How to think like a computer scientist

Allen B. Downey

C++ Version, First Edition

Contributor List

by Paul Bui

Greetings and salutations! As a busy student in my senior year at Yorktown High School (Arlington, VA), I have undertaken the assignment of contributing to this open textbook. As a sophomore, I enrolled in Computer Science, which focused on C++ programming, which I then followed up on by enrolling in AP Computer Science during my junior year. I consider myself somewhat familiar with C++ programming by now, which is why I am attempting to pass on my own knowledge of C++.

Allen B. Downey, professor of Computer Science at Wellesley College, originally wrote "How to Think Like a Computer Scientist"in Java, as a textbook for his computer science class. Over the summer of 1998, Professor Downey converted the Java version of "How to Think Like a Computer Scientist" into C++. Since then, the Java version has undergone several major changes, including the addition of Abstract Data Types such as Stacks, Queues, and Heaps. The C++ version of the open textbook however, did not receive these changes, that is...until now.

Of course, my contribution to this open textbook will not be perfect (as I am prone to human error) and will not be the last. If you feel the urge to contribute, comment, or point out errors, please contact Charles Harrison at `speedy911@mindspring.com`. If your contribution, comment, and/or error is legitimate, then you shall be added to this "comprehensive" list of contributors:

Jonah Cohen Jonah wrote the Perl scripts to convert the LaTeX source for this book into beautiful html. He will also contribute to the book several chapters in cooperation with Paul Bui; his major work will be a chapter concerning Object Oriented Programming in C++. His web page is `http://jonah.ticalc.org` and his email is `jonah@ibiblio.org`

Paul Bui Paul Bui will be contributing to the book along with Jonah Cohen, various chapters concerning pointers, references, and templates. Other chapters that involved several Abstract Data Types (Stacks, Queues, Priority Queues, and Heaps), which have already been written for Java will be converted by Paul to C++. His web page is `http://www.paul.bui.as` and his email is `paulbui1@hotmail.com`

Charles Harrison Charles Harrison is currently maintaining the online text.

He corrects errors both technically and grammaticaly. He also serves as the main for help and information about the C++ portion of the Open Book Project. His email is `speedy911@mindspring.com`

Peter Bui Peter Bui has contributed miscellaneous text corrections.

Donald Oellerich Donald Oellerich has contributed miscellaneous text corrections.

Drew Stephens Drew Stephens has helped on countless occasions to perfect the content and accuracy of the online text.

Contents

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 tradeoffs 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."

Of course, the other goal of this book is to prepare you for the Computer Science AP Exam. We may not take the most direct approach to that goal, though. For example, there are not many exercises in this book that are similar to the AP questions. On the other hand, if you understand the concepts in this book, along with the details of programming in C++, you will have all the tools you need to do well on the exam.

1.1 What is a programming language?

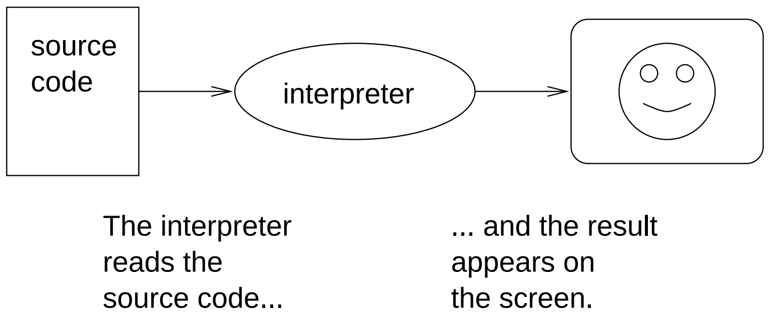
The programming language you will be learning is C++, because that is the language the AP exam is based on, as of 1998. Before that, the exam used Pascal. Both C++ and Pascal are **high-level languages**; other high-level languages you might have heard of are Java, C and FORTRAN.

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.

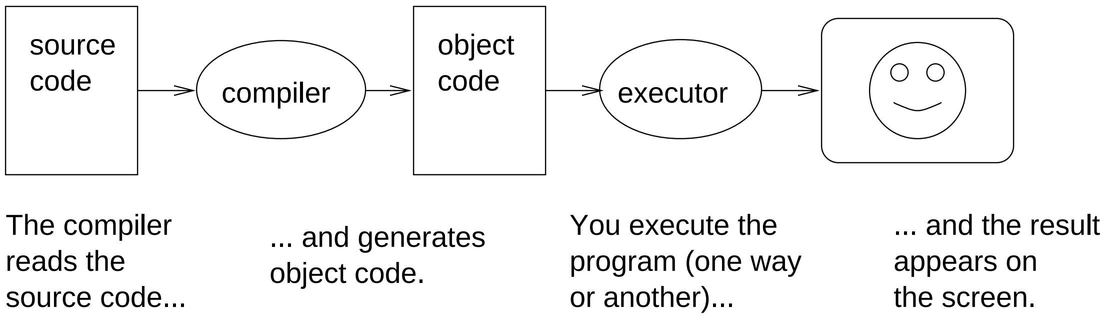
There are two ways to translate a program; **interpreting** or **compiling**. An interpreter is a program that reads a high-level program and does what it says. In effect, it translates the program line-by-line, alternately reading lines and carrying out commands.


source code
interpreter
The interpreter reads the source code...
... and the result appears on the screen.

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` to contain the object code, or `program.exe` to contain the executable.


source code
compiler
object code
executor
The compiler reads the source code...
... and generates object code.
You execute the program (one way or another)...
... and the result appears on the screen.

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 development environments), 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.

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.

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.

testing: Check for certain conditions and execute the appropriate sequence of statements.

repetition: Perform some action repeatedly, usually with some variation.

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.

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.

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. 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.h>

// main: generate some simple output

void main ()
{
  cout << "Hello, world." << endl;
}
```

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

The second 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 third line, you can ignore the word `void` 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.

`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 `cout` and a string, and that causes the string to be displayed.

`endl` is a special symbol that represents the end of a line. When you send an `endl` to `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.

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 squiggly-braces ({ and }) to group things together. In this case, the output statement is enclosed in squiggly-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. The details of how to do that depend on your programming environment, but from now on in this book I will assume that you know how to do it.

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.h. Is that what you meant, by any chance?"

Unfortunately, few compilers are so accomodating. 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 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.h>

// main: generate some simple output

void main ()
{
  cout << "Hello, world." << endl;     // output one line
  cout << "How are you?" << endl;      // output another
}
```

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

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`:

```
void main ()
{
  cout << "Goodbye, ";
  cout << "cruel world!" << endl;
}
```

In this case the output appears on a single line as `Goodbye, cruel world!`. Notice that there is a space between the word "Goodbye," 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:

```
void main ()
{
cout<<"Goodbye, ";
cout<<"cruel world!"<<endl;
}
```

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:

```
void main(){cout<<"Goodbye, ";cout<<"cruel world!"<<endl;}
```

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.

2.2 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:

```
  cout << 17 << 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:

```
  cout << '}' << endl;
```

This example outputs a single close squiggly-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 `fred` that has type `char`.

```
    char fred;
```

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 `second` 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**.

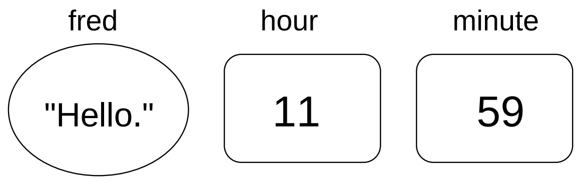
```
    firstLetter = 'a';   // give firstLetter the value 'a'
    hour = 11;           // assign the value 11 to hour
    minute = 59;         // set minute to 59
```

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 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 = ':';

cout << "The current time is ";
```

```
  cout << hour;
  cout << colon;
  cout << minute;
  cout << endl;
```

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 current time is 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: `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 = ':';

  cout << "The current time is " << hour << colon << minute << endl;
```

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

2.6 Keywords

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 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`, `return` 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) on September 1, 1998. 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 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.

2.7 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.

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

```
  int hour, minute;
  hour = 11;
  minute = 59;
  cout << "Number of minutes since midnight: ";
  cout << hour*60 + minute << endl;
  cout << "Fraction of the hour that has passed: ";
  cout << minute/60 << endl;
```

would generate the following output:

```
Number of minutes since midnight: 719
Fraction of the hour that has passed: 0
```

The first line is what we expected, but the second line is odd. The value of the variable `minute` is 59, and 59 divided by 60 is 0.98333, 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:

```
  cout << "Percentage of the hour that has passed: ";
  cout << minute*100/60 << endl;
```

The result is:

```
Percentage of the hour that has passed: 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 in the next chapter.

2.8 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 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.

2.9 Operators for characters

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

```
char letter;
letter = 'a' + 1;
cout << letter << 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';
cout << number << 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.

2.10 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:

```
    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:

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

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

```
  int percentage;
  percentage = (minute * 100) / 60;
```

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;`.

2.11 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.

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.

Chapter 3

Function

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 `doubles` 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;
```

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**.

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 x = 1.1;
```

because the variable on the left is an `int` and the value on the right is a `double`. But it is easy to forget this 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.

3.2 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 like the syntax for a function call. For example:

```
double pi = 3.14159;
int x = int (pi);
```

The `int` function 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.

3.3 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 = log (17.0);
    double angle = 1.5;
    double height = 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 = 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 math **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.h` using an **include** statement:

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

`iostream.h` contains information about input and output (I/O) streams, including the object named `cout`.

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

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

3.4 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.

3.5 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. The first couple of functions we are going to write also have no parameters, so the syntax looks like this:

```
  void newLine () {
    cout << 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:

```
void main ()
{
  cout << "First Line." << endl;
  newLine ();
  cout << "Second Line." << endl;
}
```

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:

```
void main ()
{
  cout << "First Line." << endl;
  newLine ();
  newLine ();
  newLine ();
  cout << "Second Line." << endl;
}
```

Or we could write a new function, named `threeLine`, that prints three new lines:

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

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

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 `cout << 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?

3.6 Definitions and uses

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

```
#include <iostream.h>

void newLine ()
{
  cout << endl;
}

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

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

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.

3.7 Programs with multiple functions

When you look at a class definition 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.

3.8 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 class definition, the parameter list indicates the type of each parameter. For example:

```
void printTwice (char phil) {
  cout << phil << phil << 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:

```
void main () {
  printTwice ('a');
}
```

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:

```
void main () {
  char argument = 'b';
  printTwice (argument);
}
```

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').

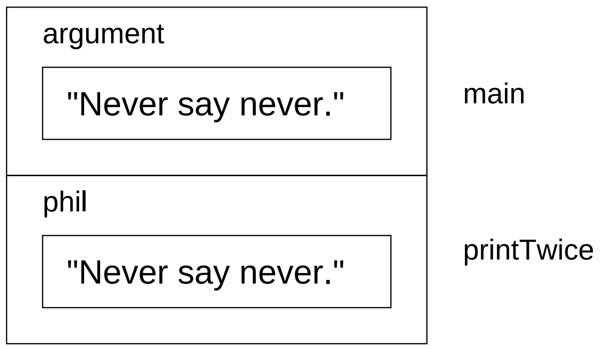
The value you provide as an argument must have the same type as the parameter of the function you call. This rule is important, but it is sometimes confusing because C++ sometimes converts arguments from one type to another automatically. For now you should learn the general rule, and we will deal with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

3.9 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 state diagram for `printTwice` looks like this:



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 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`.

3.10 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) {
  cout << hour;
  cout << ":";
  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)`.

3.11 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 a couple of chapters. 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.

3.12 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`.

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.

Chapter 4

Conditionals and recursion

4.1 The modulus 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.

4.2 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) {
  cout << "x is positive" << endl;
}
```

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`:

```
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 `=>`.

The two sides of a condition operator have to be the same type. You can only compare `ints` to `ints` and `doubles` to `doubles`. Unfortunately, at this point you can't compare `Strings` at all! There is a way to compare `Strings`, but we won't get to it for a couple of chapters.

4.3 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. The syntax looks like:

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

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.

As an aside, 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) {
    cout << "x is even" << endl;
  } else {
    cout << "x is odd" << endl;
  }
}
```

Now you have a function named `printParity` that will display an appropriate message for any integer you care to provide. 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!!!
```

4.4 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) {
    cout << "x is positive" << endl;
  } else if (x < 0) {
    cout << "x is negative" << endl;
  } else {
    cout << "x is zero" << 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 squiggly-braces lined up, you are less likely to make syntax errors and you can find them more quickly if you do.

4.5 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) {
    cout << "x is zero" << endl;
  } else {
    if (x > 0) {
      cout << "x is positive" << endl;
    } else {
      cout << "x is negative" << 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.

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.

4.6 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 <math.h>

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

  double result = log (x);
  cout << "The log of x is " << 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 `math.h`.

4.7 Recursion

I mentioned in the last chapter 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, look at the following function:

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

```
void main ()
{
  countdown (3);
}
```

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 () {
  cout << endl;
}

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

Although these work, they would not be much help if I wanted to output 2 newlines, or 106. A better alternative would be

```
void nLines (int n) {
  if (n > 0) {
    cout << 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**.

4.8 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).

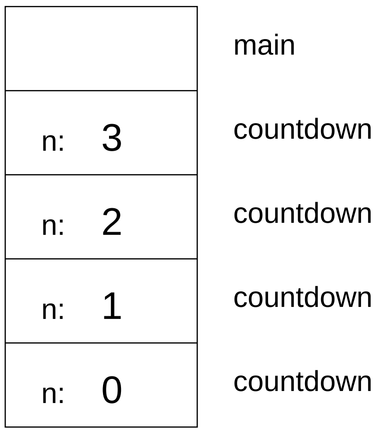
You should write a small program that recurses forever and run it to see what happens.

4.9 Stack diagrams for recursive functions

In the previous chapter 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`.

4.10 Glossary

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 (`%`).

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.

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 5

Fruitful functions

5.1 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.

5.2 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,

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{5.1}$$

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 doubles. 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.

5.3 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));
}
```

5.4 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 accidently 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.

5.5 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).

5.6 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.

5.7 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.

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

Finally, the NOT operator has the effect of negating or inverting a bool expression, so `!evenFlag` is true if `evenFlag` 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) {
      cout << "x is a positive single digit." << endl;
    }
  }
```

5.8 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`. (There is a way to fix that using the `boolalpha` flag, but it is too hideous to mention.)

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;
  }
```

5.9 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.

5.10 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 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:

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

(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;
  }
}
```

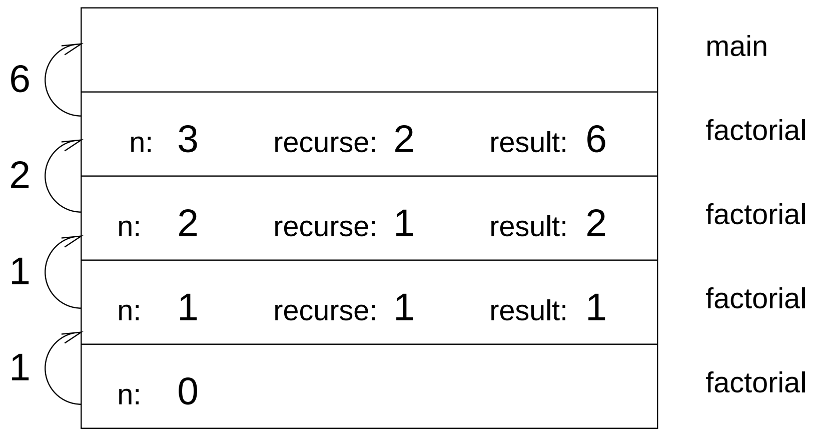
If we look at the flow of execution for this program, it is similar to `nLines` from the previous chapter. If we call `factorial` with the value 3:

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:


6
2
1
1
n: 3 recurse: 2 result: 6
n: 2 recurse: 1 result: 2
n: 1 recurse: 1 result: 1
n: 0
main
factorial
factorial
factorial
factorial

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.

5.11 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 5.8 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!

5.12 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:

$$\begin{aligned} fibonacci(0) &= 1 \\ fibonacci(1) &= 1 \\ fibonacci(n) &= fibonacci(n-1) + fibonacci(n-2); \end{aligned}$$

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.

5.13 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.

Chapter 6

Iteration

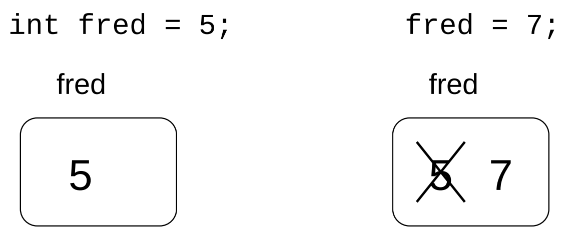
6.1 Multiple assignment

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.

6.2 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.

We have seen programs that use recursion to perform repetition, such as nLines and countdown. This type of repetition is called **iteration**, and C++ provides several language features that make it easier to write iterative programs.

The two features we are going to look at are the while statement and the for statement.

6.3 The while statement

Using a while statement, we can rewrite countdown:

```
void countdown (int n) {
  while (n > 0) {
    cout << n << endl;
    n = n-1;
  }
  cout << "Blastoff!" << 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!"'

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 squiggly-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:

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

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!

6.4 Tables

One of the things loops are good for is generating tabular data. For example, before computers were readily available, people had to calculate logarithms, sines and cosines, and other common mathematical functions by hand. To make that easier, there were books containing long tables where you could find the values of various functions. Creating these tables was slow and boring, and the result tended to be full of errors.

When computers appeared on the scene, one of the initial reactions was, "This is great! We can use the computers to generate the tables, so there will be no errors." That turned out to be true (mostly), but shortsighted. Soon thereafter computers and calculators were so pervasive that the tables became obsolete.

Well, almost. It turns out that for some operations, computers use tables of values to get an approximate answer, and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the original Intel Pentium used to perform floating-point division.

Although a "log table" is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and their logarithms in the right column:

```
  double x = 1.0;
  while (x < 10.0) {
    cout << x << "\t" << log(x) << "\n";
    x = x + 1.0;
  }
```

The sequence `\t` represents a **tab** character. The sequence `\n` represents a newline character. These sequences can be included anywhere in a string, although in these examples the sequence is the whole string.

A tab character causes the cursor to shift to the right until it reaches one of the **tab stops**, which are normally every eight characters. As we will see in a minute, tabs are useful for making columns of text line up.

A newline character has exactly the same effect as `endl`; it causes the cursor to move on to the next line. Usually if a newline character appears by itself, I use `endl`, but if it appears as part of a string, I use `\n`.

The output of this program is

```
1       0
2       0.693147
3       1.09861
4       1.38629
5       1.60944
6       1.79176
7       1.94591
8       2.07944
```

```
9       2.19722
```

If these values seem odd, remember that the `log` function uses base e. Since powers of two are so important in computer science, we often want to find logarithms with respect to base 2. To do that, we can use the following formula:

$$\log_2 x = \frac{log_e x}{log_e 2}$$

Changing the output statement to

```
      cout << x << "\t" << log(x) / log(2.0) << endl;
```

yields

```
1       0
2       1
3       1.58496
4       2
5       2.32193
6       2.58496
7       2.80735
8       3
9       3.16993
```

We can see that 1, 2, 4 and 8 are powers of two, because their logarithms base 2 are round numbers. If we wanted to find the logarithms of other powers of two, we could modify the program like this:

```
  double x = 1.0;
  while (x < 100.0) {
    cout << x << "\t" << log(x) / log(2.0) << endl;
    x = x * 2.0;
  }
```

Now instead of adding something to `x` each time through the loop, which yields an arithmetic sequence, we multiply `x` by something, yielding a **geometric** sequence. The result is:

```
1       0
2       1
4       2
8       3
16      4
32      5
64      6
```

Because we are using tab characters between the columns, the position of the second column does not depend on the number of digits in the first column.

Log tables may not be useful any more, but for computer scientists, knowing the powers of two is! As an exercise, modify this program so that it outputs the powers of two up to 65536 (that's 2^{16}). Print it out and memorize it.

6.5 Two-dimensional tables

A two-dimensional table is a table where you choose a row and a column and read the value at the intersection. A multiplication table is a good example. Let's say you wanted to print a multiplication table for the values from 1 to 6.

A good way to start is to write a simple loop that prints the multiples of 2, all on one line.

```
  int i = 1;
  while (i <= 6) {
    cout << 2*i << "   ";
    i = i + 1;
  }
  cout << endl;
```

The first line initializes a variable named `i`, which is going to act as a counter, or **loop variable**. As the loop executes, the value of `i` increases from 1 to 6, and then when `i` is 7, the loop terminates. Each time through the loop, we print the value `2*i` followed by three spaces. By omitting the `endl` from the first output statement, we get all the output on a single line.

The output of this program is:

```
2   4   6   8   10   12
```

So far, so good. The next step is to **encapsulate** and **generalize**.

6.6 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 4.3 and `isSingleDigit` in Section 5.8.

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.

6.7 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.

6.8 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.

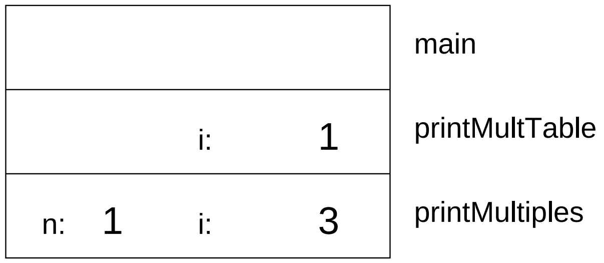
6.9 Local variables

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.

6.10 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.

6.11 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.

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

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.

Chapter 7

Strings and things

7.1 Containers for strings

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 some concepts we have not covered yet, so for the most part we are going to avoid them.

The string type we are going to use is called `pstring`, which is an open-source alternative to a type created specifically for the Computer Science AP Exam. The pclasses, as the entire group of open-source classes are called, can be found in the appendix section of this book. You can also visit their homepage at `http://www.ibiblio.org/obp/pclasses/`.

You might be wondering what is meant 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 `pstring` class contains all the functions that apply to `pstring`s.

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 `pstring`s instead of C strings.

7.2 `pstring` variables

You can create a variable with type `pstring` in the usual ways:

```
pstring first;
first = "Hello, ";
```

```
  pstring second = "world.";
```

The first line creates an `pstring` 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 an `pstring` variable, they are converted automatically to `pstring` values.

We can output strings in the usual way:

```
  cout << first << second << endl;
```

In order to compile this code, you will have to include the header file for the `pstring` class, and you will have to add the file `pstring.cpp` to the list of files you want to compile. The details of how to do this depend on your programming environment.

Before proceeding, you should type in the program above and make sure you can compile and run it.

7.3 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++ uses square brackets (`[` and `]`) for this operation:

```
  pstring fruit = "banana";
  char letter = fruit[1];
  cout << letter << endl;
```

The expression `fruit[1]` indicates 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];
```

7.4 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];
```

7.5 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 an `pstring` as an argument and that outputs the letters backwards, all on one line.

7.6 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 `[]` operator 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.

7.7 The `find` function

The `pstring` 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.

```
  pstring 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 -1.

In addition, there is a version of `find` that takes another `pstring` as an argument and that finds the index where the substring appears in the string. For example,

```
  pstring fruit = "banana";
  int index = fruit.find("nan");
```

This example returns the value 2.

You should remember from Section 5.4 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.

7.8 Our own version of `find`

If we are looking for a letter in an `pstring`, 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 (pstring s, char c, int i)
{
  while (i<s.length()) {
    if (s[i] == c) return i;
    i = i + 1;
  }
  return -1;
}
```

Instead of invoking this function on an `pstring`, like the first version of `find`, we have to pass the `pstring` as the first argument. The other arguments are the character we are looking for and the index where we should start.

7.9 Looping and counting

The following program counts the number of times the letter `'a'` appears in a string:

```
  pstring 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.

7.10 Increment and decrement operators

Incrementing and decrementing 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 `pstring`s, and neither *should* be used on `bool`s.

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:

```
cout << i++ << 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. If you really want to know, you can try it.

Using the increment operators, we can rewrite the letter-counter:

```
int index = 0;
while (index < length) {
  if (fruit[index] == 'a') {
    count++;
  }
  index++;
}
```

It is a common error to write something like

```
index = index++;             // WRONG!!
```

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

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

7.11 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:

```
  pstring fruit = "banana";
  pstring bakedGood = " nut bread";
  pstring 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

```
  pstring dessert = "banana" + " nut bread";
```

because both operands are C strings. As long as one of the operands is an `pstring`, though, C++ will automatically convert the other.

It is also possible to concatenate a character onto the beginning or end of an `pstring`. 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:

```
  pstring 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 `pstring`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.

7.12 `pstring`s are mutable

You can change the letters in an `pstring` one at a time using the `[]` operator on the left side of an assignment. For example,

```
pstring greeting = "Hello, world!";
greeting[0] = 'J';
cout << greeting << endl;
```

produces the output `Jello, world!`.

7.13 `pstring`s are comparable

All the comparison operators that work on `int`s and `double`s also work on `pstrings`. 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 `pstring` 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.

7.14 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 `ctype.h`.

```
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 `pstringToUpper` and `pstringToLower` that take a single `pstring` as a parameter, and that modify the string by converting all the letters to upper or lower case. The return type should be `void`.

7.15 Other `pstring` functions

This chapter does not cover all the `pstring` functions. Two additional ones, `c_str` and `substr`, are covered in Section 23.2 and Section 23.4.

7.16 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, and `pstring`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. `pstring`s are different in the sense that they are made up of smaller pieces, the characters. Thus, `pstring`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.

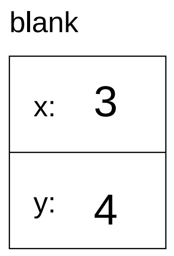
It is a common error to leave off the semi-colon at the end of a structure definition. It might seem odd to put a semi-colon after a squiggly-brace, but you'll get used to it.

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 squiggly 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, and there is no good answer. I'm sorry.

On the other hand, 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;
}
```

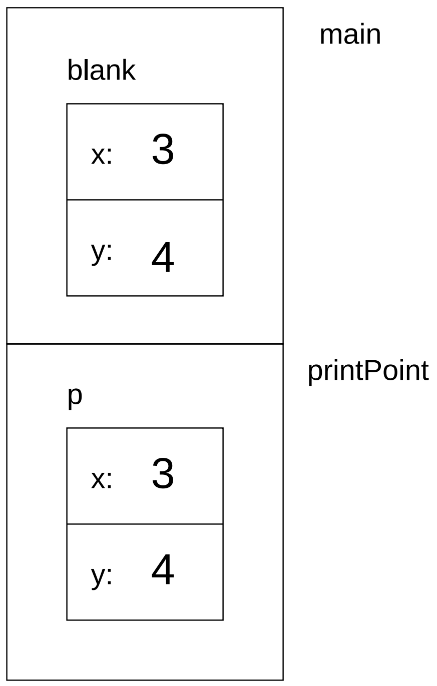
`printPoint` takes a point as an argument and outputs it in the standard format. If you call `printPoint (blank)`, it will output `(3, 4)`.

As a second example, we can rewrite the `distance` function from Section 5.2 so that it takes two `Points` as parameters instead of four `doubles`.

```
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 Pass 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 ppropriate.

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 Pass 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;
}
```

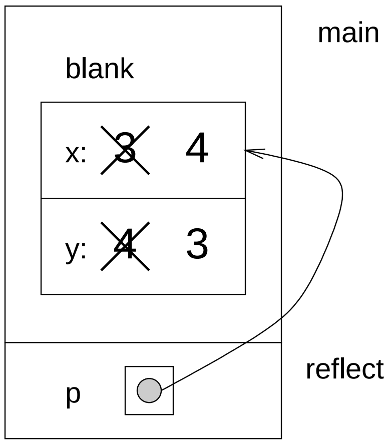
Now we can call the function in the usual way:

```
  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.

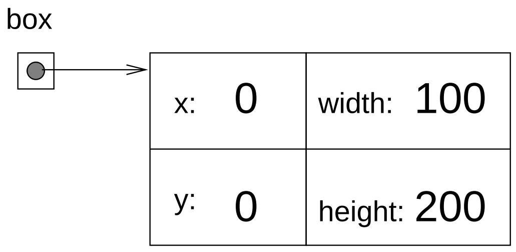
The most common choice in existing programs is to specify the upper left corner of the rectangle and the size. To do that in C++, we will define a structure that contains a `Point` and two doubles.

```
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 squiggly 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.h, 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:

```
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 `pstring`:

```
  pstring 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 `pstring` called `getline`.

```
  pstring 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 `pstring` 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 `stdlib.h`. We will get to that in Section 23.4.

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.

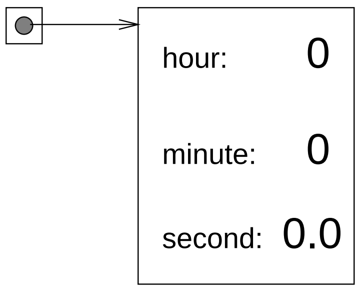
Here's what the structure definition looks like:

```
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`.

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 `Times` 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 ppropriate 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 pproach 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 ppropriately 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 pproach 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 pproach might be called a functional programming style.

9.9 Incremental development versus planning

In this chapter I have demonstrated an pproach 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 pproach 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 pproach to the whole problem is to convert `Times` into `doubles` and take advantage of the fact that the computer already knows how to do arithmetic with `doubles`. 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 `Times` to find the duration between them. The naive pproach 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 pproaches.

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 an 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 so 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). An `pstring` 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 that appears on the AP exam is called `pvector`. In order to use it, you have to include the header file `pvector.h`; 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:

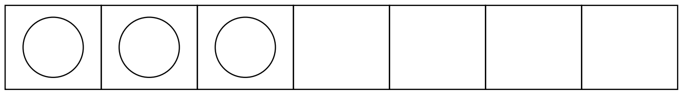
```
pvector<int> count;
pvector<double> doubleVector;
```

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 length is zero). It is more common to specify the length of the vector in parentheses:

```
pvector<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 an `pvector` 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 `pvectors` that takes two parameters; the second is a "fill value," the value that will be assigned to each of the elements.

```
pvector<int> count (4, 0);
```

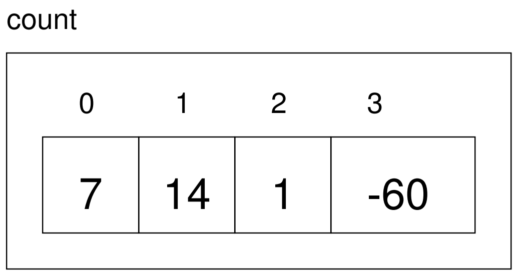
This statement creates a vector of four elements and initializes all of them to zero.

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 `pstring`. As with `pstrings`, 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 `pvector`s, which is called a copy constructor because it takes one `pvector` as an argument and creates a new vector that is the same size, with the same elements.

```
  pvector<int> copy (count);
```

Although this syntax is legal, it is almost never used for `pvector`s because there is a better alternative:

```
  pvector<int> copy = count;
```

The `=` operator works on `pvector`s in pretty much the way you would expect.

10.3 `for` loops

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 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:

```
for (int 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 length

There are only a couple of functions you can invoke on an `pvector`. One of them is very useful, though: `length`. Not surprisingly, it returns the length 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.

```
for (int i = 0; i < count.length(); i++) {
  cout << count[i] << endl;
}
```

The last time the body of the loop gets executed, the value of `i` is `count.length() - 1`, which is the index of the last element. When `i` is equal to `count.length()`, the condition fails and the body is not executed, which is a good thing, since it would cause a run-time error.

10.5 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 `stdlib.h`, 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:

```
  for (int i = 0; i < 4; i++) {
    int x = random ();
    cout << x << endl;
  }
```

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.

10.6 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.

10.7 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 `ints`, and fills it with random values between 0 and `upperBound-1`.

```
pvector<int> randomVector (int n, int upperBound) {
  pvector<int> vec (n);
  for (int i = 0; i<vec.length(); i++) {
    vec[i] = random () % upperBound;
  }
  return vec;
}
```

The return type is `pvector<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 pvector<int>& vec) {
  for (int i = 0; i<vec.length(); i++) {
    cout << vec[i] << " ";
  }
}
```

Notice that it is legal to pass `pvectors` 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;
  pvector<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`.

10.8 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 7.9 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 pply 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 pvector<int>& vec, int value) {
  int count = 0;
  for (int i=0; i< vec.length(); i++) {
    if (vec[i] == value) count++;
  }
  return count;
}
```

10.9 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;
  pvector<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.

10.10 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 length 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;
pvector<int> vector = randomVector (numValues, upperBound);
pvector<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.

10.11 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:

```
pvector<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.

10.12 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.

10.13 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 objecs).

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 pparent 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.

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 squiggly-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 squiggly-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 ppropriate 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 pparent 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.h>
#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.h>
```

```
#include "Time.h"

void 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;
  }
}
```

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.h>
```

`iostream.h` 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 `pstring`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 `Cards`. 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 `pstrings`. You can create a vector of `pstrings` the same way you create an vector of other types:

```
pvector<pstring> suits (4);
```

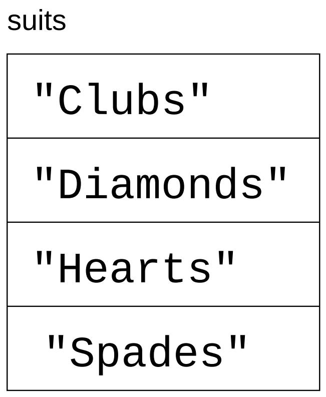
Of course, in order to use `pvectors` and `pstrings`, you will have to include the header files for both.

`pvectors` are a little different from `pstrings` in this regard. The file `pvector.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 `pstrings`, the compiler generates different code to handle that kind of vector. As a result, it is usually sufficient to include the header file `pvector.h`; you do not have to compile `pvector.cpp` at all! Unfortunately, if you do, you are likely to get a long stream of error messages. I hope this helps you avoid an unpleasant surprise, but the details in your development environment may differ.

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:



We can build a similar vector to decode the ranks. Then we can select the ppropriate 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
{
  pvector<pstring> suits (4);
  suits[0] = "Clubs";
  suits[1] = "Diamonds";
  suits[2] = "Hearts";
  suits[3] = "Spades";

  pvector<pstring> 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 ppropriate 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 pples 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 `Cards`) 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
```

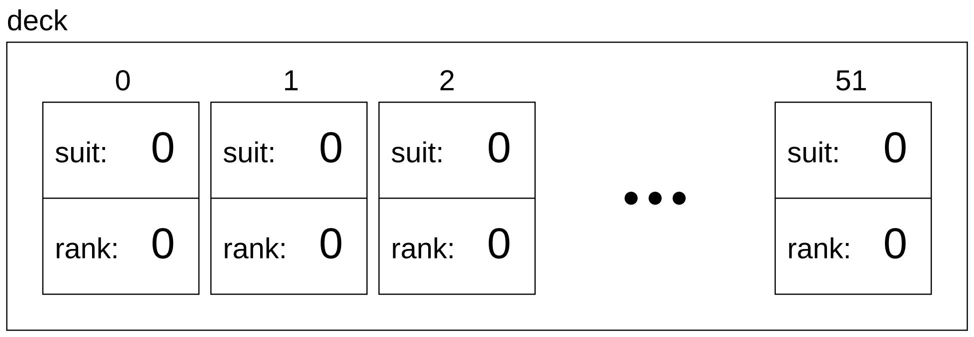
According to `isGreater`, aces are less than deuces (2s). As an exercise, fix it so that aces are ranked higher than Kings, as they are in most card games.

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:

```
pvector<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);
pvector<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 Cards.

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 pvector<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 pvector<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 Cards 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 pvector<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:

```
  pvector<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 pvector<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 pvector<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 pvector<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`:

```
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. 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 '+':
  perform_addition ();
  break;
case '*':
  perform_multiplication ();
  break;
default:
  cout << "I only know how to perform addition and multiplication" << endl;
  break;
}
```

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

```
if (symbol == '+') {
  perform_addition ();
} else if (symbol == '*') {
  perform_multiplication ();
} else {
  cout << "I only know how to perform addition and multiplication" << endl;
}
```

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.

`switch` statements work with integers, characters, and 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.

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

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 `Cards`.

The structure definition looks like this

```
struct Deck {
  pvector<Card> cards;

  Deck (int n);
};

Deck::Deck (int size)
{
  pvector<Card> temp (size);
  cards = temp;
}
```

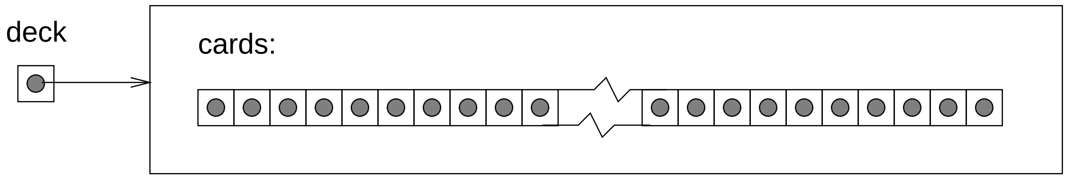
The name of the instance variable is `cards` to help distinguish the `Deck` object from the vector of `Cards` that it contains.

For now there is only one constructor. It creates a local variable named `temp`, which it initializes by invoking the constructor for the `pvector` 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 ()
{
  pvector<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 Decks 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 Cards and Decks 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 {
  pvector<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 10.5 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 10.5, 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 10.8.

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

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

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 `pstring` and `pvector` 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; }
};
```

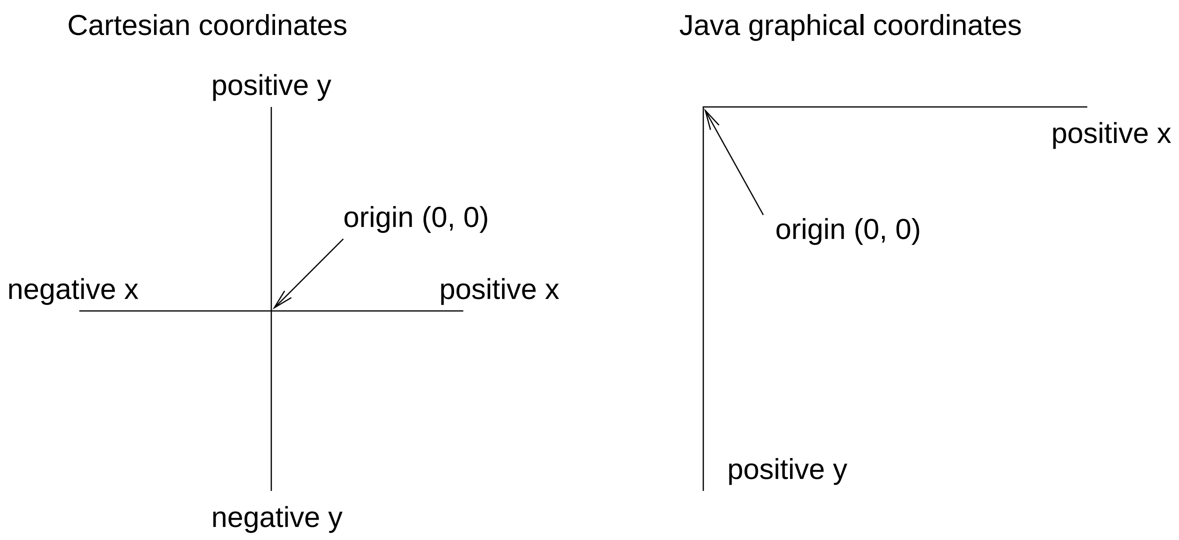
Because this is a `class` definition, the instance variables `real` and `imag` are private, and we have to include the label `public:` to allow client code to invoke the constructors.

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

Object-oriented programming

Jonah Cohen

15.1 Programming languages and styles

There are many programming languages in the world, and almost as many programming styles (sometimes called paradigms). Three styles that have appeared in this book are procedural, functional, and object-oriented. Although C++ is usually thought of as an object-oriented language, it is possible to write C++ programs in any style. The style I have demonstrated in this book is pretty much procedural. Existing C++ programs and C++ system libraries are written in a mixture of all three styles, but they tend to be more object-oriented than the programs in this book.

It's not easy to define what object-oriented programming is, but here are some of its characteristics:

- Object definitions (classes) usually correspond to relevant real-world objects. For example, in Chapter 13.3, the creation of the `Deck` class was a step toward object-oriented programming.
- The majority of functions are member functions (the kind you invoke on an object) rather than nonmember functions (the kind you just invoke). So far all the functions we have written have been nonmember functions. In this chapter we will write some member functions.
- The language feature most associated with object-oriented programming is **inheritance**. I will cover inheritance later in this chapter.

Recently object-oriented programming has become quite popular, and there are people who claim that it is superior to other styles in various ways. I hope that by exposing you to a variety of styles I have given you the tools you need to understand and evaluate these claims.

15.2 Member and nonmember functions

There are two types of functions in C++, called **nonmember functions** and **member functions**. So far, every function we have written has been a nonmember function. Member functions are declared inside class defintions. Any function declared outside of a class is a nonmember function.

Although we have not written any member functions, we have invoked some. Whenever you invoke a function "on" an object, it's a member function. Also, the functions we invoked on `pstring`s in Chapter 7 were member functions.

Anything that can be written as a nonmember function can also be written as a member function, and vice versa. Sometimes it is just more natural to use one or the other. For reasons that will be clear soon, member functions are often shorter than the corresponding nonmember functions.

15.3 The current object

When you invoke a function on an object, that object becomes **the current object**. Inside the function, you can refer to the instance variables of the current object by name, without having to specify the name of the object.

You can also refer to the current object through the keyword `this`. We have already seen `this` in an assignment operator in Section 11.2. However, the `this` keyword is implicit most of the time, so you will rarely find any need for it.

15.4 Complex numbers

Continuing the example from the previous 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. Thus, $i \cdot i = -1$.

The following is a class definition for a new object type called `Complex`:

```
class Complex
{
private:
  double real, imag;

public:
  Complex () {
    real = 0.0;  imag = 0.0;
  }

  Complex (double r, double i) {
    real = r;  imag = i;
  }
};
```

There should be nothing surprising here. The instance variables are two `doubles` that contain the real and imaginary parts. The two constructors are the usual kind: one takes no parameters and assigns default values to the instance variables, the other takes parameters that are identical to the instance variables.

In `main`, or anywhere else we want to create `Complex` objects, we have the option of creating the object and then setting the instance variables, or doing both at the same time:

```
    Complex x;
    x.real = 1.0;
    x.imag = 2.0;
    Complex y (3.0, 4.0);
```

15.5 A function on `Complex` numbers

Let's look at some of the operations we might want to perform on complex numbers. The absolute value of a complex number is defined to be $\sqrt{x^2 + y^2}$. The `abs` function is a pure function that computes the absolute value. Written as a nonmember function, it looks like this:

```
  // nonmember function
  double abs (Complex c) {
    return sqrt (c.real * c.real + c.imag * c.imag);
  }
```

This version of `abs` calculates the absolute value of `c`, the `Complex` object it receives as a parameter. The next version of `abs` is a member function; it calculates the absolute value of the current object (the object the function was invoked on). Thus, it does not receive any parameters:

```
class Complex
{
private:
  double real, image;

public:
  // ...constructors

  // member function
  double abs () {
    return sqrt (real*real + imag*imag);
  }
};
```

I removed the unnecessary parameter to indicate that this is a member function. Inside the function, I can refer to the instance variables `real` and `imag` by name without having to specify an object. C++ knows implicitly that I am referring to the instance variables of the current object. If I wanted to make it explicit, I could have used the keyword `this`:

```
class Complex
{
private:
  double real, image;

public:
  // ...constructors

  // member function
  double abs () {
    return sqrt (this->real * this->real + this->imag * this->imag);
  }
};
```

But that would be longer and not really any clearer. To invoke this function, we invoke it on an object, for example

```
    Complex y (3.0, 4.0);
    double result = y.abs ();
```

15.6 Another function on `Complex` numbers

Another operation we might want to perform on complex numbers is addition. You can add complex numbers by adding the real parts and adding the imaginary parts. Written as a nonmember function, that looks like:

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

To invoke this function, we would pass both operands as arguments:

```
    Complex sum = Add (x, y);
```

Written as a member function, it would take only one argument, which it would add to the current object:

```
  Complex Add (Complex& b) {
    return Complex (real + b.real, imag + b.imag);
  }
```

Again, we can refer to the instance variables of the current object implicitly, but to refer to the instance variables of b we have to name b explicitly using dot notation. To invoke this function, you invoke it on one of the operands and pass the other as an argument.

```
    Complex sum = x.Add (y);
```

From these examples you can see that the current object (this) can take the place of one of the parameters. For this reason, the current object is sometimes called an **implicit** parameter.

15.7 A modifier

As yet another example, we'll look at conjugate, which is a modifier function that transforms a Complex number into its complex conjugate. The complex conjugate of $x + yi$ is $x - yi$.

As a nonmember function, this looks like:

```
  void conjugate (Complex& c) {
    c.imag = -c.imag;
  }
```

As a member function, it looks like

```
  void conjugate () {
    imag = -imag;
  }
```

By now you should be getting the sense that converting a function from one kind to another is a mechanical process. With a little practice, you will be able to do it without giving it much thought, which is good because you should not be constrained to writing one kind of function or the other. You should be equally familiar with both so that you can choose whichever one seems most appropriate for the operation you are writing.

For example, I think that Add should be written as a nonmember function because it is a symmetric operation of two operands, and it makes sense for both operands to appear as parameters. It just seems odd to invoke the function on one of the operands and pass the other as an argument. (Actually, in the next section you'll learn of a method called **operator overloading** which eliminates the need for explicitly calling functions like Add.)

On the other hand, simple operations that apply to a single object can be written most concisely as member functions (even if they take some additional arguments).

15.8 Operator overloading and <<

There are two operators that are common to many object types: << and =. << converts the object to some reasonable string representation so it can be outputted, and = is used to copy objects.

When you output an object using cout, C++ checks to see whether you have provided a << definition for that object. If it can't find one, it will refuse to compile and give an error such as

```
complex.cpp:11: no match for `_IO_ostream_withassign & << Complex &'
```

Here is what << might look like for the Complex class:

```
ostream& operator << (ostream& os, Complex& num) {
  os << num.real << " + " << num.imag << "i";
  return os;
}
```

Whenever you pass an object to an output stream such as cout, C++ invokes the << operator on that object and outputs the result. In this case, the output is 1 + 2i.

The return type for << is ostream&, which is the datatype of a cout object. By returning the os object (which, like ostream, is just an abbreviation of output stream), you can string together multiple << commands such as

```
cout << "Your two numbers are " << num1 << " and " << num2;
```

To illustrate why that's a good thing, consider what you would be forced to do if you didn't return the ostream object:

```
cout << "Your two numbers are ";
cout << num1;
cout << " and ";
cout << num2;
```

Because the first example allows stringing << statements together, all the display code fits easily on one line. The output from both statements is the same, displaying "Your two numbers are 3 + 2i and 1 + 5i".

This version of << does not look good if the imaginary part is negative. As an exercise, fix it.

15.9 The = operator

Unlike the << operator, which refuses to output classes that haven't defined their own definition of that function, every class comes with its own =, or assignment, operator. This default operator simply copies every data member from one class instance to the other by using the = operator on each member variable.

When you create a new object type, you can provide your own definition of = by including a member function called `operator =`. For the `Complex` class, this looks like:

```
const Complex& operator = (Complex& b) {
  real = b.real;
  imag = b.imag;
  return *this;
}
```

By convention, = is always a member function. It returns the current object. (Remember `this` from Section 11.2?) This is similar to how << returns the ostream object, because it allows you to string together several = statements:

```
Complex a, b, c;
c.real = 1.0;
c.imag = 2.0;
a = b = c;
```

In the above example, c is copied to b, and then b is copied to a. The result is that all three variables contain the data originally stored in c. While not used as often as stringing together << statements, this is still a useful feature of C++.

The purpose of the `const` in the return type is to prevent assignments such as:

```
(a = b) = c;
```

This is a tricky statement, because you may think it should just assign c to a and b like the earlier example. However, in this case the parentheses actually mean that the *result* of the statement `a = b` is being assigned a new value, which would actually assign it to a and bypass b altogether. By making the return type `const`, we prevent this from happening.

15.10 Invoking one member function from another

As you might expect, it is legal and common to invoke one member function from another. For example, to normalize a complex number, you divide through (both parts) by the absolute value. It may not be obvious why this is useful, but it is.

Let's write the function `normalize` as a member function, and let's make it a modifier.

```
void normalize () {
  double d = this->abs();
```

```
    real = real/d;
    imag = imag/d;
  }
```

The first line finds the absolute value of the current object by invoking `abs` on the current object. In this case I named the current object explicitly, but I could have left it out. If you invoke one member function within another, C++ assumes that you are invoking it on the current object.

As an exercise, rewrite `normalize` as a pure function. Then rewrite it as a nonmember function.

15.11 Oddities and errors

If you have both member functions and nonmember functions in the same class definition, it is easy to get confused. A common way to organize a class definition is to put all the constructors at the beginning, followed by all the member functions and then all the nonmember functions.

You can have a member function and a nonmember function with the same name, as long as they do not have the same number and types of parameters. As with other kinds of overloading, C++ decides which version to invoke by looking at the arguments you provide.

Since there is no current object in a nonmember function, it is an error to use the keyword `this`. If you try, you might get an error message like: "Undefined variable: this." Also, you cannot refer to instance variables without using dot notation and providing an object name. If you try, you might get "Can't make a static reference to nonstatic variable..." This is not one of the better error messages, since it uses some non-standard language. For example, by "nonstatic variable" it means "instance variable." But once you know what it means, you know what it means.

15.12 Inheritance

The language feature that is most often associated with object-oriented programming is **inheritance**. Inheritance is the ability to define a new class that is a modified version of a previously-defined class (including built-in classes).

The primary advantage of this feature is that you can add new functions or instance variables to an existing class without modifying the existing class. This is particularly useful for built-in classes, since you can't modify them even if you want to.

The reason inheritance is called "inheritance" is that the new class inherits all the instance variables and functions of the existing class. Extending this metaphor, the existing class is sometimes called the **parent** class and the new class is called the **subclass**.

15.13 Message class

An an example of inheritance, we are going to take a message class and create a subclass of error messages. That is, we are going to create a new class called `ErrorMessage` that will have all the instance variables and functions of a `Message`, plus an additional member variable, `errorCode`, which will be displayed when the object is outputted.

The `Message` class definition looks like this:

```
class Message
{
  protected:
    pstring source;      //source of message
    pstring message;     //text in message

  public:
    //constructor
    Message(const pstring& src, const pstring& msg) {
      source = src;      //initialize source
      message = msg;     //initialize message
    }

    //convert message to pstring
    virtual pstring getMessage() const {
      return source + ": " + message;
    }
};
```

And that's all there is in the whole class definition. A `Message` has two protected member variables: the source of the message and the text of the message. The constructor initializes these member variables from the two `pstring`s passed as arguments.

You probably noticed that there is a `const` floating in free-space after the `getMessage` function declaration. When variables are declared `const` (such as in the `Message` constructor), it indicates that the function can't modify their values. However, a `const` after a function declaration means that the function itself is `const`! Only member functions can use this feature, because what it means is that the function can't modify any member variables of its class. Think of it as if `*this` is marked `const`.

The `virtual` indicator at the beginning of `getMessage` is a very important feature of inheritance. When a function is marked `virtual`, it allows that function to be redefined in subclasses. We will use this feature to change the behavior of `getMessage` in the `ErrorMessage` class.

Now here is an example of an `ErrorMessage` class which extends the functionality of a basic `Message`:

```
class ErrorMessage : public Message
```

```
{
  protected:
    pstring errorCode;        //error messages have error codes

  public:
    //constructor
    ErrorMessage(const pstring& ec, const pstring& src, const pstring& msg) {
      errorCode = ec;         //initialize error code
      source = src;           //initialize source
      message = msg;          //initialize message
    }

    //convert message to pstring
    virtual pstring getMessage() const {
      return "ERROR " + errorCode + ": " + source + ": " + message;
    }
};
```

The class declaration indicates that `ErrorMessage` inherits from `Message`. A colon followed by the keyword `public` is used to identify the parent class.

The `ErrorMessage` class has one additional member variable for an error code, which is added to the string returned from the `getMessage` function. It would serve to notify a user of the error code associated with whatever message they received. The constructor of `ErrorMessage` initializes both the original member variables, `source` and `message`, and the new `errorCode` variable.

An important thing to note is that the `getMessage` function has been redefined in `ErrorMessage`. Now the returned string includes the error code of the message. Suppose we want to overload the `<<` operator to call `getMessage` in order to display messages.

```
ostream& operator << (ostream& os, const Message& msg) {
  return os << msg.getMessage();
}
```

This function will take any `Message` object and display it by calling its `getMessage` function. Since the `ErrorMessage` class is inherited from the `Message` class, what that means is that every `ErrorMessage` object is also a `Message` object! This allows you to use `displayMessage` like this:

```
ErrorMessage error ("1234", "Hard drive", "Out of space");
cout << error << endl;
```

The code first creates an `ErrorMessage` object with three strings for the source, message, and error code. Then the error message is passed to `<<`. Inside `<<`, the `Message` object's `getMessage` function is called in order to get a string representation of the object for output. The resulting output is:

```
ERROR 1234: Hard drive: Out of space
```

Even though the function thinks the object is a `Message`, and has probably never even heard of the `ErrorMessage` class, it is still calling a function defined in `ErrorMessage`. This is all because of the `virtual` keyword used in the `getMessage` declaration. All functions that are ever going to be redefined in subclasses *must* be declared virtual. Otherwise, `<<` would not realize the object is an `ErrorMessage` and would go ahead and call the `getMessage` defined in `Message` instead.

As an excercise, remove the `virtuals` and recompile the program. See if you can predict the output before running it.

15.14 Object-oriented design

Inheritance is a powerful feature. Some programs that would be complicated without inheritance can be written concisely and simply with it. Also, inheritance can facilitate code reuse, since you can customize the behavior of build-in classes without having to modify them.

On the other hand, inheritance can make programs difficult to read, since it is sometimes not clear, when a function is invoked, where to find the definition. For example, in a GUI environment you could call the `Redraw` function on a `Scrollbar` object, yet that particular function was defined in `WindowObject`, the parent of the parent of the parent of the parent of `Scrollbar`.

Also, many of the things that can be done using inheritance can be done almost as elegantly (or more so) without it.

15.15 Glossary

dot notation: The method C++ uses to refer to member variables and functions. The format is `className.memberName`.

member function: A function that is declared within the class defintion of an object. It is invoked directly on an object using dot notation.

nonmember function: A function defined outside any class defintion. Nonmember functions are not invoked on objects and they do not have a current object.

current object: The object on which a member function is invoked. Inside the function, the current object is referred to by the pointer `this`.

`this`: The keyword that refers to a pointer to the current object.

`virtual`: The keyword that is used by any function defined in a parent class that can be overloaded in subclasses.

implicit: Anything that is left unsaid or implied. Within a member function, you can refer to the instance variables implicitly (without naming the object).

explicit: Anything that is spelled out completely. Within a nonmember function, all references to the instance variables have to be explicit.

Chapter 16

Pointers and References

Paul Bui

I suppose the easiest way to explain pointers and references is to jump right into an example. Let's first take a look at some Algebra:

$$x = 1$$

In Algebra, when you use a variable, it is essentially a letter or designation that you use to store some number. In programming, the variable in the equation above must be on the left side. You've probably noticed by now that the compiler won't let you do something like this:

```
1 = x;
```

And if you didn't know this...now you know, and knowing is half the battle. The reason why you receive a compile-time error like "lvalue required in..." is because the left hand side of the equation, traditionally referred to as the **lvalue**, must be an address in memory. Think about it for a second. If you wanted to store some data somewhere, you first need to know where you're going to store it before the action can take place. The **lvalue** is the address of the place in memory where you're going to store the information and/or data of the right hand side of the equation, better known as the **rvalue**.

In C++, you will most likely at one point or another, deal with memory management. To manipulate addresses, C++ has two mechanisms: **pointers** and **references**.

16.1 What are pointers and references?

Pointers and references are essentially variables that hold memory addresses as their values. You learned before about the various different data types such as: `int`, `double`, and `char`. Pointers and references hold the addresses in memory of where you find the data of the various data types that you have declared and assigned. The two mechanisms, pointers and references, have different syntax and different traditional uses.

16.2 Declaring pointers and references

When declaring a pointer to an object or data type, you basically follow the same rules of declaring variables and data types that you have been using, only now, to declare a pointer of `SOMETYPE`, you tack on an asterix `*` between the data type and its variable.

```
SOMETYPE* sometype;

int* x;
```

To declare a reference, you do the exact same thing you did to declare a pointer, only this time, rather than using an asterix `*`, use instead an ampersand `&`.

```
SOMETYPE& sometype;

int& x;
```

As you probably have already learned, spacing in C++ does not matter, so the following pointer declarations are identical:

```
SOMETYPE*  sometype;
SOMETYPE * sometype;
SOMETYPE  *sometype;
```

The following reference declarations are identical as well:

```
SOMETYPE&  sometype;
SOMETYPE & sometype;
SOMETYPE  &sometype;
```

16.3 The "address of" operator

Although declaring pointers and references look similar, assigning them is a whole different story. In C++, there is another operator that you'll get to know intimately, the "address of" operator, which is denoted by the ampersand `&` symbol. The "address of" operator does exactly what it says, it returns the "address of" a variable, a symbolic constant, or a element in an array, in the form of a pointer of the corresponding type. To use the "address of" operator, you tack it on in front of the variable that you wish to have the address of returned.

```
SOMETYPE* x = &sometype; // must be used as rvalue
```

Now, do not confuse the "address of" operator with the declaration of a reference. Because use of operators is restricted to rvalues, or to the right hand side of the equation, the compiler knows that `&SOMETYPE` is the "address of" operator being used to denote the return of the address of `SOMETYPE` as a pointer.

Furthermore, if you have a function which has a pointer as an argument, you may use the "address of" operator on a variable to which you have not already set a pointer to point. By doing this, you do not necessarily have to declare a pointer just so that it is used as an argument in a function, the "address of" operator returns a pointer and thus can be used in that case too.

```
SOMETYPE MyFunc(SOMETYPE *x)
{
  cout << *x << endl;
}

int main()
{
  SOMETYPE i;

  MyFunc(&i);

  return 0;
}
```

16.4 Assigning pointers and references

pointers!assignment of references!assignment of

As you saw in the syntax of using the "address of" operator, a pointer is assigned to the return value of the "address of" operator. Because the return value of an "address of" operator is a pointer, everything works out and your code should compile. To assign a pointer, it must be given an address in memory as the rvalue, else, the compiler will give you an error.

```
int x;
int* px = &x;
```

The above piece of code shows a variable `x` of type `int` being declared, and then a pointer `px` being declared and assigned to the address in memory of `x`. The pointer `px` essentially "points" to `x` by storing its address in memory. Keep in mind that when declaring a pointer, the pointer needs to be of the same type pointer as the variable or constant from which you take the address.

Now here is where you begin to see the differences between pointers and references. To assign a pointer to an address in memory, you had to have used the "address of" operator to return the address in memory of the variable as a pointer. A reference however, does not need to use the "address of" operator to be assigned to an address in memory. To assign an address in memory of a variable to a reference, you just need to use the variable as the rvalue.

```
int x;
int& rx = x;
```

The above piece of code shows a variable `x` of type `int` being declared, and then a reference `rx` being declared and assigned to "refer to" `x`. Notice how the address of `x` is stored in `rx`, or "referred to" by `rx` without the use of any operators, just the variable. You must also follow the same rule as pointers, wherein you must declare the same type reference as the variable or constant to which you refer.

Hypothetically, if you wanted to see what output a pointer would be...

```
#include <iostream.h>

int main()
{
  int someNumber = 12345;
  int* ptrSomeNumber = &someNumber;

  cout << "someNumber = " << someNumber << endl;
  cout << "ptrSomeNumber = " << ptrSomeNumber << endl;

  return 0;
}
```

If you compiled and ran the above code, you would have the variable `someNumber` output `12345` while `ptrSomeNumber` would output some hexadecimal number (addresses in memory are represented in hex). Now, if you wanted to `cout` the value pointed to by `ptrSomeNumber`, you would use this code:

```
#include <iostream.h>

int main()
{
  int someNumber = 12345;
  int* ptrSomeNumber = &someNumber;

  cout << "someNumber = " << someNumber << endl;
  cout << "ptrSomeNumber points to " << *ptrSomeNumber << endl;

  return 0;
}
```

So basically, when you want to use, modify, or manipulate the value pointed to by pointer `x`, you denote the value/variable with `*x`.

Here is a quick list of things you can do with pointers and references:

- You can assign pointers to "point to" addresses in memory
- You can assign references to "refer to" variables or constants
- You can copy the values of pointers to other pointers
- You can modify the values stored in the memory pointed to or referred to by pointers and/or references, respectively
- You can also increment or decrement the addresses stored in pointers
- You can pass pointers and/or references to functions (Further information on "Passing by reference" can be found HERE)

16.5 The Null pointer

Remember how you can assign a character or string to be **null**? If you don't remember, check out HERE. The **null** character in a string denotes the end of a string, however, if a pointer were to be assigned to the `null` pointer, it points to nothing. The null pointer is often denoted by `0` or `null`. The `null` pointer is often used in conditions and/or in logical operations.

```
#include <iostream.h>

int main()
{
  int x = 12345;
  int* px = &x;

  while (px) {
    cout << "Pointer px points to something\n";
    px = 0;
  }

  cout << "Pointer px points to null, nothing, nada!\n";

  return 0;
}
```

If pointer `px` is NOT null, then it is pointing to something, however, if the pointer is null, then it is pointing to nothing. The null pointer becomes very useful when you must test the state of a pointer, whether it has a value or not.

16.6 Dynamic Memory Allocation

You have probably wondered how programmers allocate memory efficiently without knowing, prior to running the program, how much memory will be necessary. Here is when the fun starts with dynamic memory allocation.

Several sections ago, we learned about assigning pointers using the "address of" operator because it returned the address in memory of the variable or constant in the form of a pointer. Now, the "address of" operator is NOT the only operator that you can use to assign a pointer. In C++ you have yet another operator that returns a pointer, which is the **new** operator. The `new` operator allows the programmer to allocate memory for a specific data type, struct, class, etc, and gives the programmer the address of that allocated sect of memory in the form of a pointer. The `new` operator is used as an rvalue, similar to the "address of" operator. Take a look at the code below to see how the `new` operator works.

```
int n = 10;
SOMETYPE *parray, *pS;
int *pint;

parray = new SOMETYPE[n];
pS = new SOMETYPE;
pint = new int;
```

By assigning the pointers to an allocated sect of memory, rather than having to use a variable declaration, you basically override the "middleman" (the variable declaration. Now, you can allocate memory dynamically without having to know the number of variables you should declare. If you looked at the above piece of code, you can use the `new` operator to allocate memory for arrays too, which comes quite in handy when we need to manipulate the sizes of large arrays and or classes efficiently. The memory that your pointer points to because of the `new` operator can also be "deallocated," not destroyed but rather, freed up from your pointer. The `delete` operator is used in front of a pointer and frees up the address in memory to which the pointer is pointing.

```
delete parray;
delete pint;
```

The memory pointed to by `parray` and `pint` have been freed up, which is a very good thing because when you're manipulating multiple large arrays, you try to avoid losing the memory someplace by leaking it. Any allocation of memory needs to be properly deallocated or a leak will occur and your program won't run efficiently. Essentially, every time you use the `new` operator on something, you should use the `delete` operator to free that memory before exiting. The `delete` operator, however, not only can be used to delete a pointer allocated with the `new` operator, but can also be used to "delete" a null pointer, which prevents attempts to delete non-allocated memory (this actions compiles and does nothing).

The `new` and `delete` operators do not have to be used in conjunction with each other within the same function or block of code. It is proper and often advised to write functions that allocate memory and other functions that deallocate memory.

16.7 Returning pointers and/or references from functions

When declaring a function, you must declare it in terms of the type that it will return, for example:

```
int MyFunc(); // returns an int
SOMETYPE MyFunc(); // returns a SOMETYPE

int* MyFunc(); // returns a pointer to an int
SOMETYPE *MyFunc(); // returns a pointer to a SOMETYPE
SOMETYPE &MyFunc(); // returns a reference to a SOMETYPE
```

Woah, my a-paul-igies, I didn't mean to jump right into it, but I'm pretty sure that if you're understanding pointers, the declaration of a function that returns a pointer or a reference should seem relatively logical. The above piece of code shows how to basically declare a function that will return a reference or a pointer.

```
SOMETYPE *MyFunc(int *p)
{
   ...
   ...
   return p;
}

SOMETYPE &MyFunc(int &r)
{
  ...
  ...
  return r;
}
```

Within the body of the function, the `return` statement should NOT return a pointer or a reference that has the address in memory of a local variable that was declared within the function, else, as soon as the function exits, all local variables ar destroyed and your pointer or reference will be pointing to some place in memory that you really do not care about. Having a dangling pointer like that is quite inefficient and dangerous outside of your function.

However, within the body of your function, if your pointer or reference has the address in memory of a data type, struct, or class that you dynamically allocated the memory for, using the `new` operator, then returning said pointer or reference would be reasonable.

```
SOMETYPE *MyFunc() //returning a pointer that has a dynamically
{ //allocated memory address is proper code
   int *p = new int[5];
   ...
   ...
   return p;
}
```

16.8 Glossary

pointer: a variable that holds an address in memory. Similar to a reference, however, pointers have different syntax and traditional uses from references.

reference: a variable that holds an address in memory. Similar to a pointer, however, references have different syntax and traditional uses from pointers.

"address of" operator: an operator that returns the address in memory of a variable.

dynamic memory allocation: the explicit allocation of contiguous blocks of memory at any time in a program.

new: an operator that returns a pointer of the appropriate data type, which points to the reserved place.

delete: an operator that returns the memory pointed to by a pointer to the free store (a special pool of free memory that each program has)

Chapter 17

Templates

Paul Bui

Now that you have a decent amount of experiencing coding, allow me to ask you a question. Have you noticed how many functions that perform the same tasks look similar? For example, if you wrote a function that prints an `int`, you would have to have the `int` declared first. This way, the possibility of error in your code is reduced, however, it gets somewhat annoying to have to create different versions of functions just to handle all the different data types you use. Oh wait...we've got templates.

Parameterized types, better known as templates, allow the programmer [you] to create one function that can handle many different types. Instead of having to take into account every data type, you have one arbitrary parameter name that the compiler then replaces with the different data types that you wish the function to use, manipulate, etc.

17.1 Syntax for Templates

Templates are pretty easy to use, just look at the syntax:

```
template <class TYPEPARAMTER>
```

`TYPEPARAMETER is just the arbitrary typeparameter name that you want to use in your function.  Let's say you want to create a swap function that can handle more than one data type...something that looks like this:`

```
template <class SOMETYPE>
void swap (SOMETYPE &x, SOMETYPE &b)
{
  SOMETYPE temp = a;
  a = b;
  b = temp;
}
```

The function you see above looks really similar to any other swap function, with the differences being the template <class SOMETYPE> line before the function definition and the instances of SOMETYPE in the code. Everywhere you would normally need to have the name or class of the datatype that you're using, you now replace with the arbitrary name that you used in the template <class SOMETYPE>. For example, if you had "SUPERDUPERTYPE" instead of "SOMETYPE," the code would look something like this:

```
template <class SUPERDUPERTYPE>
void swap (SUPERDUPERTYPE &x, SUPERDUPERTYPE &y)
{
  SUPERDUPERTYPE temp = x;
  x = y;
  y = temp;
}
```

As you can see, you can use whatever label you wish for the template typeparameter, as long as it is not a reserved word.

If you want to have more than one template typeparameter, then the syntax would be:

```
template <class SOMETYPE1, class SOMETYPE2, ...>
```

17.2 Templates and Classes

templates!in classes

Let's say that rather than creating a puny little templated function, you would rather use templates in a class, so that the class may handle more than one datatype. If you've noticed, pmatrix and pvector are both able to handle creating matrices and vectors of int, double, and etc. This is because there is a line, template <class SOMETYPE> in the line preceding the declaration of the class. Just take a look:

```
template <class SOMETYPE>
```

```
class pmatrix {
  ...
  ...
  ...
};
```

If you want to declare a function that will return your typeparameter then replace the return type with your typeparameter name.

```
template <class SOMETYPE>
SOMETYPE printFunction();
```

17.3 The Pitfalls of Templates

Ok, so now you probably think that templates are the coolest things in the world. There is however, the all too familiar problem of getting the actual code to work. The templated aspects of your function will only work if the type that you are using already has the constructors, operators, and etc. defined. For example, if you were to use the += operator with your typeparamter:

```
SOMETYPE += x;
```

However, if the datatype, class, or struct that you use SOMETYPE to represent does not have a += operator defined, then the compiler, specifically, the linker will give you an error and your start losing hair. Don't worry, this is all a part of the sometimes seemingly difficult process of getting templates and the like to work. Have fun.

17.4 Glossary

templates: Also known as parameterized types, templates allow the programmer to save time and space in source code by simplifying code through overloading functions with an arbitrary typeparameter.

typeparameter: The typeparameter is the arbitrary label or name that you use in your template to represent the various datatypes, structs, or classes.

Chapter 18

Linked lists

18.1 References in objects

One of the more interesting qualities of an object is that an object can contain a reference to another object of the same type. There is a common data structure, the **list**, that takes advantage of this feature.

Lists are made up of **nodes**, where each node contains a pointer or reference to the next node in the list. In addition, each node usually contains a unit of data called the **cargo**. In our first example, the cargo will be a single integer, but later we will write a **generic** list that can contain objects of any type.

18.2 Revenge of the Node

As usual when we write a new class, we'll start with the instance variables, one or two constructors and toString so that we can test the basic mechanism of creating and displaying the new type.

```
struct Node {

    public:

      int cargo;
      Node* next;

      Node () {
          cargo = 0;
          next = null;
      }
      Node (int Cargo, Node Next) {
            cargo = Cargo;
            next = Next;
      }

      pstring toString () {
        pstring s;
        for(; n; n/=10)
        s = char(n%10 + '0') + s;
        return s;
      }
}
```

The declarations of the instance variables follow naturally from the specification, and the rest follows mechanically from the instance variables. The expression cargo + "" is an awkward but concise way to convert an integer to a String.

To test the implementation so far, we would put something like this in main:

```
    Node node = new Node (1, null);
    cout << node.cargo;
```

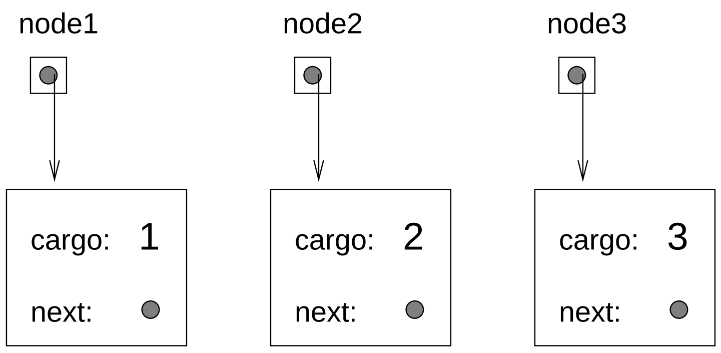
The result is simply

```
1
```

To make it interesting, we need a list with more than one node!

```
    Node node1 = new Node (1, null);
    Node node2 = new Node (2, null);
    Node node3 = new Node (3, null);
```

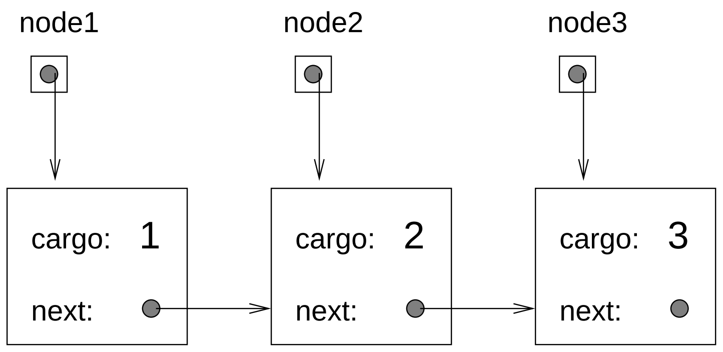
This code creates three nodes, but we don't have a list yet because the nodes are not **linked**. The state diagram looks like this:



To link up the nodes, we have to make the first node refer to the second and the second node refer to the third.

```
node1.next = node2;
node2.next = node3;
node3.next = null;
```

The reference of the third node is null, which indicates that it is the end of the list. Now the state diagram looks like:



Now we know how to create nodes and link them into lists. What might be less clear at this point is why. Now that we're a little more familiar with the use of the struct Node, we can now introduce you to the other style of notation. The use of dot notation can be replaced by the more well-known ->

For example:

```
node1->next = node2;
node2->next = node3;
node3->next = null;
```

18.3 Lists as collections

The thing that makes lists useful is that they are a way of assembling multiple objects into a single entity, sometimes called a collection. In the example, the first node of the list serves as a reference to the entire list.

If we want to pass the list as a parameter, all we have to pass is a reference to the first node. For example, the method printList takes a single node as an argument. Starting with the head of the list, it prints each node until it gets to the end (indicated by the null reference).

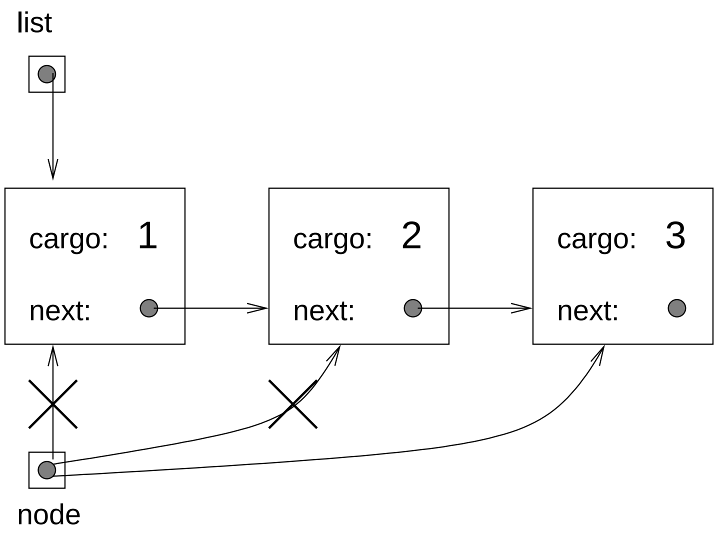
```
void printList (Node *list) {
        Node *node = list;

        while (node != null) {
            cout << node->cargo; << endl;
            node = node->next;
        }
    }
```

To invoke this method we just have to pass a reference to the first node:

```
        printList (node1);
```

Inside printList we have a reference to the first node of the list, but there is no variable that refers to the other nodes. We have to use the next value from each node to get to the next node.

This diagram shows the value of list and the values that node takes on:



This way of moving through a list is called a **traversal**, just like the similar pattern of moving through the elements of an array. It is common to use a loop variable like node to refer to each of the nodes in the list in succession.

The output of this method is

```
123
```

By convention, lists are printed in parentheses with commas between the elements, as in (1, 2, 3). As an exercise, modify printList so that it generates output in this format.

As another exercise, rewrite printList using a for loop instead of a while loop.

18.4 Lists and recursion

Recursion and lists go together like fava beans and a nice Chianti. For example, here is a recursive algorithm for printing a list backwards:

1. Separate the list into two pieces: the first node (called the head) and the rest (called the tail).
2. Print the tail backwards.
3. Print the head.

Of course, Step 2, the recursive call, assumes that we have a way of printing a list backwards. But *if* we assume that the recursive call works---the leap of faith---then we can convince ourselves that this algorithm works.

All we need is a base case, and a way of proving that for any list we will eventually get to the base case. A natural choice for the base case is a list with a single element, but an even better choice is the empty list, represented by null.

```
printBackward (Node *list) {
    if (list == null) return;

    Node *head = list;
    Node *tail = list->next;

    printBackward (tail);
    cout << head->cargo;
}
```

The first line handles the base case by doing nothing. The next two lines split the list into head and tail. The last two lines print the list.

We invoke this method exactly as we invoked printList:

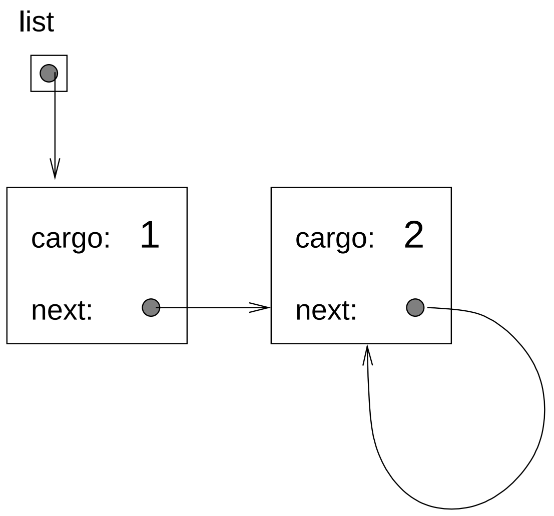
```
printBackward (node1);
```

The result is a backwards list.

Can we prove that this method will always terminate? In other words, will it always reach the base case? In fact, the answer is no. There are some lists that will make this method crash.

18.5 Infinite lists

There is nothing to prevent a node from referring back to an earlier node in the list, including itself. For example, this figure shows a list with two nodes, one of which refers to itself.



If we invoke printList on this list, it will loop forever. If we invoke printBackward it will recurse infinitely. This sort of behavior makes infinite lists difficult to work with.

Nevertheless, they are occasionally useful. For example, we might represent a number as a list of digits and use an infinite list to represent a repeating fraction.

Regardless, it is problematic that we cannot prove that printList and printBackward terminate. The best we can do is the hypothetical statement, ‘‘If the list contains no loops, then these methods will terminate.’’ This sort of claim is called a **precondition**. It imposes a constraint on one of the parameters and describes the behavior of the method if the constraint is satisfied. We will see more examples soon.

18.6 The fundamental ambiguity theorem

There is a part of printBackward that might have raised an eyebrow:

```
        Node *head = list;
        Node *tail = list->next;
```

After the first assignment, head and list have the same type and the same value. So why did I create a new variable?

The reason is that the two variables play different roles. We think of head as a reference to a single node, and we think of list as a reference to the first node of a list. These ''roles'' are not part of the program; they are in the mind of the programmer.

The second assignment creates a new reference to the second node in the list, but in this case we think of it as a list. So, even though head and tail have the same type, they play different roles.

This ambiguity is useful, but it can make programs with lists difficult to read. I often use variable names like node and list to document how I intend to use a variable, and sometimes I create additional variables to disambiguate.

I could have written printBackward without head and tail, but I think it makes it harder to understand:

```
void printBackward (Node *list) {
    if (list == null) return;

    printBackward (list->next);
    cout << list->cargo;
}
```

Looking at the two function calls, we have to remember that printBackward treats its argument as a list and print treats its argument as a single object.

Always keep in mind the **fundamental ambiguity theorem**:

> A variable that refers to a node might treat the node as a single object or as the first in a list of nodes.

18.7 Object methods for nodes

You might have wondered why printList and printBackward are class methods. I have made the claim that anything that can be done with class methods can also be done with object methods; it's just a question of which form is cleaner.

In this case there is a legitimate reason to choose class methods. It is legal to send null as an argument to a class method, but it is not legal to invoke an object method on a null object.

```
Node *node = null;
```

```
printList (node);       // legal
node.printList ();      // NullPointerException
```

This limitation makes it awkward to write list-manipulating code in a clean, object-oriented style. A little later we will see a way to get around this, though.

18.8 Modifying lists

Obviously one way to modify a list is to change the cargo of one on the nodes, but the more interesting operations are the ones that add, remove, or reorder the nodes.

As an example, we'll write a method that removes the second node in the list and returns a reference to the removed node.

```
    Node* removeSecond (Node *list) {
        Node *first = list;
        Node *second = list->next;

        // make the first node refer to the third
        first->next = second->next;

        // separate the second node from the rest of the list
        second->next = null;
        return second;
    }
```

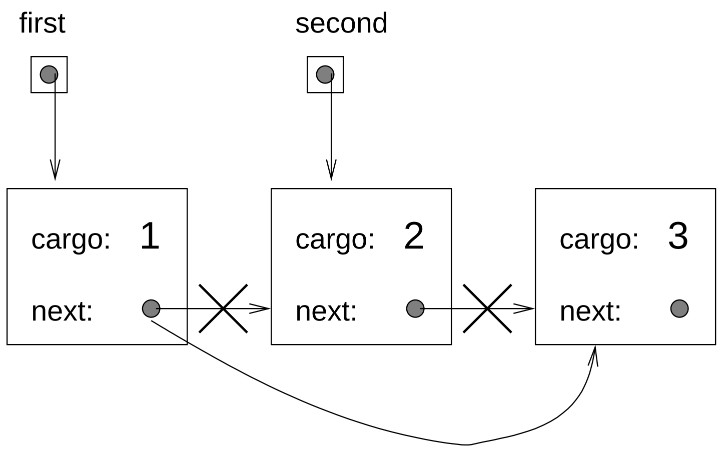
Again, I am using temporary variables to make the code more readable. Here is how to use this method.

```
        printList (node1);
        Node *removed = removeSecond (node1);
        printList (removed);
        printList (node1);
```

The output is

```
(1, 2, 3)           the original list
(2)                 the removed node
(1, 3)              the modified list
```

Here is a state diagram showing the effect of this operation.



What happens if we invoke this method and pass a list with only one element (a **singleton**)? What happens if we pass the empty list as an argument? Is there a precondition for this method?

18.9 Wrappers and helpers

For some list operations it is useful to divide the labor into two methods. For example, to print a list backwards in the conventional list format, (3, 2, 1) we can use the printBackwards method to print 3, 2, but we need a separate method to print the parentheses and the first node. We'll call it printBackwardNicely.

```
    void printBackwardNicely (Node *list) {
        cout << '(';

        if (list != null) {
            Node *head = list;
            Node *tail = list->next;
            printBackward (tail);
            cout << head->cargo;
        }
        cout << ')';
    }
```

Again, it is a good idea to check methods like this to see if they work with special cases like an empty list or a singleton.

Elsewhere in the program, when we use this method, we will invoke printBackwardNicely directly and it will invoke printBackward on our behalf. In that sense, printBackwardNicely acts as a **wrapper**, and it uses printBackward as a helper.

18.10 The LinkedList class

There are a number of subtle problems with the way we have been implementing lists. In a reversal of cause and effect, I will propose an alternative implementation first and then explain what problems it solves.

First, we will create a new class called LinkedList. Its instance variables are an integer that contains the length of the list and a reference to the first node in the list. LinkedList objects serve as handles for manipulating lists of Node objects.

```
class LinkedList {

  public:
    int length;
    Node* head;

    LinkedList () {
        length = 0;
        head = null;
    }
};
```

One nice thing about the LinkedList class is that it gives us a natural place to put wrapper functions like printBackwardNicely, which we can make an object method in the LinkedList class.

```
    public void printBackward () {
        cout << '(';

        if (head != null) {
            Node *tail = head->next;
            Node.printBackward (tail);
            cout << head->cargo;
        }
        cout << ')';
    }
```

Just to make things confusing, I renamed printBackwardNicely. Now there are two methods named printBackward: one in the Node class (the helper) and one in the LinkedList class (the wrapper). In order for the wrapper to invoke the helper, it has to identify the class explicitly (Node.printBackward).

So, one of the benefits of the LinkedList class is that it provides a nice place to put wrapper functions. Another is that it makes it easier to add or remove the first element of a list. For example, addFirst is an object method for LinkedLists; it takes an int as an argument and puts it at the beginning of the list.

```
    void addFirst (int i) {
      Node *node = new Node (i, head);
      head = node;
      length++;
    }
```

As always, to check code like this it is a good idea to think about the special cases. For example, what happens if the list is initially empty?

18.11 Invariants

Some lists are ''well-formed;'' others are not. For example, if a list contains a loop, it will cause many of our methods to crash, so we might want to require that lists contain no loops. Another requirement is that the length value in the LinkedList object should be equal to the actual number of nodes in the list.

Requirements like this are called **invariants** because, ideally, they should be true of every object all the time. Specifying invariants for objects is a useful programming practice because it makes it easier to prove the correctness of code, check the integrity of data structures, and detect errors.

One thing that is sometimes confusing about invariants is that there are some times when they are violated. For example, in the middle of addFirst, after we have added the node, but before we have incremented length, the invariant is violated. This kind of violation is acceptable; in fact, it is often impossible to modify an object without violating an invariant for at least a little while. Normally the requirement is that every method that violates an invariant must restore the invariant.

If there is any significant stretch of code in which the invariant is violated, it is important for the comments to make that clear, so that no operations are performed that depend on the invariant.

18.12 Glossary

list: A data structure that implements a collection using a sequence of linked nodes.

node: An element of a list, usually implemented as an object that contains a reference to another object of the same type.

cargo: An item of data contained in a node.

link: An object reference embedded in an object.

generic data structure: A kind of data structure that can contain data of any type.

precondition: An assertion that must be true in order for a method to work correctly.

invariant: An assertion that should be true of an object at all times (except maybe while the object is being modified).

wrapper method: A method that acts as a middle-man between a caller and a helper method, often offering an interface that is cleaner than the helper method's.

Chapter 19

Stacks

19.1 Abstract data types

The data types we have looked at so far are all concrete, in the sense that we have completely specified how they are implemented. For example, the Card class represents a card using two integers. As I discussed at the time, that is not the only way to represent a card; there are many alternative implementations.

An **abstract data type**, or ADT, specifies a set of operations (or methods) and the semantics of the operations (what they do) but it does not not specify the implementation of the operations. That's what makes it abstract.

Why is that useful?

- It simplifies the task of specifying an algorithm if you can denote the operations you need without having to think at the same time about how the operations are performed.
- Since there are usually many ways to implement an ADT, it might be useful to write an algorithm that can be used with any of the possible implementations.
- Well-known ADTs, like the Stack ADT in this chapter, are often implemented in standard libraries so they can be written once and used by many programmers.
- The operations on ADTs provide a common high-level language for specifying and talking about algorithms.

When we talk about ADTs, we often distinguish the code that uses the ADT, called the **client** code, from the code that implements the ADT, called **provider** code because it provides a standard set of services.

19.2 The Stack ADT

In this chapter we will look at one common ADT, the stack. A stack is a collection, meaning that it is a data structure that contains multiple elements. Other collections we have seen include arrays and lists.

As I said, an ADT is defined by the operations you can perform on it. Stacks can perform only the following operations:

constructor: Create a new, empty stack.

push: Add a new item to the stack.

pop: Remove and return an item from the stack. The item that is returned is always the last one that was added.

empty: Check whether the stack is empty.

A stack is sometimes called a ‘‘last in, first out,’’ or LIFO data structure, because the last item added is the first to be removed.

19.3 The pstack Class

Although we have the pclasses that provide for a class called pstack that implements the Stack ADT. You should make some effort to keep these two things---the ADT and the pclass implementation---straight.

Then the syntax for constructing a new pstack is

```
pstack stack;
```

Initially the stack is empty, as we can confirm with the empty method, which returns a boolean:

```
cout << stack.empty ();
```

A stack is a generic data structure, which means that we can add any type of item to it. In the pclass implementation, though, we can only add object types. For our first example, we’ll use Node objects, as defined in the previous chapter. Let’s start by creating and printing a short list.

```
LinkedList list;
list.addFirst (3);
list.addFirst (2);
list.addFirst (1);
list.print ();
```

The output is (1, 2, 3). To put a Node object onto the stack, use the push method:

```
pstack.push (node);
```

The following loop traverses the list and pushes all the nodes onto the stack:

```
    for (Node *node = list.head; node != null; node = node->next) {
        pstack.push (node);
    }
```

We can remove an element from the stack with the overloaded pop method.

```
    pstack.pop ();
    //  or
    pstack.pop (itemType &item);
```

The return type from pop is void. That's because the stack implementation doesn't need to keep the item around because it is to be removed from the stack.

The following loop is a common idiom for popping all the elements from a stack, stopping when it is empty:

```
    while (!pstack.empty ()) {
        cout << pstack.top() << ' ';
        pstack.pop();
    }
```

The output is 3 2 1. In other words, we just used a stack to print the elements of a list backwards! Granted, it's not the standard format for printing a list, but using a stack it was remarkably easy to do.

You should compare this code to the implementations of printBackward in the previous chapter. There is a natural parallel between the recursive version of printBackward and the stack algorithm here. The difference is that printBackward uses the run-time stack to keep track of the nodes while it traverses the list, and then prints them on the way back from the recursion. The stack algorithm does the same thing, just using a pstack object instead of the run-time stack.

19.4 Postfix expressions

In most programming languages, mathematical expressions are written with the operator between the two operands, as in 1+2. This format is called **infix**. An alternate format used by some calculators is called **postfix**. In postfix, the operator follows the operands, as in 1 2+.

The reason postfix is sometimes useful is that there is a natural way to evaluate a postfix expression using a stack.

- Starting at the beginning of the expression, get one term (operator or operand) at a time.
  - If the term is an operand, push it on the stack.
  - If the term is an operator, pop two operands off the stack, perform the operation on them, and push the result back on the stack.
- When we get to the end of the expression, there should be exactly one operand left on the stack. That operand is the result.

As an exercise, apply this algorithm to the expression 1 2 + 3 *.

This example demonstrates one of the advantages of postfix: there is no need to use parentheses to control the order of operations. To get the same result in infix, we would have to write (1 + 2) * 3. As an exercise, write a postfix expression that is equivalent to 1 + 2 * 3?

19.5 Parsing

In order to implement the algorithm from the previous section, we need to be able to traverse a string and break it into operands and operators. This process is an example of **parsing**

If we were to break the string up into smaller parts, we would need a specific character to use as a boundary between the chucks. A character that marks a boundary is called a **delimiter**.

So let's quickly build a parsing function that will store the various chunks of a pstring into a pvector<pstring>.

```
pvector<pstring> parse(pstring string, char delim) {

  pvector<pstring> stringParsed;
  if (string.length() == 0)
    return stringParsed.resize(0);

  for (int i = 0, j = 0; i < string.length(); i++)
  {
    if (string[i] != delim || string[i] != '\n')
      stringParsed[j] += string[i];
    else
    {
      cout << stringParsed[j] << endl;
      j++;
      stringParsed.resize(j+1);
    }
  }

  return stringParsed;
}
```

The function above accepts a pstring to be parsed and a char to be used as a delimiter, so that whenever the delim character appears in the string, the chunk is saved as a new pstring element in the pvector<pstring>.

Passing a string through the function with a space delimiter would look like this:

```
  pstring string = "Here are four tokens.";

  pvector<pstring> = parse(string, ' ');
```

The output of the parser is:

```
Here
are
four
tokens.
```

For parsing expressions, we have the option of specifying additional characters that will be used as delimiters:

```
bool checkDelim(char ch, pstring delim) {

  for (int i = 0; i < delim.length(); i++)
  {
    if (ch == delim[i])
      return true;
  }
  return false;
}

pvector<pstring> parse(pstring string, pstring delim) {

  pvector<pstring> stringParsed;

  if (string.length() == 0)
    return stringParsed.resize(0);

  for (int i = 0, j = 0; i < string.length(); i++)
  {
    if (!checkDelim(string[i], delim) || string[i] != '\n')
      stringParsed[j] += string[i];
    else
    {
      cout << stringParsed[j] << endl;
      j++;
      stringParsed.resize(j+1);
    }
  }

  return stringParsed;
}
```

An example of using the above functions can be seen below:Using the above functions would

```
    pstring string = "11 22+33*";
    pstring delim = " +-*/";
    pvector<pstring> stringParsed = parse(string, delim);
```

The new function checkDelim checks for whether or not a given char is a delimiter. Now the output is:

```
11
22
33
```

19.6 Implementing ADTs

One of the fundamental goals of an ADT is to separate the interests of the provider, who writes the code that implements the ADT, and the client, who uses the ADT. The provider only has to worry about whether the implementation is correct---in accord with the specification of the ADT---and not how it will be used.

Conversely, the client *assumes* that the implementation of the ADT is correct and doesn't worry about the details. When you are using one of Java's built-in classes, you have the luxury of thinking exclusively as a client.

When you implement an ADT, on the other hand, you also have to write client code to test it. In that case, you sometimes have to think carefully about which role you are playing at a given instant.

In the next few sections we will switch gears and look at one way of implementing the Stack ADT, using an array. Start thinking like a provider.

19.7 Array implementation of the Stack ADT

The instance variables for this implementation is a templated array, which is why we will use the pvector class. It will contain the items on the stack, and an integer index which will keep track of the next available space in the array. Initially, the array is empty and the index is 0.

To add an element to the stack (push), we'll copy a reference to it onto the stack and increment the index. To remove an element (pop) we have to decrement the index first and then copy the element out.

Here is the class definition:

```
class Stack {

    pvector<ITEMTYPE> array;
    int index;

    public:

      Stack () {
        array.resize(128);
        index = 0;
      }
};
```

The above code contains the type ITEMTYPE which is essentially just a quick way of saying, "INSERT DATATYPE HERE" because pvectors are templated and can handle various types. ITEMTYPE is not in actuality a type, just so you know. In the future, you can replace ITEMTYPE with any other data type, class, or struct.

As usual, once we have chosen the instance variables, it is a mechanical process to write a constructor. For now, the default size is 128 items. Later we will consider better ways of handling this.

Checking for an empty stack is trivial.

```
    bool empty () {
        return array.length() == 0;
    }
```

It it important to remember, though, that the number of elements in the stack is not the same as the size of the array. Initially the size is 128, but the number of elements is 0.

The implementations of push and pop follow naturally from the specification.

```
    void push (ITEMTYPE item) {
        array[index] = item;
        index++;
    }

    ITEMTYPE pop () {
        index--;
        return array[index];
    }
```

To test these methods, we can take advantage of the client code we used to exercise pstack.

If everything goes according to plan, the program should work without any additional changes. Again, one of the strengths of using an ADT is that you can change implementations without changing client code.

19.8 Resizing arrays

A weakness of this implementation is that it chooses an arbitrary size for the array when the Stack is created. If the user pushes more than 128 items onto the stack, it will cause an ArrayIndexOutOfBounds exception.

An alternative is to let the client code specify the size of the array. This alleviates the problem, but it requires the client to know ahead of time how many items are needed, and that is not always possible.

A better solution is to check whether the array is full and make it bigger when necessary. Since we have no idea how big the array needs to be, it is a reasonable strategy to start with a small size and increase the size by 1 each time it overflows.

Here's the improved version of push:

```
    void push (ITEMTYPE item) {
        if (full()) resize ();

        // at this point we can prove that index < array.length

        array[index] = item;
        index++;
    }
```

Before putting the new item in the array, we check if the array is full. If so, we invoke resize. After the if statement, we know that either (1) there was room in the array, or (2) the array has been resized and there is room. If full and resize are correct, then we can prove that index < array.length, and therefore the next statement cannot cause an exception.

Now all we have to do is implement full and resize.

```
    private:
      bool full () {
        return index == (array.length()-1);
      }

      void resize () {
        array.resize(array.length()+1);
      }
```

Both methods are declared private, which means that they cannot be invoked from another class, only from within this one. This is acceptable, since there is no reason for client code to use these functions, and desirable, since it enforces the boundary between the implementation and the client.

The implementation of full is trivial; it just checks whether the index has gone beyond the range of valid indices.

The implementation of resize is straightforward, with the caveat that it assumes that the old array is full. In other words, that assumption is a precondition of this method. It is easy to see that this precondition is satisfied, since the only way resize is invoked is if full returns true, which can only happen if index == array.length.

At the end of resize, we replace the old array with the new one (causing the old to be garbage collected). The new array.length is twice as big as the old, and index hasn't changed, so now it must be true that index < array.length. This assertion is a **postcondition** of resize: something that must be true when the method is complete (as long as its preconditions were satisfied).

Preconditions, postconditions, and invariants are useful tools for analyzing programs and demonstrating their correctness. In this example I have demonstrated a programming style that facilitates program analysis and a style of documentation that helps demonstrate correctness.

19.9 Glossary

abstract data type (ADT): A data type (usually a collection of objects) that is defined by a set of operations, but that can be implemented in a variety of ways.

client: A program that uses an ADT (or the person who wrote the program).

provider: The code that implements an ADT (or the person who wrote it).

private: A Java keyword that indicates that a method or instance variable cannot be accessed from outside the current class definition.

infix: A way of writing mathematical expressions with the operators between the operands.

postfix: A way of writing mathematical expressions with the operators after the operands.

parse: To read a string of characters or tokens and analyze their grammatical structure.

delimiter: A character that is used to separate tokens, like the punctuation in a natural language.

predicate: A mathematical statement that is either true or false.

postcondition: A predicate that must be true at the end of a method (provided that the preconditions were true at the beginning).

Chapter 20

Queues and Priority Queues

This chapter presents two ADTs: Queues and Priority Queues. In real life a **queue** is a line of customers waiting for service of some kind. In most cases, the first customer in line is the next customer to be served. There are exceptions, though. For example, at airports customers whose flight is leaving imminently are sometimes taken from the middle of the queue. Also, at supermarkets a polite customer might let someone with only a few items go first.

The rule that determines who goes next is called a **queueing discipline**. The simplest queueing discipline is called **FIFO**, for ''first-in-first-out.'' The most general queueing discipline is **priority queueing**, in which each customer is assigned a priority, and the customer with the highest priority goes first, regardless of the order of arrival. The reason I say this is the most general discipline is that the priority can be based on anything: what time a flight leaves, how many groceries the customer has, or how important the customer is. Of course, not all queueing disciplines are ''fair,'' but fairness is in the eye of the beholder.

The Queue ADT and the Priority Queue ADT have the same set of operations and their interfaces are the same. The difference is in the semantics of the operations: a Queue uses the FIFO policy, and a Priority Queue (as the name suggests) uses the priority queueing policy.

As with most ADTs, there are a number of ways to implement queues Since a queue is a collection of items, we can use any of the basic mechanisms for storing collections: arrays, lists, or vectors. Our choice among them will be based in part on their performance--- how long it takes to perform the operations we want to perform--- and partly on ease of implementation.

20.1 The queue ADT

The queue ADT is defined by the following operations:

constructor: Create a new, empty queue.

insert: Add a new item to the queue.

remove: Remove and return an item from the queue. The item that is returned is the first one that was added.

empty: Check whether the queue is empty.

To demonstrate a queue implementation, I will take advantage of the LinkedList class from Chapter 18. Also, I will assume that we have a class named Customer that defines all the information about each customer, and the operations we can perform on customers.

As far as our implementation goes, it does not matter what kind of object is in the Queue, so we can make it generic. Here is what the implementation looks like.

```
class Queue {
    public:

      LinkedList list;

      Queue () {
        list = new List ();
      }

      bool empty () {
        return list.empty ();
      }

      void insert (Node* node) {
        list.addLast (node);
      }

      Node* remove () {
        return list.removeFirst ();
      }
};
```

A queue object contains a single instance variable, which is the list that implements it. For each of the other methods, all we have to do is invoke one of the methods from the LinkedList class.

20.2 Veneer

An implementation like this is called a **veneer**. In real life, veneer is a thin coating of good quality wood used in furniture-making to hide lower quality wood underneath. Computer scientists use this metaphor to describe a small piece of code that hides the details of an implementation and provides a simpler, or more standard, interface.

This example demonstrates one of the nice things about a veneer, which is that it is easy to implement, and one of the dangers of using a veneer, which is the **performance hazard**!

Normally when we invoke a method we are not concerned with the details of its implementation. But there is one ‘‘detail’’ we might want to know---the performance characteristics of the method. How long does it take, as a function of the number of items in the list?

First let's look at removeFirst.

```
     Node* removeFirst () {
        Node* result = head;
        if (head != null) {
            head = head->next;
        }
        return result;
    }
```

There are no loops or function calls here, so that suggests that the run time of this method is the same every time. Such a method is called a **constant time** operation. In reality, the method might be slightly faster when the list is empty, since it skips the body of the conditional, but that difference is not significant.

The performance of addLast is very different.

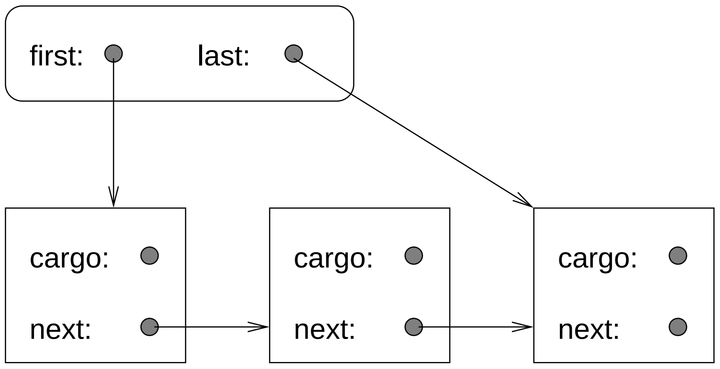
```
    void addLast (Node* node) {
        // special case: empty list
        if (head == null) {
            head = node;
            node->next = null;
            return;
        }
        Node* last;
        for (last = head; last->next != null; last = last->next) {
            // traverse the list to find the last node
        }
        last->next = node;
        node->next = null;
    }
```

The first conditional handles the special case of adding a new node to an empty list. In this case, again, the run time does not depend on the length of the list. In the general case, though, we have to traverse the list to find the last element so we can make it refer to the new node.

This traversal takes time proportional to the length of the list. Since the run time is a linear function of the length, we would say that this method is **linear time**. Compared to constant time, that's very bad.

20.3 Linked Queue

We would like an implementation of the Queue ADT that can perform all operations in constant time. One way to accomplish that is to implement a **linked queue**, which is similar to a linked list in the sense that it is made up of zero or more linked Node objects. The difference is that the queue maintains a reference to both the first and the last node, as shown in the figure.



Here's what a linked Queue implementation looks like:

```
class Queue {
    public:
       Node *first, *last;

    Queue () {
        first = null;
        last = null;
    }

    boolean empty () {
        return first == null;
    }
};
```

So far it is straightforward. In an empty queue, both first and last are null. To check whether a list is empty, we only have to check one of them.

insert is a little more complicated because we have to deal with several special cases.

```
    void insert (Node* node) {
        Node* node = new Node (node->cargo, null);
        if (last != null) {
            last->next = node;
        }
        last = node;
        if (first == null) {
            first = last;
        }
    }
```

The first condition checks to make sure that last refers to a node; if it does then we have to make it refer to the new node.

The second condition deals with the special case where the list was initially empty. In this case both first and last refer to the new node.

remove also deals with several special cases.

```
    Node* remove () {
        Node* result = first;
        if (first != null) {
            first = first.next;
        }
        if (first == null) {
            last = null;
        }
        return result;
    }
```

The first condition checks whether there were any nodes in the queue. If so, we have to copy the next node into first. The second condition deals with the special case that the list is now empty, in which case we have to make last null.

As an exercise, draw diagrams showing both operations in both the normal case and in the special cases, and convince yourself that they are correct.

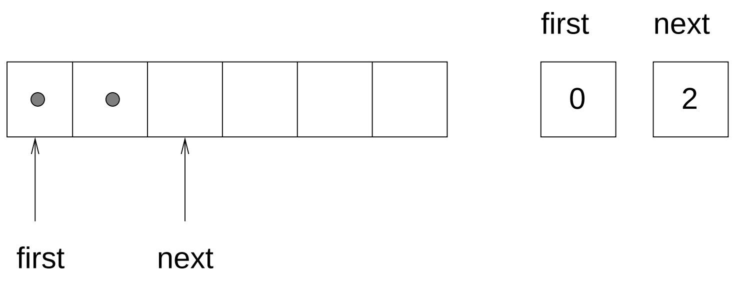
Clearly, this implementation is more complicated than the veneer implementation, and it is more difficult to demonstrate that it is correct. The advantage is that we have achieved the goal: both insert and remove are constant time.

20.4 Circular buffer

Another common implementation of a queue is a **circular buffer**. ''Buffer'' is a general name for a temporary storage location, although it often refers to an array, as it does in this case. What it means to say a buffer is ''circular'' should become clear in a minute.

The implementation of a circular buffer is similar to the array implementation of a stack, as in Section 19.7. The queue items are stored in an array, and we use indices to keep track of where we are in the array. In the stack implementation, there was a single index that pointed to the next available space. In the queue implementation, there are two indices: first points to the space in the array that contains the first customer in line and next points to the next available space.

The following figure shows a queue with two items (represented by dots).



There are two ways to think of the variables first and last. Literally, they are integers, and their values are shown in boxes on the right. Abstractly, though, they are indices of the array, and so they are often drawn as arrows pointing to locations in the array. The arrow representation is convenient, but you should remember that the indices are not references; they are just integers.

Here is an incomplete array implementation of a queue:

```
class Queue {
    public:

      pvector<Node> array;
    int first, next;

    Queue () {
        array.resize (128);
        first = 0;
        next = 0;
    }

    bool empty () {
      return first == next;
    }
```

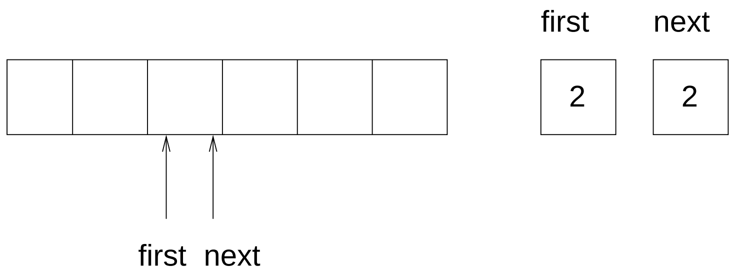
The instance variables and the constructor are straightforward, although again we have the problem that we have to choose an arbitrary size for the array. Later we will solve that problem, as we did with the stack, by resizing the array if it gets full.

The implementation of empty is a little surprising. You might have thought that first == 0 would indicate an empty queue, but that neglects the fact that the head of the queue is not necessarily at the beginning of the array. Instead, we know that the queue is empty if head equals next, in which case there are no items left. Once we see the implementation of insert and remove, that situation will more more sense.

```
  void insert (Node node) {
      array[next] = node;
      next++;
  }

  Node remove () {
      first++;
      return array[first-1];
  }
```

insert looks very much like push in Section 19.7; it puts the new item in the next available space and then increments the index.

remove is similar. It takes the first item from the queue and then increments first so it refers to the new head of the queue. The following figure shows what the queue looks like after both items have been removed.

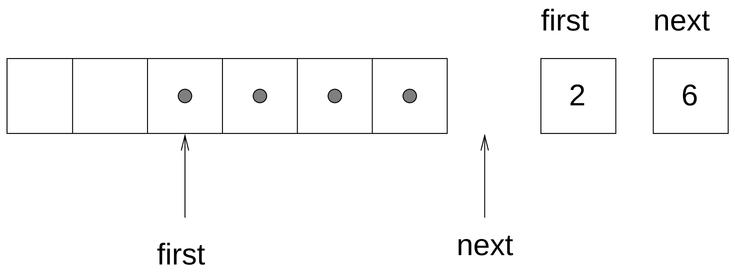


It is always true that next points to an available space. If first catches up with next and points to the same space, then first is referring to an ‘‘empty’’ location, and the queue is empty. I put ‘‘empty’’ in quotation marks because it is possible that the location that first points to actually contains a value (we do nothing to ensure that empty locations contain null); on the other hand, since we know the queue is empty, we will never read this location, so we can think of it, abstractly, as empty.

As an exercise, fix remove so that it returns null if the queue is empty.

The next problem with this implementation is that eventually it will run out of space. When we add an item we increment next and when we remove an item we increment first, but we never decrement either. What happens when we get to the end of the array?

The following figure shows the queue after we add four more items:



The array is now full. There is no ‘‘next available space,’’ so there is nowhere for next to point. One possibility is that we could resize the array, as we did with the stack implementation. But in that case the array would keep getting bigger regardless of how many items were actually in queue. A better solution is to wrap around to the beginning of the array and reuse the spaces there. This ‘‘wrap around’’ is the reason this implementation is called a circular buffer.

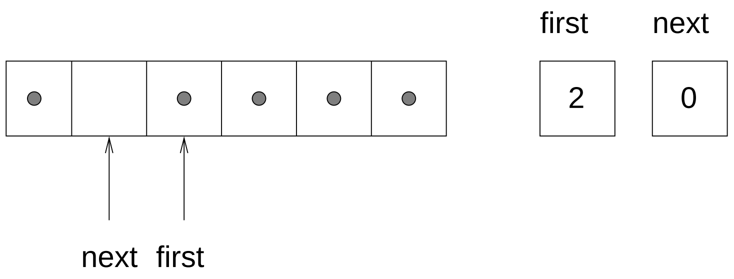
One way to wrap the index around is to add a special case whenever we increment an index:

```
        next++;
        if (next == array.length()) next = 0;
```

A fancy alternative is to use the modulus operator:

```
        next = (next + 1) % array.length();
```

Either way, we have one last problem to solve. How do we know if the queue is *really* full, meaning that we cannot insert another item? The following figure shows what the queue looks like when it is ‘‘full.’’



There is still one empty space in the array, but the queue is full because if we insert another item, then we have to increment next such that next == first, and in that case it would appear that the queue was empty!

To avoid that, we sacrifice one space in the array. So how can we tell if the queue is full?

```
        if ((next + 1) % array.length == first)
```

And what should we do if the array is full? In that case resizing the array is probably the only option.

As an exercise, put together all the code from this section and write an implementation of a queue using a circular buffer that resizes itself when necessary.

20.5 Priority queue

The Priority Queue ADT has the same interface as the Queue ADT, but different semantics. The interface is:

constructor: Create a new, empty queue.

insert: Add a new item to the queue.

remove: Remove and return an item from the queue. The item that is returned is the one with the highest priority.

empty: Check whether the queue is empty.

The semantic difference is that the item that is removed from the queue is not necessarily the first one that was added. Rather, it is whatever item in the queue has the highest priority. What the priorities are, and how they compare to each other, are not specified by the Priority Queue implementation. It depends on what the items are that are in the queue.

For example, if the items in the queue have names, we might choose them in alphabetical order. If they are bowling scores, we might choose from highest to lowest, but if they are golf scores, we would go from lowest to highest.

So we face a new problem. We would like an implementation of Priority Queue that is generic---it should work with any kind of object---but at the same time the code that implements Priority Queue needs to have the ability to compare the objects it contains.

We have seen a way to implement generic data structures using Node, but that does not solve this problem, because there is no way to compare Node unless we know what type the cargo is. So basically, to implement a priority queue, we will have to create compare functions that will compare the cargo of the nodes.

20.6 Array implementation of Priority Queue

In the implementation of the Priority Queue, every time we specify the type of the items in the queue, we specify the type of the cargo. For example, the instance variables are an array of Node and an integer:

```
class PriorityQueue {
    private:
       pvector<Node> array;
       int index;
};
```

As usual, index is the index of the next available location in the array. The instance variables are declared private so that other classes cannot have direct access to them.

The constructor and empty are similar to what we have seen before. I chose the initial size for the array arbitrarily.

```
    public:
      PriorityQueue () {
        array.resize(16);
        index = 0;
      }

      bool empty () {
        return index == 0;
      }
```

insert is similar to push:

```
    void insert (Node item) {
        if (index == array.length()) {
            resize ();
        }
        array[index] = item;
        index++;
    }
```

I omitted the implementation of resize. The only substantial method in the class is remove, which has to traverse the array to find and remove the largest item:

```
    Node remove () {
        if (index == 0) return null;

        int maxIndex = 0;

        // find the index of the item with the highest priority
        for (int i=1; i<index; i++) {
            if (array[i].cargo > array[maxIndex].cargo) {
                maxIndex = i;
            }
        }

        // move the last item into the empty slot
        index--;
        array[maxIndex] = array[index];
        return array[maxIndex];
   }
```

As we traverse the array, maxIndex keeps track of the index of the largest element we have seen so far. What it means to be the ‘‘largest’’ is determined by >.

20.7 The Golfer class

As an example of something with an unusual definition of ‘‘highest’’ priority, we’ll use golfers:

```
class Golfer {

  public:

    pstring name;
    int score;

    Golfer (pstring name, int score) {
        this->name = name;
        this->score = score;
    }
}
```

The class definition and the constructor are pretty much the same as always.

Since priority queues require some comparisons, we'll have to write a function compareTo. So let's write one:

```
    int compareTo (Golfer g1, Golfer g2) {

        int a = g1.score;
        int b = g2.score;

        // for golfers, low is good!
        if (a<b) return 1;
        if (a>b) return -1;
        return 0;
    }
```

Finally, we can create some golfers:

```
        Golfer* tiger = new Golfer ("Tiger Woods", 61);
        Golfer* phil = new Golfer ("Phil Mickelson", 72);
        Golfer* hal = new Golfer ("Hal Sutton", 69);
```

And put them in the queue:

```
        pq.insert (tiger);
        pq.insert (phil);
        pq.insert (hal);
```

When we pull them out:

```
        while (!pq.empty ()) {
            golfer = pq.remove ();
            cout << golfer->name << ' ' << golfer->score;
        }
```

They appear in descending order (for golfers):

```
        Tiger Woods 61
        Hal Sutton 69
        Phil Mickelson 72
```

Ok, so now that we've got a priority queue done for golfers...FORE!!!

20.8 Glossary

queue: An ordered set of objects waiting for a service of some kind.

queueing discipline: The rules that determine which member of a queue is removed next.

FIFO: ''first in, first out,'' a queueing discipline in which the first member to arrive is the first to be removed.

priority queue: A queueing discipline in which each member has a priority determined by external factors. The member with the highest priority is the first to be removed.

Priority Queue: An ADT that defines the operations one might perform on a priority queue.

veneer: A class definition that implements an ADT with method definitions that are invocations of other methods, sometimes with simple transformations. The veneer does no significant work, but it improves or standardizes the interface seen by the client.

performance hazard: A danger associated with a veneer that some of the methods might be implemented inefficiently in a way that is not apparent to the client.

constant time: An operation whose run time does not depend on the size of the data structure.

linear time: An operation whose run time is a linear function of the size of the data structure.

linked queue: An implementation of a queue using a linked list and references to the first and last nodes.

circular buffer: An implementation of a queue using an array and indices of the first element and the next available space.

abstract class: A set of classes. The abstract class specification lists the requirements a class must satisfy to be included in the set.

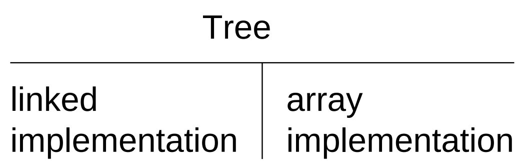
Chapter 21

Trees

This chapter presents a new data structure called a tree, some of its uses and two ways to implement it.

A possible source of confusion is the distinction between an ADT, a data structure, and an implementation of an ADT or data structure. There is no universal answer, because something that is an ADT at one level might in turn be the implementation of another ADT.

To help keep some of this straight, it is sometimes useful to draw a diagram showing the relationship between an ADT and its possible implementations. This figure shows that there are two implementations of a tree:



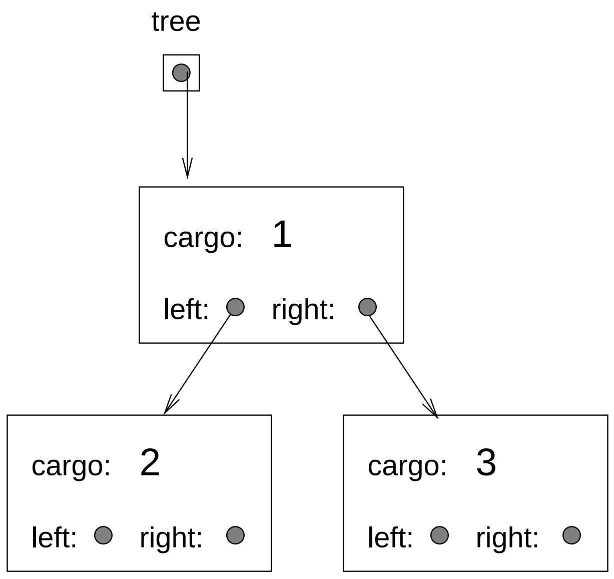
The horizontal line in the figure represents the barrier of abstraction between the ADT and its implementations.

21.1 A tree node

Like lists, trees are made up of nodes. A common kind of tree is a **binary tree**, in which each node contains a reference to two other nodes (possibly null). The class definition looks like this:

```
class Tree {
    int cargo;
    Tree *left, *right;
};
```

Like list nodes, tree nodes contain cargo: in this case a generic int. However, trees may consist of any type of cargo, so in the future you could technically substitute the int with other types and it should work, that is if the rest of your program code does not go awry. The other instance variables are named left and right, in accordance with a standard way to represent trees graphically:



The top of the tree (the node referred to by tree) is called the **root**. In keeping with the tree metaphor, the other nodes are called branches and the nodes at the tips with null references are called **leaves**. It may seem odd that we draw the picture with the root at the top and the leaves at the bottom, but that is not the strangest thing.

To make things worse, computer scientists mix in yet another metaphor: the family tree. The top node is sometimes called a **parent** and the nodes it refers to are its **children**. Nodes with the same parent are called **siblings**, and so on.

Finally, there is also a geometric vocabulary for taking about trees. I already mentioned left and right, but there is also ‘‘up’’ (toward the parent/root) and down (toward the children/leaves). Also, all the nodes that are the same distance from the root comprise a **level** of the tree.

I don’t know why we need three metaphors for talking about trees, but there it is.

21.2 Building trees

The process of assembling tree nodes is similar to the process of assembling lists. We have a constructor for tree nodes that initializes the instance variables.

```
    public Tree (int cargo, Tree* left, Tree* right) {
        this->cargo = cargo;
        this->left = left;
        this->right = right;
    }
```

We allocate the child nodes first:

```
    Tree* left = new Tree (2, null, null);
    Tree* right = new Tree (3, null, null);
```

We can create the parent node and link it to the children at the same time:

```
    Tree* tree = new Tree (1, left, right);
```

This code produces the state shown in the previous figure.

21.3 Traversing trees

By now, any time you see a new data structure, your first question should be, ‘‘How can I traverse it?’’ The most natural way to traverse a tree is recursively. For example, to add up all the integers in a tree, we could write this class method:

```
    int total (Tree tree) {
        if (tree == null) return 0;
        int cargo = tree->cargo;
        return cargo + total (tree->left) + total (tree->right);
    }
```

This is a class method because we would like to use null to represent the empty tree, and make the empty tree the base case of the recursion. If the tree is empty, the method returns 0. Otherwise it makes two recursive calls to find the total value of its two children. Finally, it adds in its own cargo and returns the total.

Although this method works, there is some difficulty fitting it into an object-oriented design. It should not appear in the Tree class because it requires the cargo to be int objects. If we make that assumption then we lose the advantages of a generic data structure.

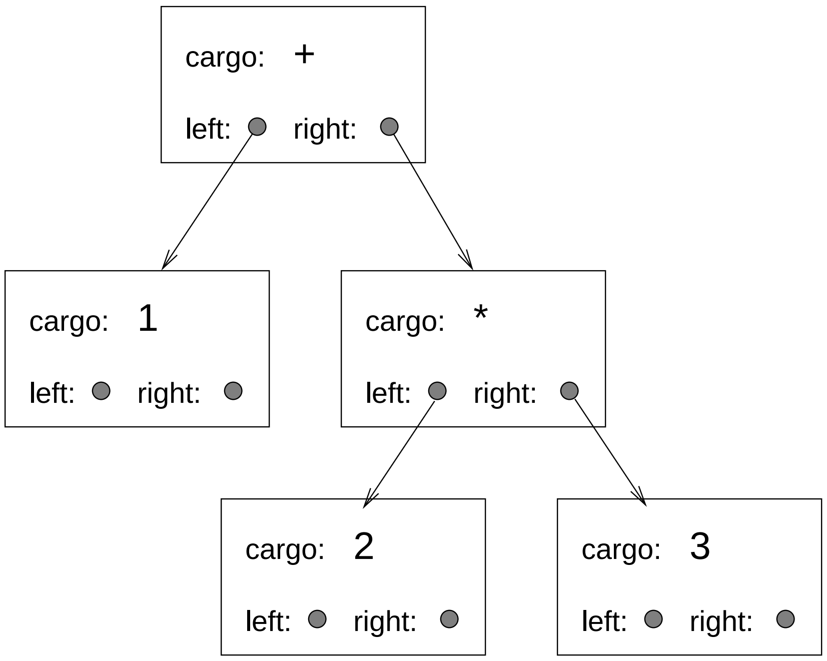
On the other hand, this code accesses the instance variables of the Tree nodes, so it ‘‘knows’’ more than it should about the implementation of the tree. If we changed that implementation later (and we will) this code would break.

Later in this chapter we will develop ways to solve this problem, allowing client code to traverse trees containing any kinds of objects without breaking the abstraction barrier between the client code and the implementation. Before we get there, let’s look at an application of trees.

21.4 Expression trees

A tree is a natural way to represent the structure of an expression. Unlike other notations, it can represent the comptation unambiguously. For example, the infix expression 1 + 2 * 3 is ambiguous unless we know that the multiplication happens before the addition.

The following figure represents the same computation:



The nodes can be operands like 1 and 2 or operators like + and *. Operands are leaf nodes; operator nodes contain references to their operands (all of these operators are **binary**, meaning they have exactly two operands).

Looking at this figure, there is no question what the order of operations is: the multiplication happens first in order to compute the first operand of the addition.

Expression trees like this have many uses. The example we are going to look at is translation from one format (postfix) to another (infix). Similar trees are used inside compilers to parse, optimize and translate programs.

123

21.5 Traversal

I already pointed out that recursion provides a natural way to traverse a tree. We can print the contents of an expression tree like this:

```
    public static void print (Tree tree) {
        if (tree == null) return;
        System.out.print (tree + " ");
        print (tree.left);
        print (tree.right);
    }
```

In other words, to print a tree, first print the contents of the root, then print the entire left subtree, then print the entire right subtree. This way of traversing a tree is called a **preorder**, because the contents of the root appear before the contents of the children.

For the example expression the output is + 1 * 2 3. This is different from both postfix and infix; it is a new notation called **prefix**, in which the operators appear before their operands.

You might suspect that if we traverse the tree in a different order we get the expression in a different notation. For example, if we print the subtrees first, and then the root node:

```
    public static void printPostorder (Tree tree) {
        if (tree == null) return;
        printPostorder (tree.left);
        printPostorder (tree.right);
        System.out.print (tree + " ");
    }
```

We get the expression in postfix (1 2 3 * +)! As the name of the previous method implies, this order of traversal is called **pos-**

torder. Finally, to traverse a tree **inorder**, we print the left tree, then the root, then the right tree:

```
    public static void printInorder (Tree tree) {
        if (tree == null) return;
        printInorder (tree.left);
        System.out.print (tree + " ");
        printInorder (tree.right);
    }
```

The result is 1 + 2 * 3, which is the expression in infix.

To be fair, I have to point out that I have omitted an important complication. Sometimes when we write an expression in infix we have to use parentheses to preserve the order of operations. So an inorder traversal is not quite sufficient to generate an infix expression.

Nevertheless, with a few improvements, the expression tree and the three recursive traversals provide a general way to translate expressions from one format to another.

21.6 Encapsulation

As I mentioned before, there is a problem with the way we have been traversing trees: it breaks down the barrier between the client code (the application that uses the tree) and the provider code (the Tree implementation). Ideally, tree code should be general; it shouldn't know anything about expression trees. And the code that generates and traverses the expression tree shouldn't know about the implementation of the trees. This design criterion is called **object encapsulation** to distinguish it from the encapsulation we saw in Section ??, which we might call **method encapsulation**.

In the current version, the Tree code knows too much about the client. Instead, the Tree class should provide the general capability of traversing a tree in various ways. As it traverses, it should perform operations on each node that are specified by the client.

To facilitate this separation of interests, we will create a new abstract class, called Visitable. The items stored in a tree will be required to be visitable, which means that they define a method named visit that does whatever the client wants done to each node. That way the Tree can perform the traversal and the client can perform the node operations.

Here are the steps we have to perform to wedge an abstract class between a client and a provider:

1. Define an abstract class that specifies the methods the provider code will need to invoke on its components.
2. Write the provider code in terms of the new abstract class, as opposed to generic Objects.
3. Define a concrete class that belongs to the abstract class and that implements the required methods as appropriate for the client.
4. Write the client code to use the new concrete class.

The next few sections demonstrate these steps.

21.7 Defining an abstract class

An abstract class definition looks a lot like a concrete class definition, except that it only specifies the interface of each method and not an implementation. The definition of Visitable is

```
public interface Visitable {
    public void visit ();
}
```

That's it! The word interface is Java's keyword for an abstract class. The definition of visit looks like any other method definition, except that it has no body. This definition specifies that any class that implements Visitable has to have a method named visit that takes no parameters and that returns void.

Like other class definitions, abstract class definitions go in a file with the same name as the class (in this case Visitable.java).

21.8 Implementing an abstract class

If we are using an expression tree to generate infix, then ''visiting'' a node means printing its contents. Since the contents of an expression tree are tokens, we'll create a new concrete class called Token that implements Visitable

```
public class Token implements Visitable {
    String str;

    public Token (String str) {
        this.str = str;
    }

    public void visit () {
        System.out.print (str + " ");
    }
}
```

When we compile this class definition (which is in a file named Token.java), the compiler checks whether the methods provided satisfy the requirements specified by the abstract class. If not, it will produce an error message. For example, if we misspell the name of the method that is supposed to be visit, we might get something like, ‘‘class Token must be declared abstract. It does not define void visit() from interface Visitable.’’

The next step is to modify the parser to put Token objects into the tree instead of Strings. Here is a small example:

```
    String expr = "1 2 3 * +";
    StringTokenizer st = new StringTokenizer (expr, " +-*/", true);
    String token = st.nextToken();
    Tree tree = new Tree (new Token (token), null, null));
```

This code takes the first token in the string and wraps it in a Token object, then puts the Token into a tree node. If the Tree requires the cargo to be Visitable, it will convert the Token to be a Visitable object. When we remove the Visitable from the tree, we will have to cast it back into a Token.

As an exercise, write a version of printPreorder called visitPreorder that traverses the tree and invokes visit on each node in preorder.

21.9 Array implementation of trees

What does it mean to ‘‘implement’’ a tree? So far we have only seen one implementation of a tree, a linked data structure similar to a linked list. But there are other structures we would like to identify as trees. Anything that can perform the basic set of tree operations should be recognized as a tree.

So what are the tree operation? In other words, how do we define the Tree ADT?

constructor: Build an empty tree.

empty: Is this tree the empty tree?

left: Return the left child of this node, or an empty tree if there is none.

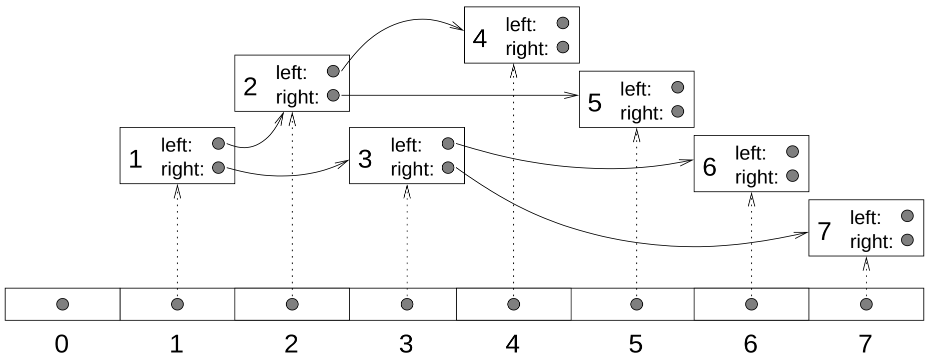
right: Return the left child of this node, or an empty tree if there is none.

parent: Return the parent of this node, or an empty tree if this node is the root.

In the implementation we have seen, the empty tree is represented by the special value null. left and right are performed by accessing the instance variables of the node. We have not implemented parent yet (you might think about how to do it).

There is another implementation of trees that uses arrays and indices instead of objects and references. To see how it works, we will start by looking at a hybrid implementation that uses both arrays and objects.

This figure shows a tree like the ones we have been looking at, although it is laid out sideways, with the root at the left and the leaves on the right. At the bottom there is an array of references that refer to the objects in the trees.



In this tree the cargo of each node is the same as the array index of the node, but of course that is not true in general. You might notice that array index 1 refers to the root node and array index 0 is empty. The reason for that will become clear soon.

So now we have a tree where each node has a unique index. Furthermore, the indices have been assigned to the nodes according to a deliberate pattern, in order to achieve the following results:

1. The left child of the node with index i has index $2i$.
2. The right child of the node with index i has index $2i + 1$.

3. The parent of the node with index i has index $i/2$ (rounded down).

Using these formulas, we can implement left, right and parent just by doing arithmetic; we don't have to use the references at all!

Since we don't use the references, we can get rid of them, which means that what used to be a tree node is now just cargo and nothing else. That means we can implement the tree as an array of cargo objects; we don't need tree nodes at all.

Here's what one implementation looks like:

```
public class Tree {
    Object[] array;

    public Tree () {
        array = new Object [128];
    }
```

No surprises so far. The instance variable is an array of Objects. The constructor initializes this array with an arbitrary initial size (we can always resize it later).

To check whether a tree is empty, we check whether the root node is null. Again, the root node is located at index 1.

```
    public boolean empty () {
        return (array[1] == null);
    }
```

The implementation of left, right and parent is just arithmetic:

```
    public int left (int i) {  return 2*i;  }
    public int right (int i) {  return 2*i + 1;  }
    public int parent (int i) {  return i/2;  }
```

Only one problem remanins. The node ''references'' we have are not really references; they are integer indices. To access the cargo itself, we have to get or set an element of the array. For that kind of operation, it is often a good idea to provide methods that perform simple error checking before accessing the data structure.

```
    public Object getCargo (int i) {
        if (i < 0 || i >= array.length) return null;
        return array[i];
    }
    public void setCargo (int i, Object obj) {
        if (i < 0 || i >= array.length) return;
        array[i] = obj;
    }
```

Methods like this are often called **accessor methods** because they provide access to a data structure (the ability to get and set elements) without letting the client see the details of the implementation.

Finally we are ready to build a tree. In another class (the client), we would write

```
    Tree tree = new Tree ();
    int root = 1;
    tree.setCargo (root, "cargo for root");
```

The constructor builds an empty tree. In this case we assume that the client knows that the index of the root is 1 although it would be preferable for the tree implementation to provide that information. Anyway, invoking setCargo puts the string "cargo for root" into the root node.

To add children to the root node:

```
    tree.setCargo (tree.left (root), "cargo for left");
    tree.setCargo (tree.right (root), "cargo for right");
```

In the tree class we could provide a method that prints the contents of the tree in preorder.

```
    public void printPreorder (int i) {
        if (getNode (i) == null) return;
        System.out.println (getNode (i));
        printPreorder (left (i));
        printPreorder (right (i));
    }
```

We invoke this method from the client by passing the root as a parameter.

```
    tree.print (root);
```

The output is

```
cargo for root
cargo for left
cargo for right
```

This implementation provides the basic operations required to be a tree, but it leaves a lot to be desired. As I pointed out, we expect the client to have a lot of information about the implementation, and the interface the client sees, with indices and all, is not very pretty.

Also, we have the usual problem with array implementations, which is that the initial size of the array is arbitrary and it might have to be resized. This last problem can be solved by replacing the array with a Vector.

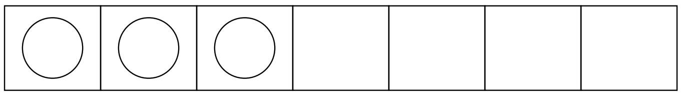
21.10 The Vector class

The Vector is a built-in Java class in the java.util package. It is an implementation of an array of Objects, with the added feature that it can resize itself automatically, so we don't have to.

The Vector class provides methods named get and set that are similar to the getCargo and setCargo methods we wrote for the Tree class. You should review the other Vector operations by consulting the online documentation.

Before using the Vector class, you should understand a few concepts. Every Vector has a capacity, which is the amount of space that has been allocated to store values, and a size, which is the number of values that are actually in the vector.

The following figure is a simple diagram of a Vector that contains three elements, but it has a capacity of seven.



In general, it is the responsibility of the client code to make sure that the vector has sufficient *size* before invoking set or get. If you try to access an element that does not exist (in this case the elements with indices 3 through 6), you will get an ArrayIndexOutOfBounds exception.

The Vector methods use the add and insert automatically increase the size of the Vector, but set does not. The resize method adds null elements to the end of the Vector to get to the given size.

Most of the time the client doesn't have to worry about capacity. Whenever the size of the Vector changes, the capacity is updated automatically. For performance reasons, some applications might want to take control of this function, which is why there are additional methods for increasing and decreasing capacity.

Because the client code has no access to the implementation of a vector, it is not clear how we should traverse one. Of course, one possibility is to use a loop variable as an index into the vector:

```
        for (int i=0; i<v.size(); i++) {
            System.out.println (v.get(i));
        }
```

There's nothing wrong with that, but there is another way that serves to demonstrate the Iterator class. Vectors provide a method named iterator that returns an Iterator object that makes it possible to traverse the vector.

21.11 The Iterator class

Iterator is an abstract class in the java.util package. It specifies three methods:

hasNext: Does this iteration have more elements?

next: Return the next element, or throw an exception if there is none.

remove: Remove from the collection the last element that was returned.

The following example uses an iterator to traverse and print the elements of a vector.

```
        Iterator iterator = vector.iterator ();
        while (iterator.hasNext ()) {
            System.out.println (iterator.next ());
        }
```

Once the Iterator is created, it is a separate object from the original Vector. Subsequent changes in the Vector are not reflected in the Iterator. In fact, if you modify the Vector after creating an Iterator, the Iterator becomes invalid. If you access the Iterator again, it will cause a ConcurrentModification exception.

In a previous section we used the Visitable abstract class to allow a client to traverse a data structure without knowing the details of its implementation. Iterators provide another way to do the same thing. In the first case, the provider performs the iteration and invokes client code to ‘‘visit’’ each element.

In the second case the provider gives the client an object that it can use to select elements one at a time (albeit in an order controlled by the provider).

As an exercise, write a concrete class named PreIterator that implements the Iterator interface, and write a method named preorderIterator for the Tree class that returns a PreIterator that selects the elements of the Tree in preorder.

21.12 Glossary

binary tree: A tree in which each node refers to 0, 1, or 2 dependent nodes.

root: The top-most node in a tree, to which no other nodes refer.

leaf: A bottom-most node in a tree, which refers to no other nodes.

parent: The node that refers to a given node.

child: One of the nodes referred to by a node.

level: The set of nodes equidistant from the root.

prefix notation: A way of writing a mathematical expression with each operator appearing before its operands.

preorder: A way to traverse a tree, visiting each node before its children.

postorder: A way to traverse a tree, visiting the children of each node before the node itself.

inorder: A way to traverse a tree, visiting the left subtree, then the root, then the right subtree.

class variable: A static variable declared outside of any method. It is accessible from any method.

binary operator: An operator that takes two operands.

object encapsulation: The design goal of keeping the implementations of two objects as separate as possible. Neither class should have to know the details of the implementation of the other.

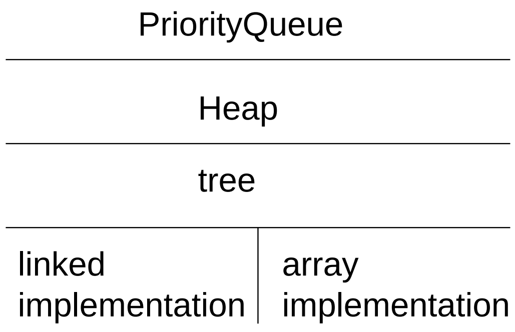
method encapsulation: The design goal of keeping the interface of a method separate from the details of its implementation.

Chapter 22

Heap

22.1 The Heap

A heap is a special kind of tree that happens to be an efficient implementation of a priority queue. This figure shows the relationships among the data structures in this chapter.



Ordinarily we try to maintain as much distance as possible between an ADT and its implementation, but in the case of the Heap, this barrier breaks down a little. The reason is that we are interested in the performance of the operations we implement. For each implementation there are some operations that are easy to implement and efficient, and others that are clumsy and slow.

It turns out that the array implementation of a tree works particularly well as an implementation of a Heap. The operations the array performs well are exactly the operations we need to implement a Heap.

To understand this relationship, we will proceed in a few steps. First, we need to develop ways of comparing the performance of various implementations. Next, we will look at the operations Heaps perform. Finally, we will compare the Heap implementation of a Priority Queue to the others (arrays and lists) and see why the Heap is considered particularly efficient.

22.2 Performance analysis

When we compare algorithms, we would like to have a way to tell when one is faster than another, or takes less space, or uses less of some other resource. It is hard to answer those questions in detail, because the time and space used by an algorithm depend on the implementation of the algorithm, the particular problem being solved, and the hardware the program runs on.

The objective of this section is to develop a way of talking about performance that is independent of all of those things, and only depends on the algorithm itself. To start, we will focus on run time; later we will talk about other resources.

Our decisions are guided by a series of constraints:

1. First, the performance of an algorithm depends on the hardware it runs on, so we usually don't talk about run time in absolute terms like seconds. Instead, we usually count the number of abstract operations the algorithm performs.

2. Second, performance often depends on the particular problem we are trying to solve -- some problems are easier than others. To compare algorithms, we usually focus on either the worst-case scenario or an average (or common) case.

3. Third, performance depends on the size of the problem (usually, but not always, the number of elements in a collection). We address this dependence explicitly by expressing run time as a function of problem size.

4. Finally, performance depends on details of the implementation like object allocation overhead and method invocation overhead. We usually ignore these details because they don't affect the rate at which the number of abstract operations increases with problem size.

To make this process more concrete, consider two algorithms we have already seen for sorting an array of integers. The first is **selection sort**, which we saw in Section 13.7. Here is the pseudocode we used there.

```
    selectionsort (array) {
        for (int i=0; i<array.length(); i++) {
          // find the lowest item at or to the right of i
          // swap the ith item and the lowest item
        }
    }
```

To perform the operations specified in the pseudocode, we wrote helper methods named findLowest() and swap. In **pseudocode**, findLowest() looks like this

```
    // find the index of the lowest item between
    // i and the end of the array

    findLowest (array, i) {
        // lowest contains the index of the lowest item so far
        lowest = i;
        for (int j=i+1; j<array.length(); j++) {
          // compare the jth item to the lowest item so far
          // if the jth item is lower, replace lowest with j
        }
        return lowest;
    }
```

And swap looks like this:

```
    swap (i, j) {
        // store a reference to the ith card in temp
        // make the ith element of the array refer to the jth card
        // make the jth element of the array refer to temp
    }
```

To analyze the performance of this algorithm, the first step is to decide what operations to count. Obviously, the program does a lot of things: it increments i, compares it to the length of the deck, it searches for the largest element of the array, etc. It is not obvious what the right thing is to count.

It turns out that a good choice is the number of times we compare two items. Many other choices would yield the same result in the end, but this is easy to do and we will find that it allows us to compare most easily with other sort algorithms.

The next step is to define the ‘‘problem size.’’ In this case it is natural to choose the size of the array, which we’ll call n.

Finally, we would like to derive an expression that tells us how many abstract operations (specifically, comparisons) we have to do, as a function of n.

We start by analyzing the helper methods. swap copies several references, but it doesn’t perform any comparisons, so we ignore the time spent performing swaps. findLowest starts at i and traverses the array, comparing each item to lowest. The number of items we look at is $n - i$, so the total number of comparisons is $n - i - 1$.

Next we consider how many times findLowest gets invoked and what the value of i is each time. The last time it is invoked, i is $n-2$ so the number of comparisons is 1. The previous iteration performs 2 comparisons, and so on. During the first iteration, i is 0 and the number of comparisons is $n - 1$.

So the total number of comparisons is $1 + 2 + \cdots + n - 1$. This sum is equal to $n^2/2 - n/2$. To describe this algorithm, we would typically ignore the lower order term ($n/2$) and say that the total amount of work is proportional to n^2. Since the leading order term is quadratic, we might also say that this algorithm is **quadratic time**.

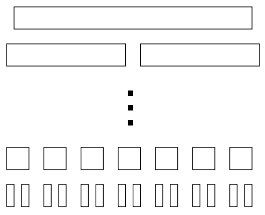
22.3 Analysis of mergesort

In Section 13.10 I claimed that mergesort takes time that is proportional to $n \log n$, but I didn't explain how or why. Now I will.

Again, we start by looking at pseudocode for the algorithm. For mergesort, it's

```
  mergeSort (array) {
    // find the midpoint of the array
    // divide the array into two halves
    // sort the halves recursively
    // merge the two halves and return the result
  }
```

At each level of the recursion, we split the array in half, make two recursive calls, and then merge the halves. Graphically, the process looks like this:



# arrays	items per array	# merges	comparisons per merge	total work
1	n	1	n–1	~n
2	n/2	2	n/2–1	~n
⋮	⋮	⋮	⋮	⋮
n/2	2	n/2	2–1	~n
n	1	0	0	

Each line in the diagram is a level of the recursion. At the top, a single array divides into two halves. At the bottom, n

arrays (with one element each) are merged into $n/2$ arrays (with 2 elements each).

The first two columns of the table show the number of arrays at each level and the number of items in each array. The third column shows the number of merges that take place at each level of recursion. The next column is the one that takes the most thought: it shows the number of comparisons each merge performs.

If you look at the pseudocode (or your implementation) of merge, you should convince yourself that in the worst case it takes $m - 1$ comparisons, where m is the total number items being merged.

The next step is to multiply the number of merges at each level by the amount of work (comparisons) per merge. The result is the total work at each level. At this point we take advantage of a small trick. We know that in the end we are only interested in the leading-order term in the result, so we can go ahead and ignore the -1 term in the comparisons per merge. If we do that, then the total work at each level is simply n.

Next we need to know the number of levels as a function of n. Well, we start with an array of n items and divide it in half until it gets to 1. That's the same as starting at 1 and multiplying by 2 until we get to n. In other words, we want to know how many times we have to multiply 2 by itself before we get to n. The answer is that the number of levels, l, is the logarithm, base 2, of n.

Finally, we multiply the amount of work per level, n, by the number of levels, $\log_2 n$ to get $n \log_2 n$, as promised. There isn't a good name for this functional form; most of the time people just say, ''en log en.''

It might not be obvious at first that $n \log_2 n$ is better than n^2, but for large values of n, it is. As an exercise, write a program that prints $n \log_2 n$ and n^2 for a range of values of n.

22.4 Overhead

Performance analysis takes a lot of handwaving. First we ignored most of the operations the program performs and counted only comparisons. Then we decided to consider only worst case performance. During the analysis we took the liberty of rounding a few things off, and when we finished, we casually discarded the lower-order terms.

When we interpret the results of this analysis, we have to keep all this hand-waving in mind. Because mergesort is $n \log_2 n$, we consider it a better algorithm than selection sort, but that doesn't mean that mergesort is *always* faster. It just means that eventually, if we sort bigger and bigger arrays, mergesort will win.

How long that takes depends on the details of the implementation, including the additional work, besides the comparisons we counted, that each algorithm performs. This extra work is sometimes called **overhead**. It doesn't affect the performance analysis, but it does affect the run time of the algorithm.

For example, our implementation of mergesort actually allocates subarrays before making the recursive calls and then lets them get garbage collected after they are merged. Looking again at the diagram of mergesort, we can see that the total amount of space that gets allocated is proportional to $n \log_2 n$, and the total number of objects that get allocated is about $2n$. All that allocating takes time.

Even so, it is most often true that a bad implementation of a good algorithm is better than a good implementation of a bad algorithm. The reason is that for large values of n the good algorithm is better and for small values of n it doesn't matter because both algorithms are good enough.

As an exercise, write a program that prints values of $1000 n \log_2 n$ and n^2 for a range of values of n. For what value of n are they equal?

22.5 Priority Queue implementations

In Chapter 20 we looked at an implementation of a Priority Queue based on an array. The items in the array are unsorted, so it is easy to add a new item (at the end), but harder to remove an item, because we have to search for the item with the highest priority.

An alternative is an implementation based on a sorted list. In this case when we insert a new item we traverse the list and put the new item in the right spot. This implementation takes advantage of a property of lists, which is that it is easy to insert a new node into the middle. Similarly, removing the item with the highest priority is easy, provided that we keep it at the beginning of the list.

Performance analysis of these operations is straightforward. Adding an item to the end of an array or removing a node from the beginning of a list takes the same amount of time regardless of the number of items. So both operations are constant time.

Any time we traverse an array or list, performing a constant-time operation on each element, the run time is proportional to the number of items. Thus, removing something from the array and adding something to the list are both linear time.

So how long does it take to insert and then remove n items from a Priority Queue? For the array implementation, n insertions takes time proportional to n, but the removals take longer. The first removal has to traverse all n items; the second has to traverse $n - 1$, and so on, until the last removal, which only has to look at 1 item. Thus, the total time is $1 + 2 + \cdots + n$, which is (still) $n^2/2 - n/2$. So the total for the insertions and the removals is the sum of a linear function and a quadratic function. The leading term of the result is quadratic.

The analysis of the list implementation is similar. The first insertion doesn't require any traversal, but after that we have to traverse at least part of the list each time we insert a new item. In general we don't know how much of the list we will have to traverse, since it depends on the data and what order they are inserted, but we can assume that on average we have to traverse half of the list. Unfortunately, even traversing half of the list is still a linear operation.

So, once again, to insert and remove n items takes time proportional to n^2. Thus, based on this analysis we cannot say which implementation is better; both the array and the list yield quadratic run times.

If we implement a Priority Queue using a heap, we can perform both insertions and removals in time proportional to $logn$. Thus the total time for n items is $n \log n$, which is better than n^2. That's why, at the beginning of the chapter, I said that a heap is a particularly efficient implementation of a Priority Queue.

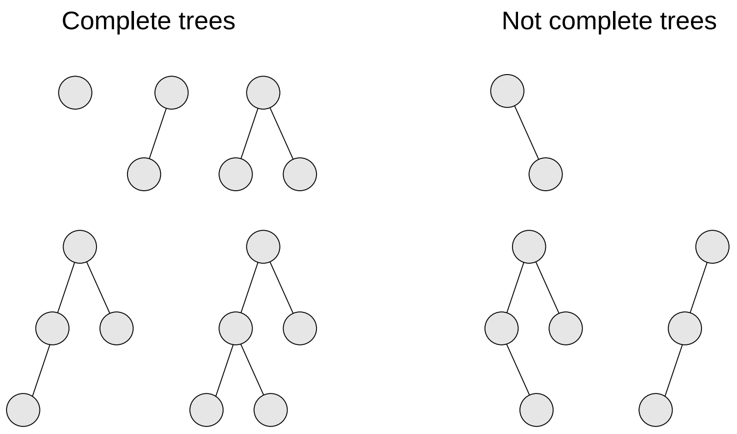
22.6 Definition of a Heap

A heap is a special kind of tree. It has two properties that are not generally true for other trees:

completeness: The tree is complete, which means that nodes are added from top to bottom, left to right, without leaving any spaces.

heapness: The item in the tree with the highest priority is at the top of the tree, and the same is true for every subtree.

Both of these properties bear a little explaining. This figure shows a number of trees that are considered complete or not complete:



An empty tree is also considered complete. We can define completeness more rigorously by comparing the height of the subtrees. Recall that the **height** of a tree is the number of levels.

Starting at the root, if the tree is complete, then the height of the left subtree and the height of the right subtree should be equal, or the left subtree may be taller by one. In any other case, the tree cannot be complete.

Furthermore, if the tree is complete, then the height relationship between the subtrees has to be true for every node in the tree.

It is natural to write these rules as a recursive method:

```
int height(Tree* head)
{
  if(head==null) return 0;
  int right = height(head->right), left = height(head->left);
  if(right > left) return right + 1;
  return left + 1;
}

bool isComplete (Tree *tree) {
    // the null tree is complete
    if (tree == null) return true;

    int leftHeight = height (tree->left);
    int rightHeight = height (tree->right);
    int diff = leftHeight - rightHeight

    // check the root node
```

```
        if (diff < 0 || diff > 1) return false;

        // check the children
        if (!isComplete (tree->left)) return false;
        return isComplete (tree->right);
    }
```

For this example I used the linked implementation of a tree. As an exercise, write the same method for the array implementation. Also as an exercise, write the height method. The height of a null tree is 0 and the height of a leaf node is 1.

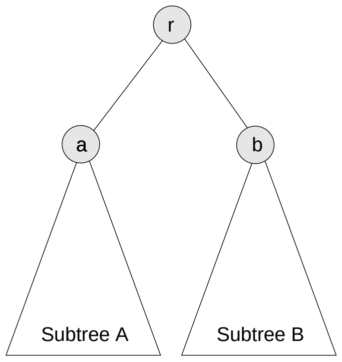
The **heap property** is similarly recursive. In order for a tree to be a heap, the largest value in the tree has to be at the root, ***and*** the same has to be true for each subtree. As another exercise, write a method that checks whether a tree has the heap property.

22.7 Heap remove

It might seem odd that we are going to remove things from the heap before we insert any, but I think removal is easier to explain.

At first glance, we might think that removing an item from the heap is a constant time operation, since the item with the highest priority is always at the root. The problem is that once we remove the root node, we are left with something that is no longer a heap. Before we can return the result, we have to restore the heap property. We call this operation reheapify.

The situation is shown in the following figure:



The root node has priority r and two subtrees, A and B. The value at the root of Subtree A is a and the value at the root of Subtree B is b.

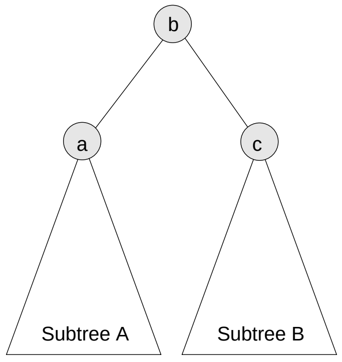
We assume that before we remove r from the tree, the tree is a heap. That implies that r is the largest value in the heap and that a and b are the largest values in their respective subtrees.

Once we remove r, we have to make the resulting tree a heap again. In other words we need to make sure it has the properties of completeness and heapness.

The best way to ensure completeness is to remove the bottom-most, right-most node, which we'll call c and put its value at the root. In a general tree implementation, we would have to traverse the tree to find this node, but in the array implementation, we can find it in constant time because it is always the last (non-null) element of the array.

Of course, the chances are that the last value is not the highest, so putting it at the root breaks the heapness property. Fortunately it is easy to restore. We know that the largest value in the heap is either a or b. Therefore we can select whichever is larger and swap it with the value at the root.

Arbitrarily, let's say that b is larger. Since we know it is the highest value left in the heap, we can put it at the root and put c at the top of Subtree B. Now the situation looks like this:



Again, c is the value we copied from the last entry in the array and b is the highest value left in the heap. Since we haven't changed Subtree A at all, we know that it is still a heap. The only problem is that we don't know if Subtree B is a heap, since we just stuck a (probably low) value at its root.

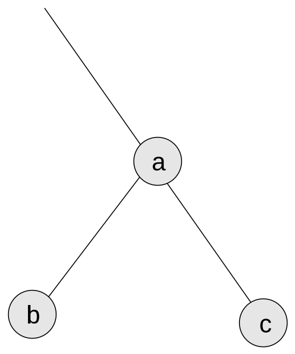
Wouldn't it be nice if we had a method that could reheapify Subtree B? Wait... we do!

22.8 Heap insert

Inserting a new item in a heap is a similar operation, except that instead of trickling a value down from the top, we trickle it up from the bottom.

Again, to guarantee completeness, we add the new element at the bottom-most, rightmost position in the tree, which is the next available space in the array.

Then to restore the heap property, we compare the new value with its neighbors. The situation looks like this:



The new value is c. We can restore the heap property of this subtree by comparing c to a. If c is smaller, then the heap property is satisfied. If c is larger, then we swap c and a. The swap satisfies the heap property because we know that c must also be bigger than b, because c > a and a > b.

Now that the subtree is reheapified, we can work our way up the tree until we reach the root.

22.9 Performance of heaps

For both insert and remove, we perform a constant time operation to do the actual insertion and removal, but then we have to reheapify the tree. In one case we start at the root and work our way down, comparing items and then recursively reheapifying one of the subtrees. In the other case we start at a leaf and work our way up, again comparing elements at each level of the tree.

As usual, there are several operations we might want to count, like comparisons and swaps. Either choice would work; the real issue is the number of levels of the tree we examine and how much work we do at each level. In both cases we keep examining levels of the tree until we restore the heap property, which means we might only visit one, or in the worst case we might have to visit them all. Let's consider the worst case.

At each level, we perform only constant time operations like comparisons and swaps. So the total amount of work is proportional to the number of levels in the tree, a.k.a. the height.

So we might say that these operations are linear with respect to the height of the tree, but the ''problem size'' we are interested in is not height, it's the number of items in the heap.

As a function of n, the height of the tree is $log_2 n$. This is not true for all trees, but it is true for complete trees. To see why, think of the number of nodes on each level of the tree. The first level contains 1, the second contains 2, the third contains 4, and so on. The ith level contains 2^i nodes, and the total number in all levels up to i is $2^i - 1$. In other words, $2^h = n$, which means that $h = log_2 n$.

Thus, both insertion and removal take **logarithmic** time. To insert and remove n items takes time proportional to $n \log_2 n$.

22.10 Heapsort

The result of the previous section suggests yet another algorithm for sorting. Given n items, we insert them into a Heap and then remove them. Because of the Heap semantics, they come out in order. We have already shown that this algorithm, which is called **heapsort**, takes time proportional to $n \log_2 n$, which is better than selection sort and the same as mergesort.

As the value of n gets large, we expect heapsort to be faster than selection sort, but performance analysis gives us no way to know whether it will be faster than mergesort. We would say that the two algorithms have the same **order of growth** because they grow with the same functional form. Another way to say the same thing is that they belong to the same **complexity class**.

Complexity classes are sometimes written in ''big-O notation''. For example, $\mathcal{O}(n^2)$, pronounced ''oh of en squared'' is the set of all functions that grow no faster than n^2 for large values of n. To say that an algorithm is $\mathcal{O}(n^2)$ is the same as saying that it is quadratic. The other complexity classes we have seen, in decreasing order of performance, are:

$\mathcal{O}(1)$	constant time
$\mathcal{O}(\log n)$	logarithmic
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \log n)$	''en log en''
$\mathcal{O}(n^2)$	quadratic
$\mathcal{O}(2^n)$	exponential

So far none of the algorithms we have looked at are **exponential**. For large values of n, these algorithms quickly become impractical. Nevertheless, the phrase ''exponential growth'' appears frequently in even non-technical language. It is frequently misused so I wanted to include its technical meaning.

People often use ''exponential'' to describe any curve that is increasing and accelerating (that is, one that has positive slope and curvature). Of course, there are many other curves that fit this description, including quadratic functions (and higher-order polynomials) and even functions as undramatic as $n \log n$. Most of these curves do not have the (often detrimental) explosive behavior of exponentials.

As an exercise, compare the behavior of $1000n^2$ and 2^n as the value of n increases.

22.11 Glossary

selection sort: The simple sorting algorithm in Section 13.7.

mergesort: A better sorting algorithm from Section 13.10.

heapsort: Yet another sorting algorithm.

complexity class: A set of algorithms whose performance (usually run time) has the same order of growth.

order of growth: A set of functions with the same leading-order term, and therefore the same qualitative behavior for large values of n.

overhead: Additional time or resources consumed by a programming performing operations other than the abstract operations considered in performance analysis.

Chapter 23

File Input/Output and pmatrices

In this chapter we will develop a program that reads and writes files, parses input, and demonstrates the pmatrix 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.

23.1 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.h, 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.

23.2 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;
  pstring 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:

```
  pstring 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 pstring to a native C string. Because the ifstream constructor expects a C string as an argument, we have to convert the pstring.

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.

23.3 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;
  }
```

23.4 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 ppearance 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 pstring member function; it takes two arguments, the starting index of the substring and the length.

```
void processLine (const pstring& line)
{
  // the character we are looking for is a quotation mark
  char quote = '\"';

  // store the indices of the quotation marks in a vector
  pvector<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;
  pstring city1 = line.substr (quoteIndex[0]+1, len1);
  int len2 = quoteIndex[3] - quoteIndex[2] - 1;
  pstring city2 = line.substr (quoteIndex[2]+1, len2);
  int len3 = line.length() - quoteIndex[2] - 1;
  pstring 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.

23.5 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 stdlib.h.

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 pstring& s)
{
  pstring 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 7.9, 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.

23.6 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 pstrings, which are collections of characters, and pvectors 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 pstrings and pvectors 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 pvectors.

Here is the beginning of a class definition for a Set.

```
class Set {
private:
  pvector<pstring> elements;
  int numElements;

public:
  Set (int n);

  int getNumElements () const;
  pstring getElement (int i) const;
  int find (const pstring& s) const;
  int add (const pstring& s);
};
```

```
Set::Set (int n)
{
  pvector<pstring> temp (n);
  elements = temp;
  numElements = 0;
}
```

The instance variables are an pvector 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 pvector. Usually it will be smaller.

The Set constructor takes a single parameter, which is the initial size of the pvector. 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 pvector, it checks to make sure the index is greater than or equal to zero and less than the length of the pvector. 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 pvector.

```
pstring 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 pstring& s) const
{
  for (int i=0; i<numElements; i++) {
    if (elements[i] == s) return i;
  }
  return -1;
}
```

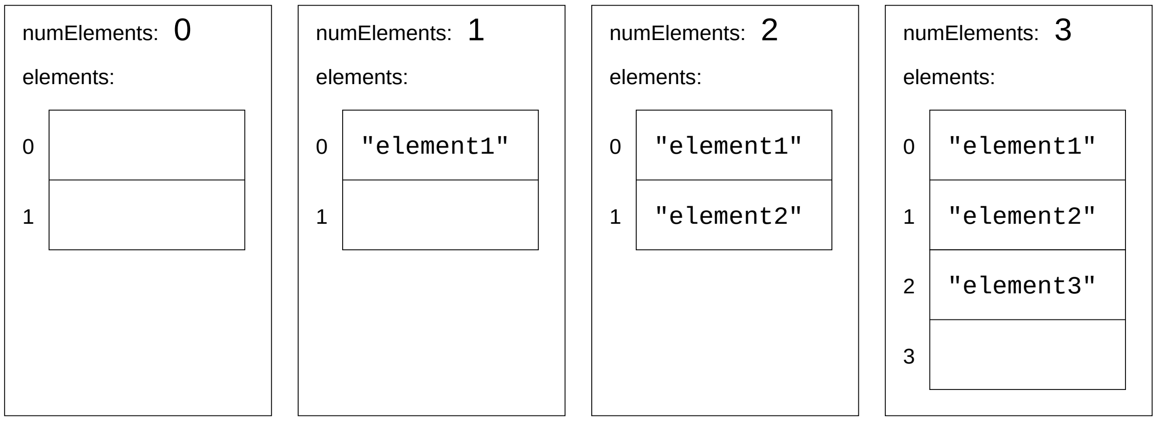
So that leaves us with add. Often the return type for something like add would be void, but in this case it might be useful to make it return the index of the element.

```
int Set::add (const pstring& s)
{
  // if the element is already in the set, return its index
  int index = find (s);
  if (index != -1) return index;

  // if the pvector 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 pvector, 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.

23.7 pmatrix

An pmatrix is similar to an pvector 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:

```
  pmatrix<char> m1;
  pmatrix<int> m2 (3, 4);
  pmatrix<double> m3 (rows, cols, 0.0);
  pmatrix<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 pmatrix as a parameter.

Just as with pvectors, we can make pmatrixes with any type of elements (including pvectors, and even pmatrixes).

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;
  }
```

23.8 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:

```
  pmatrix<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.

23.9 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 pmatrix 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 pmatrix 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;
  pmatrix<int> distances;

public:
  DistMatrix (int rows);

  void add (const pstring& city1, const pstring& city2, int dist);
  int distance (int i, int j) const;
  int distance (const pstring& city1, const pstring& city2) const;
  pstring cityName (int i) const;
  int numCities () const;
  void print ();
};
```

Using this interface simplifies main:

```
void main ()
{
  pstring line;
  ifstream infile ("distances");
  DistMatrix distances (2);

  while (true) {
    getline (infile, line);
    if (infile.eof()) break;
    processLine (line, distances);
  }

  distances.print ();
}
```

It also simplifies processLine:

```
void processLine (const pstring& line, DistMatrix& distances)
{
  char quote = '\"';
  pvector<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;
  pstring city1 = line.substr (quoteIndex[0]+1, len1);
  int len2 = quoteIndex[3] - quoteIndex[2] - 1;
  pstring city2 = line.substr (quoteIndex[2]+1, len2);
  int len3 = line.length() - quoteIndex[2] - 1;
  pstring 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.

23.10 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.

Appendix A

Quick reference for pclasses

These class definitions are copied from the pclasses web page,
http://www.ibiblio.org/obp/pclasses/, with a few minor formatting
changes.

A.1 pstring

```
template <class T>
class pstringT
{

public:
    pstringT<T>();                              //default constructor
    pstringT<T>(const pstringT<T> &);           //copy constructor
    pstringT<T>(const T *copy);                 //copy constructor from C-style str
    pstringT<T>(T ch);                          //copy constructor from single char
    virtual ~pstringT<T>();                     //destructor

    inline T * c\_str() const;                   //returns a null-terminated, C-sty
    inline int length() const;                  //returns the number of characters

    int find(const pstringT<T> &str) const;     //return index of str or -1 if not
    int find(const T ch) const;                 //return index of ch or -1 if not f
    pstringT<T> substr(int pos, int len) const; //returns substring from pos of len

    T & operator [] (int n);                    //access a character (mutable)
    const T operator [] (int n) const;          //access a character (immutable)

    const pstringT<T> & operator = (const pstringT<T> &);   //assignment operator
    const pstringT<T> & operator = (const T * const);       //assignment operator f
    const pstringT<T> & operator = (const T);               //assignment operator f
    const pstringT<T> & operator += (const pstringT<T> &);  //concatenation operator
    const pstringT<T> & operator += (const T * const);      //concatenation operator from C-style
    const pstringT<T> & operator += (const T);              //concatenation operator from single

protected:
    T *mystring;

};

//typedef for regular pstrings
typedef pstringT<char> pstring;

//concatenation operators
template <class T> pstringT<T> operator + (const pstringT<T> &, const pstringT<T> &);
template <class T> pstringT<T> operator + (const pstringT<T> &, T);
template <class T> pstringT<T> operator + (T, const pstringT<T> &);
template <class T> pstringT<T> operator + (const pstringT<T> &, const T * const);
template <class T> pstringT<T> operator + (const T * const, const pstringT<T> &);

//stream operators
template <class T> inline ostream & operator << (ostream &, const pstringT<T> &);
template <class T> istream & operator >> (istream &, pstringT<T> &);
template <class T> istream & getline(istream &, pstringT<T> &);

//comparison operators
template <class T> inline bool operator == (const pstringT<T> &, const pstringT<T> &);
template <class T> inline bool operator != (const pstringT<T> &, const pstringT<T> &);
template <class T> inline bool operator <  (const pstringT<T> &, const pstringT<T> &);
template <class T> inline bool operator <= (const pstringT<T> &, const pstringT<T> &);
template <class T> inline bool operator >  (const pstringT<T> &, const pstringT<T> &);
template <class T> inline bool operator >= (const pstringT<T> &, const pstringT<T> &);

template <class T>
ostream & operator << (ostream &os, const pstringT<T> &out)
{
    return os << out.c\_str() << flush;
}

template <class T>
istream & operator >> (istream &is, pstringT<T> &in)
{
    fflush(stdin);
    T input\_buffer[4096];
    is >> input\_buffer;
    in = input\_buffer;
    return is;
}

template <class T>
istream & getline(istream &is, pstringT<T> &to\_get)
{
    fflush(stdin);
    T getline\_buffer[4096];
    getline(is, getline\_buffer, 4095);
    to\_get = getline\_buffer;
    return is;
}

template <class T>
pstringT<T>::pstringT()
{
    mystring = new T[1];
    mystring[0] = 0;
}

template <class T>
pstringT<T>::pstringT(const T *copy)
{
    mystring = new T[strlen(copy) + 1]; //allocate memory
    strcpy(mystring, copy);                 //copy string
}

template <class T>
pstringT<T>::pstringT(T ch)
{
    mystring = new T[2];
    mystring[0] = ch;
    mystring[1] = 0;
}

template <class T>
pstringT<T>::pstringT(const pstringT<T> & to\_create\_from)
{
    mystring = new T[to\_create\_from.length()+1];
    strcpy(mystring, to\_create\_from.c\_str());   //copy string
}

template <class T>
pstringT<T>::~pstringT<T>()
{
    delete[] mystring;
```

```
}

template <class T>
T* pstringT<T>::c\_str() const
{
    return mystring;
}

template <class T>
int pstringT<T>::length() const
{
    return strlen(mystring);
}

template <class T>
int pstringT<T>::find(const pstringT<T> & str) const
{
    int i, j, endsearch = length() - str.length() + 1;
    for(i = 0; i < endsearch; i++)
    {
        for(; i < endsearch && mystring[i] != str[0]; i++);
        if(i == endsearch)
            break;

        for(j = 0; j < str.length() && mystring[i+j] == str[j]; j++);
        if(j == str.length())
            return i;
    }
    return -1;
}

template <class T>
int pstringT<T>::find(const T ch) const
{
    for(int i = 0; i < length(); i++)
        if(mystring[i] == ch)
            return i;

    return -1;
}

template <class T>
pstringT<T> pstringT<T>::substr(int pos, int len) const
{
    if(pos < 0 || len < 0 || pos >= length())
    {
        cerr << "\nError: substring (" << pos << "," << len
             << ") out of bounds for string \"" << mystring << '\"' << endl;
        exit(1);
    }
    if(pos + len > length())
        len = length() - pos;
    T *result = new T[len + 1];
    memcpy(result, mystring + pos, len * sizeof(T));
    result[len] = 0;
    pstringT<T> to\_return(result);
    delete[] result;
    return to\_return;
}

template <class T>
const pstringT<T> & pstringT<T>::operator = (const T * const to\_copy)
{
    delete[] mystring;                  //deallocate mem
    mystring = new T[strlen(to\_copy)+1];
    strcpy(mystring, to\_copy);          //copy string
    return *this;
}

template <class T>
const pstringT<T> & pstringT<T>::operator = (const T ch)
{
    delete[] mystring;                  //deallocate memory
    mystring = new T[2];
    mystring[0] = ch;
    mystring[1] = 0;
    return *this;
}

template <class T>
const pstringT<T> & pstringT<T>::operator = (const pstringT<T> &copy)
{
    return *this = copy.c\_str();        //call T pointer copier
}

template <class T>
const pstringT<T> & pstringT<T>::operator += (const T * const to\_append)
{
    T *newbuffer = new T[length() + strlen(to\_append) + 1];
    strcpy(newbuffer, mystring);
    strcat(newbuffer, to\_append);
    delete[] mystring;
    mystring = newbuffer;
    return *this;
}

template <class T>
const pstringT<T> & pstringT<T>::operator += (const pstringT<T> &to\_append)
{
    return *this += to\_append.c\_str();  //append T pointer
}

template <class T>
const pstringT<T> & pstringT<T>::operator += (const T to\_append)
{
    T *newstring = new T[length()+2];
    strcpy(newstring, mystring);
    delete[] mystring;
    mystring = newstring;                //points to new string
    mystring[length()+1] = 0;       //null terminator
    mystring[length()] = to\_append;
    return *this;
}

template <class T>
pstringT<T> operator + (const pstringT<T> & lval, const pstringT<T> & rval)
{
    pstringT<T> to\_return(lval);
    return to\_return += rval;
}

template <class T>
pstringT<T> operator + (const pstringT<T> & lval, const T & rval)
{
    pstringT<T> to\_return(lval);
    return to\_return += rval;
}

template <class T>
pstringT<T> operator + (T lval, const pstringT<T> & rval)
{
    pstringT<T> to\_return(lval);
    return to\_return += rval;
}

template <class T>
pstringT<T> operator + (const pstringT<T> & lval, const T * const rval)
{
    pstringT<T> to\_return(lval);
    return to\_return += rval;
}

template <class T>
pstringT<T> operator + (const T * const lval, const pstringT<T> & rval)

{
    pstringT<T> to\_return(lval);
    return to\_return += rval;
}

template <class T>
T & pstringT<T>::operator [] (int n)
{
    if(n<0 || n>=length())
    {
        cerr << "\nError: index out of range: " << n << " in string \""
             << mystring << "\" of length " << length() << endl;
        exit(1);
    }
    return mystring[n];
}

template <class T>
const T pstringT<T>::operator [] (int n) const
{
    if(n<0 || n>=length())
    {
        cerr << "\nError: index out of range: " << n << " in string \""
             << mystring << "\" of length " << length() << endl;
        exit(1);
    }
    return mystring[n];
}

template <class T>
bool operator == (const pstringT<T> &lval, const pstringT<T> &rval)
{
    return strcmp(lval.c\_str(), rval.c\_str()) == 0;
}

template <class T>
bool operator != (const pstringT<T> &lval, const pstringT<T> &rval)
{
    return strcmp(lval.c\_str(),rval.c\_str()) != 0;
```

```
}

template <class T>
bool operator < (const pstringT<T> &lval, const pstringT<T> &rval)
{
    return strcmp(lval.c\_str(), rval.c\_str()) < 0;
}

template <class T>
bool operator <= (const pstringT<T> &lval, const pstringT<T> &rval)
{
    return strcmp(lval.c\_str(), rval.c\_str()) <= 0;
}

template <class T>
bool operator > (const pstringT<T> &lval, const pstringT<T> &rval)
{
    return strcmp(lval.c\_str(), rval.c\_str()) > 0;
}

template <class T>
bool operator >= (const pstringT<T> &lval, const pstringT<T> &rval)
{
    return strcmp(lval.c\_str(), rval.c\_str()) >= 0;
}
```

A.2 pvector

```
template <class T>
class pvector
{

public:
    pvector();                                 //default constructor
    pvector(int size);                         //constructor with specific dimension
    pvector(int size, const T &fill\_val);      //create a pvector with a default fill value
    pvector(const pvector<T> &);               //copy constructor
    virtual ˜pvector();                        //destructor

    void resize(int new\_size);                 //resize the vector

    inline int length() const;                 //returns number of elements in pvector

    T & operator [] (int index);               //access a particular array element (mutable)
    const T & operator [] (int index) const;   //access a particular array element (immutable)
    const pvector<T> & operator = (const pvector<T> &); //assignment operator

protected:
    T *array;
    int len;

};

template <class T>
pvector<T>::pvector() :array(0), len(0)
{}

template <class T>
pvector<T>::pvector(int size)
{
    if(size <= 0)
    {
        cerr << "\nError: invalid pvector dimension: " << size << endl;
        exit(1);
    }
    array = new T[size];
    len = size;
}

template <class T>
pvector<T>::pvector(int size, const T &fill\_val)
{
    array = new T[size];
    len = size;

    for(int i=0; i<size; i++)
        array[i] = fill\_val;
}

template <class T>
pvector<T>::pvector(const pvector<T> &vec)
{
    array = new T[vec.length()];
    for(int i=0; i<vec.length(); i++)
        array[i] = vec[i];
    len = vec.length();
}

template <class T>
pvector<T>::˜pvector()
```

```
{
    delete[] array;
}

template <class T>
int pvector<T>::length() const
{
    return len;
}

template <class T>
T & pvector<T>::operator [] (int index)
{
    if(index < 0 || index >= length())
    {
        cerr << "\nError: index out of range: " << index
             << " in pvector of length " << length() << endl;
        exit(1);
    }
    return array[index];
}

template <class T>
const T & pvector<T>::operator [] (int index) const
{
    if(index < 0 || index >= length())
    {
        cerr << "\nError: index out of range: " << index
             << " in pvector of length " << length() << endl;
        exit(1);
    }
    return array[index];
}

template <class T>
const pvector<T> & pvector<T>::operator = (const pvector<T> & vec)
{
    delete[] array;
    array = new T[vec.length()];
    for(int i=0; i<vec.length(); i++)
        array[i] = vec[i];
    len = vec.length();
    return *this;
}

template <class T>
```

```
void pvector<T>::resize(int new\_size)
{
    if(new\_size <= 0)
    {
        cerr << "\nError: invalid pvector dimension: " << new\_size << endl;
        exit(1);
    }

    T *newarray = new T[new\_size];

    int minsize = (new\_size<len)?new\_size:len;
    for(int i=0; i<minsize; i++)
        newarray[i] = array[i];

    delete[] array;
    array = newarray;
    len = new\_size;
}
```

A.3 pmatrix

```
template <class T>
class pmatrix
{

public:
    pmatrix();                                          //default constructor
    pmatrix(int rows, int cols);                        //constructor with dimensio
    pmatrix(int rows, int cols, const T & fillvalue);   //constructor with default
    pmatrix(const pmatrix<T> &);                        //copy constructor
    virtual ~pmatrix();                                 //destructor

    void resize(int rows, int cols);                    //change matrix dimensions
    inline int numrows() const;                         //returns number of rows in
    inline int numcols() const;                         //returns number of columns

    inline pvector<T> & operator [] (int index);            //access a particular a
    inline const pvector<T> & operator [] (int index) const;//access a particular a

    const pmatrix<T> & operator = (const pmatrix<T> &);     //assignment operator

protected:
    pvector< pvector<T> > matrix;

};
```

```
template <class T>
pmatrix<T>::pmatrix()
{}

template <class T>
pmatrix<T>::pmatrix(int rows, int cols)
{
    resize(rows,cols);
}

template <class T>
pmatrix<T>::pmatrix(int rows, int cols, const T & fillvalue)
{
    resize(rows,cols);
    for(int x = 0; x < rows; x++)
        for(int y = 0; y < cols; y++)
            matrix[x][y] = fillvalue;
}

template <class T>
pmatrix<T>::pmatrix(const pmatrix<T> & copy)
{
    *this = copy;
}

template <class T>
pmatrix<T>::~pmatrix()
{}

template <class T>
int pmatrix<T>::numrows() const
{
    return matrix.length();
}

template <class T>
int pmatrix<T>::numcols() const
{
    return (matrix.length())?matrix[0].length():0;
}

template <class T>
void pmatrix<T>::resize(int rows, int cols)
{
    matrix.resize(rows);
    for(int x = 0; x < rows; x++)
        matrix[x].resize(cols); //resize each individual vector
}

template <class T>
const pmatrix<T> & pmatrix<T>::operator = (const pmatrix<T> & copy)
{
    matrix = copy.matrix;       //copy vector of vectors
    return *this;
}

template <class T>
const pvector<T> & pmatrix<T>::operator [] (int index) const
{
    return matrix[index];
}

template <class T>
pvector<T> & pmatrix<T>::operator [] (int index)
{
    return matrix[index];
}
```

A.4 pstack

```
template <class T>
class pstack
{

public:
    pstack();                                         //default constructor
    pstack(const pstack<T> &);                        //copy constructor
    virtual ~pstack();                                //destructor

    void push(T storage);                             //push data onto stack
    void pop(T & storage);                            //pop data off stack and store in p
    const T pop();                                    //pop data off stack and return
    const T top() const;                              //returns top value without popping

    void makeEmpty();                                 //empty the stack

    inline bool isEmpty() const;                      //returns true if stack is empty
    inline int length() const;                        //returns number of elements on sta

    const pstack<T> & operator = (const pstack<T> &);   //assignment operator
```

```
protected:
    struct node
    {
        T data;
        node *next;
        node() :next(0) {}
        node(const T & a) :next(0), data(a) {}
    } *sp;
    int size;

};

template <class T>
pstack<T>::pstack() :sp(0), size(0)
{}

template <class T>
pstack<T>::pstack(const pstack<T> &copy) :sp(0), size(0)
{
    *this = copy;
}

template <class T>
pstack<T>::~pstack()
{
    makeEmpty();
}

//push a value onto the stack
template <class T>
void pstack<T>::push(T storage)
{
    //create new data
    node *newnode = new node;

    newnode->next=sp;
    newnode->data=storage;
    sp = newnode;
    ++size;
}

//pop a value off of stack
template <class T>
void pstack<T>::pop(T & storage)
{
    if(isEmpty())
    {
        cerr << "\nError: accessing empty stack through method pstack::pop\n";
        exit(1);
    }

    //store data
    storage = sp->data;

    //delete node and move stack pointer back one
    node *temp = sp;
    sp = sp->next;
    delete temp;
    --size;
}

//pop a value off of stack and return it
template <class T>
const T pstack<T>::pop()
{
    T val;
    pop(val);
    return val;
}

template <class T>
const T pstack<T>::top() const
{
    if(isEmpty())
    {
        cerr << "\nError: accessing empty stack through method pstack::top\n";
        exit(1);
    }

    return sp->data;
}

template <class T>
int pstack<T>::length() const
{
    return size;
}

template <class T>
void pstack<T>::makeEmpty()
{
    node *temp;
    //iterate through list, deleting each elements
    while(sp!=0)
    {
        temp = sp;
        sp = sp->next;
        delete temp;
    }
    size = 0;
}

template <class T>
bool pstack<T>::isEmpty() const
{
    return size == 0;
}

template <class T>
const pstack<T> & pstack<T>::operator = (const pstack<T> & copy)
{
    makeEmpty();
    if(copy.isEmpty())
        return *this;

    sp = new node(copy.sp->data);
    node *newnode,*end = sp;
    for(newnode = copy.sp->next; newnode; newnode = newnode->next)
    {
        end->next = new node(newnode->data);
        end = end->next;
    }
    size = copy.size;
    return *this;
}
```

A.5 pqueue

```
template <class T>
class pqueue
{

public:
    pqueue();                               //default constructor
    pqueue(const pqueue<T> &);              //copy constructor
    virtual ~pqueue();                      //destructor
    void enqueue(const T &data);            //enqueue data to queue
    void dequeue(T &storage);               //storage holds the data that is dequeu
    const T dequeue();                      //dequeue to return value
    const T & front() const;                //returns top value without dequeueing

    void makeEmpty();                       //empty the queue

    inline bool isEmpty() const;            //returns true if queue is empty
    inline int length() const;              //find out the number of queued items

    const pqueue<T> & operator = (const pqueue<T> &);   //assignment operator

protected:
    struct node
    {
        T data;
        node *next;
        node() :next(0) {}
        node(const T &a) :next(0), data(a) {}
    } *head, *tail;
    int size;

};

template <class T>
pqueue<T>::pqueue() :head(0), tail(0), size(0)
{}

template <class T>
pqueue<T>::pqueue(const pqueue<T> &copy) :head(0), tail(0), size(0)
{
    *this = copy;
}

template <class T>
pqueue<T>::~pqueue()
{
    makeEmpty();
}

template <class T>
void pqueue<T>::enqueue(const T &data)
{
    node *newnode = new node(data); //end of queue
    if(size==0) //make sure queue exists
    {
        head = newnode;
        tail = head;
    }
    else
    {
        tail->next = newnode;
        tail = newnode;
    }
    ++size;
}

template <class T>
void pqueue<T>::dequeue(T &storage)
{
    if(head==0)
    {
        cerr << "\nError: accessing empty queue through method pqueue::dequeue\n";
        exit(1);
    }

    storage = head->data;               //save to var
    node *temp = head;                  //make a new node
    head = head->next;                  //iterate
    delete temp;                        //delete the node
    --size;                             //decrement size
}

template <class T>
const T pqueue<T>::dequeue()
{
    T val;              //store to temporary variable
    dequeue(val);       //dequeue
    return val;
}

template <class T>
const T & pqueue<T>::front() const
{
    if(head==0)
    {
        cerr << "\nError: accessing empty queue through method pqueue::front\n";
        exit(1);
    }
    return head->data;
```

```
}

template <class T>
void pqueue<T>::makeEmpty()
{
    node *temp;
    while(head)
    {
        temp = head;
        head = head->next;
        delete temp;
    }
    size = 0;
}

template <class T>
const pqueue<T> & pqueue<T>::operator = (const pqueue<T> & copy)
{
    makeEmpty();
    if(copy.isEmpty())
        return *this;

    head = new node(copy.head->data);
    tail = head;
    node *newnode;
    for(newnode = copy.head->next; newnode; newnode = newnode->next)
    {
        tail->next = new node(newnode->data);
        tail = tail->next;
    }
    size = copy.size;
    return *this;
}

template <class T>
int pqueue<T>::length() const
{
    return size;
}

template <class T>
bool pqueue<T>::isEmpty() const
{
    return size == 0;
}
```

Index