = 20;
  int upperBound = 10;
  vector<int> vector = randomVector (numValues, upperBound);
  printVector (vector);
```

On my machine the output is

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

which is pretty random-looking. Your results may differ.

If these numbers are really random, we expect each digit to appear the same number of times—twice each. In fact, the number 6 appears five times, and the numbers 4 and 8 never appear at all.

Do these results mean the values are not really uniform? It's hard to tell. With so few values, the chances are slim that we would get exactly what we

expect. But as the number of values increases, the outcome should be more predictable.

To test this theory, we'll write some programs that count the number of times each value appears, and then see what happens when we increase `numValues`.

10.9 Counting

A good approach to problems like this is to think of simple functions that are easy to write, and that might turn out to be useful. Then you can combine them into a solution. This approach is sometimes called **bottom-up design**. Of course, it is not easy to know ahead of time which functions are likely to be useful, but as you gain experience you will have a better idea.

Also, it is not always obvious what sort of things are easy to write, but a good approach is to look for subproblems that fit a pattern you have seen before.

Back in Section 7.12 we looked at a loop that traversed a string and counted the number of times a given letter appeared. You can think of this program as an example of a pattern called "traverse and count." The elements of this pattern are:

- A set or container that can be traversed, like a string or a vector.
- A test that you can apply to each element in the container.
- A counter that keeps track of how many elements pass the test.

In this case, I have a function in mind called `howMany` that counts the number of elements in a vector that equal a given value. The parameters are the vector and the integer value we are looking for. The return value is the number of times the value appears.

```
int howMany (const vector<int>& vec, int value) {
  int count = 0;
  for (int i=0; i< vec.size(); i++) {
    if (vec[i] == value) count++;
  }
  return count;
}
```

10.10 Checking the other values

`howMany` only counts the occurrences of a particular value, and we are interested in seeing how many times each value appears. We can solve that problem with a loop:

```
  int numValues = 20;
  int upperBound = 10;
```

```
vector<int> vector = randomVector (numValues, upperBound);

cout << "value\thowMany";

for (int i = 0; i<upperBound; i++) {
  cout << i << '\t' << howMany (vector, i) << endl;
}
```

Notice that it is legal to declare a variable inside a `for` statement. This syntax is sometimes convenient, but you should be aware that a variable declared inside a loop only exists inside the loop. If you try to refer to `i` later, you will get a compiler error.

This code uses the loop variable as an argument to `howMany`, in order to check each value between 0 and 9, in order. The result is:

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

Again, it is hard to tell if the digits are really appearing equally often. If we increase `numValues` to 100,000 we get the following:

```
value   howMany
0       10130
1       10072
2       9990
3       9842
4       10174
5       9930
6       10059
7       9954
8       9891
9       9958
```

In each case, the number of appearances is within about 1% of the expected value (10,000), so we conclude that the random numbers are probably uniform.

10.11 A histogram

It is often useful to take the data from the previous tables and store them for later access, rather than just print them. What we need is a way to store 10 integers. We could create 10 integer variables with names like `howManyOnes`, `howManyTwos`, etc. But that would require a lot of typing, and it would be a real pain later if we decided to change the range of values.

A better solution is to use a vector with size 10. That way we can create all ten storage locations at once and we can access them using indices, rather than ten different names. Here's how:

```
int numValues = 100000;
int upperBound = 10;
vector<int> vector = randomVector (numValues, upperBound);
vector<int> histogram (upperBound);

for (int i = 0; i<upperBound; i++) {
  int count = howMany (vector, i);
  histogram[i] = count;
}
```

I called the vector **histogram** because that's a statistical term for a vector of numbers that counts the number of appearances of a range of values.

The tricky thing here is that I am using the loop variable in two different ways. First, it is an argument to `howMany`, specifying which value I am interested in. Second, it is an index into the histogram, specifying which location I should store the result in.

10.12 A single-pass solution

Although this code works, it is not as efficient as it could be. Every time it calls `howMany`, it traverses the entire vector. In this example we have to traverse the vector ten times!

It would be better to make a single pass through the vector. For each value in the vector we could find the corresponding counter and increment it. In other words, we can use the value from the vector as an index into the histogram. Here's what that looks like:

```
vector<int> histogram (upperBound, 0);

for (int i = 0; i<numValues; i++) {
  int index = vector[i];
  histogram[index]++;
}
```

The first line initializes the elements of the histogram to zeroes. That way, when we use the increment operator (`++`) inside the loop, we know we are starting from zero. Forgetting to initialize counters is a common error.

As an exercise, encapsulate this code in a function called `histogram` that takes a vector and the range of values in the vector (in this case 0 through 10), and that returns a histogram of the values in the vector.

10.13 Random seeds

If you have run the code in this chapter a few times, you might have noticed that you are getting the same "random" values every time. That's not very random!

One of the properties of pseudorandom number generators is that if they start from the same place they will generate the same sequence of values. The starting place is called a **seed**; by default, C++ uses the same seed every time you run the program.

While you are debugging, it is often helpful to see the same sequence over and over. That way, when you make a change to the program you can compare the output before and after the change.

If you want to choose a different seed for the random number generator, you can use the `srand` function. It takes a single argument, which is an integer between 0 and `RAND_MAX`.

For many applications, like games, you want to see a different random sequence every time the program runs. A common way to do that is to use a library function like `gettimeofday` to generate something reasonably unpredictable and unrepeatable, like the number of milliseconds since the last second tick, and use that number as a seed. The details of how to do that depend on your development environment.

10.14 Glossary

vector: A named collection of values, where all the values have the same type, and each value is identified by an index.

element: One of the values in a vector. The `[]` operator selects elements of a vector.

index: An integer variable or value used to indicate an element of a vector.

constructor: A special function that creates a new object and initializes its instance variables.

deterministic: A program that does the same thing every time it is run.

pseudorandom: A sequence of numbers that appear to be random, but which are actually the product of a deterministic computation.

seed: A value used to initialize a random number sequence. Using the same seed should yield the same sequence of values.

bottom-up design: A method of program development that starts by writing small, useful functions and then assembling them into larger solutions.

histogram: A vector of integers where each integer counts the number of values that fall into a certain range.

Chapter 11

Member functions

11.1 Objects and functions

C++ is generally considered an object-oriented programming language, which means that it provides features that support object-oriented programming.

It's not easy to define object-oriented programming, but we have already seen some features of it:

1. Programs are made up of a collection of structure definitions and function definitions, where most of the functions operate on specific kinds of structures (or objects).

2. Each structure definition corresponds to some object or concept in the real world, and the functions that operate on that structure correspond to the ways real-world objects interact.

For example, the `Time` structure we defined in Chapter 9 obviously corresponds to the way people record the time of day, and the operations we defined correspond to the sorts of things people do with recorded times. Similarly, the `Point` and `Rectangle` structures correspond to the mathematical concept of a point and a rectangle.

So far, though, we have not taken advantage of the features C++ provides to support object-oriented programming. Strictly speaking, these features are not necessary. For the most part they provide an alternate syntax for doing things we have already done, but in many cases the alternate syntax is more concise and more accurately conveys the structure of the program.

For example, in the `Time` program, there is no obvious connection between the structure definition and the function definitions that follow. With some examination, it is apparent that every function takes at least one `Time` structure as a parameter.

This observation is the motivation for **member functions**. Member function differ from the other functions we have written in two ways:

1. When we call the function, we **invoke** it on an object, rather than just call it. People sometimes describe this process as "performing an operation on an object," or "sending a message to an object."

2. The function is *declared* inside the `struct` definition, in order to make the relationship between the structure and the function explicit.

In the next few sections, we will take the functions from Chapter 9 and transform them into member functions. One thing you should realize is that this transformation is purely mechanical; in other words, you can do it just by following a sequence of steps.

As I said, anything that can be done with a member function can also be done with a nonmember function (sometimes called a **free-standing** function). But sometimes there is an advantage to one over the other. If you are comfortable converting from one form to another, you will be able to choose the best form for whatever you are doing.

11.2 `print`

In Chapter 9 we defined a structure named `Time` and wrote a function named `printTime`

```
struct Time {
  int hour, minute;
  double second;
};

void printTime (const Time& time) {
  cout << time.hour << ":" << time.minute << ":" << time.second << endl;
}
```

To call this function, we had to pass a `Time` object as a parameter.

```
  Time currentTime = { 9, 14, 30.0 };
  printTime (currentTime);
```

To make `printTime` into a member function, the first step is to change the name of the function from `printTime` to `Time::print`. The `::` operator separates the name of the structure from the name of the function; together they indicate that this is a function named `print` that can be invoked on a `Time` structure.

The next step is to eliminate the parameter. Instead of passing an object as an argument, we are going to invoke the function on an object.

As a result, inside the function, we no longer have a parameter named `time`. Instead, we have a **current object**, which is the object the function is invoked on. We can refer to the current object using the C++ keyword `this`.

One thing that makes life a little difficult is that `this` is actually a **pointer** to a structure, rather than a structure itself. A pointer is similar to a reference,

but I don't want to go into the details of using pointers yet. The only pointer operation we need for now is the * operator, which converts a structure pointer into a structure. In the following function, we use it to assign the value of `this` to a local variable named `time`:

```
void Time::print () {
  Time time = *this;
  cout << time.hour << ":" << time.minute << ":" << time.second << endl;
}
```

The first two lines of this function changed quite a bit as we transformed it into a member function, but notice that the output statement itself did not change at all.

In order to invoke the new version of `print`, we have to invoke it on a `Time` object:

```
  Time currentTime = { 9, 14, 30.0 };
  currentTime.print ();
```

The last step of the transformation process is that we have to declare the new function inside the structure definition:

```
struct Time {
  int hour, minute;
  double second;

  void Time::print ();
};
```

A **function declaration** looks just like the first line of the function definition, except that it has a semi-colon at the end. The declaration describes the **interface** of the function; that is, the number and types of the arguments, and the type of the return value. This works the same way as the function prototype that we saw in Section 5.6.

When you declare a function, you are making a promise to the compiler that you will, at some point later on in the program, provide a definition for the function. This definition is sometimes called the **implementation** of the function, since it contains the details of how the function works. If you omit the definition, or provide a definition that has an interface different from what you promised, the compiler will complain.

11.3 Implicit variable access

Actually, the new version of `Time::print` is more complicated than it needs to be. We don't really need to create a local variable in order to refer to the instance variables of the current object.

If the function refers to `hour`, `minute`, or `second`, all by themselves with no dot notation, C++ knows that it must be referring to the current object. So we could have written:

```
void Time::print ()
{
  cout << hour << ":" << minute << ":" << second << endl;
}
```

This kind of variable access is called "implicit" because the name of the object does not appear explicitly. Features like this are one reason member functions are often more concise than nonmember functions.

11.4 Another example

Let's convert `increment` to a member function. Again, we are going to transform one of the parameters into the implicit parameter called `this`. Then we can go through the function and make all the variable accesses implicit.

```
void Time::increment (double secs) {
  second += secs;

  while (second >= 60.0) {
    second -= 60.0;
    minute += 1;
  }
  while (minute >= 60) {
    minute -= 60.0;
    hour += 1;
  }
}
```

By the way, remember that this is not the most efficient implementation of this function. If you didn't do it back in Chapter 9, you should write a more efficient version now.

To declare the function, we can just copy the first line into the structure definition:

```
struct Time {
  int hour, minute;
  double second;

  void Time::print ();
  void Time::increment (double secs);
};
```

And again, to call it, we have to invoke it on a `Time` object:

```
  Time currentTime = { 9, 14, 30.0 };
  currentTime.increment (500.0);
  currentTime.print ();
```

The output of this program is `9:22:50`.

11.5 Yet another example

The original version of `convertToSeconds` looked like this:

```
double convertToSeconds (const Time& time) {
  int minutes = time.hour * 60 + time.minute;
  double seconds = minutes * 60 + time.second;
  return seconds;
}
```

It is straightforward to convert this to a member function:

```
double Time::convertToSeconds () const {
  int minutes = hour * 60 + minute;
  double seconds = minutes * 60 + second;
  return seconds;
}
```

The interesting thing here is that the implicit parameter should be declared `const`, since we don't modify it in this function. But it is not obvious where we should put information about a parameter that doesn't exist. The answer, as you can see in the example, is after the parameter list (which is empty in this case).

The `print` function in the previous section should also declare that the implicit parameter is `const`.

11.6 A more complicated example

Although the process of transforming functions into member functions is mechanical, there are some oddities. For example, `after` operates on two `Time` structures, not just one, and we can't make both of them implicit. Instead, we have to invoke the function on one of them and pass the other as an argument.

Inside the function, we can refer to one of the them implicitly, but to access the instance variables of the other we continue to use dot notation.

```
bool Time::after (const Time& time2) const {
  if (hour > time2.hour) return true;
  if (hour < time2.hour) return false;

  if (minute > time2.minute) return true;
```

```
  if (minute < time2.minute) return false;

  if (second > time2.second) return true;
  return false;
}
```

To invoke this function:

```
  if (doneTime.after (currentTime)) {
    cout << "The bread will be done after it starts." << endl;
  }
```

You can almost read the invocation like English: "If the done-time is after the current-time, then..."

11.7 Constructors

Another function we wrote in Chapter 9 was `makeTime`:

```
Time makeTime (double secs) {
  Time time;
  time.hour = int (secs / 3600.0);
  secs -= time.hour * 3600.0;
  time.minute = int (secs / 60.0);
  secs -= time.minute * 60.0;
  time.second = secs;
  return time;
}
```

Of course, for every new type, we need to be able to create new objects. In fact, functions like `makeTime` are so common that there is a special function syntax for them. These functions are called **constructors** and the syntax looks like this:

```
Time::Time (double secs) {
  hour = int (secs / 3600.0);
  secs -= hour * 3600.0;
  minute = int (secs / 60.0);
  secs -= minute * 60.0;
  second = secs;
}
```

First, notice that the constructor has the same name as the class, and no return type. The arguments haven't changed, though.

Second, notice that we don't have to create a new time object, and we don't have to return anything. Both of these steps are handled automatically. We can refer to the new object—the one we are constructing—using the keyword `this`,

or implicitly as shown here. When we write values to `hour`, `minute` and `second`, the compiler knows we are referring to the instance variables of the new object.

To invoke the constructor, you use syntax that is a cross between a variable declaration and a function call:

```
Time time (seconds);
```

This statement declares that the variable `time` has type `Time`, and it invokes the constructor we just wrote, passing the value of `seconds` as an argument. The system allocates space for the new object and the constructor initializes its instance variables. The result is assigned to the variable `time`.

11.8 Initialize or construct?

Earlier we declared and initialized some `Time` structures using curly-braces:

```
Time currentTime = { 9, 14, 30.0 };
Time breadTime = { 3, 35, 0.0 };
```

Now, using constructors, we have a different way to declare and initialize:

```
Time time (seconds);
```

These two functions represent different programming styles, and different points in the history of C++. Maybe for that reason, the C++ compiler requires that you use one or the other, and not both in the same program.

If you define a constructor for a structure, then you have to use the constructor to initialize all new structures of that type. The alternate syntax using curly-braces is no longer allowed.

Fortunately, it is legal to overload constructors in the same way we overloaded functions. In other words, there can be more than one constructor with the same "name," as long as they take different parameters. Then, when we initialize a new object the compiler will try to find a constructor that takes the appropriate parameters.

For example, it is common to have a constructor that takes one parameter for each instance variable, and that assigns the values of the parameters to the instance variables:

```
Time::Time (int h, int m, double s)
{
  hour = h;  minute = m;  second = s;
}
```

To invoke this constructor, we use the same funny syntax as before, except that the arguments have to be two integers and a `double`:

```
Time currentTime (9, 14, 30.0);
```

11.9 One last example

The final example we'll look at is `addTime`:

```
Time addTime2 (const Time& t1, const Time& t2) {
  double seconds = convertToSeconds (t1) + convertToSeconds (t2);
  return makeTime (seconds);
}
```

We have to make several changes to this function, including:

1. Change the name from `addTime` to `Time::add`.

2. Replace the first parameter with an implicit parameter, which should be declared `const`.

3. Replace the use of `makeTime` with a constructor invocation.

Here's the result:

```
Time Time::add (const Time& t2) const {
  double seconds = convertToSeconds () + t2.convertToSeconds ();
  Time time (seconds);
  return time;
}
```

The first time we invoke `convertToSeconds`, there is no apparent object! Inside a member function, the compiler assumes that we want to invoke the function on the current object. Thus, the first invocation acts on `this`; the second invocation acts on `t2`.

The next line of the function invokes the constructor that takes a single `double` as a parameter; the last line returns the resulting object.

11.10 Header files

It might seem like a nuisance to declare functions inside the structure definition and then define the functions later. Any time you change the interface to a function, you have to change it in two places, even if it is a small change like declaring one of the parameters `const`.

There is a reason for the hassle, though, which is that it is now possible to separate the structure definition and the functions into two files: the **header file**, which contains the structure definition, and the implementation file, which contains the functions.

Header files usually have the same name as the implementation file, but with the suffix `.h` instead of `.cpp`. For the example we have been looking at, the header file is called `Time.h`, and it contains the following:

```
struct Time {
  // instance variables
  int hour, minute;
  double second;

  // constructors
  Time (int hour, int min, double secs);
  Time (double secs);

  // modifiers
  void increment (double secs);

  // functions
  void print () const;
  bool after (const Time& time2) const;
  Time add (const Time& t2) const;
  double convertToSeconds () const;
};
```

Notice that in the structure definition I don't really have to include the prefix `Time::` at the beginning of every function name. The compiler knows that we are declaring functions that are members of the `Time` structure.

`Time.cpp` contains the definitions of the member functions (I have elided the function bodies to save space):

```
#include <iostream>
using namespace std;
#include "Time.h"

Time::Time (int h, int m, double s)  ...

Time::Time (double secs) ...

void Time::increment (double secs) ...

void Time::print () const ...

bool Time::after (const Time& time2) const ...

Time Time::add (const Time& t2) const ...

double Time::convertToSeconds () const ...
```

In this case the definitions in `Time.cpp` appear in the same order as the declarations in `Time.h`, although it is not necessary.

On the other hand, it is necessary to include the header file using an `include` statement. That way, while the compiler is reading the function definitions, it

knows enough about the structure to check the code and catch errors.

Finally, `main.cpp` contains the function `main` along with any functions we want that are not members of the `Time` structure (in this case there are none):

```
#include <iostream>
using namespace std;
#include "Time.h"

int main ()
{
  Time currentTime (9, 14, 30.0);
  currentTime.increment (500.0);
  currentTime.print ();

  Time breadTime (3, 35, 0.0);
  Time doneTime = currentTime.add (breadTime);
  doneTime.print ();

  if (doneTime.after (currentTime)) {
    cout << "The bread will be done after it starts." << endl;
  }
  return 0;
}
```

Again, `main.cpp` has to include the header file.

It may not be obvious why it is useful to break such a small program into three pieces. In fact, most of the advantages come when we are working with larger programs:

Reuse: Once you have written a structure like `Time`, you might find it useful in more than one program. By separating the definition of `Time` from `main.cpp`, you make is easy to include the `Time` structure in another program.

Managing interactions: As systems become large, the number of interactions between components grows and quickly becomes unmanageable. It is often useful to minimize these interactions by separating modules like `Time.cpp` from the programs that use them.

Separate compilation: Separate files can be compiled separately and then linked into a single program later. The details of how to do this depend on your programming environment. As the program gets large, separate compilation can save a lot of time, since you usually need to compile only a few files at a time.

For small programs like the ones in this book, there is no great advantage to splitting up programs. But it is good for you to know about this feature,

especially since it explains one of the statements that appeared in the first program we wrote:

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

`iostream` is the header file that contains declarations for `cin` and `cout` and the functions that operate on them. When you compile your program, you need the information in that header file.

The implementations of those functions are stored in a library, sometimes called the "Standard Library" that gets linked to your program automatically. The nice thing is that you don't have to recompile the library every time you compile a program. For the most part the library doesn't change, so there is no reason to recompile it.

11.11 Glossary

member function: A function that operates on an object that is passed as an implicit parameter named `this`.

nonmember function: A function that is not a member of any structure definition. Also called a "free-standing" function.

invoke: To call a function "on" an object, in order to pass the object as an implicit parameter.

current object: The object on which a member function is invoked. Inside the member function, we can refer to the current object implicitly, or by using the keyword `this`.

this: A keyword that refers to the current object. `this` is a pointer, which makes it difficult to use, since we do not cover pointers in this book.

interface: A description of how a function is used, including the number and types of the parameters and the type of the return value.

function declaration: A statement that declares the interface to a function without providing the body. Declarations of member functions appear inside structure definitions even if the definitions appear outside.

implementation: The body of a function, or the details of how a function works.

constructor: A special function that initializes the instance variables of a newly-created object.

Chapter 12

Vectors of Objects

12.1 Composition

By now we have seen several examples of composition (the ability to combine language features in a variety of arrangements). One of the first examples we saw was using a function invocation as part of an expression. Another example is the nested structure of statements: you can put an `if` statement within a `while` loop, or within another `if` statement, etc.

Having seen this pattern, and having learned about vectors and objects, you should not be surprised to learn that you can have vectors of objects. In fact, you can also have objects that contain vectors (as instance variables); you can have vectors that contain vectors; you can have objects that contain objects, and so on.

In the next two chapters we will look at some examples of these combinations, using `Card` objects as a case study.

12.2 `Card` objects

If you are not familiar with common playing cards, now would be a good time to get a deck, or else this chapter might not make much sense. There are 52 cards in a deck, each of which belongs to one of four suits and one of 13 ranks. The suits are Spades, Hearts, Diamonds and Clubs (in descending order in Bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen and King. Depending on what game you are playing, the rank of the Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is pretty obvious what the instance variables should be: `rank` and `suit`. It is not as obvious what type the instance variables should be. One possibility is `apstring`s, containing things like `"Spade"` for suits and `"Queen"` for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. By "encode," I do not mean what some people think, which is to encrypt, or translate into a secret code. What a computer scientist means by "encode" is something like "define a mapping between a sequence of numbers and the things I want to represent." For example,

Spades	$\mapsto$	3
Hearts	$\mapsto$	2
Diamonds	$\mapsto$	1
Clubs	$\mapsto$	0

The symbol $\mapsto$ is mathematical notation for "maps to." The obvious feature of this mapping is that the suits map to integers in order, so we can compare suits by comparing integers. The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack	$\mapsto$	11
Queen	$\mapsto$	12
King	$\mapsto$	13

The reason I am using mathematical notation for these mappings is that they are not part of the C++ program. They are part of the program design, but they never appear explicitly in the code. The class definition for the `Card` type looks like this:

```
struct Card
{
  int suit, rank;

  Card ();
  Card (int s, int r);
};

Card::Card () {
  suit = 0;  rank = 0;
}

Card::Card (int s, int r) {
  suit = s;  rank = r;
}
```

There are two constructors for `Card`s. You can tell that they are constructors because they have no return type and their name is the same as the name of the structure. The first constructor takes no arguments and initializes the instance variables to a useless value (the zero of clubs).

The second constructor is more useful. It takes two parameters, the suit and rank of the card.

The following code creates an object named `threeOfClubs` that represents the 3 of Clubs:

```
Card threeOfClubs (0, 3);
```

The first argument, `0` represents the suit Clubs, the second, naturally, represents the rank 3.

12.3 The printCard function

When you create a new type, the first step is usually to declare the instance variables and write constructors. The second step is often to write a function that prints the object in human-readable form.

In the case of `Card` objects, "human-readable" means that we have to map the internal representation of the rank and suit onto words. A natural way to do that is with a vector of `apstring`s. You can create a vector of `apstring`s the same way you create an vector of other types:

```
apvector<apstring> suits (4);
```

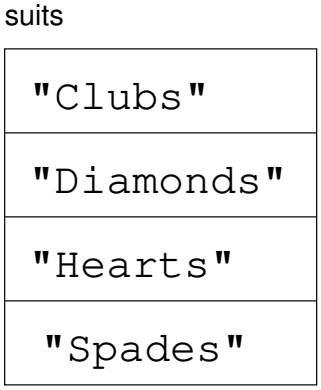
Of course, in order to use `apvector`s and `apstring`s, you will have to include the header files for both[1].

To initialize the elements of the vector, we can use a series of assignment statements.

```
suits[0] = "Clubs";
suits[1] = "Diamonds";
suits[2] = "Hearts";
suits[3] = "Spades";
```

A state diagram for this vector looks like this:

[1] `apvector`s are a little different from `apstring`s in this regard. The file `apvector.cpp` contains a template that allows the compiler to create vectors of various kinds. The first time you use a vector of integers, the compiler generates code to support that kind of vector. If you use a vector of `apstring`s, the compiler generates different code to handle that kind of vector. As a result, it is usually sufficient to include the header file `apvector.h`; you do not have to compile `apvector.cpp` at all! Unfortunately, if you do, you are likely to get a long stream of error messages. I hope this footnote helps you avoid an unpleasant surprise, but the details in your development environment may differ.



We can build a similar vector to decode the ranks. Then we can select the appropriate elements using the `suit` and `rank` as indices. Finally, we can write a function called `print` that outputs the card on which it is invoked:

```
void Card::print () const
{
  apvector<apstring> suits (4);
  suits[0] = "Clubs";
  suits[1] = "Diamonds";
  suits[2] = "Hearts";
  suits[3] = "Spades";

  apvector<apstring> ranks (14);
  ranks[1] = "Ace";
  ranks[2] = "2";
  ranks[3] = "3";
  ranks[4] = "4";
  ranks[5] = "5";
  ranks[6] = "6";
  ranks[7] = "7";
  ranks[8] = "8";
  ranks[9] = "9";
  ranks[10] = "10";
  ranks[11] = "Jack";
  ranks[12] = "Queen";
  ranks[13] = "King";

  cout << ranks[rank] << " of " << suits[suit] << endl;
}
```

The expression `suits[suit]` means "use the instance variable `suit` from the current object as an index into the vector named `suits`, and select the appropriate string."

Because `print` is a `Card` member function, it can refer to the instance variables of the current object implicitly (without having to use dot notation to specify the object). The output of this code

```
Card card (1, 11);
card.print ();
```

is `Jack of Diamonds`.

You might notice that we are not using the zeroeth element of the `ranks` vector. That's because the only valid ranks are 1–13. By leaving an unused element at the beginning of the vector, we get an encoding where 2 maps to "2", 3 maps to "3", etc. From the point of view of the user, it doesn't matter what the encoding is, since all input and output uses human-readable formats. On the other hand, it is often helpful for the programmer if the mappings are easy to remember.

12.4 The `equals` function

In order for two cards to be equal, they have to have the same rank and the same suit. Unfortunately, the `==` operator does not work for user-defined types like `Card`, so we have to write a function that compares two cards. We'll call it `equals`. It is also possible to write a new definition for the `==` operator, but we will not cover that in this book.

It is clear that the return value from `equals` should be a boolean that indicates whether the cards are the same. It is also clear that there have to be two `Card`s as parameters. But we have one more choice: should `equals` be a member function or a free-standing function?

As a member function, it looks like this:

```
bool Card::equals (const Card& c2) const
{
  return (rank == c2.rank && suit == c2.suit);
}
```

To use this function, we have to invoke it on one of the cards and pass the other as an argument:

```
Card card1 (1, 11);
Card card2 (1, 11);

if (card1.equals(card2)) {
  cout << "Yup, that's the same card." << endl;
}
```

This method of invocation always seems strange to me when the function is something like `equals`, in which the two arguments are symmetric. What I

mean by symmetric is that it does not matter whether I ask "Is A equal to B?" or "Is B equal to A?" In this case, I think it looks better to rewrite `equals` as a nonmember function:

```
bool equals (const Card& c1, const Card& c2)
{
  return (c1.rank == c2.rank && c1.suit == c2.suit);
}
```

When we call this version of the function, the arguments appear side-by-side in a way that makes more logical sense, to me at least.

```
  if (equals (card1, card2)) {
    cout << "Yup, that's the same card." << endl;
  }
```

Of course, this is a matter of taste. My point here is that you should be comfortable writing both member and nonmember functions, so that you can choose the interface that works best depending on the circumstance.

12.5 The `isGreater` function

For basic types like `int` and `double`, there are comparison operators that compare values and determine when one is greater or less than another. These operators (`<` and `>` and the others) don't work for user-defined types. Just as we did for the `==` operator, we will write a comparison function that plays the role of the `>` operator. Later, we will use this function to sort a deck of cards.

Some sets are totally ordered, which means that you can compare any two elements and tell which is bigger. For example, the integers and the floating-point numbers are totally ordered. Some sets are unordered, which means that there is no meaningful way to say that one element is bigger than another. For example, the fruits are unordered, which is why we cannot compare apples and oranges. As another example, the `bool` type is unordered; we cannot say that `true` is greater than `false`.

The set of playing cards is partially ordered, which means that sometimes we can compare cards and sometimes not. For example, I know that the 3 of Clubs is higher than the 2 of Clubs because it has higher rank, and the 3 of Diamonds is higher than the 3 of Clubs because it has higher suit. But which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit.

In order to make cards comparable, we have to decide which is more important, rank or suit. To be honest, the choice is completely arbitrary. For the sake of choosing, I will say that suit is more important, because when you buy a new deck of cards, it comes sorted with all the Clubs together, followed by all the Diamonds, and so on.

With that decided, we can write `isGreater`. Again, the arguments (two `Card`s) and the return type (boolean) are obvious, and again we have to choose between a member function and a nonmember function. This time, the arguments are not symmetric. It matters whether we want to know "Is A greater than B?" or "Is B greater than A?" Therefore I think it makes more sense to write `isGreater` as a member function:

```
bool Card::isGreater (const Card& c2) const
{
  // first check the suits
  if (suit > c2.suit) return true;
  if (suit < c2.suit) return false;

  // if the suits are equal, check the ranks
  if (rank > c2.rank) return true;
  if (rank < c2.rank) return false;

  // if the ranks are also equal, return false
  return false;
}
```

Then when we invoke it, it is obvious from the syntax which of the two possible questions we are asking:

```
  Card card1 (2, 11);
  Card card2 (1, 11);

  if (card1.isGreater (card2)) {
    card1.print ();
    cout << "is greater than" << endl;
    card2.print ();
  }
```

You can almost read it like English: "If card1 isGreater card2 ..." The output of this program is

```
Jack of Hearts
is greater than
Jack of Diamonds
```

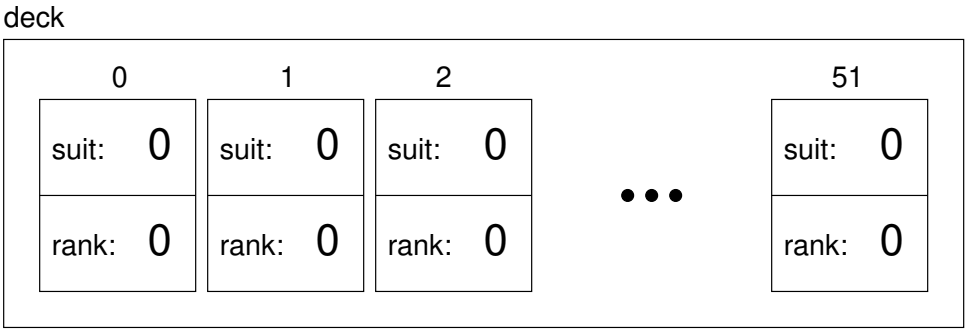
According to `isGreater`, aces are less than deuces (2s). As an exercise, fix it so that aces are ranked higher than Kings, as they are in most card games.

12.6 Vectors of cards

The reason I chose `Cards` as the objects for this chapter is that there is an obvious use for a vector of cards—a deck. Here is some code that creates a new deck of 52 cards:

```
apvector<Card> deck (52);
```

Here is the state diagram for this object:



The three dots represent the 48 cards I didn't feel like drawing. Keep in mind that we haven't initialized the instance variables of the cards yet. In some environments, they will get initialized to zero, as shown in the figure, but in others they could contain any possible value.

One way to initialize them would be to pass a `Card` as a second argument to the constructor:

```
Card aceOfSpades (3, 1);
apvector<Card> deck (52, aceOfSpades);
```

This code builds a deck with 52 identical cards, like a special deck for a magic trick. Of course, it makes more sense to build a deck with 52 different cards in it. To do that we use a nested loop.

The outer loop enumerates the suits, from 0 to 3. For each suit, the inner loop enumerates the ranks, from 1 to 13. Since the outer loop iterates 4 times, and the inner loop iterates 13 times, the total number of times the body is executed is 52 (13 times 4).

```
int i = 0;
for (int suit = 0; suit <= 3; suit++) {
  for (int rank = 1; rank <= 13; rank++) {
    deck[i].suit = suit;
    deck[i].rank = rank;
    i++;
  }
}
```

I used the variable `i` to keep track of where in the deck the next card should go.

Notice that we can compose the syntax for selecting an element from an array (the `[]` operator) with the syntax for selecting an instance variable from an object (the dot operator). The expression `deck[i].suit` means "the suit of the ith card in the deck".

As an exercise, encapsulate this deck-building code in a function called `buildDeck` that takes no parameters and that returns a fully-populated vector of `Card`s.

12.7 The printDeck function

Whenever you are working with vectors, it is convenient to have a function that prints the contents of the vector. We have seen the pattern for traversing a vector several times, so the following function should be familiar:

```
void printDeck (const apvector<Card>& deck) {
  for (int i = 0; i < deck.length(); i++) {
    deck[i].print ();
  }
}
```

By now it should come as no surprise that we can compose the syntax for vector access with the syntax for invoking a function.

Since `deck` has type `apvector<Card>`, an element of `deck` has type `Card`. Therefore, it is legal to invoke `print` on `deck[i]`.

12.8 Searching

The next function I want to write is `find`, which searches through a vector of `Card`s to see whether it contains a certain card. It may not be obvious why this function would be useful, but it gives me a chance to demonstrate two ways to go searching for things, a `linear` search and a `bisection` search.

Linear search is the more obvious of the two; it involves traversing the deck and comparing each card to the one we are looking for. If we find it we return the index where the card appears. If it is not in the deck, we return -1.

```
int find (const Card& card, const apvector<Card>& deck) {
  for (int i = 0; i < deck.length(); i++) {
    if (equals (deck[i], card)) return i;
  }
  return -1;
}
```

The loop here is exactly the same as the loop in `printDeck`. In fact, when I wrote the program, I copied it, which saved me from having to write and debug it twice.

Inside the loop, we compare each element of the deck to `card`. The function returns as soon as it discovers the card, which means that we do not have to traverse the entire deck if we find the card we are looking for. If the loop terminates without finding the card, we know the card is not in the deck and return `-1`.

To test this function, I wrote the following:

```
apvector<Card> deck = buildDeck ();

int index = card.find (deck[17]);
cout << "I found the card at index = " << index << endl;
```

The output of this code is

```
I found the card at index = 17
```

12.9 Bisection search

If the cards in the deck are not in order, there is no way to search that is faster than the linear search. We have to look at every card, since otherwise there is no way to be certain the card we want is not there.

But when you look for a word in a dictionary, you don't search linearly through every word. The reason is that the words are in alphabetical order. As a result, you probably use an algorithm that is similar to a bisection search:

1. Start in the middle somewhere.
2. Choose a word on the page and compare it to the word you are looking for.
3. If you found the word you are looking for, stop.
4. If the word you are looking for comes after the word on the page, flip to somewhere later in the dictionary and go to step 2.
5. If the word you are looking for comes before the word on the page, flip to somewhere earlier in the dictionary and go to step 2.

If you ever get to the point where there are two adjacent words on the page and your word comes between them, you can conclude that your word is not in the dictionary. The only alternative is that your word has been misfiled somewhere, but that contradicts our assumption that the words are in alphabetical order.

In the case of a deck of cards, if we know that the cards are in order, we can write a version of `find` that is much faster. The best way to write a bisection search is with a recursive function. That's because bisection is naturally recursive.

The trick is to write a function called `findBisect` that takes two indices as parameters, `low` and `high`, indicating the segment of the vector that should be searched (including both `low` and `high`).

1. To search the vector, choose an index between `low` and `high`, and call it `mid`. Compare the card at `mid` to the card you are looking for.

2. If you found it, stop.

3. If the card at `mid` is higher than your card, search in the range from `low` to `mid-1`.

4. If the card at `mid` is lower than your card, search in the range from `mid+1` to `high`.

Steps 3 and 4 look suspiciously like recursive invocations. Here's what this all looks like translated into C++:

```
int findBisect (const Card& card, const apvector<Card>& deck,
                int low, int high) {
  int mid = (high + low) / 2;

  // if we found the card, return its index
  if (equals (deck[mid], card)) return mid;

  // otherwise, compare the card to the middle card
  if (deck[mid].isGreater (card)) {
    // search the first half of the deck
    return findBisect (card, deck, low, mid-1);
  } else {
    // search the second half of the deck
    return findBisect (card, deck, mid+1, high);
  }
}
```

Although this code contains the kernel of a bisection search, it is still missing a piece. As it is currently written, if the card is not in the deck, it will recurse forever. We need a way to detect this condition and deal with it properly (by returning `-1`).

The easiest way to tell that your card is not in the deck is if there are *no* cards in the deck, which is the case if `high` is less than `low`. Well, there are still cards in the deck, of course, but what I mean is that there are no cards in the segment of the deck indicated by `low` and `high`.

With that line added, the function works correctly:

```
int findBisect (const Card& card, const apvector<Card>& deck,
                int low, int high) {

  cout << low << ", " << high << endl;

  if (high < low) return -1;

  int mid = (high + low) / 2;
```

```
  if (equals (deck[mid], card)) return mid;

  if (deck[mid].isGreater (card)) {
    return findBisect (card, deck, low, mid-1);
  } else {
    return findBisect (card, deck, mid+1, high);
  }
}
```

I added an output statement at the beginning so I could watch the sequence of recursive calls and convince myself that it would eventually reach the base case. I tried out the following code:

```
  cout << findBisect (deck, deck[23], 0, 51));
```

And got the following output:

```
0, 51
0, 24
13, 24
19, 24
22, 24
I found the card at index = 23
```

Then I made up a card that is not in the deck (the 15 of Diamonds), and tried to find it. I got the following:

```
0, 51
0, 24
13, 24
13, 17
13, 14
13, 12
I found the card at index = -1
```

These tests don't prove that this program is correct. In fact, no amount of testing can prove that a program is correct. On the other hand, by looking at a few cases and examining the code, you might be able to convince yourself.

The number of recursive calls is fairly small, typically 6 or 7. That means we only had to call `equals` and `isGreater` 6 or 7 times, compared to up to 52 times if we did a linear search. In general, bisection is much faster than a linear search, especially for large vectors.

Two common errors in recursive programs are forgetting to include a base case and writing the recursive call so that the base case is never reached. Either error will cause an infinite recursion, in which case C++ will (eventually) generate a run-time error.

12.10 Decks and subdecks

Looking at the interface to `findBisect`

```
int findBisect (const Card& card, const apvector<Card>& deck,
int low, int high) {
```

it might make sense to treat three of the parameters, `deck`, `low` and `high`, as a single parameter that specifies a **subdeck**.

This kind of thing is quite common, and I sometimes think of it as an **abstract parameter**. What I mean by "abstract," is something that is not literally part of the program text, but which describes the function of the program at a higher level.

For example, when you call a function and pass a vector and the bounds `low` and `high`, there is nothing that prevents the called function from accessing parts of the vector that are out of bounds. So you are not literally sending a subset of the deck; you are really sending the whole deck. But as long as the recipient plays by the rules, it makes sense to think of it, abstractly, as a subdeck.

There is one other example of this kind of abstraction that you might have noticed in Section 9.3, when I referred to an "empty" data structure. The reason I put "empty" in quotation marks was to suggest that it is not literally accurate. All variables have values all the time. When you create them, they are given default values. So there is no such thing as an empty object.

But if the program guarantees that the current value of a variable is never read before it is written, then the current value is irrelevant. Abstractly, it makes sense to think of such a variable as "empty."

This kind of thinking, in which a program comes to take on meaning beyond what is literally encoded, is a very important part of thinking like a computer scientist. Sometimes, the word "abstract" gets used so often and in so many contexts that it is hard to interpret. Nevertheless, abstraction is a central idea in computer science (as well as many other fields).

A more general definition of "abstraction" is "The process of modeling a complex system with a simplified description in order to suppress unnecessary details while capturing relevant behavior."

12.11 Glossary

encode: To represent one set of values using another set of values, by constructing a mapping between them.

abstract parameter: A set of parameters that act together as a single parameter.

Chapter 13

Objects of Vectors

13.1 Enumerated types

In the previous chapter I talked about mappings between real-world values like rank and suit, and internal representations like integers and strings. Although we created a mapping between ranks and integers, and between suits and integers, I pointed out that the mapping itself does not appear as part of the program.

Actually, C++ provides a feature called and **enumerated type** that makes it possible to (1) include a mapping as part of the program, and (2) define the set of values that make up the mapping. For example, here is the definition of the enumerated types `Suit` and `Rank`:

FIXME - change to class enums

```
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES };

enum Rank { ACE=1, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE,
TEN, JACK, QUEEN, KING };
```

By default, the first value in the enumerated type maps to 0, the second to 1, and so on. Within the `Suit` type, the value `CLUBS` is represented by the integer 0, `DIAMONDS` is represented by 1, etc.

The definition of `Rank` overrides the default mapping and specifies that `ACE` should be represented by the integer 1. The other values follow in the usual way.

Once we have defined these types, we can use them anywhere. For example, the instance variables `rank` and `suit` are can be declared with type `Rank` and `Suit`:

```
struct Card
{
  Rank rank;
  Suit suit;
```

```
  Card (Suit s, Rank r);
};
```

That the types of the parameters for the constructor have changed, too. Now, to create a card, we can use the values from the enumerated type as arguments:

```
Card card (DIAMONDS, JACK);
```

By convention, the values in enumerated types have names with all capital letters. This code is much clearer than the alternative using integers:

```
Card card (1, 11);
```

Because we know that the values in the enumerated types are represented as integers, we can use them as indices for a vector. Therefore the old `print` function will work without modification. We have to make some changes in `buildDeck`, though:

```
int index = 0;
for (Suit suit = CLUBS; suit <= SPADES; suit = Suit(suit+1)) {
  for (Rank rank = ACE; rank <= KING; rank = Rank(rank+1)) {
    deck[index].suit = suit;
    deck[index].rank = rank;
    index++;
  }
}
```

In some ways, using enumerated types makes this code more readable, but there is one complication. Strictly speaking, we are not allowed to do arithmetic with enumerated types, so `suit++` is not legal. On the other hand, in the expression `suit+1`, C++ automatically converts the enumerated type to integer. Then we can take the result and typecast it back to the enumerated type:

```
suit = Suit(suit+1);
rank = Rank(rank+1);
```

Actually, there is a better way to do this—we can define the `++` operator for enumerated types—but that is beyond the scope of this book.

13.2 `switch` statement

It's hard to mention enumerated types without mentioning `switch` statements, because they often go hand in hand. We saw switch statements in Section 3.9, but we only used it with characters and integers. It also works well with enumerated types. For example, to convert a `Suit` to the corresponding string, we could use something like:

```
  switch (suit) {
  case CLUBS:     return "Clubs";
  case DIAMONDS:  return "Diamonds";
  case HEARTS:    return "Hearts";
  case SPADES:    return "Spades";
  default:        return "Not a valid suit";
  }
```

In this case we don't need `break` statements because the `return` statements cause the flow of execution to return to the caller instead of falling through to the next case.

13.3 Decks

In the previous chapter, we worked with a vector of objects, but I also mentioned that it is possible to have an object that contains a vector as an instance variable. In this chapter I am going to create a new object, called a `Deck`, that contains a vector of `Card`s.

The structure definition looks like this

```
struct Deck {
  apvector<Card> cards;

  Deck (int n);
};

Deck::Deck (int size)
{
  apvector<Card> temp (size);
  cards = temp;
}
```

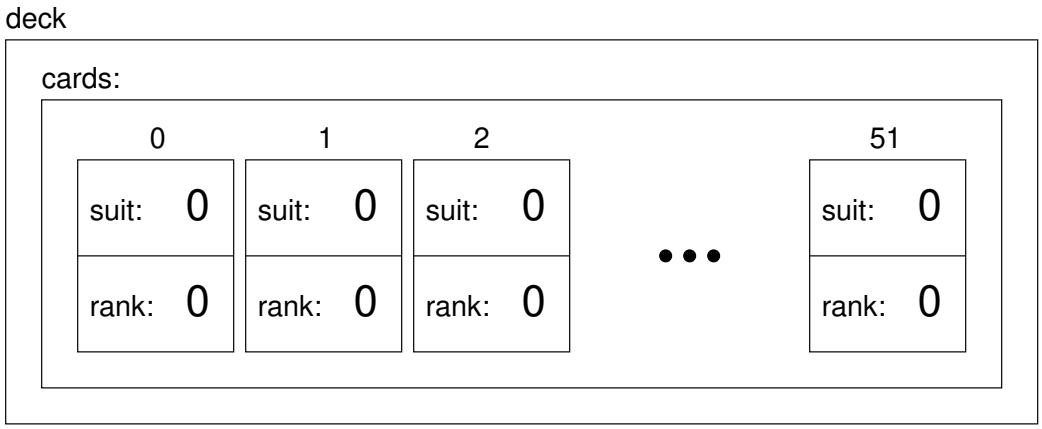
The name of the instance variable is `cards` to help distinguish the `Deck` object from the vector of `Card`s that it contains.

For now there is only one constructor. It creates a local variable named `temp`, which it initializes by invoking the constructor for the `apvector` class, passing the size as a parameter. Then it copies the vector from `temp` into the instance variable `cards`.

Now we can create a deck of cards like this:

```
  Deck deck (52);
```

Here is a state diagram showing what a `Deck` object looks like:



The object named `deck` has a single instance variable named `cards`, which is a vector of `Card` objects. To access the cards in a deck we have to compose the syntax for accessing an instance variable and the syntax for selecting an element from an array. For example, the expression `deck.cards[i]` is the ith card in the deck, and `deck.cards[i].suit` is its suit. The following loop

```
for (int i = 0; i<52; i++) {
  deck.cards[i].print();
}
```

demonstrates how to traverse the deck and output each card.

13.4 Another constructor

Now that we have a `Deck` object, it would be useful to initialize the cards in it. From the previous chapter we have a function called `buildDeck` that we could use (with a few adaptations), but it might be more natural to write a second `Deck` constructor.

```
Deck::Deck ()
{
  apvector<Card> temp (52);
  cards = temp;

  int i = 0;
  for (Suit suit = CLUBS; suit <= SPADES; suit = Suit(suit+1)) {
    for (Rank rank = ACE; rank <= KING; rank = Rank(rank+1)) {
      cards[i].suit = suit;
      cards[i].rank = rank;
      i++;
    }
  }
```

```
}
```

Notice how similar this function is to `buildDeck`, except that we had to change the syntax to make it a constructor. Now we can create a standard 52-card deck with the simple declaration `Deck deck;`

13.5 `Deck` member functions

Now that we have a `Deck` object, it makes sense to put all the functions that pertain to `Deck`s in the `Deck` structure definition. Looking at the functions we have written so far, one obvious candidate is `printDeck` (Section 12.7). Here's how it looks, rewritten as a `Deck` member function:

```
void Deck::print () const {
  for (int i = 0; i < cards.length(); i++) {
    cards[i].print ();
  }
}
```

As usual, we can refer to the instance variables of the current object without using dot notation.

For some of the other functions, it is not obvious whether they should be member functions of `Card`, member functions of `Deck`, or nonmember functions that take `Card`s and `Deck`s as parameters. For example, the version of `find` in the previous chapter takes a `Card` and a `Deck` as arguments, but you could reasonably make it a member function of either type. As an exercise, rewrite `find` as a `Deck` member function that takes a `Card` as a parameter.

Writing `find` as a `Card` member function is a little tricky. Here's my version:

```
int Card::find (const Deck& deck) const {
  for (int i = 0; i < deck.cards.length(); i++) {
    if (equals (deck.cards[i], *this)) return i;
  }
  return -1;
}
```

The first trick is that we have to use the keyword `this` to refer to the `Card` the function is invoked on.

The second trick is that C++ does not make it easy to write structure definitions that refer to each other. The problem is that when the compiler is reading the first structure definition, it doesn't know about the second one yet.

One solution is to declare `Deck` before `Card` and then define `Deck` afterwards:

```
// declare that Deck is a structure, without defining it
struct Deck;
```

```
// that way we can refer to it in the definition of Card
struct Card
{
  int suit, rank;

  Card ();
  Card (int s, int r);

  void print () const;
  bool isGreater (const Card& c2) const;
  int find (const Deck& deck) const;
};

// and then later we provide the definition of Deck
struct Deck {
  apvector<Card> cards;

  Deck ();
  Deck (int n);
  void print () const;
  int find (const Card& card) const;
};
```

13.6 Shuffling

For most card games you need to be able to shuffle the deck; that is, put the cards in a random order. In Section 10.6 we saw how to generate random numbers, but it is not obvious how to use them to shuffle a deck.

One possibility is to model the way humans shuffle, which is usually by dividing the deck in two and then reassembling the deck by choosing alternately from each deck. Since humans usually don't shuffle perfectly, after about 7 iterations the order of the deck is pretty well randomized. But a computer program would have the annoying property of doing a perfect shuffle every time, which is not really very random. In fact, after 8 perfect shuffles, you would find the deck back in the same order you started in. For a discussion of that claim, see `http://www.wiskit.com/marilyn/craig.html` or do a web search with the keywords "perfect shuffle."

A better shuffling algorithm is to traverse the deck one card at a time, and at each iteration choose two cards and swap them.

Here is an outline of how this algorithm works. To sketch the program, I am using a combination of C++ statements and English words that is sometimes called **pseudocode**:

```
  for (int i=0; i<cards.length(); i++) {
    // choose a random number between i and cards.length()
```

```
    // swap the ith card and the randomly-chosen card
  }
```

The nice thing about using pseudocode is that it often makes it clear what functions you are going to need. In this case, we need something like `randomInt`, which chooses a random integer between the parameters `low` and `high`, and `swapCards` which takes two indices and switches the cards at the indicated positions.

You can probably figure out how to write `randomInt` by looking at Section 10.6, although you will have to be careful about possibly generating indices that are out of range.

You can also figure out `swapCards` yourself. I will leave the remaining implementation of these functions as an exercise to the reader.

13.7 Sorting

Now that we have messed up the deck, we need a way to put it back in order. Ironically, there is an algorithm for sorting that is very similar to the algorithm for shuffling.

Again, we are going to traverse the deck and at each location choose another card and swap. The only difference is that this time instead of choosing the other card at random, we are going to find the lowest card remaining in the deck.

By "remaining in the deck," I mean cards that are at or to the right of the index `i`.

```
for (int i=0; i<cards.length(); i++) {
  // find the lowest card at or to the right of i
  // swap the ith card and the lowest card
}
```

Again, the pseudocode helps with the design of the **helper functions**. In this case we can use `swapCards` again, so we only need one new one, called `findLowestCard`, that takes a vector of cards and an index where it should start looking.

This process, using pseudocode to figure out what helper functions are needed, is sometimes called **top-down design**, in contrast to the bottom-up design I discussed in Section 10.9.

Once again, I am going to leave the implementation up to the reader.

13.8 Subdecks

How should we represent a hand or some other subset of a full deck? One easy choice is to make a `Deck` object that has fewer than 52 cards.

We might want a function, `subdeck`, that takes a vector of cards and a range of indices, and that returns a new vector of cards that contains the specified subset of the deck:

```
Deck Deck::subdeck (int low, int high) const {
  Deck sub (high-low+1);

  for (int i = 0; i<sub.cards.length(); i++) {
    sub.cards[i] = cards[low+i];
  }
  return sub;
}
```

To create the local variable named `subdeck` we are using the `Deck` constructor that takes the size of the deck as an argument and that does not initialize the cards. The cards get initialized when they are copied from the original deck.

The length of the subdeck is `high-low+1` because both the low card and high card are included. This sort of computation can be confusing, and lead to "off-by-one" errors. Drawing a picture is usually the best way to avoid them.

As an exercise, write a version of `findBisect` that takes a subdeck as an argument, rather than a deck and an index range. Which version is more error-prone? Which version do you think is more efficient?

13.9 Shuffling and dealing

In Section 13.6 I wrote pseudocode for a shuffling algorithm. Assuming that we have a function called `shuffleDeck` that takes a deck as an argument and shuffles it, we can create and shuffle a deck:

```
  Deck deck;               // create a standard 52-card deck
  deck.shuffle ();         // shuffle it
```

Then, to deal out several hands, we can use `subdeck`:

```
  Deck hand1 = deck.subdeck (0, 4);
  Deck hand2 = deck.subdeck (5, 9);
  Deck pack = deck.subdeck (10, 51);
```

This code puts the first 5 cards in one hand, the next 5 cards in the other, and the rest into the pack.

When you thought about dealing, did you think we should give out one card at a time to each player in the round-robin style that is common in real card games? I thought about it, but then realized that it is unnecessary for a computer program. The round-robin convention is intended to mitigate imperfect shuffling and make it more difficult for the dealer to cheat. Neither of these is an issue for a computer.

This example is a useful reminder of one of the dangers of engineering metaphors: sometimes we impose restrictions on computers that are unnecessary, or expect capabilities that are lacking, because we unthinkingly extend a metaphor past its breaking point. Beware of misleading analogies.

13.10 Mergesort

In Section 13.7, we saw a simple sorting algorithm that turns out not to be very efficient. In order to sort n items, it has to traverse the vector n times, and each traversal takes an amount of time that is proportional to n. The total time, therefore, is proportional to n^2.

In this section I will sketch a more efficient algorithm called **mergesort**. To sort n items, mergesort takes time proportional to $n \log n$. That may not seem impressive, but as n gets big, the difference between n^2 and $n \log n$ can be enormous. Try out a few values of n and see.

The basic idea behind mergesort is this: if you have two subdecks, each of which has been sorted, it is easy (and fast) to merge them into a single, sorted deck. Try this out with a deck of cards:

1. Form two subdecks with about 10 cards each and sort them so that when they are face up the lowest cards are on top. Place both decks face up in front of you.

2. Compare the top card from each deck and choose the lower one. Flip it over and add it to the merged deck.

3. Repeat step two until one of the decks is empty. Then take the remaining cards and add them to the merged deck.

The result should be a single sorted deck. Here's what this looks like in pseudocode:

```
Deck merge (const Deck& d1, const Deck& d2) {
  // create a new deck big enough for all the cards
  Deck result (d1.cards.length() + d2.cards.length());

  // use the index i to keep track of where we are in
  // the first deck, and the index j for the second deck
  int i = 0;
  int j = 0;

  // the index k traverses the result deck
  for (int k = 0; k<result.cards.length(); k++) {

    // if d1 is empty, d2 wins; if d2 is empty, d1 wins;
    // otherwise, compare the two cards

    // add the winner to the new deck
  }
  return result;
}
```

I chose to make `merge` a nonmember function because the two arguments are symmetric.

The best way to test `merge` is to build and shuffle a deck, use subdeck to form two (small) hands, and then use the sort routine from the previous chapter to sort the two halves. Then you can pass the two halves to `merge` to see if it works.

If you can get that working, try a simple implementation of `mergeSort`:

```
Deck Deck::mergeSort () const {
  // find the midpoint of the deck
  // divide the deck into two subdecks
  // sort the subdecks using sort
  // merge the two halves and return the result
}
```

Notice that the current object is declared `const` because `mergeSort` does not modify it. Instead, it creates and returns a new `Deck` object.

If you get that version working, the real fun begins! The magical thing about mergesort is that it is recursive. At the point where you sort the subdecks, why should you invoke the old, slow version of `sort`? Why not invoke the spiffy new `mergeSort` you are in the process of writing?

Not only is that a good idea, it is *necessary* in order to achieve the performance advantage I promised. In order to make it work, though, you have to add a base case so that it doesn't recurse forever. A simple base case is a subdeck with 0 or 1 cards. If `mergesort` receives such a small subdeck, it can return it unmodified, since it is already sorted.

The recursive version of `mergesort` should look something like this:

```
Deck Deck::mergeSort (Deck deck) const {
  // if the deck is 0 or 1 cards, return it

  // find the midpoint of the deck
  // divide the deck into two subdecks
  // sort the subdecks using mergesort
  // merge the two halves and return the result
}
```

As usual, there are two ways to think about recursive programs: you can think through the entire flow of execution, or you can make the "leap of faith." I have deliberately constructed this example to encourage you to make the leap of faith.

When you were using `sort` to sort the subdecks, you didn't feel compelled to follow the flow of execution, right? You just assumed that the `sort` function would work because you already debugged it. Well, all you did to make `mergeSort` recursive was replace one sort algorithm with another. There is no reason to read the program differently.

Well, actually you have to give some thought to getting the base case right and making sure that you reach it eventually, but other than that, writing the recursive version should be no problem. Good luck!

13.11 Glossary

helper function: Often a small function that does not do anything enormously useful by itself, but which helps another, more useful, function.

bottom-up design: A method of program development that uses pseudocode to sketch solutions to large problems and design the interfaces of helper functions.

mergesort: An algorithm for sorting a collection of values. Mergesort is faster than the simple algorithm in the previous chapter, especially for large collections.

Chapter 14

Classes and invariants

14.1 Private data and classes

I have used the word "encapsulation" in this book to refer to the process of wrapping up a sequence of instructions in a function, in order to separate the function's interface (how to use it) from its implementation (how it does what it does).

This kind of encapsulation might be called "functional encapsulation," to distinguish it from "data encapsulation," which is the topic of this chapter. Data encapsulation is based on the idea that each structure definition should provide a set of functions that apply to the structure, and prevent unrestricted access to the internal representation.

One use of data encapsulation is to hide implementation details from users or programmers that don't need to know them.

For example, there are many possible representations for a `Card`, including two integers, two strings and two enumerated types. The programmer who writes the `Card` member functions needs to know which implementation to use, but someone using the `Card` structure should not have to know anything about its internal structure.

As another example, we have been using `apstring` and `apvector` objects without ever discussing their implementations. There are many possibilities, but as "clients" of these libraries, we don't need to know.

In C++, the most common way to enforce data encapsulation is to prevent client programs from accessing the instance variables of an object. The keyword `private` is used to protect parts of a structure definition. For example, we could have written the `Card` definition:

```
struct Card
{
private:
  int suit, rank;
```

```
public:
  Card ();
  Card (int s, int r);

  int getRank () const { return rank; }
  int getSuit () const { return suit; }
  void setRank (int r) { rank = r; }
  void setSuit (int s) { suit = s; }
};
```

There are two sections of this definition, a private part and a public part. The functions are public, which means that they can be invoked by client programs. The instance variables are private, which means that they can be read and written only by `Card` member functions.

It is still possible for client programs to read and write the instance variables using the **accessor functions** (the ones beginning with `get` and `set`). On the other hand, it is now easy to control which operations clients can perform on which instance variables. For example, it might be a good idea to make cards "read only" so that after they are constructed, they cannot be changed. To do that, all we have to do is remove the `set` functions.

Another advantage of using accessor functions is that we can change the internal representations of cards without having to change any client programs.

14.2 What is a class?

In most object-oriented programming languages, a **class** is a user-defined type that includes a set of functions. As we have seen, structures in C++ meet the general definition of a class.

But there is another feature in C++ that also meets this definition; confusingly, it is called a `class`. In C++, a class is just a structure whose instance variables are private by default. For example, I could have written the `Card` definition:

```
class Card
{
  int suit, rank;

public:
  Card ();
  Card (int s, int r);

  int getRank () const { return rank; }
  int getSuit () const { return suit; }
  int setRank (int r) { rank = r; }
  int setSuit (int s) { suit = s; }
};
```

I replaced the word `struct` with the word `class` and removed the `private:` label. This result of the two definitions is exactly the same.

In fact, anything that can be written as a `struct` can also be written as a `class`, just by adding or removing labels. There is no real reason to choose one over the other, except that as a stylistic choice, most C++ programmers use `class`.

Also, it is common to refer to all user-defined types in C++ as "classes," regardless of whether they are defined as a `struct` or a `class`.

14.3 Complex numbers

As a running example for the rest of this chapter we will consider a class definition for complex numbers. Complex numbers are useful for many branches of mathematics and engineering, and many computations are performed using complex arithmetic. A complex number is the sum of a real part and an imaginary part, and is usually written in the form $x + yi$, where x is the real part, y is the imaginary part, and i represents the square root of -1.

The following is a class definition for a user-defined type called `Complex`:

```
class Complex
{
  double real, imag;

public:
  Complex () { }
  Complex (double r, double i) { real = r;  imag = i; }
};
```

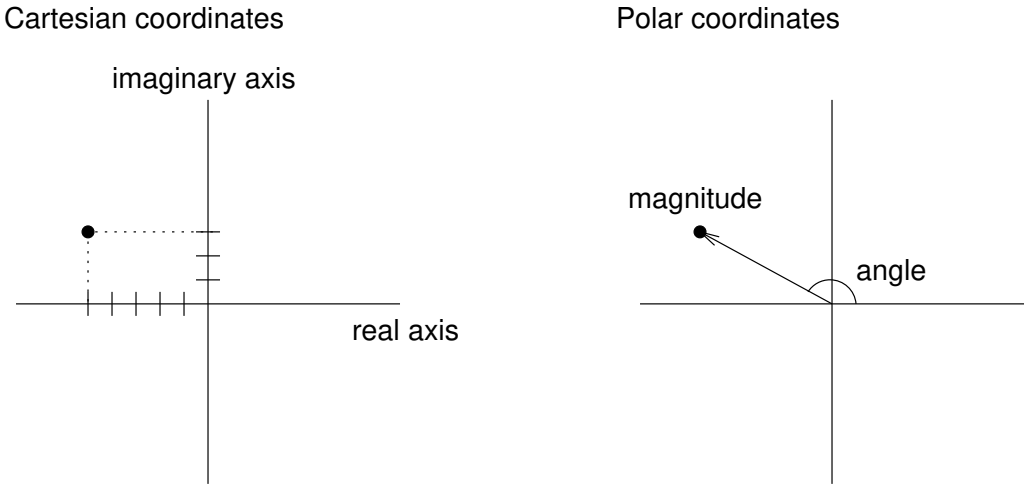
Because this is a `class` definition, the instance variables `real` and `imag` are private, and we have to include the label `public:` to allow client code to invoke the constructors.

As usual, there are two constructors: one takes no parameters and does nothing; the other takes two parameters and uses them to initialize the instance variables.

So far there is no real advantage to making the instance variables private. Let's make things a little more complicated; then the point might be clearer.

There is another common representation for complex numbers that is sometimes called "polar form" because it is based on polar coordinates. Instead of specifying the real part and the imaginary part of a point in the complex plane, polar coordinates specify the direction (or angle) of the point relative to the origin, and the distance (or magnitude) of the point.

The following figure shows the two coordinate systems graphically.



Complex numbers in polar coordinates are written $re^{i\theta}$, where r is the magnitude (radius), and θ is the angle in radians.

Fortunately, it is easy to convert from one form to another. To go from Cartesian to polar,

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ \theta &= \arctan(y/x) \end{aligned}$$

To go from polar to Cartesian,

$$\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \end{aligned}$$

So which representation should we use? Well, the whole reason there are multiple representations is that some operations are easier to perform in Cartesian coordinates (like addition), and others are easier in polar coordinates (like multiplication). One option is that we can write a class definition that uses *both* representations, and that converts between them automatically, as needed.

```
class Complex
{
  double real, imag;
  double mag, theta;
  bool cartesian, polar;

public:
  Complex () { cartesian = false;  polar = false; }

  Complex (double r, double i)
```

```
  {
    real = r;  imag = i;
    cartesian = true;  polar = false;
  }
};
```

There are now six instance variables, which means that this representation will take up more space than either of the others, but we will see that it is very versatile.

Four of the instance variables are self-explanatory. They contain the real part, the imaginary part, the angle and the magnitude of the complex number. The other two variables, `cartesian` and `polar` are flags that indicate whether the corresponding values are currently valid.

For example, the do-nothing constructor sets both flags to false to indicate that this object does not contain a valid complex number (yet), in either representation.

The second constructor uses the parameters to initialize the real and imaginary parts, but it does not calculate the magnitude or angle. Setting the `polar` flag to false warns other functions not to access `mag` or `theta` until they have been set.

Now it should be clearer why we need to keep the instance variables private. If client programs were allowed unrestricted access, it would be easy for them to make errors by reading uninitialized values. In the next few sections, we will develop accessor functions that will make those kinds of mistakes impossible.

14.4 Accessor functions

By convention, accessor functions have names that begin with `get` and end with the name of the instance variable they fetch. The return type, naturally, is the type of the corresponding instance variable.

In this case, the accessor functions give us an opportunity to make sure that the value of the variable is valid before we return it. Here's what `getReal` looks like:

```
double Complex::getReal ()
{
  if (cartesian == false) calculateCartesian ();
  return real;
}
```

If the `cartesian` flag is true then `real` contains valid data, and we can just return it. Otherwise, we have to call `calculateCartesian` to convert from polar coordinates to Cartesian coordinates:

```
void Complex::calculateCartesian ()
{
```

```
  real = mag * cos (theta);
  imag = mag * sin (theta);
  cartesian = true;
}
```

Assuming that the polar coordinates are valid, we can calculate the Cartesian coordinates using the formulas from the previous section. Then we set the `cartesian` flag, indicating that `real` and `imag` now contain valid data.

As an exercise, write a corresponding function called `calculatePolar` and then write `getMag` and `getTheta`. One unusual thing about these accessor functions is that they are not `const`, because invoking them might modify the instance variables.

14.5 Output

As usual when we define a new class, we want to be able to output objects in a human-readable form. For `Complex` objects, we could use two functions:

```
void Complex::printCartesian ()
{
  cout << getReal() << " + " << getImag() << "i" << endl;
}

void Complex::printPolar ()
{
  cout << getMag() << " e^ " << getTheta() << "i" << endl;
}
```

The nice thing here is that we can output any `Complex` object in either format without having to worry about the representation. Since the output functions use the accessor functions, the program will compute automatically any values that are needed.

The following code creates a `Complex` object using the second constructor. Initially, it is in Cartesian format only. When we invoke `printCartesian` it accesses `real` and `imag` without having to do any conversions.

```
  Complex c1 (2.0, 3.0);

  c1.printCartesian();
  c1.printPolar();
```

When we invoke `printPolar`, and `printPolar` invokes `getMag`, the program is forced to convert to polar coordinates and store the results in the instance variables. The good news is that we only have to do the conversion once. When `printPolar` invokes `getTheta`, it will see that the polar coordinates are valid and return `theta` immediately.

The output of this code is:

```
2 + 3i
3.60555 e^ 0.982794i
```

14.6 A function on Complex numbers

A natural operation we might want to perform on complex numbers is addition. If the numbers are in Cartesian coordinates, addition is easy: you just add the real parts together and the imaginary parts together. If the numbers are in polar coordinates, it is easiest to convert them to Cartesian coordinates and then add them.

Again, it is easy to deal with these cases if we use the accessor functions:

```
Complex add (Complex& a, Complex& b)
{
  double real = a.getReal() + b.getReal();
  double imag = a.getImag() + b.getImag();
  Complex sum (real, imag);
  return sum;
}
```

Notice that the arguments to `add` are not `const` because they might be modified when we invoke the accessors. To invoke this function, we would pass both operands as arguments:

```
  Complex c1 (2.0, 3.0);
  Complex c2 (3.0, 4.0);

  Complex sum = add (c1, c2);
  sum.printCartesian();
```

The output of this program is

```
5 + 7i
```

14.7 Another function on Complex numbers

Another operation we might want is multiplication. Unlike addition, multiplication is easy if the numbers are in polar coordinates and hard if they are in Cartesian coordinates (well, a little harder, anyway).

In polar coordinates, we can just multiply the magnitudes and add the angles. As usual, we can use the accessor functions without worrying about the representation of the objects.

```
Complex mult (Complex& a, Complex& b)
{
  double mag = a.getMag() * b.getMag()
```

```
  double theta = a.getTheta() + b.getTheta();
  Complex product;
  product.setPolar (mag, theta);
  return product;
}
```

A small problem we encounter here is that we have no constructor that accepts polar coordinates. It would be nice to write one, but remember that we can only overload a function (even a constructor) if the different versions take different parameters. In this case, we would like a second constructor that also takes two `double`s, and we can't have that.

An alternative it to provide an accessor function that *sets* the instance variables. In order to do that properly, though, we have to make sure that when `mag` and `theta` are set, we also set the `polar` flag. At the same time, we have to make sure that the `cartesian` flag is unset. That's because if we change the polar coordinates, the cartesian coordinates are no longer valid.

```
void Complex::setPolar (double m, double t)
{
  mag = m;  theta = t;
  cartesian = false;  polar = true;
}
```

As an exercise, write the corresponding function named `setCartesian`.

To test the `mult` function, we can try something like:

```
  Complex c1 (2.0, 3.0);
  Complex c2 (3.0, 4.0);

  Complex product = mult (c1, c2);
  product.printCartesian();
```

The output of this program is

```
-6 + 17i
```

There is a lot of conversion going on in this program behind the scenes. When we call `mult`, both arguments get converted to polar coordinates. The result is also in polar format, so when we invoke `printCartesian` it has to get converted back. Really, it's amazing that we get the right answer!

14.8 Invariants

There are several conditions we expect to be true for a proper `Complex` object. For example, if the `cartesian` flag is set then we expect `real` and `imag` to contain valid data. Similarly, if `polar` is set, we expect `mag` and `theta` to be valid. Finally, if both flags are set then we expect the other four variables to

be consistent; that is, they should be specifying the same point in two different formats.

These kinds of conditions are called `invariants`, for the obvious reason that they do not vary—they are always supposed to be true. One of the ways to write good quality code that contains few bugs is to figure out what invariants are appropriate for your classes, and write code that makes it impossible to violate them.

One of the primary things that data encapsulation is good for is helping to enforce invariants. The first step is to prevent unrestricted access to the instance variables by making them private. Then the only way to modify the object is through accessor functions and modifiers. If we examine all the accessors and modifiers, and we can show that every one of them maintains the invariants, then we can prove that it is impossible for an invariant to be violated.

Looking at the `Complex` class, we can list the functions that make assignments to one or more instance variables:

```
the second constructor
calculateCartesian
calculatePolar
setCartesian
setPolar
```

In each case, it is straightforward to show that the function maintains each of the invariants I listed. We have to be a little careful, though. Notice that I said "maintain" the invariant. What that means is "If the invariant is true when the function is called, it will still be true when the function is complete."

That definition allows two loopholes. First, there may be some point in the middle of the function when the invariant is not true. That's ok, and in some cases unavoidable. As long as the invariant is restored by the end of the function, all is well.

The other loophole is that we only have to maintain the invariant if it was true at the beginning of the function. Otherwise, all bets are off. If the invariant was violated somewhere else in the program, usually the best we can do is detect the error, output an error message, and exit.

14.9 Preconditions

Often when you write a function you make implicit assumptions about the parameters you receive. If those assumptions turn out to be true, then everything is fine; if not, your program might crash.

To make your programs more robust, it is a good idea to think about your assumptions explicitly, document them as part of the program, and maybe write code that checks them.

For example, let's take another look at `calculateCartesian`. Is there an assumption we make about the current object? Yes, we assume that the `polar`

flag is set and that `mag` and `theta` contain valid data. If that is not true, then this function will produce meaningless results.

One option is to add a comment to the function that warns programmers about the **precondition**.

```
void Complex::calculateCartesian ()
// precondition: the current object contains valid polar coordinates
and the polar flag is set
// postcondition: the current object will contain valid Cartesian
coordinates and valid polar coordinates, and both the cartesian
flag and the polar flag will be set
{
  real = mag * cos (theta);
  imag = mag * sin (theta);
  cartesian = true;
}
```

At the same time, I also commented on the **postconditions**, the things we know will be true when the function completes.

These comments are useful for people reading your programs, but it is an even better idea to add code that *checks* the preconditions, so that we can print an appropriate error message:

```
void Complex::calculateCartesian ()
{
  if (polar == false) {
    cout <<
    "calculateCartesian failed because the polar representation is invalid"
 << endl;
    exit (1);
  }
  real = mag * cos (theta);
  imag = mag * sin (theta);
  cartesian = true;
}
```

The `exit` function causes the program to quit immediately. The return value is an error code that tells the system (or whoever executed the program) that something went wrong.

This kind of error-checking is so common that C++ provides a built-in function to check preconditions and print error messages. If you include the `assert.h` header file, you get a function called `assert` that takes a boolean value (or a conditional expression) as an argument. As long as the argument is true, `assert` does nothing. If the argument is false, assert prints an error message and quits. Here's how to use it:

```
void Complex::calculateCartesian ()
```

```
{
  assert (polar);
  real = mag * cos (theta);
  imag = mag * sin (theta);
  cartesian = true;
  assert (polar && cartesian);
}
```

The first `assert` statement checks the precondition (actually just part of it); the second `assert` statement checks the postcondition.

In my development environment, I get the following message when I violate an assertion:

```
Complex.cpp:63: void Complex::calculatePolar(): Assertion 'cartesian' failed.
Abort
```

There is a lot of information here to help me track down the error, including the file name and line number of the assertion that failed, the function name and the contents of the assert statement.

14.10 Private functions

In some cases, there are member functions that are used internally by a class, but that should not be invoked by client programs. For example, `calculatePolar` and `calculateCartesian` are used by the accessor functions, but there is probably no reason clients should call them directly (although it would not do any harm). If we wanted to protect these functions, we could declare them `private` the same way we do with instance variables. In that case the complete class definition for `Complex` would look like:

```
class Complex
{
private:
  double real, imag;
  double mag, theta;
  bool cartesian, polar;

  void calculateCartesian ();
  void calculatePolar ();

public:
  Complex () { cartesian = false;  polar = false; }

  Complex (double r, double i)
  {
    real = r;  imag = i;
```

```
    cartesian = true;  polar = false;
  }

  void printCartesian ();
  void printPolar ();

  double getReal ();
  double getImag ();
  double getMag ();
  double getTheta ();

  void setCartesian (double r, double i);
  void setPolar (double m, double t);
};
```

The `private` label at the beginning is not necessary, but it is a useful reminder.

14.11 Glossary

class: In general use, a class is a user-defined type with member functions. In C++, a class is a structure with private instance variables.

accessor function: A function that provides access (read or write) to a private instance variable.

invariant: A condition, usually pertaining to an object, that should be true at all times in client code, and that should be maintained by all member functions.

precondition: A condition that is assumed to be true at the beginning of a function. If the precondition is not true, the function may not work. It is often a good idea for functions to check their preconditions, if possible.

postcondition: A condition that is true at the end of a function.

Chapter 15

File Input/Output and apmatrixes

In this chapter we will develop a program that reads and writes files, parses input, and demonstrates the `apmatrix` class. We will also implement a data structure called `Set` that expands automatically as you add elements.

Aside from demonstrating all these features, the real purpose of the program is to generate a two-dimensional table of the distances between cities in the United States. The output is a table that looks like this:

```
Atlanta 0
Chicago 700     0
Boston  1100    1000    0
Dallas  800     900     1750    0
Denver  1450    1000    2000    800     0
Detroit 750     300     800     1150    1300    0
Orlando 400     1150    1300    1100    1900    1200    0
Phoenix 1850    1750    2650    1000    800     2000    2100    0
Seattle 2650    2000    3000    2150    1350    2300    3100    1450    0
        Atlanta Chicago Boston  Dallas  Denver  Detroit Orlando Phoenix Seattle
```

The diagonal elements are all zero because that is the distance from a city to itself. Also, because the distance from A to B is the same as the distance from B to A, there is no need to print the top half of the matrix.

15.1 Streams

To get input from a file or send output to a file, you have to create an `ifstream` object (for input files) or an `ofstream` object (for output files). These objects are defined in the header file `fstream`, which you have to include.

A **stream** is an abstract object that represents the flow of data from a source like the keyboard or a file to a destination like the screen or a file.

We have already worked with two streams: `cin`, which has type `istream`, and `cout`, which has type `ostream`. `cin` represents the flow of data from the keyboard to the program. Each time the program uses the `>>` operator or the `getline` function, it removes a piece of data from the input stream.

Similarly, when the program uses the `<<` operator on an `ostream`, it adds a datum to the outgoing stream.

15.2 File input

To get data from a file, we have to create a stream that flows from the file into the program. We can do that using the `ifstream` constructor.

```
ifstream infile ("file-name");
```

The argument for this constructor is a string that contains the name of the file you want to open. The result is an object named `infile` that supports all the same operations as `cin`, including `>>` and `getline`.

```
int x;
apstring line;

infile >> x;               // get a single integer and store in x
getline (infile, line);    // get a whole line and store in line
```

If we know ahead of time how much data is in a file, it is straightforward to write a loop that reads the entire file and then stops. More often, though, we want to read the entire file, but don't know how big it is.

There are member functions for `ifstreams` that check the status of the input stream; they are called `good`, `eof`, `fail` and `bad`. We will use `good` to make sure the file was opened successfully and `eof` to detect the "end of file."

Whenever you get data from an input stream, you don't know whether the attempt succeeded until you check. If the return value from `eof` is `true` then we have reached the end of the file and we know that the last attempt failed. Here is a program that reads lines from a file and displays them on the screen:

```
apstring fileName = ...;
ifstream infile (fileName.c_str());

if (infile.good() == false) {
  cout << "Unable to open the file named " << fileName;
  exit (1);
}

while (true) {
  getline (infile, line);
  if (infile.eof()) break;
```

```
    cout << line << endl;
  }
```

The function `c_str` converts an `apstring` to a native C string. Because the `ifstream` constructor expects a C string as an argument, we have to convert the `apstring`.

Immediately after opening the file, we invoke the `good` function. The return value is `false` if the system could not open the file, most likely because it does not exist, or you do not have permission to read it.

The statement `while(true)` is an idiom for an infinite loop. Usually there will be a `break` statement somewhere in the loop so that the program does not really run forever (although some programs do). In this case, the `break` statement allows us to exit the loop as soon as we detect the end of file.

It is important to exit the loop between the input statement and the output statement, so that when `getline` fails at the end of the file, we do not output the invalid data in `line`.

15.3 File output

Sending output to a file is similar. For example, we could modify the previous program to copy lines from one file to another.

```
ifstream infile ("input-file");
ofstream outfile ("output-file");

if (infile.good() == false || outfile.good() == false) {
  cout << "Unable to open one of the files." << endl;
  exit (1);
}

while (true) {
  getline (infile, line);
  if (infile.eof()) break;
  outfile << line << endl;
}
```

15.4 Parsing input

In Section 1.4 I defined "parsing" as the process of analyzing the structure of a sentence in a natural language or a statement in a formal language. For example, the compiler has to parse your program before it can translate it into machine language.

In addition, when you read input from a file or from the keyboard you often have to parse it in order to extract the information you want and detect errors.

For example, I have a file called `distances` that contains information about the distances between major cities in the United States. I got this information from a randomly-chosen web page

`http://www.jaring.my/usiskl/usa/distance.html`

so it may be wildly inaccurate, but that doesn't matter. The format of the file looks like this:

```
"Atlanta"       "Chicago"       700
"Atlanta"       "Boston"        1,100
"Atlanta"       "Chicago"       700
"Atlanta"       "Dallas"        800
"Atlanta"       "Denver"        1,450
"Atlanta"       "Detroit"       750
"Atlanta"       "Orlando"       400
```

Each line of the file contains the names of two cities in quotation marks and the distance between them in miles. The quotation marks are useful because they make it easy to deal with names that have more than one word, like "San Francisco."

By searching for the quotation marks in a line of input, we can find the beginning and end of each city name. Searching for special characters like quotation marks can be a little awkward, though, because the quotation mark is a special character in C++, used to identify string values.

If we want to find the first appearance of a quotation mark, we have to write something like:

```
  int index = line.find ('\"');
```

The argument here looks like a mess, but it represents a single character, a double quotation mark. The outermost single-quotes indicate that this is a character value, as usual. The backslash (\) indicates that we want to treat the next character literally. The sequence `\"` represents a quotation mark; the sequence `\'` represents a single-quote. Interestingly, the sequence `\\` represents a single backslash. The first backslash indicates that we should take the second backslash seriously.

Parsing input lines consists of finding the beginning and end of each city name and using the `substr` function to extract the cities and distance. `substr` is an `apstring` member function; it takes two arguments, the starting index of the substring and the length.

```
void processLine (const apstring& line)
{
  // the character we are looking for is a quotation mark
  char quote = '\"';
```

```
  // store the indices of the quotation marks in a vector
  apvector<int> quoteIndex (4);

  // find the first quotation mark using the built-in find
  quoteIndex[0] = line.find (quote);

  // find the other quotation marks using the find from Chapter 7
  for (int i=1; i<4; i++) {
    quoteIndex[i] = find (line, quote, quoteIndex[i-1]+1);
  }

  // break the line up into substrings
  int len1 = quoteIndex[1] - quoteIndex[0] - 1;
  apstring city1 = line.substr (quoteIndex[0]+1, len1);
  int len2 = quoteIndex[3] - quoteIndex[2] - 1;
  apstring city2 = line.substr (quoteIndex[2]+1, len2);
  int len3 = line.length() - quoteIndex[2] - 1;
  apstring distString = line.substr (quoteIndex[3]+1, len3);

  // output the extracted information
  cout << city1 << "\t" << city2 << "\t" << distString << endl;
}
```

Of course, just displaying the extracted information is not exactly what we want, but it is a good starting place.

15.5 Parsing numbers

The next task is to convert the numbers in the file from strings to integers. When people write large numbers, they often use commas to group the digits, as in 1,750. Most of the time when computers write large numbers, they don't include commas, and the built-in functions for reading numbers usually can't handle them. That makes the conversion a little more difficult, but it also provides an opportunity to write a comma-stripping function, so that's ok. Once we get rid of the commas, we can use the library function `atoi` to convert to integer. `atoi` is defined in the header file `cstdlib`.

To get rid of the commas, one option is to traverse the string and check whether each character is a digit. If so, we add it to the result string. At the end of the loop, the result string contains all the digits from the original string, in order.

```
int convertToInt (const apstring& s)
{
  apstring digitString = "";

  for (int i=0; i<s.length(); i++) {
```

```
    if (isdigit (s[i])) {
      digitString += s[i];
    }
  }
  return atoi (digitString.c_str());
}
```

The variable `digitString` is an example of an **accumulator**. It is similar to the counter we saw in Section 7.12, except that instead of getting incremented, it gets accumulates one new character at a time, using string concatentation.

The expression

```
digitString += s[i];
```

is equivalent to

```
digitString = digitString + s[i];
```

Both statements add a single character onto the end of the existing string.

Since `atoi` takes a C string as a parameter, we have to convert `digitString` to a C string before passing it as an argument.

15.6 The `Set` data structure

A data structure is a container for grouping a collection of data into a single object. We have seen some examples already, including `apstring`s, which are collections of characters, and `apvector`s which are collections on any type.

An ordered set is a collection of items with two defining properties:

Ordering: The elements of the set have indices associated with them. We can use these indices to identify elements of the set.

Uniqueness: No element appears in the set more than once. If you try to add an element to a set, and it already exists, there is no effect.

In addition, our implementation of an ordered set will have the following property:

Arbitrary size: As we add elements to the set, it expands to make room for new elements.

Both `apstring`s and `apvector`s have an ordering; every element has an index we can use to identify it. Both none of the data structures we have seen so far have the properties of uniqueness or arbitrary size.

To achieve uniqueness, we have to write an `add` function that searches the set to see if it already exists. To make the set expand as elements are added, we can take advantage of the `resize` function on `apvector`s.

Here is the beginning of a class definition for a `Set`.

```
class Set {
private:
  apvector<apstring> elements;
  int numElements;

public:
  Set (int n);

  int getNumElements () const;
  apstring getElement (int i) const;
  int find (const apstring& s) const;
  int add (const apstring& s);
};

Set::Set (int n)
{
  apvector<apstring> temp (n);
  elements = temp;
  numElements = 0;
}
```

The instance variables are an `apvector` of strings and an integer that keeps track of how many elements there are in the set. Keep in mind that the number of elements in the set, `numElements`, is not the same thing as the size of the `apvector`. Usually it will be smaller.

The `Set` constructor takes a single parameter, which is the initial size of the `apvector`. The initial number of elements is always zero.

`getNumElements` and `getElement` are accessor functions for the instance variables, which are private. `numElements` is a read-only variable, so we provide a `get` function but not a `set` function.

```
int Set::getNumElements () const
{
  return numElements;
}
```

Why do we have to prevent client programs from changing `getNumElements`? What are the invariants for this type, and how could a client program break an invariant. As we look at the rest of the `Set` member function, see if you can convince yourself that they all maintain the invariants.

When we use the `[]` operator to access the `apvector`, it checks to make sure the index is greater than or equal to zero and less than the length of the `apvector`. To access the elements of a set, though, we need to check a stronger condition. The index has to be less than the number of elements, which might be smaller than the length of the `apvector`.

```
apstring Set::getElement (int i) const
```

```
{
  if (i < numElements) {
    return elements[i];
  } else {
    cout << "Set index out of range." << endl;
    exit (1);
  }
}
```

If `getElement` gets an index that is out of range, it prints an error message (not the most useful message, I admit), and exits.

The interesting functions are `find` and `add`. By now, the pattern for traversing and searching should be old hat:

```
int Set::find (const apstring& s) const
{
  for (int i=0; i<numElements; i++) {
    if (elements[i] == s) return i;
  }
  return -1;
}
```

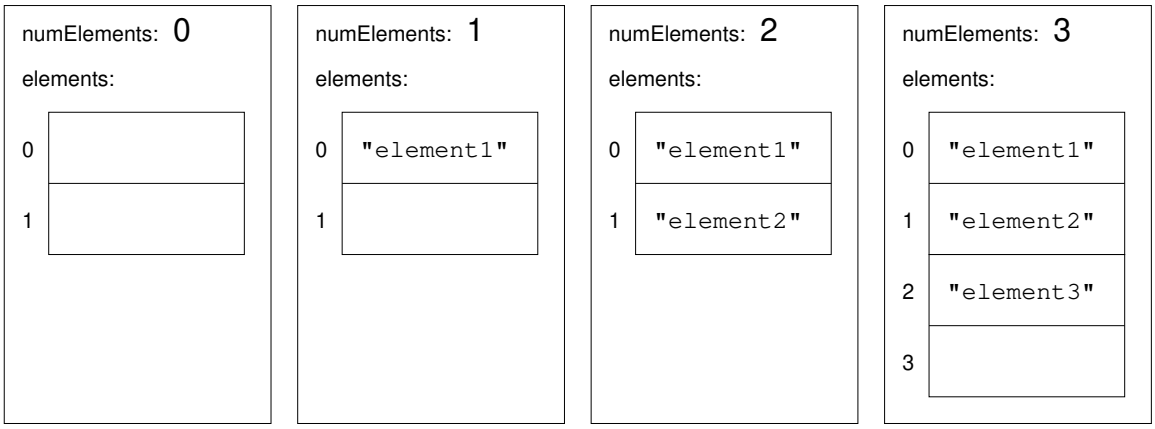
So that leaves us with `add`. Often the return type for something like `add` would be void, but in this case it might be useful to make it return the index of the element.

```
int Set::add (const apstring& s)
{
  // if the element is already in the set, return its index
  int index = find (s);
  if (index != -1) return index;

  // if the apvector is full, double its size
  if (numElements == elements.length()) {
    elements.resize (elements.length() * 2);
  }

  // add the new elements and return its index
  index = numElements;
  elements[index] = s;
  numElements++;
  return index;
}
```

The tricky thing here is that `numElements` is used in two ways. It is the number of elements in the set, of course, but it is also the index of the next element to be added.

It takes a minute to convince yourself that that works, but consider this: when the number of elements is zero, the index of the next element is 0. When the number of elements is equal to the length of the `apvector`, that means that the vector is full, and we have to allocate more space (using `resize`) before we can add the new element.

Here is a state diagram showing a `Set` object that initially contains space for 2 elements.



Now we can use the `Set` class to keep track of the cities we find in the file. In `main` we create the `Set` with an initial size of 2:

```
Set cities (2);
```

Then in `processLine` we add both cities to the `Set` and store the index that gets returned.

```
int index1 = cities.add (city1);
int index2 = cities.add (city2);
```

I modified `processLine` to take the `cities` object as a second parameter.

15.7 apmatrix

An `apmatrix` is similar to an `apvector` except it is two-dimensional. Instead of a length, it has two dimensions, called `numrows` and `numcols`, for "number of rows" and "number of columns."

Each element in the matrix is indentified by two indices; one specifies the row number, the other the column number.

To create a matrix, there are four constructors:

```
apmatrix<char> m1;
apmatrix<int> m2 (3, 4);
apmatrix<double> m3 (rows, cols, 0.0);
apmatrix<double> m4 (m3);
```

The first is a do-nothing constructor that makes a matrix with both dimensions 0. The second takes two integers, which are the initial number of rows and columns, in that order. The third is the same as the second, except that it takes an additional parameter that is used to initialized the elements of the matrix. The fourth is a copy constructor that takes another `apmatrix` as a parameter.

Just as with `apvectors`, we can make `apmatrix`es with any type of elements (including `apvector`s, and even `apmatrix`es).

To access the elements of a matrix, we use the `[]` operator to specify the row and column:

```
m2[0][0] = 1;
m3[1][2] = 10.0 * m2[0][0];
```

If we try to access an element that is out of range, the program prints an error message and quits.

The `numrows` and `numcols` functions get the number of rows and columns. Remember that the row indices run from 0 to `numrows() -1` and the column indices run from 0 to `numcols() -1`.

The usual way to traverse a matrix is with a nested loop. This loop sets each element of the matrix to the sum of its two indices:

```
for (int row=0; row < m2.numrows(); row++) {
  for (int col=0; col < m2.numcols(); col++) {
    m2[row][col] = row + col;
  }
}
```

This loop prints each row of the matrix with tabs between the elements and newlines between the rows:

```
for (int row=0; row < m2.numrows(); row++) {
  for (int col=0; col < m2.numcols(); col++) {
    cout << m2[row][col] << "\t";
  }
  cout << endl;
}
```

15.8 A distance matrix

Finally, we are ready to put the data from the file into a matrix. Specifically, the matrix will have one row and one column for each city.

We'll create the matrix in `main`, with plenty of space to spare:

```
apmatrix<int> distances (50, 50, 0);
```

Inside `processLine`, we add new information to the matrix by getting the indices of the two cities from the `Set` and using them as matrix indices:

```
int dist = convertToInt (distString);
int index1 = cities.add (city1);
int index2 = cities.add (city2);

distances[index1][index2] = distance;
distances[index2][index1] = distance;
```

Finally, in `main` we can print the information in a human-readable form:

```
for (int i=0; i<cities.getNumElements(); i++) {
  cout << cities.getElement(i) << "\t";

  for (int j=0; j<=i; j++) {
    cout << distances[i][j] << "\t";
  }
  cout << endl;
}

cout << "\t";
for (int i=0; i<cities.getNumElements(); i++) {
  cout << cities.getElement(i) << "\t";
}
cout << endl;
```

This code produces the output shown at the beginning of the chapter. The original data is available from this book's web page.

15.9 A proper distance matrix

Although this code works, it is not as well organized as it should be. Now that we have written a prototype, we are in a good position to evaluate the design and improve it.

What are some of the problems with the existing code?

1. We did not know ahead of time how big to make the distance matrix, so we chose an arbitrary large number (50) and made it a fixed size. It would be better to allow the distance matrix to expand in the same way a `Set` does. The `apmatrix` class has a function called `resize` that makes this possible.

2. The data in the distance matrix is not well-encapsulated. We have to pass the set of city names and the matrix itself as arguments to `processLine`, which is awkward. Also, use of the distance matrix is error prone because we have not provided accessor functions that perform error-checking. It

might be a good idea to take the `Set` of city names and the `apmatrix` of distances, and combine them into a single object called a `DistMatrix`.

Here is a draft of what the header for a `DistMatrix` might look like:

```
class DistMatrix {
private:
  Set cities;
  apmatrix<int> distances;

public:
  DistMatrix (int rows);

  void add (const apstring& city1, const apstring& city2, int dist);
  int distance (int i, int j) const;
  int distance (const apstring& city1, const apstring& city2) const;
  apstring cityName (int i) const;
  int numCities () const;
  void print ();
};
```

Using this interface simplifies `main`:

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


int main ()
{
  apstring line;
  ifstream infile ("distances");
  DistMatrix distances (2);

  while (true) {
    getline (infile, line);
    if (infile.eof()) break;
    processLine (line, distances);
  }

  distances.print ();
  return 0;
}
```

It also simplifies `processLine`:

```
void processLine (const apstring& line, DistMatrix& distances)
```

```
{
  char quote = '\"';
  apvector<int> quoteIndex (4);
  quoteIndex[0] = line.find (quote);
  for (int i=1; i<4; i++) {
    quoteIndex[i] = find (line, quote, quoteIndex[i-1]+1);
  }

  // break the line up into substrings
  int len1 = quoteIndex[1] - quoteIndex[0] - 1;
  apstring city1 = line.substr (quoteIndex[0]+1, len1);
  int len2 = quoteIndex[3] - quoteIndex[2] - 1;
  apstring city2 = line.substr (quoteIndex[2]+1, len2);
  int len3 = line.length() - quoteIndex[2] - 1;
  apstring distString = line.substr (quoteIndex[3]+1, len3);
  int distance = convertToInt (distString);

  // add the new datum to the distances matrix
  distances.add (city1, city2, distance);
}
```

I will leave it as an exercise to you to implement the member functions of `DistMatrix`.

15.10 Glossary

ordered set: A data structure in which every element appears only once and every element has an index that identifies it.

stream: A data structure that represents a "flow" or sequence of data items from one place to another. In C++ streams are used for input and output.

accumulator: A variable used inside a loop to accumulate a result, often by getting something added or concatenated during each iteration.

Chapter 16

Advanced Classes

16.1 Inheritance

16.2 Polymorphism

virtual override final default

16.3 Overloading Operators

16.3.1 Overloading Parentheses

16.4 Copy/Move

16.5 Smart Pointers

Chapter 17

Advanced Topics

17.1 Function Pointer

Just like you have pointers to variables, you can even have pointers to functions. (C++ uses this behind the scenes to implement object member and virtual functions.)

Function pointers are pointers that instead of pointing to data, point to a certain location in the executable code. Initializing a function pointer looks very similar to creating a function:

int (*match)(int key1, int key2);

The big difference is the extra * before the function name and that the function name and the star are surrounded by parentheses.

This definition is a little tricky to understand. It gets even harder to understand when you have a function pointer returned from another function.

```
int (*HowToMatch(const int type)) (int, int) {
 if (type == 1) return &match1;
    else return &match2;
}
```

Most people use a typedef when they create a function pointer to make it look more like a variable. For example, you can create match like:

```
typedef int (*Match)(int,int);

Match match;
```

And, the function that returns a function pointer would just be:

```
Match HowToMatch(const int type) {
 if (type = = 1) return &match1;
    else return &match2;
}
```

You may have noticed in the earlier code, to initialize the function pointer to a function, just use =. For example, assuming match_int is a function that will be later in the program.

```
int match_int(int key1, int key2);

match = match_int;
or
match = &match_int;
```

To call a function using a function pointer, dereference the pointer using the asterisk symbol or simply use the pointer variable name as if it were the function name:

```
retval = (*match)(x, y);

or

retval = match(x, y);
```

Pointers to functions can be passed to functions, returned from functions, stored in arrays, and assigned to other function pointers. They are often used for callback functions and in algorithms. They are quite powerful, but they do have some problems. The definition of the pointers can be confusing [1] The functions can not keep any state information, so a new function would have to be created for each individual use. Each function would be taking up a namespace. Also, the compiler will not know what function will be set inside the pointer, so will not be able to call the function inline.

For all those reasons, more recent versions of C++ have improved methods for the same or similar behavior.

17.2 Function Objects

Function Objects, also called functors, are a way to create a function via a class. This subject was introduced when we introduced overloading operators in 16.3.1. This particular object uses the overloaded parenthesis operator in order to allow the function object to be "called".

```
//this creates the function object
class SayYayNumPlus {
  int num;  // This allows the function to have a "state"
public:
  SayYayNumPlus(int t):num(t){}
  void operator()(int i) const {
        std::cout << "Yay "<< i+num << '\n';
```

[1] std::function, described in Section 17.4, was created in C++11 to avoid the need to use typedef.

```
        }
};

// The following would be in main
  SayYayNumPlus plus25(25);  // create the function object
                             // and put 25 in the "num"
  plus25(10);   // Output "Yay 35"
```

The above example shows the power of this object. The constructor allows this class to keep a value that can be used later. Then, in main, the object was created and then can later be used.

These objects can be passed into functions like a function pointer. Unlike function pointers, the compiler know what function it will call and can inline the function call.

17.3 Lambda Expressions

Lambda expressions, also known as Lambda functions, and lambda were introduced to C++ in C++11. This feature allows for inline unnamed functions that can be passed as parameters to other functions, or called where it was defined. Lambdas tend to be used when sending code to algorithms (like a sorting algorithm), or to asynchronous functions (code that will run at a different time.)

Here is an the format that many lambda expressions take:

```
[capture](parameters){
   function body
}
```

Here is an example of a lambda expression being used to discover the smallest item in a range.[2] The code takes the vector, named v, and finds the smallest item in the vector. It uses the expression in order to compare each item. Min_Element takes each element in turn and passes it as a parameter to the lambda expression. The expression returns the result of the comparison.

```
  std::vector<int> v{1, 7, 3, 4, 0};

  auto l = std::min_element(v.begin(), v.end(),
              [](int x, int y) { return x < y; }); // lambda
  std::cout << "Min is " << *l << "\n";
```

[2]Normally you would not need this lambda expression because the default compare (using <) or the standard library compare function object would have been fine with this example. Lambda expressions or functors are usually needed when the default comparison would not work.

17.3.1 Generic Lambda

In C++14, the above expression can be changed to a generic lambda by using "auto" for the parameter types. This will allow this call to be used by many different types, not just int. [3]:

```
[](auto x, auto y) { return x < y; }
```

17.3.2 Constant expressions

C++17 allows lambda expressions to be constexpr if it can be evaluated at compile time. If the compiler discovers that it can, the lambda will automatically be changed to constexpr. But, you can declare it explicitly.

```
[]() constexpr {
    function body
}
```

For example, if there is something that will always return the same value:

```
auto w = []() constexpr { return 7;}
```

This should automatically be set to a constexpr, but it will not hurt for you to declare it anyway.

You may have noticed in some of the above examples that you can set a variable to a lambda expression.

```
auto var = [] () {
    std::cout << "Hello, I'm in a lambda expression";
    };

var(); // This is how you call it
```

17.3.3 Return Values

In simple cases, the return value of a lambda expression can be deduced by the compiler so it does not need to be specified. But, more complex code needs to be explicitly listed. That is possible by using " ->". This would be the format with the return type added.

```
[capture](parameters)->returntype {
   function body
}
```

Below is an example when you would need to set the return type explicitly. If you notice, one return could be returning an integer, and another could be returning a double. The return value needs to be set explicitly because the compiler might not be able to guess what was expected:

[3] Internally, this turns this code to an unnamed, templated functor.

```
auto whattodo = []  (int num1, int num2,
        string whichmath) -> double {

  if (whichmath == "avg") {
    // returns double here
    return (num1 + num2) / 2.0;
  }
  else {
    // returns int here
    return num1 + num2;
  }
};
```

17.3.4 Captures

Normally, the code in the lambda body can not access variables from the functions that are around it. If the body needs to access one of those variables, they can be listed in the capture clause. There are many ways that this can be done.

[] No variables in the surrounding area are used in the lambda body. This is what we had in previous examples.

[**varname**] The variable listed will be captured by value. This means that it will make a copy of the value where it can be used later.

[**& varname**] The variable listed will be captured by reference. That means that the lambda expression can change the value of the variable

[**&**] All variables listed in the body are captured by reference. Instead of having an empty [], the & symbol is put there. This means that that if the variable is changed in the lambda expression, it will be changed outside the expression. No variables need to be listed specifically.

One thing that you need to watch out is when you run a lambda that captures a variable by reference and it runs asynchronously. It is very important that the variable that is captured does not leave scope before the lambda is called. If it does, you may get an access violation error.

[=] All variables listed in the body are captured by value. All the variables that were captured are copied and will work in parallel or asynchronously. It is acceptable for the variables that were captured by value to have left scope before the expression runs.

It is also possible to list variables that will be using a different method of capture. For example: [=, &varname]. That would capture everything but varname by value. Varname would be by reference. [&, varname] would capture everything by reference except varname, which would be by value.

Generalized captures

C++14 added the capability to create and initialize a variable in the capture section. This is great for something that you wanted to be able to change, but it would have no longer been in scope by the time the expression ran. For example:

```
// make a int array of size 7
auto ptr = std::make_unique<int[]>(7);

auto lambda = [value = std::move(ptr)] {
    /* Something in here that uses "value"*/
    };
```

17.3.5 Mutable

If a variable is captured by value, it is treated like a constant. If code in the lambda body changes the value, the compile flags it as an error. You can put the word "mutable" in your lambda expression definition. This allows you to change the variables that were captured by value. NOTE: It will not be changed outside the lambda, only inside the lambda.

17.4 std::function

C++ introduced different ways to do function pointers since C++11. In order to use this technique, the code needs to:

```
#include <functional>
```

The variables that are created this way can store function pointers, lambda expressions, function objects, and bind expressions. Once those are stored, these are now callable.

To use the template, it follows this pattern:

```
std::function<retval(params)> fvar;
```

The format is a lot easier than a typical function pointer. The return value (in the pattern it is retval) is the first thing listed. The items inside the parentheses are the parameter types. The parameters do not need to be named here, only the types.

Here is an example using a function pointer:

```
// This is a normal function
void print_num(int i) { std::cout << i << '\n'; }

// Here it uses the printnum function, and f_display
// is set to that function.
  std::function<void(int)> f_display = print_num;

// Using the variable, it can now call
// the print_num function
  f_display(35);
```

This tool also works with function objects.

```
// Here is the functor's definition
struct PrintNum {
  void operator()(int i) const { std::cout << i << '\n'; }
};

  // store a functor
  std::function<void(int)> f_functor = PrintNum();
  // call the functor
  f_functor(75);
```

And this shows how it can work with a Lambda expression.

```
int main() {
  // store a lambda
  std::function<void()>
    f_display_89 = []() { print_num(89); };
  f_display_89();
```

17.4.1 Problems with returned reference

If you are using std::function that has a returned reference with a lambda expression that does not specify a return value, there can be problems. Quite often it will connect the return value to a variable that will lose scope as soon as the expression is over. This will not flag an error with the compiler until C++23.

17.4.2 Predefined functions

Before you create your own lambda expression or functor, you may want to check if what you were planning on making already exists. There are many simple checks, like less, greater, equal_to and more defined in the functional header.

Here is an example:

```
template <typename X, typename Y, typename Z = std::less<>>
bool isless(X x, Y y, Z l = Z{})
{
    return l(x, y);
}

// and in a function, you can have:
std::cout << std::boolalpha << isless(1, 2);
```

17.5 Glossary

lambda expression: A technique to create a function inline without naming it.

functor: A function object. This is a class (or a struct) that has overloaded the parenthesis operator in order to allow it to act like a function.

function pointer: a variable that is connected to a function instead of data.

Appendix A

Integrated Development Environments

C++ has standards so code should work in different compilers. But, the environments that can compile and run your code can be different. Here are some descriptions on how to use some of the popular development environments. This appendix has information on Replit and Visual Studio Code.

A.1 Installing

Before you use a compiler, there is usually some setup. Some need to be installed on your computer. Others need an account to be created. Here are the steps to get your compiler ready to be used.

A.1.1 Replit

Replit does not need to be installed. It is an online coding environment. This IDE will work well for those using a Chrome book. The one thing that you will need to do is go to `http://replit.com` and create an account there.

A.1.2 Visual Studio Code

There are many steps to get C++ to work with Visual Studio code. Visual Studio Code works in Windows, macOS and Linux. First, download and install the tool. The tool is available at this website: `https://code.visualstudio.com/`. Click the appropriate down button for your computer.

Once it is installed, you will need to install the C/C++ extension for VS Code. You can install the C/C++ extension by searching for 'c++' in the Extensions view (Ctrl+Shift+X).

Microsoft Compiler

The next steps are to get it working with the Microsoft compiler. If you have a recent version of the compiler installed, you can use the tools and you are done with the installation. If not, you will need to do the following:

Go online to the Visual Studio Downloads page. On that page, scroll down until you see `Tools for Visual Studio` under the All Downloads section and click the download button for "Build Tools for Visual Studio 2022".

This will download and launch the Visual Studio Installer, which will bring up a dialog showing the available Visual Studio Build Tools workloads. Check the Desktop development with C++ workload and select Install. Once it is done, you should be ready to create a project.

For more information, check this page: `https://code.visualstudio.com/docs/cpp/config-msvc`

GNU compiler

FIXME The information can be found here: `https://code.visualstudio.com/docs/cpp/config-mingw`

A.2 Setting up a project

Want to code on your own? This section will some helpful instructions on how to start coding with some common coding environments.

A.2.1 Replit

Once you log in to you account, you should be able to see the full menu. Look for the "Create" button that is on the top left. The button should look like figure A.1: Once you click that button, another menu will open. Here you will

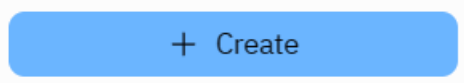


Figure A.1: The Create button for Replit

have a chance to choose the language that you will be using. Click on the area under the "Template" in the left top section of the menu. Look for the C++ template. Once the template is chosen, click on the "Title" area and choose a name for your project. If you have the template and the name, the next step is to click the "Create Repl" button. Soon after, you will be in the development environment.

For more information, there is the documentation from replit which explains the startup in more detail. `https://docs.replit.com/programming-ide/introduction-to-the-workspace`

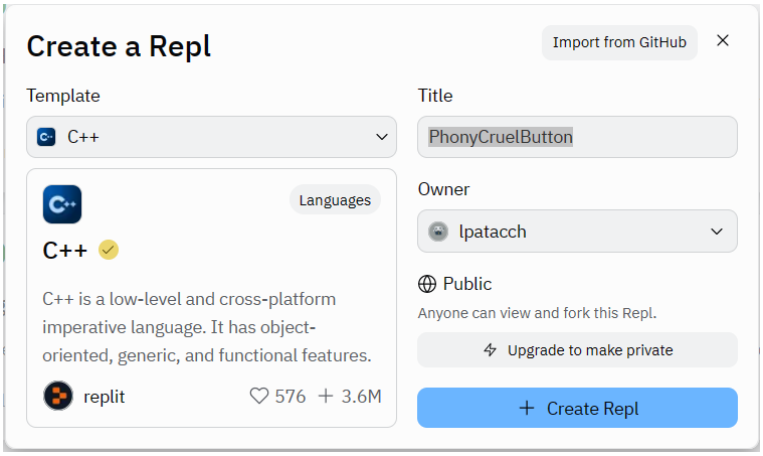


Figure A.2: Template and Naming your Replit

A.2.2 Visual Studio Code

If you are running the Microsoft compiler, you will need to run the "Developer Command Prompt for Visual Studio". That will open up a command line. If you have a GNU compiler, a regular command line will work instead.

From there, go to the area where you want your project to be. Then, make the directory (mkdir), go into the directory (cd) and run visual studio code from that location (code). For example, if you want a "newproj" project.

```
mkdir newproj
cd newproj
code .
```

Visual Studio Code will now be running. You can click the "new file" button and it will add a file to your project.

A.3 Settings

Most Integrated Development Environments work well as soon as you install them. They are designed so you do not have to worry about all the various options the compiler and linker can use. This section will explain options you may want to set on your IDE to help you code. NOTE: Your IDE will work fine without these changes. They are the ones that I have found particularly helpful.

A.3.1 Replit

Turning on warnings

One particular feature I like to use is to "show all warnings". If you remember, warnings are a way that the compiler can show you that it noticed some code that could be an error. The code does not have a problem with it's syntax, but many times code in this pattern is a logic error.

For example, this code compiles without an error on Replit:

```
if (x=0)           \\ There is only 1 equal sign
{
  std::cout<<"woo";
}
```

It only has one equal sign, so instead of comparing the variable to zero, it set the variable to zero. Technically, that code does not have a syntax problem. But, very few people would want to set a value here. They were expecting the code to compare values.

Luckily, you can set options in the Replit environment to let you know that there could be a mistake.

To make your environment less cluttered, Replit hides the configuration files as a default. But, you can show these files so you can edit them. First, click the three dots next to the word "Files" on the left. That should open a menu with an option to "Show hidden files". Click that option. The menu should look like Figure A.3

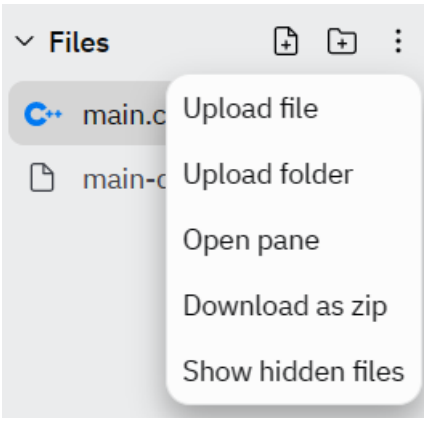


Figure A.3: The Menu that Displays when you press the three Dots

The file that you will be editing is called "Makefile". This file holds the special options that are used when compiling and linking. Look for a line in the file matches this:

```
override CXXFLAGS += -g -Wno-everything
```

The -Wno-everything flag is telling the compiler to ignore all warnings. That is not what we want. Instead, change that line to:

```
override CXXFLAGS += -g -Weverything
```

Instead of hiding all the warnings, it will show them instead. There are other flags that do not show as many warnings, but I prefer to see all the possible problems with my code.

Using a certain version of the C++ standard

Although Replit does compile C++ code, it defaults to an older version of the language. C++ has continually added new features as the years have gone on. For example, smart pointers were not added to the language until 2011.

At the time of the writing of this book, the default version of C++ that Replit uses is the 2014 standard. If you want to use some of the newer features, you need to tell it to use a different version of the standard. That is achievable by changing the Makefile again. Look for the same overide CXXFLAGS line that you changed in the last section. Change that line to the following line:

```
override CXXFLAGS += -g -std=c++20 -Weverything -Wno-c++98-compat
```

The important part is the `-std=` part of the line. That is telling the compiler what standard to use. c++20 is telling it to use the 2020 version of the standard. I also added the `-Wnoc++98-compat"` to the line. That shuts off any warnings complaining that the code is not compatible with the 1998 version of C++. That isn't necessary to get the 2020 version to run, but I found it helpful to remove the warnings that were no longer pertinent. If I am using the 2020 version of the standard because I want to use the new language features, I am not worried about if my code works with a 1998 version of the compiler.

A.3.2 Visual Studio Code

At the time of the writing of this book, Visual Studio Code defaults to C++14, If you want to use any of the newer features, you need to tell the compiler. To turn on the different standard, open the tasks.json file. There will be a section listed as: "args". Put the following in the list of items in that section.

```
"/std:c++20",
```

That should turn on the C++20 standard. If you want the latest that is available, you can use this instead:

```
"/std:c++latest",
```

to use the most recent version.

That will change the way the tool compiles, but it will not change the IntelliSense (this is what makes the red squiggles underneath errors). That setting

is found in the c_cpp_properties.json file. If your project does not have that yet, Visual Studio Code can make one for you. First, type Ctrl-Shft-P to get the menu. Then, choose C/C++: Edit Configurations (UI). That will open up a C/C++ Configurations menu. Scroll down until you get to the "C++ standard" section. From there, you can change it to whatever standard you want. The code in this book has been tested with the C++23 Intellisense.

Index

absolute value, 77
abstract parameter, 169
abstraction, 169
accessor function, 184, 187, 194
accumulator, 200, 207
algorithm, 130, 131
ambiguity, 6
apmatrix, 203
apstring
    vector of, 159
argument, 59, 66, 74
arithmetic
    base 60, 129
    complex, 185
    floating-point, 30, 129
    integer, 20
array, 94
    std, 95
    traditional, 94
assert, 192
assignment, 14, 23, 27
atoi, 199

backslash, 198
bisection search, 166
body, 56
    loop, 49
bool, 82, 93
boolean, 81
bottom-up design, 140, 177, 181
braced initialization, 16, 30
break statement, 39, 197
bug, 4

C string, 197
c_str, 197
call, 74
call by reference, 115, 118
call by value, 114
Card, 157
Cartesian coordinate, 185
cast
    static, 30
character
    classification, 199
    special sequence, 198
character operator, 22
cin, 119, 196
class, 183, 184, 194
    Card, 157
    Complex, 185
    string, 106, 107
client programs, 183
cmath, 60
code block, 33, 45
code blocks, 53
comment, 8
comparable, 162
comparison
    operator, 31
    string, 107
comparison operator, 81, 162
compile, 2, 10
compile-time error, 4, 77
complete ordering, 162
Complex, 185
complex number, 185
composition, 22, 27, 61, 79, 117, 157
concatenate, 109
concatentation, 200
conditional, 31, 45
    alternative, 33
    chained, 34, 45
    nested, 34, 45

conditional operator, 37
constexpr, 214
constructor, 133, 144, 155, 158, 173, 174, 178, 185, 187, 201, 204
convention, 172
convert
    to integer, 199
coordinate, 185
    Cartesian, 185
    polar, 185
correctness, 168
counter, 105, 109, 140
cout, 119, 196
current object, 146

data encapsulation, 183, 191, 201
data structure, 200
dead code, 76, 93
dealing, 178
debugging, 4, 10, 77
deck, 163, 169, 173
declaration, 14, 112
decrement, 105, 109, 126
default, 38
detail hiding, 183
deterministic, 137, 144
diagram
    stack, 73, 85
    state, 73, 85
distribution, 138
division
    integer, 20
do-while statement, 53
double (floating-point), 29
Doyle, Arthur Conan, 5

efficiency, 179
element, 134, 144
else, 33
encapsulation, 87, 89, 93, 105, 164
    data, 183, 191, 201
    functional, 183
encode, 157, 169
encrypt, 157
end of file, 196
enumerated type, 171
eof, 196
error, 10
    compile-time, 4, 77
    logic, 4
    run-time, 4, 102, 127
    syntax, 4
exit, 192
expression, 19, 22, 27, 59, 61, 135

factorial, 87
file
    input, 196
file output, 197
fill-in function, 128
find, 104, 165, 198
findBisect, 166
flag, 82, 187
floating-point, 45
floating-point number, 29
flowchart, 41, 45
for each, 135
for loop
    range based, 103
for statement, 55
formal language, 6, 10
frabjuous, 84
fruitful function, 69, 75
function, 74, 89, 125
    accessor, 184, 187
    bool, 82
    definition, 61
    fill-in, 128
    for objects, 124
    fruitful, 69, 75
    helper, 177, 181
    main, 61
    Math, 59
    member, 145, 155, 175
    modifier, 127
    multiple parameter, 68
    nonmember, 146, 155
    pure function, 125
    Time, 68
    void, 75
function objects, 212
functional programming, 131

functions
    vector, 136
functor, 209, 212

generalization, 87, 90, 93, 105, 130
getline, 197
good, 196

header file, 60, 195
hello world, 7
helper function, 177, 181
high-level language, 1, 10
histogram, 142, 144
Holmes, Sherlock, 5

if, 31
if else, 33
ifstream, 196
immutable, 106
implementation, 155, 183
increment, 105, 109, 126
incremental development, 77, 129
index, 102, 109, 135, 144, 164, 203
infinite loop, 49, 56, 197
infinite recursion, 72, 74, 168
initalization
    braced, 16
initialization, 29, 74, 81
    braced, 30
input, 3
    keyboard, 119
instance, 131
instance variable, 121, 131, 173, 185, 187
integer division, 20
interface, 155, 183
interpret, 2, 10
invariant, 190, 194
invoke, 155
iostream, 60
isdigit, 199
isGreater, 162
istream, 196
iteration, 3, 47, 56

keyword, 18, 27

lambda expressions, 213
language
    complete, 83
    formal, 6
    high-level, 1
    low-level, 1
    natural, 6
    programming, 1
    safe, 4
leap of faith, 86, 180
length
    string, 101
linear search, 165
Linux, 5
literalness, 6
local variable, 89, 93
logic error, 4
logical operator, 36, 82
loop, 3, 49, 56, 135
    body, 49
    counting, 105, 140
    do-while, 53
    for, 55
    for each, 135
    infinite, 49, 56, 197
    nested, 164, 174, 204
    search, 165
    while, 47
loop variable, 90, 102, 135
low-level language, 1, 10

main, 61
map to, 157
mapping, 171
math, 3
math function, 59
    acos, 75
    exp, 75
    fabs, 77
    sin, 75
matrix, 203
mean, 138
member function, 145, 155, 175
mergesort, 179, 181
modifier, 127, 131
modulus, 21, 27

multiple assignment, 23

natural language, 6, 10
nested loop, 164
nested structure, 35, 36, 117, 157
newline, 11, 72
nondeterministic, 137
nonmember function, 146, 155

object, 109, 124
    current, 146
    output, 124
    vector of, 163
object-oriented programming, 184
ofstream, 197
operand, 20, 27
operator, 8, 19, 27
    >>, 119
    character, 22
    comparison, 31, 81, 162
    conditional, 37, 93
    decrement, 50, 126
    increment, 50, 126
    logical, 36, 82, 93
    modulus, 21
    remainder, 21
    ternary, 37
order of operations, 21
ordered set, 207
ordering, 162, 200
ostream, 196
output, 3, 8, 11, 124, 188
overloading, 80, 93, 178

parameter, 66, 74, 114
    abstract, 169
    multiple, 68
parameter passing, 114, 115, 118
parse, 6, 10
parsing, 197
parsing number, 199
partial ordering, 162
pass by reference, 121
pass by value, 121
pattern
    accumulator, 200, 207
    counter, 140
    eureka, 165
pi, 75
poetry, 7
Point, 111
pointer, 146
polar coordinate, 185
portability, 1
postcondition, 191, 194
precedence, 21
precondition, 191, 194
print
    Card, 159
    vector of Cards, 165
printCard, 159
printDeck, 165, 175
private, 183, 185
    function, 193
problem-solving, 10
processing, 3
program development, 77, 93
    bottom-up, 140, 177, 181
    encapsulation, 89
    incremental, 129
    planning, 129
    top-down, 177
programming language, 1
programming style, 129
prose, 7
prototyping, 129
pseudocode, 40, 45, 176
pseudorandom, 144
public, 185
pure function, 125, 131, 189

random, 143
random number, 137, 177
rank, 157
Rectangle, 116
recursion, 70, 74, 83, 167, 180
    infinite, 72, 74, 168
recursive, 72
redundancy, 6
reference, 112, 115, 118, 121, 177
remainder, 21
repetition, 3

replit, 3, 8, 11, 222
representation, 183
resize, 205
return, 70, 75, 118
    inside loop, 165
return type, 93
return value, 75, 93
rounding, 30
rules
    precedence, 21
    variable naming, 18
    variable value, 15
run-time error, 4, 72, 102, 127, 135, 168, 192, 202, 204

safe language, 4
same, 161
scaffolding, 78, 93
searching, 165
seed, 143, 144
selection, 3
semantics, 4, 10, 36
Set, 200
set
    ordered, 207
shuffling, 176, 178
size
    vector, 136
sorting, 177, 179
spans, 97
special character, 198
stack, 73, 85
state, 112
state diagram, 112, 159, 164, 173, 203
statement, 3, 27
    assignment, 14, 23
    break, 39, 197
    comment, 8
    conditional, 31
    declaration, 14, 112
    do-while, 53
    for, 55
    for each, 135
    initialization, 81
    output, 8, 11, 124
    return, 70, 75, 118, 165
    switch, 38, 172
    while, 47
static cast, 30
statistics, 138
std::array, 95
stream, 119, 195, 207
    status, 196
string, 11, 106, 107
    concatentation, 200
    length, 101
    native C, 197
struct, 111, 123, 184
    as parameter, 114
    as return type, 118
    instance variable, 112
    operations, 113
    Point, 111
    Rectangle, 116
    Time, 123
structure, 121
structure definition, 175
subdeck, 169, 177
suit, 157
swapCards, 177
switch statement, 38, 45, 172
syntax, 4, 10
syntax error, 4

tab, 27
temporary variable, 76
ternary operator, 37
testing, 3, 168, 180
this, 146
Time, 123
to_array, 96
top-down design, 177
traverse, 102, 109, 165
    counting, 105, 140
Turing, Alan, 84
type, 13, 27
    `bool`, 81
    double, 29
    enumerated, 171
    int, 20
    string, 11
    vector, 133

typecasting, 30

value, 13, 14, 27
    boolean, 81
variable, 14, 27
    instance, 173, 185, 187
    local, 89, 93
    loop, 90, 102, 135
    naming rules, 18
    temporary, 76
variable naming rules, 18
vector, 133, 144
    copying, 135
    element, 134
    functions, 136
    of apstring, 159
    of Cards, 173
    of object, 163
    size, 136
void, 75, 93, 125

warnings, 222
while statement, 47