Introduction to Computer Programming

With C++

Spring 2014

Alexis Maciel
Department of Computer Science
Clarkson University

Contents

Preface

These notes are for a first course on computer programming. Before we start, it is useful to ask what exactly is computer programming and why is it important?

Computers and computer-related technology, usually called *information technology*, is everywhere in modern society, so much so that it is very easy to take it for granted. But most of this technology is at most a few decades old and its widespread use has had a profound impact in how people live their lives.

Consider the writing of formal documents, such as reports — and these notes. These used to be typewritten. Editing such documents was painful since it often required retyping many pages. Spell-checking was tedious since it involved looking up words one by one in a dictionary. And the documents didn't look very professional since most people could not afford the services of a professional typesetter and printer. Nowadays, everybody — not only businesses but individuals too — has access to typesetting programs such as OpenOffice Writer, Microsoft Word and LaTeX that are either free or relatively cheap. These programs make it easy to edit, spell-check and produce documents that can look as good as a professionally typeset book.

Consider access to information. Looking up information that wasn't in the home encyclopedia or in the day's newspaper typically required a trip to the local public library. And if that one was too small, a trip to the main city library was required — unless you lived in a small town and then you were out of luck. What this meant is that a lot of information was, for all practical purposes, inaccessible to most people. Nowadays, with the World Wide Web, most people have access to pretty much all the knowledge in the world, from their homes or anywhere on their smart phones, through programs called Web browsers. And search engines allow you to find what you need almost instantly.

Consider communication. People who didn't live near each other used to

communicate mainly by letter and, occasionally, by phone. Nowadays, with email, "letters" can be sent without the hassle of paper, envelopes and stamps, and without the delay of the postal system. Friends can keep in touch easily, and instantly, through programs such as facebook. And you can broadcast your thoughts to the entire world — or at least to anybody who cares — through your own blog.

We could go on and on. There's eCommerce, animation, games, simulators, expert systems such as IBM's Jeopardy-playing Watson, as well as business computing and scientific and engineering computation. And it's not over. (For example, see *Exponentials R Us: Seven Computer Science Game-Changers from the 2000s, and Seven More to Come* by Ed Lazowska [Laz].[1])

Most of these technological breakthroughs are made possible by three key ingredients: digital data, extensive communication networks, and automation.

Digital data, whether it is text, images, audio or video, is data that is represented as sequences of 0's and 1's. This makes it easier to store, edit and transmit the data. And this data can be easily transmitted and accessed because most people's computers and smart phones are connected together through vast and powerful *communication networks*. This is what enables the Internet and the World Wide Web. But none of this is possible without *automation*: the fact that computers can automatically perform complex tasks at incredible speeds.

The focus of this course is on automation. More precisely, a computer's *hardware*, its physical components, consists mainly of *memory* and a *processor* (which is often called a central processing unit or CPU). Memory holds data. A computer's memory also holds the instructions that tell the processor what to do. These instructions are arranged into *programs*, which are sequences of instructions designed to accomplish a particular task. A computer's programs are called the *software* of the computer.

As mentioned earlier, these notes are an introduction to computer programming, the writing of computer programs. Which means that we will focus on software. After reading these notes and working on the exercises, you will be able to create relatively simple programs. You will also be prepared to continue your study of computer science and software development, including the creation of much larger and more complex programs.

These notes use the programming language C++ but the main focus is not on the language itself (even though you will learn all the relevant features

[1] Link on the course web site.

of C++). The main focus is on programming: most of the concepts and techniques you will learn apply equally well to programming in any language.

Feedback on these notes is welcome. Please send comments to `alexis@clarkson.edu`.

Chapter 1

Basic Concepts

In this chapter, we will create our first program, a running pace calculator. In the process, we will learn how to make programs perform input and output as well as conditional execution. We will also be introduced to the important concept of a variable.

1.1 A Running Pace Calculator

Runners often like to know how fast they run but not in miles per hour. Instead, runners typically like to know their pace in minutes per mile. In addition, they like to express that pace in minutes and seconds, not in minutes and a fraction of a minute. For example, a running pace of 8 minutes and 15 seconds per mile would be expressed as 8:15 and not 8.25.

Calculating running paces like these is a bit tricky to do mentally. Therefore, in this chapter, we will create a calculator for this purpose. Figure 1.1 shows a sample session of the calculator. In this example, the user answered the questions `Distance?` and `Time?` with the numbers `3.5` and `0:28:45`. The rest of the text is produced by the program.

Note that this is only a first version of the calculator. Later in this chapter, we will add some other features to it — and learn the concepts necessary to program them.

```
Welcome to the running pace calculator.

Please enter distances in miles and times in the format
hours:minutes:seconds, as in 0:32:27.

Distance? 3.5
Time? 0:28:45

Pace: 8:13 minutes per mile.
```

Figure 1.1: A sample session of the running pace calculator

1.2 High-Level Languages and Compilers

As mentioned in the preface, a **program** is a sequence of instructions that tell the computer how to accomplish a particular task. These instructions are executed by the computer's processor. Each processor is designed to execute a specific set of instructions. This set of instructions constitutes the *language* of the computer and this type of programming language is called **machine language**.

Conceptually, a computer's processor turns out to be a surprisingly simple device. In particular, the instructions it can execute are very simple. It's the fact that processors can execute these instructions at extremely high speed that gives computers their amazing power. The fact that these instructions are very simple makes it very tedious for people to write programs directly in machine language. So computer scientists have designed **high-level languages**, such as C++, in which instructions correspond to more complex concepts. It is much easier to write a complex program in a high-level language than in machine language.

However, computers do not understand high-level languages. Therefore, programs written in a high-level language must be translated into machine language by a special program called a **compiler**. The compiler also verifies that the high-level program is written properly, according to the rules of the language.

Some additional terminology. Programming instructions are often referred to as **code**. Code written in a high-level language is often called **source code**. The machine language code produced by a compiler is called **object**

code. The machine language program produced by a compiler is often called an **executable**.

Study Questions

1.2.1. What is a computer program?

1.2.2. What is a machine language?

1.2.3. What is a high-level language?

1.2.4. What two tasks do compilers perform?

1.2.5. What is code, source code and object code?

1.2.6. What is an executable?

1.3 Output and Data

We now start writing our running pace calculator. We will do it very gradually and explain what we are doing as we go along.

Let's start by making the program display on the computer's screen the welcome sentence:

```
Welcome to the running pace calculator.
```

This can be accomplished as follows:

```
std::cout << "Welcome to the running pace calculator.";
```

The first element in this code, `std::cout` (usually read as *standard c-out*), is an **output stream**. An output stream is normally associated with an output device and data inserted into (or sent to) the stream will be sent to the output device, in the order that it is inserted into the stream. The output stream `std::cout` is called *standard output* and is normally associated with the computer's screen. Therefore, data sent to `std::cout` will appear on the screen.

In the above code, we are sending the **string**

```
"Welcome to the running pace calculator."
```

to `std::cout`. A string is simply a sequence of characters. The double quotes serve to mark the beginning and end of the string. Those quotes are not part of the string and will not appear on the screen.

The string is sent to `std::cout` by using the **output operator** <<, which is also called the **stream insertion operator**. Operators in C++ are like mathematical operators such as + and ×. They have operands and perform an operations on those operands. In C++, output operators are binary operators, meaning that they have two operands. The left operand is an output stream and the right operand is data:

```
output_stream << data;
```

The job of the operator is to send the data that's on the right to the stream that's on the left.

The output operator can output other types of data besides strings. One example is numbers such as `3`, `-5` and `25.5`. Another example is single characters such as `'a'`, `'B'` and `' '`. This last character is a blank space. Single characters are surrounded by single quotes, just as strings are surrounded by double quotes.

Note that it is possible to output more than one piece of data with a single output statement, as in

```
cout << 1 << 2 << 3;
```

Therefore, the general form of an output statement is as follows:

```
output_stream << data1 << data2 << ... ;
```

The data is sent to the stream in the order that it appears in the output statement, from left to right.

The last element in the code

```
std::cout << "Welcome to the running pace calculator.";
```

is a semicolon. In C++, most instructions are called **statements**. And most C++ code consists of sequences of statements. The semicolon is used to mark the end of each statement. As you will soon see, C++ programs typically contain lots of semicolons...

Returning to our program, after the welcome sentence, we need to print the instruction sentence. This can be done with a single output statement as follows:

```
#include <iostream>

int main()
{
    std::cout << "Welcome to the running pace calculator.\n\n"
              << "Please enter distances in miles and times in the "
              << "format\n"
              << "hours:minutes:seconds, as in 0:28:45.\n\n";

    return 0;
}
```

Figure 1.2: The beginning of the running pace calculator

```
std::cout << "Welcome to the running pace calculator.\n\n"
          << "Please enter distances in miles and times in the "
          << "format\n"
          << "hours:minutes:seconds, as in 0:28:45.\n\n";
```

The characters `\n` at the end of some of these strings. This is called an **escape sequence**. The escape sequence `\n` represents the *new line character*. This special character causes the program's output to move to the next line. Two new line characters in a row produce a blank line in the output. Other useful escapes sequences are `\"` and `\\`. The first one represents the double quotes character while the second one represents a backslash.

The first line of the instruction sentence was split into two strings. This was simply to avoid a very long line of code. Long lines of code don't fit within the margins of these notes but, more importantly, they make programs harder to read, just as very long lines of text would make a document hard to read.

Note that individual strings cannot extend beyond the end of the line. So the following code is not valid:

```
std::cout << "Please enter distances in miles and times in the
             format\n";
```

So we now know how to write code that will produce the welcome and instruction sentences. To be able to compile and run this code, we need to wrap it with some additional code, as shown in Figure 1.2. We will explain this additional code in the next section.

Study Questions

1.3.1. What is an output stream associated with?

1.3.2. What is the output (or stream insertion) operator used for?

1.3.3. In what order does the data sent to a stream appear on the output device?

1.3.4. What are most instructions called in C++?

1.3.5. What symbol is used to separate statements in C++?

1.3.6. What is a string?

1.3.7. Besides strings, what are two other types of data that can be printed by the output operator?

1.3.8. What character does the escape sequence `\n` represent?

Exercises

1.3.9. Create a program that prints your name and hometown, as in the following example:

```
Alexis Maciel
Potsdam, NY
```

1.4 Structure of a Simple C++ Program

We now explain the additional code that was included in Figure 1.2. First, `std::cout` is actually not part of the C++ language itself. It is part of a **library** called `iostream`. A library is a collection of prewritten code. This particular library is one of C++'s *standard libraries* and contains code that allows us to perform input and output. To be able to use this code, we must include the library into our program by using the *directive*

```
#include <iostream>
```

Second, C++ programs must contain at least one **function** called `main`. In our program, that function is represented by the code

```
int main()
{
    ...

    return 0;
}
```

Functions are an important concept that we will study in detail later. For now, all we will say is that every program must contain a `main` function and that our code must be placed within that function as indicated in Figure 1.2.

We end this section with a few additional comments. First, in C++ programs, *white space*, such as blank spaces and new lines, is only used to separate the various program elements. In particular, the amount of white space is not important. The particular layout of our program is only for the benefit of human readers. In principle, as far as the compiler is concerned, the code could all be written on a single line.[1]

Second, C++ code is often bundled into groups called **namespaces**. For example, the standard output stream `cout` belongs to the `std` (standard) namespace. Normally, to access a component of a namespace, you need to specify the namespace, as in `std::cout`. An alternative is to tell the compiler, once and for all, that we will be using `std::cout`. This is done with the *declaration*

```
using std::cout;
```

This declaration can be placed at the top of the file, right after the directive that includes the `iostream` library. This then makes it possible to simply write `cout` anywhere in the program, as shown in Figure 1.3.

Third, C++ programs usually include **comments** that describe the entire program or explain the purpose of a line or section of code. In the program shown in Figure 1.3, there are two comments, which begin with `//` and end at the end of the line. The first comment describes the entire program while the second one states the purpose of the following output statement. Comments are ignored by the compiler but can be very useful to human readers, including the person writing the program. In particular, every non-trivial line of code should be accompanied by a comment. (Of course, what is trivial for one person may not be for another...)

[1] In practice, this may not work because of internal compiler limitations.

```
// Running pace calculator.

#include <iostream>
using std::cout;

int main()
{
    // Print welcome and instructions.
    cout << "Welcome to the running pace calculator.\n\n"
         << "Please enter distances in miles and times in the "
         << "format\n"
         << "hours:minutes:seconds, as in 0:28:45.\n\n";

    return 0;
}
```

Figure 1.3: Running pace calculator with a `using` declaration and a comment

Finally, if the program of Figure 1.3 is executed on his own on a computer running the Windows operating system, the result will normally be that a *Command Prompt* window will flash on the monitor without letting the user see the output of the program. The simplest way to prevent this behavior is to include the following statement just before `return 0`:

```
std::system("pause");  // For Windows only.
```

This will cause the program to pause and wait until the user presses a key before closing the window. The `system` function is defined in the library `cstdlib` which must then be included in the program:

```
#include <cstdlib>  // For system.
```

Study Questions

1.4.1. What is a library?

1.4.2. What C++ standard library relates to input and output?

1.4.3. What is the purpose of the directive `#include <iostream>`?

1.4.4. What is a namespace?

1.4.5. What is the purpose of the declaration `using std::cout`?

1.4.6. Where does a comment begun with `//` terminate?

1.4.7. How can we prevent a Windows Command Prompt window from closing automatically when a program terminates?

1.5 Input and Variables

After the running pace calculator prints the welcome and instruction sentences, it needs to ask the user to enter a distance and time, as shown in Figure 1.1. Asking for the distance is easy

```
cout << "Distance? ";
```

The user will then enter a number. The program will eventually use that number to compute the running pace but, in the mean time, the program needs to *remember* that number. The easiest way to do that is for the program to write the number to the computer's memory. And the easiest way to do that is through the important concept of a **variable**.

A programming variable is a memory location to which a name has been associated. For example, the code

```
double distance;
```

allocates (or reserves) a memory location and associates the name `distance` with that location.

The above code is called a **variable declaration**. The general form of a simple variable declaration is

```
Type name;
```

In the above example, the type of `distance` is `double`, which is appropriate for real numbers. Other possible types are `int`, for integers, `char`, for individual characters, and `string`, for strings.

Once a variable is declared, it can be used to *store* data. For example, our running pace calculator can read the distance entered by the user and store that number in the variable `distance` as follows:

```
std::cin >> distance;
```

The first element of that statement, `std::cin` (usually read as *standard c-in*), is an **input stream**. An input stream is normally associated with an input device and data extracted (or read) from the stream will come from that device, in the order that it was entered on the device. The input stream `std::cin` is called *standard input* and is normally associated with the computer's keyboard. Therefore, data read from `std::cin` will come from the keyboard.

Data can be read from an input stream by using the **input operator** >>, which is also called the **stream extraction operator**. The left operand of the input operator is an input stream and the right operand is a variable:

```
input_stream >> variable;
```

The input operator causes the program to pause and wait for data to become available from the input device. The input operator then stores that data in the variable. The data coming from the device must be of the same type as the variable. (If it's not, a reading error occurs. We will learn how to detect and handle reading errors later in these notes.)

A single input statement can be used to read multiple values:

```
input_stream >> variable1 >> variable2 >> ... ;
```

The data is read from the stream in the order given by the variables, from left to right.

Note that `std::cin` is a *buffered* input stream. This means that the user must press the *Enter* key for any data typed on the keyboard to become available from `std::cin`. After the user presses *Enter*, the data is stored in a temporary memory location called a **buffer**. The input operator causes data to be extracted from that buffer.

For example, consider the following input statement:

```
std::cin >> x >> y >> z;
```

where the variables `x`, `y` and `z` are of type `double`. Now suppose that the user types `3.5 28` followed by the *Enter* key. The first input operator will cause the number `3.5` to be removed from the buffer and stored in the variable `x`. The second input operator will cause the number `28` to be removed from the buffer and stored in the variable `y`. The third input operator will cause the program to wait for the user to enter more data on the keyboard.

Figure 1.4 shows a version of the running pace calculator that reads a distance and time from the user. The instruction

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

int main()
{
    cout << "Welcome to the running pace calculator.\n\n"
         << "Please enter distances in miles and times in the "
         << "format\n"
         << "hours:minutes:seconds, as in 0:28:45.\n\n";

    cout << "Distance? ";
    double distance;
    cin >> distance;

    cout << "Time? ";
    int hours;
    cin >> hours;
    cin.get(); // colon
    int minutes;
    cin >> minutes;
    cin.get(); // colon
    int seconds;
    cin >> seconds;

    cout << distance << '\n'
         << hours << ':' << minutes << ':' << seconds << '\n';

    return 0;
}
```

Figure 1.4: Running pace calculator with reading of data from the user

```
cin.get();
```

causes a single character to be read from standard input. A comment is included in the program that indicates that this is to read the colons that occurs in the times.

The last output statement in the program of Figure 1.4 is not part of the running pace calculator. It was added to the program temporarily to allow us to test that the data was read and stored properly. If the user enters `3.5` and `0:28:45`, this will cause the program to print

```
3.5
0:28:45
```

Note how printing a variable causes the value of the variable, not its name, to be printed. This can be further illustrated by the fact that the statement

```
cout << "distance = " << distance << '\n';
```

would cause the program to print

```
distance = 3.5
```

Study Questions

1.5.1. What is a variable?

1.5.2. What C++ type is typically used for real numbers?

1.5.3. What is an input stream associated with?

1.5.4. What is the input (or stream extraction) operator used for?

1.5.5. When data is entered on the keyboard, where is it temporarily stored?

1.5.6. How can a single character be read from `cin`?

```
Please enter your birthday using numbers only.
Year: 1994
Month: 7
Day: 17

You were born on 7/17/1994.
```

Figure 1.5: Sample session for Exercise 1.5.8

Exercises

1.5.7. Create a program that asks the user for his or her age, as in the following example:

```
How old are you? 19
You are 19 years old.
```

The number `19` on the first line is typed by the user while the other text is produced by the program.

1.5.8. Create a program that asks the user for his or her birthday. The program should be behave as shown in Figure 1.5.

1.6 Arithmetic Operations

Now that our running pace calculator is able to read a distance and time from the user, the next step is to have the program calculate and print the running pace.

First, we convert the time into a total number of minutes:

```
double total_minutes =
    hours*60 + minutes + seconds/60.0;
```

This computes the total number of minutes that corresponds to the time and stores that value in a new variable called `total_minutes`.

The general form of the above variable declaration is

```
Type name = initial_value;
```

This creates a variable and sets its initial value. The initial value can be given by any expression that produces a value of the right type.

In our example, the initial value is given by an arithmetic expression. This expression contains three arithmetic operators: `*`, `+` and `/`. They correspond to multiplication, addition and division, respectively. Another common arithmetic operator is –, for subtraction.

Note that when used on two integers, the division operator performs *integer division*, meaning that the fractional part of the result is dropped. For example, the expression `45/60` evaluates to `0`, not `0.75`. This is why, in the above code, we divide the seconds by `60.0` instead of `60`. This turns one of the integers into a real number and causes the operator to perform real division.

The running pace can then be computed and printed as follows:

```
double pace = total_minutes/distance;
```

This pace will have a fractional part, as in 8.75. We must now convert this into minutes and seconds, as in 8:45.

This can be done as follows

```
int pace_minutes = pace;
int pace_seconds = std::round((pace - pace_minutes) * 60);
```

The first instruction initializes the integer variable `pace_minutes` to a `double`. This causes `pace_minutes` to be initialized to the integer part of the `double`. In other words, the fractional part of the `double` is lost. For example, if `pace` is 8.75, then `pace_minutes` is initialized to 8.

The second instruction computes the number of seconds that is represented by the fractional part of the pace. For example, if `pace` is 8.75 then the following computation will occur

$$(8.75 - 8) \cdot 60 = 0.75 \cdot 60 = 45$$

Note that the number of seconds is rounded to the nearest integer by using the function `std::round`. This function is defined in the library `cmath`.

We briefly mentioned the concept of a function in Section 1.4. All we said there is that `main` was a function and that every program needed to include a `main` function. Let's say a bit more now. A function is a block of code that performs a particular task. In this case, the function `round` takes a number and rounds it to the nearest integer. The number to be rounded is given to the

function as an argument, as in `round(x)`. We say that the function is *called* with argument `x`. Note that the function call `round(x)` is an expression that evaluates to the rounded value of `x`. For example, `round(34.72)` evaluates to 35. We also say that the function *returns* the rounded value of its argument. For example, `round(34.72)` returns 35.

All that is left to do now is to print the running pace:

```
cout << "\nPace: " << pace_minutes << ':' << pace_seconds
     << " minutes per mile.\n";
```

Figure 1.6 shows the complete first version of our running pace calculator.

Study Questions

1.6.1. What are the four basic arithmetic operators?

1.6.2. What happens when the division operator `/` is used on two integers?

Exercises

1.6.3. Create a program that calculates a person's age, as in the following example:

```
What year were you born in? 1995
You are 18 years old.
```

Assume that the current date is December 31, 2013.

1.6.4. Create a program that converts temperatures in degrees Celsius to degrees Fahrenheit. The program should be behave as follows:

```
Temperature in degrees Celsius? 20
The equivalent in degrees Fahrenheit is 68.
```

Use the following formula:

$$F = C \times 9/5 + 32$$

where F is the temperature in degrees Fahrenheit and C is the temperature in degrees Celsius.

```
#include <cmath>

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

int main()
{
    cout << "Welcome to the running pace calculator.\n\n"
         << "Please enter distances in miles and times in the "
         << "format\n"
         << "hours:minutes:seconds, as in 0:28:45.\n\n";

    cout << "Distance? ";
    double distance;
    cin >> distance;

    cout << "Time? ";
    int hours;
    cin >> hours;
    cin.get(); // colon
    int minutes;
    cin >> minutes;
    cin.get(); // colon
    int seconds;
    cin >> seconds;

    double total_minutes = hours*60 + minutes + seconds/60.0;

    double pace = total_minutes/distance;
    int pace_minutes = pace;
    int pace_seconds = std::round((pace - pace_minutes) * 60);

    cout << "\nPace: " << pace_minutes << ':' << pace_seconds
         << " minutes per mile.\n";

    return 0;
}
```

Figure 1.6: The first version of the running pace calculator

```
What is your income? 42000
How many dependents do you have? 2

Your income tax is 7700.
Your effective tax rate is 18.3%.
```

Figure 1.7: Sample session of the income tax program of Exercise 1.6.5

```
Please enter your birthday using numbers only.
Year: 1995
Month: 7
Day: 17

You are 18 years old.
```

Figure 1.8: Sample session of the age calculating program of Exercise 1.6.6

1.6.5. Create a program that computes the user's income tax. The program should be behave as shown in Figure 1.7. Use the following formula to calculate the tax:

$$T = (I - 10000 - D \times 5000) \times 35\%$$

where T is the tax, I is the income and D is the number of dependents. The effective tax rate is T/I.

1.6.6. Expand the age calculating program so it behaves as shown in Figure 1.8. Assume that the current date is January 22, 2014. *Hint*: Convert dates to a single number of days.

1.7 More on Variables

We have described a programming variable as a memory location to which a name has been associated. This corresponds closely with what happens during the execution of a program. Under this perspective, we say that a variable *holds* a value, and that a value is *stored* in a variable.

But there is another way of viewing variables: a variable can be viewed as being simply a pair that consists of a name and a value. This is a more abstract

view since it doesn't refer to the computer's memory. In this perspective, we say that a variable *has* a value and that a variable is *set* to a value.

Both perspectives are valid and useful. Some programmers prefer one over the other but most switch freely between the two. You should be comfortable with both.

Some of the variable names we used in the running pace calculator are `distance`, `hours`, `total_minutes`, `pace` and `pace_seconds`. Each of these names describes the value held by the variable. These names are descriptive without being too verbose. This makes the program easier to read and understand. And that's important because it makes it easier to work on the program and reduces the chances that we'll make mistakes.

Variable names must also follow certain rules. First, they must consist of a letter or underscore (`_`) followed by any number of letters, underscores and digits. Second, certain names are *reserved* in C++ and cannot be used as variable names. These include names such as `int`, `double`, `return` and `using`.

Note that all of the above variable names consist of lowercase letters with words separated by underscores. This is the style that we will use in these notes for most variable names. It will help us distinguish variables from other elements of our code. It's useful to choose a particular naming style and use it consistently.

In the previous section, when we declared the variables `total_minutes`, `pace`, `pace_minutes` and `pace_seconds`, we also initialized them right away. For example,

```
double pace = total_minutes/distance;
```

It turns out that it's a good idea to initialize every variable as soon as it is declared, so we don't forget to do it later. One reasonable exception is if the variable will be initialized in the statement that immediately follows the declaration. For example, this is what we did with `distance`:

```
double distance;
cin >> distance;
```

In C++, every variable must have a type. The type of a variable determines how much memory will be allocated for it, what operations can be performed on this variable, and how those operations will be performed. It is better for the type of a variable to correspond as precisely as possible to the type of value

```
Welcome to the running pace calculator.

Please enter distances as a number with units
(either mi or K). Enter times in the format
hours:minutes:seconds, as in 0:28:45.

Distance? 5K
Time? 0:28:45

Pace: 9:12 minutes per mile.
```

Figure 1.9: A sample session of the revised running pace calculator

that the variable will hold. For example, since the value of `hours` will be an integer, it is better for this variable to be of type `int` instead of `double`. This typically causes the variable to use less computer memory, it ensures that only appropriate operations are used on the variable, and that the operations used on the variable are performed correctly and efficiently.

Study Questions

1.7.1. Why should variable names be descriptive?

1.7.2. What rules must variable names follow?

1.7.3. Why is it better to initialize variables as soon as they are declared?

1.7.4. What are three advantages of the fact that every C++ variable must have a type?

1.8 Conditional Statements

In this section, we are going to expand the running pace calculator so the user can enter distances in either miles or kilometers, as shown in Figure 1.9.

First, when the program reads the distance, it will have to read a string in addition to a number. This can be done as follows:

```
string units;
cin >> units;
```

Note that the `string` data type is not built into the language. Instead, it is defined in the library `string` and included in the namespace `std`. Therefore, the above code will compile only if the following is included in our program:

```
#include <string>
using std::string;
```

Now, in our program, we already have code that computes the pace when the distance is given in miles. To be able to use that code even if the user enters the distance in kilometers, we would need to test to see if the user entered the distance in kilometers and, if that's the case, convert the distance from kilometers to miles. This can be done as follows:

```
if (units == "K") distance = distance / 8 * 5;
```

The above is an `if` statement, which is a particular type of **conditional statement**. The general form of an `if` statement is

```
if (condition) statement
```

The effect of an `if` statement is that the *`statement`* will be executed if the *`condition`* is true. On the other hand, if the *`condition`* is false, the *`statement`* will not be executed. The condition of an `if` statement is sometimes called the *test* of the `if` statement.

The condition `units == "K"` is an example of a **Boolean expression**, an expression that evaluates to either `true` or `false`. This particular Boolean expression evaluates to `true` if the value of `units` is equal to `"K"`.

Boolean expressions typically include *comparison operators* such as == (equal to). Other examples of operators are < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to) and != (not equal to). The == and != operators can be applied to strings as well as numbers and single characters. The inequality operators can of course be used with numbers but they also work with strings by using alphabetical order. The inequality operators should not be used with single characters.

In the above `if` statement, the statement

```
distance = distance / 8 * 5;
```

is an example of an **assignment statement**. The general form of an assignment statement is

```
variable = expression;
```

The equality symbol (=) represents the *assignment operator*. (Be careful not to confuse it with the equality testing operator ==.) When an assignment statement is executed, the expression on the right is evaluated and the resulting value is assigned to the variable on the left.

The expression on the right can be as simple as a value, as in `distance = 10`, or another variable, as in `distance = other_distance`. It can also be more complex. In our case, the expression refers to the variable itself and causes `distance` to be assigned its current value divided by 8 and multiplied by 5. For example, if the current value of `distance` is 4, then the expression on the right evaluates to 2.5 and this value becomes the new value of `distance`.

Now, if the user enters anything besides `K` as units, the program will simply assume that the user meant miles. Let's fix that. First, we will make the program also accept either `km` or `kms` for kilometers. This can be done by modifying the earlier test:

```
if (units == "K" || units == "km" || units == "kms")
    distance = distance / 8 * 5;
```

The symbol `||` represents the *logical OR* operator. This is an example of a **Boolean operator**, an operator that applies to Boolean expressions. In general, if *e1* and *e2* are Boolean expressions, then the expression

```
e1 || e2
```

evaluates to `true` if either *e1* or *e2* (or both) evaluate to `true`. In our case, the condition of the above `if` statement evaluates to `true` if `units` is equal to `K`, `km` or `kms`.

Two other examples of Boolean operators are the *logical AND* operator `&&` and the *logical NOT* operator `!`. If *e1* and *e2* are Boolean expressions, then

```
e1 && e2
```

evaluates to `true` if both *e1* and *e2* evaluate to `true`. If *e* is a Boolean expression, then

```
!e
```

evaluates to `true` if *e* is `false`. Note that Boolean operators are sometimes called **logical operators**.

Just like arithmetic operators, Boolean and comparison operators are subject to precedence rules. Table 1.1 lists the rules that apply to all to the arithmetic, Boolean and comparison operators we have seen so far in these notes. For example, the table indicates that == has higher precedence than `||`. This implies that the Boolean expression

```
units == "K" || units == "km" || units == "kms"
```

will be interpreted as

```
(units == "K") || (units == "km") || (units == "kms")
```

which is what we want.

The associativity properties given in Table 1.1 specify what happens when several operators with the same precedence occur one after the other. For example, the fact that addition and subtraction have left-to-right associativity implies that the expression $8 - 5 + 3$ will be interpreted as $(8 - 5) + 3$ and not as $8 - (5 + 3)$. Note that these last two expressions have different values.

Parentheses can always be used to override the normal precedence rules. But note that parentheses can also be used to make complicated expressions easier to understand.

Now, in our running pace calculator, it is still true that if the user enters as units anything besides `K`, `km` or `kms`, the distance will be assumed to be in miles. We can fix that by adding another test to the program right after the reading of the units, as shown in Figure 1.10. If the units are not equal to one of `mi`, `miles`, `mile`, `K`, `km` or `kms`, then the program prints an error message. Otherwise, the program asks the user for a time and computes the running pace.

This code uses a more general kind of `if` statement that can be called an `if-else` statement The general form of an `if-else` statement is

```
if (condition)
    statement
else
    statement
```

Precedence	**Operator**	**Description**	**Associativity**
1	!	Logical NOT	Right-to-left
	−	Unary minus	
2	*	Multiplication	Left-to-right
	/	Division	
3	+	Addition	Left-to-right
	−	Subtraction	
4	<	Less than	Left-to-right
	<=	Less than or equal	
	>	Greater than	
	>=	Greater than or equal	
5	==	Equal	Left-to-right
	!=	Not equal	
6	&&	Logical AND	Left-to-right
7	||	Logical OR	Left-to-right

Table 1.1: Precedence rules

```
if (units != "mi" && units != "miles" && units != "mile" &&
    units != "K" && units != "km" && units != "kms") {
    cout << "The units \"" << units << "\" are not recognized.\n";
} else {
    cout << "Time? ";
    int hours;
    cin >> hours;
    cin.get(); // colon
    int minutes;
    cin >> minutes;
    cin.get(); // colon
    int seconds;
    cin >> seconds;

    if (units == "K" || units == "km" || units == "kms")
        distance = distance / 8 * 5;

    double total_minutes = hours*60 + minutes + seconds/60.0;

    double pace = total_minutes/distance;
    int pace_minutes = pace;
    int pace_seconds = std::round((pace - pace_minutes) * 60);

    cout << "\nPace: " << pace_minutes << ':';
    if (pace_seconds < 10) cout << '0';
    cout << pace_seconds << " minutes per mile.\n";
}
```

Figure 1.10: Testing for valid units

If the *condition* is `true`, then the first *statement* is executed; otherwise, the second *statement* is executed. The first statement is called the `if` *branch* of the `if-else` statement. The second statement is called the `else` branch.

In the code of Figure 1.10, the `else` branch consists of more than one statement. This is possible as long as those statements are enclosed within braces. In other words, an `if-else` statement can also have the following form:

```
if (condition) {
    statements
} else {
    statements
}
```

Simple `if` statements can also involve multiple statements:

```
if (condition) {
    statements
}
```

(A sequence of statements enclosed within braces is often called a **compound statement**.)

Note that if braces are used for one branch of an `if-else` statement, it is a good idea to also use them for the other branch, even if that branch consists of a single statement. This is what we did in Figure 1.10. It is also a good idea to use braces if a branch consists of a single statement that spans more than one line.

At the bottom of the code shown in Figure 1.10, in the printing of the running pace, we added a new `if` statement. Its purpose is to ensure that a number of seconds with a single digit is printed with a leading 0. Otherwise, a running pace of 7:01 would appear as `7:1`. This fixes an error in our program, one that we didn't catch earlier.

Programming errors, or **bugs**, as they are as often called, can be difficult to avoid and difficult to find. (What would make us think that single-digit running paces would be a problem?) Which is why it is important to test programs extensively. Since even the best programmers make mistakes, it is important for a good programmer to also be a good tester and a good debugger.

It is fairly common for the `else` branch of an `if-else` statement to consist of a single `if` statement. For example, suppose that someone's age is stored

```
if (age < 18) {
    cout << "minor";
} else {
    if (age < 65)
        cout << "adult";
    else
        cout << "senior";
}

if (age < 18)
    cout << "minor";
else if (age < 65)
    cout << "adult";
else
    cout << "senior";
```

Figure 1.11: A multipart `if` statement

in a variable called `age` and that we need to print either `minor`, `adult` or `senior`, depending on the age. This could be done as shown in Figure 1.11. That code is shown twice, using two different styles. The second style is the one that is more commonly used. It is more compact and avoids excessive indentation.

Study Questions

1.8.1. What is the general form of an `if` statement?

1.8.2. What is the general form of an `if-else` statement?

1.8.3. What is the general form of an assignment statement?

1.8.4. What is a Boolean expression?

1.8.5. What are six examples of comparison operators?

1.8.6. What are three examples of Boolean (or logical) operators?

1.8.7. What is a compound statement?

1.8.8. What is a bug?

Degrees Celsius	Interpretation
38 and above	Very hot
28 to 37	Hot
18 to 27	Comfortable
8 to 17	Cool
-12 to 7	Cold
Below -12	Very cold

Table 1.2: Interpretation of temperatures in degrees Celsius

Exercises

1.8.9. Create a program that interprets outside temperatures in degrees Celsius as follows:

```
Temperature in degrees Celsius? 20
Interpretation: comfortable.
```

Interpret temperatures according to Table 1.2.

1.8.10. Expand the Celsius to Fahrenheit conversion program of Exercise 1.6.4 so that the user can convert temperatures in both directions, as shown in Figure 1.12.

1.8.11. Modify the tax income program of Exercise 1.6.5 to include tax brackets. Define *taxable income* as

$$I - 10000 - D \times 5000$$

where I is the income and D is the number of dependents. Then compute the tax by using Table 1.3.

1.8.12. Redo Exercise 1.6.6 but this time use `if` statements.

```
Please choose:
  1. Celsius to Fahrenheit
  2. Fahrenheit to Celsius

Choice? (1 or 2) 1

Temperature in degrees Celsius? 20
The equivalent in degrees Fahrenheit is 68.
```

Figure 1.12: Sample session of the expanded temperature conversion program of Exercise 1.8.10

Taxable income	**Tax**
100,000 and above	25,000 plus 40% of taxable income above 100,000
50,000 to 100,000	10,000 plus 30% of taxable income above 50,000
50,000 or less	20% of taxable income

Table 1.3: Income tax

Chapter 2

Repetition

In this chapter, we will expand the running pace calculator by allowing the user to perform more than one calculation in a single session. In the process, we will learn how to program repetition.

2.1 Adding Repetition to the Running Pace Calculator

The running pace calculator we created in the previous chapter only allows the user to calculate one running pace. To perform another calculation, the user must restart the program. In this chapter, we will modify the running pace calculator so the user calculate more than one running pace in a single session, as shown in Figure 2.1.

To achieve this, the program needs to repeat the code that asks the user for a height and weight and performs a single running pace calculation. Since we don't know how many calculations the user will want to do, we can't simply repeat the code itself a certain number of times. What we need is a mechanism that allows the *execution* of this code (not the code itself) to be repeated. In most programming languages, this is accomplished through a construct called a **loop**.

C++ provides several types of loops. One is the `do-while` loop, whose general form is as follows:

```
do statement while (condition);
```

```
Welcome to the running pace calculator.

Please enter distances as a number with units
(either mi or K). Enter times in the format
hours:minutes:seconds, as in 0:28:45.

Distance? 5K
Time? 0:28:45

Pace: 9:12 minutes per mile.

Another calculation? (y or n) y

Distance?
```

Figure 2.1: A sample session of the revised running pace calculator

The statement is called the *body* of the loop. As in the case of `if` statements, the condition is a Boolean expression that's often called the *test* of the loop. When a `do-while` loop is executed, the body of the loop will be repeated as long as the condition is `true`.

As in the case of `if` statements, the body of a loop can be a compound statement:

```
do {
    statements
} while (condition);
```

Figure 2.2 shows how a `do-while` loop can be added to our running pace calculator. Most of the body of the loop consists of the code that was already present in the running pace calculator. That's the code that asks the user for a distance, checks if the units are valid, asks the user for a time and then computes and prints the running pace. (The reading of the time and the calculation of the pace are replaced by three dots to allow the entire loop to fit in one page.) But at the end of both branches of the `if` statement, the programs asks the user if another calculation should be performed, or if the user wants to try entering the distance again. The user's answer is stored in the variable `answer`. The test used as a condition for the loop is

```
string answer;
    // To the questions "Another calculation?" or "Try again?"
do {
    cout << "Distance? ";
    double distance;
    cin >> distance;
    string units;
    cin >> units;

    if (units != "mi" && units != "miles" && units != "mile" &&
        units != "K" && units != "km") {
        cout << "The units \"" << units << "\" are not
        recognized.\n";

        cout << "\nTry again? (y or n) ";
    } else {
        cout << "Time? ";
        ...

        cout << "\nPace: " << pace_minutes << ':';
        if (pace_seconds < 10) cout << '0';
        cout << pace_seconds << " minutes per mile.\n";

        cout << "\nAnother calculation? (y or n) ";
    }

    cin >> answer;
    cout << '\n';
} while (answer == "y" || answer == "yes");
```

Figure 2.2: Running pace calculator with a loop

```
answer == "y" || answer == "yes"
```

That is, the loop is repeated as long as the user says yes.

Note that the variable `answer` is declared *before* the beginning of the loop. An alternative would be to declare the variable inside the loop, immediately before the variable is used. This is what we do with all the other variables: we declare them just before we use them. However, this would not work here. The reason is that a variable declared inside the body of a loop exists only within the body of that loop. Such a variable is said to be **local** to the body of the loop, and the body of the loop is called the **scope** of that variable. This implies that a variable declared inside the body of a loop cannot be used in the Boolean expression that serves as the condition of that loop. Therefore, we declared `answer` before the loop because we need to test that variable in the condition of the loop.

Note that a similar observation applies to `if` statements: a variable declared within a branch of an `if` statement is local to that branch and cannot be used in the other branch of the `if` statement or outside the `if` statement.

Study Questions

2.1.1. What is the general form of a `do-while` loop?

2.1.2. What does it mean to say that a variable declared inside a loop is local to that loop?

2.1.3. What is the scope of a variable?

Exercises

2.1.4. Expand the Celsius to Fahrenheit conversion program of Exercise 1.8.10 so that the user can convert more than one temperature, as shown in Figure 2.3.

2.1.5. Modify the program of the previous exercise so it behaves as shown in Figure 2.4.

```
Please choose:
  1. Celsius to Fahrenheit
  2. Fahrenheit to Celsius

Choice? (1 or 2) 1

Temperature in degrees Celsius? 20
The equivalent in degrees Fahrenheit is 68.

More? (y or n) y

Please choose:
  1. Celsius to Fahrenheit
  2. Fahrenheit to Celsius

...

More? (y or n) n

Goodbye!
```

Figure 2.3: Sample session for Exercise 2.1.4

```
Please choose:
  1. Celsius to Fahrenheit
  2. Fahrenheit to Celsius
  3. Quit

Choice? 1

Temperature in degrees Celsius? 20
The equivalent in degrees Fahrenheit is 68.

Please choose:
  1. Celsius to Fahrenheit
  2. Fahrenheit to Celsius
  3. Quit

Choice? 3

Goodbye!
```

Figure 2.4: Sample session for Exercise 2.1.5

```
int count = 0;
do {
    cout << '-';
    ++count;
    // count equals the number of dashes that have been printed.
} while (count < n);
```

Figure 2.5: Printing a line of dashes with a `do-while` loop

2.2 More on Loops

In the previous section, we added a `do-while` loop to the running pace calculator. The purpose was to allow the user to calculate multiple running paces in one session. In this section, we explore other uses of loops as well as other forms of loops.

Suppose that `n` is an integer variable and that, for some reason, we need to print a line consisting of `n` dashes. Figure 2.5 shows one way to do this. The loop uses a variable to count the number of dashes that are printed. The counter is initialized to 0 and incremented by 1 every time a dash is printed. While the count is less than `n`, the loop continues repeating. Once the count reaches `n`, the loop stops.

The value of `count` is incremented by 1 by using the *increment* operator `++`. This is a *unary* operator whose operand must be a variable. The effect of the operator is simply to add 1 to the variable.[1]

Note that we could have incremented `count` with the following assignment statement:

```
count = count + 1;
```

[1]The increment operator has two forms: *prefix*, as in `++i`, and *postfix*, as in `i++`. When used on their own, as a statement and not as part of an expression, the two forms of the operator perform the same function but the prefix version runs a little faster. In these notes, we will only use the prefix version. When used in an expression, the difference between the two forms is in the value they evaluate to. The prefix form evaluates to the new value of the variable while the postfix form evaluates to the old value of the variable, as if the expression was evaluated before the variable was incremented. For example, if `i` is 4, then `++i` evaluates to 5 while `i++` evaluates to 4. In both cases, the value of `i` is changed to 5. Code that takes advantage of this difference between the two versions of the `++` operator tends to be harder to understand. In these notes, we will avoid this and never use the operator in expressions.

```
int count = 0;
while (count < n) {
    // count is the number of dashes that have been printed.
    cout << '-';
    ++count;
}
```

Figure 2.6: Printing a line of dashes with a `while` loop

Recall that when an assignment statement is executed, the expression on the right is evaluated and its value becomes the new value of the variable on the left. Therefore, if `count` is 3, then the expression `count + 1` evaluates to 4 and the value of `count` becomes 4.

C++ also has a *decrement* operator (--) that subtracts 1 from a variable. For example, --`count` has the same effect as `count = count - 1`.

The loop of Figure 2.5 correctly prints `n` dashes as long as the value of `n` is positive. For example, if `n` is 0, then a dash will be printed, `count` will be increased to 1, the test `count < n` will fail (because 1 is not less than 0) and the loop will stop. So if `n` is 0, the loop will incorrectly print one dash.

To fix this and have the loop print nothing if `n` is 0, we need to test `count` *before* the body of the loop is executed for the first time. This can be achieve with another form of loop called a `while` loop:

```
while (condition) statement
```

As in the case of a `do-while` loop, when a `while` loop is executed, the body of the loop will be repeated as long as the condition is `true`. The only difference is that the test is performed before the first iteration of the loop, making it possible for the body of the loop not to be executed at all. In contrast, the body of a `do-while` loop is always executed at least once.

Figure 2.6 shows a `while` loop that correctly prints `n` dashes even if `n` is 0. (In case `n` is negative, nothing will be printed, which seems reasonable.)

It is very common to write loops that must count in order to perform some action some number of times. For these situations, there is another type of loop in C++ that's usually more convenient. But understanding this other type of loop requires a change of perspective.

When the `while` loop of Figure 2.6 is executed, it performs an action (the printing of a dash) for each value of `count` in the sequence $0, 1, 2, \ldots, n-1$. So we can view the job of this loop as follows:

```
for (int i = 1; i <= n; ++i)
    cout << '-';
```

Figure 2.7: Printing a line of dashes with a `for` loop

For each value $0, 1, 2, \ldots, n-1$*, print a dash.*

This produces the same result but without the need to explicitly refer to counting.

The idea of repeating some action for each value in some sequence can be programmed in C++ by using a `for` loop. The general form of a `for` loop is as follows:

```
for (initialization; condition; update) statement
```

The initialization and update are statements. The condition is, as usual, a Boolean expression.

When a `for` loop is executed, the initialization action is performed and then the body of the loop is repeated as long as the condition is `true`. In addition, the update action is performed after each execution of the body, before the condition is reevaluated. In fact, a `for` loop is equivalent to the following `while` loop:

```
initialization
while (condition) {
    statement
    update
}
```

We said that `for` loops are often used for repeating some action for each value in a sequence. This is done as follows. The initialization part of the `for` loop is used to set a variable to the initial value of the sequence. The condition states that the variable has not gone beyond the end of the sequence. The update statement takes the variable to the next value in the sequence.

For example, Figure 2.7 shows how `n` dashes can be printed with a `for` loop. This prints a dash for each value in the sequence $1, 2, \ldots, n$. The result is that `n` dashes are printed.

The code of Figure 2.7 is somewhat shorter than the previous versions. But, more importantly, it's also simpler because it does not require the explicit

```
for (int i = 1; i < n; i = i + 1)
    cout << i << ", ";
cout << n;
```

Figure 2.8: Printing 1 through `n`

use of a counter. And this is important because simpler code is easier to understand and easier to write correctly.

As a further example of the usefulness of `for` loops, suppose that `n` is once again an integer variable and that we need to print the integers 1 through `n`, on a single line, separated by commas. For example,

```
1, 2, 3, 4, 5
```

In other words, we want to do the following:

> *For each value* $1, 2, \ldots, n$*, print that value.*

This task can be carried out by the simple `for` loop of Figure 2.8. The fact that no comma should be printed after the last number introduces a small complication. In fact, this code can be more accurately described as follows:

> *For each value* $1, 2, \ldots, n-1$*, print that value followed by a comma. Then print* n*.*

Note that when the loop stops, the variable `i` has the value `n`. But we could not have replaced the statement `cout << n` by `cout << i`. The reason is that any variable declared in the initialization of a `for` loop is local to that loop. It can be used in the condition, update and body of the loop, but not after the loop.

Study Questions

2.2.1. What is the general form of a `while` loop?

2.2.2. What is the difference between a `while` loop and a `do-while` loop?

2.2.3. What is the general form of a `for` loop?

2.2.4. What C++ loop is the most convenient for repeating an action for each value in some sequence?

2.2.5. What is the effect of `++i`?

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

Figure 2.9: Computing the sum of 1 through `n`

Exercises

2.2.6. Suppose that `n` is an integer variable. Write code that prints the first `n` even, positive integers, on a single line, separated by commas. For example, if `n` is 5, your code should print `2, 4, 6, 8, 10`.

2.2.7. Suppose that `n` is an integer variable. Write code that prints the first `n` odd, positive integers, on a single line, separated by commas. For example, if `n` is 5, your code should print `1, 3, 5, 7, 9`.

2.2.8. Suppose that `n` is an integer variable. Write code that prints the first `n` positive squares, on a single line, separated by commas. For example, if `n` is 5, your code should print `1, 4, 9, 16, 25`.

2.3 Loops and Variables

In the previous section, we used variables to control the execution of our loops. We used the variable `count` as a counter in the `while` loops that printed `n` dashes. We also had the variable `i` take on the values 1 through n, or 1 through $n-1$, in the `for` loops. In this section, we will learn that variables can be used in other ways in conjunction with loops.

For example, suppose that `n` is an integer variable and that we want to compute the sum of the first n positive integers: $1 + 2 + \cdots + n$. There is a well-known formula for this sum:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

But pretend that we don't know this formula. Or that we want to verify it. We can compute the sum as follows: start with the sum initialized to 0 and then add each value $1, 2, \ldots, n$ to the sum. This can be easily implemented by the loop shown in Figure 2.9.

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

Figure 2.10: Computing `n`!

In this loop, the variable `i` is again used to go through the sequence of values 1 through n. But the variable `sum` is used for a new purpose: it is used as an *accumulator*.

Here's another example. Suppose once again that `n` is an integer variable and that we want to compute the product of the first n positive integers: $1 \cdot 2 \cdots n$. This product is called the *factorial* of n and denoted $n!$.

There is a formula that gives a good approximation of $n!$:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

where $e \approx 2.71828$ is the base of the natural logarithm. (Don't worry if you don't know what this is.) This approximation is called *Stirling's approximation*. However, there is no formula that gives the exact value of $n!$. Fortunately, it is not hard to compute $n!$: start with a product initialized to 1 and then multiply this product by each of the values $1, 2, \ldots, n$. Figure 2.10 shows how this can be easily coded with a `for` loop. Note that this loop skips the multiplication of `factorial` by 1 since this has no effect.

In both of these loops, we had to update the value of a variable. In the first loop, we added `i` to `sum` while in the second loop, we multiplied `factorial` by `i`. C++ provides convenient operators for such operations. For example, the *add and assign* operator (`+=`) is a binary operator that adds the value of the expression on the right to the variable on the left, as in `sum += i`. Three other similar operators are *subtract and assign* (`-=`), *multiply and assign* (`*=`) and *divide and assign* (`/=`). For example, `factorial *= i` is equivalent to `factorial = factorial * i`.

Study Questions

2.3.1. What is the effect of `x += y`?

```
Please enter the various incomes of your household.
Income 1: 42000

Another income? (y or n) y
Income 2: 28000

Another income? (y or n) n

How many adults in your household? 2
How many dependents in your household? 2

Your total income is 70000.
Your taxable income is 40000.
Your income tax is 8000.
Your effective tax rate is 11.4%.
```

Figure 2.11: Sample session of the income tax program of Exercise 2.3.3

Exercises

2.3.2. Verify that the formula for the sum of the first n positive integers is correct. Do this by modifying the loop of Figure 2.9 as follows: for each i, in addition to adding `i` to `sum`, print the value of `sum` (which, at this point, equals $1 + 2 + \cdots + i$) as well as the value $i(i+1)/2$.

2.3.3. Modify the income tax program of Exercise 1.8.11 to allow the user to enter multiple incomes for a single household. The program should be behave as shown in Figure 2.11. Define *taxable income* as

$$I - A \times 10000 - D \times 5000$$

where I is the income, A is the number of adults and D is the number of dependents. Compute the tax by using Table 1.3.

2.4 Nested Loops

In an earlier section, we learned how to write a loop that prints a line consisting of `n` dashes. Now suppose that we need to print a box filled with stars ($*$). The

```
for (int i = 1; i <= height; ++i) {
    for (int j = 1; j <= width; ++j)
        cout << '*';
    cout << '\n';
}
```

Figure 2.12: Printing a box filled with stars

dimensions of the box are given by the integer variables `width` and `height`. For example, here's a box of width 5 and height 3:

```
*****
*****
*****
```

We know how to print a line consisting of `width` stars. All we need to do is repeat that `height` times. The code of Figure 2.12 achieves this.

Because the second loop is contained within the body of the first loop, we say that the second loop is *nested* within the first one. We also say that the second loop is the *inner* loop and that the first loop is the *outer* loop.

Note that we are using different variables for controlling the two loops. This is not absolutely necessary here but it makes the code clearer. Viewing the box as a table, we are using `i` for row numbers and `j` for column numbers. (If we had used `i` for both loops, within the inner loop, the `i` variable of the inner loop would have hidden the `i` variable of the outer loop.)

Now suppose that we want the inside of the box to be empty. In other words, we want the box to have a border of stars:

```
*****
*   *
*****
```

This requires the top and bottom rows to be different from the middle rows. The code of Figure 2.13 achieves this. Note how the initialization and test of the middle loop ensure that `i` goes from 2 to `height` $-$ 1.

Here's one more example of a nested loop. Suppose that we want to print the multiplication table shown in Figure 2.14. This can be easily done by the code shown in Figure 2.15.

```
// Print first row.
for (int j = 1; j <= width; ++j)
    cout << '*';
cout << '\n';

// Print middle rows.
for (int i = 2; i < height; ++i) {
    cout << '*';
    for (int j = 2; j < width; ++j)
        cout << ' ';
    cout << "*\n";
}

// Print last row.
for (int j = 1; j <= width; ++j)
    cout << '*';
cout << '\n';
```

Figure 2.13: Printing a box with a border of stars

```
 1   2   3   4   5   6   7   8   9  10  11  12
 2   4   6   8  10  12  14  16  18  20  22  24
 3   6   9  12  15  18  21  24  27  30  33  36
 4   8  12  16  20  24  28  32  36  40  44  48
 5  10  15  20  25  30  35  40  45  50  55  60
 6  12  18  24  30  36  42  48  54  60  66  72
 7  14  21  28  35  42  49  56  63  70  77  84
 8  16  24  32  40  48  56  64  72  80  88  96
 9  18  27  36  45  54  63  72  81  90  99 108
10  20  30  40  50  60  70  80  90 100 110 120
11  22  33  44  55  66  77  88  99 110 121 132
12  24  36  48  60  72  84  96 108 120 132 144
```

Figure 2.14: A multiplication table

```
for (int x = 1; x <= 12; ++x) {
    for (int y = 1; y <= 12; ++y)
        cout << std::setw(5) << x * y;
    cout << '\n';
}
```

Figure 2.15: Printing the multiplication table of Figure 2.14

In this code, `setw` is a **stream manipulator**. A stream manipulator is sent to an output stream, just like a piece of data. But instead of causing a value to be printed, a steam manipulator affects the behavior of the stream.

For example, `setw(5)` causes each number to be printed in a *field* that's five characters wide. Blank spaces are added as needed to fill the empty space. For example, `72` is printed preceded by 3 spaces.

By default, numbers are printed to the right of their respective fields. The stream manipulator `left` can change this (and `right` can bring it back to the right).

Note that `setw` only applies to the next value that's printed. On the other hand, the effect of `left` and `right` lasts until another manipulator changes it.

Standard stream manipulators such as `setw`, `left` and `right` are defined in the library `iomanip` and are part of the namespace `std`.

Study Questions

2.4.1. What is a nested loop?

2.4.2. What is the effect of the stream manipulators `setw`, `left` and `right`?

Exercises

2.4.3. Suppose that `n` is an integer variable. Write code that prints a lower left triangle of dimension `n`. Here's an example when `n` is 3:

```
*
**
***
```

2.4.4. Repeat the previous exercise for an upper right triangle:

```
***
 **
  *
```

2.4.5. Repeat again but this time print a diamond:

```
  *
 ***
*****
 ***
  *
```

Assume that `n` is odd.

Chapter 3

File Input and Output

In this chapter, we will create a pay calculator. We will learn how to read data from files and how to write data to files. We will also learn a bit more about string variables and we will be introduced to error checking.

3.1 A Pay Calculator

In most businesses, every week or two, someone finds out how many hours each employee has worked and then figures how much each employee should be paid. In a large company, this is a tedious and time-consuming task. But it's also a fairly routine task and, therefore, a perfect candidate for automation.

In this chapter, we will create a pay calculator that reads a file containing one line per employee. On each line contains an employee number followed by the number of hours that employee has worked. The program will produce another file that also contains one line per employee. Each line in that file contains an employee number, the number of hours that employee as worked as well as the amount that employee should be paid.

For example, suppose hat the program reads the input file shown in Figure 3.1. Then, assuming that every employee is paid $20 per hour, the program would produce an output file similar to that shown in Figure 3.2.

3.2 File Streams

The main new thing we will learn in the creation of the pay calculator is how to read from files and how to write to files. But this will turn out not to be

```
12 43
23 37.5
37 40.33
```

Figure 3.1: A sample input file

```
12 43 860
23 37.5 750
37 40.33 806.6
```

Figure 3.2: A sample output file

very difficult.

Let's start with file output. We already know how to display data on the computer's screen by sending the data to `cout`. This output stream is already defined for us, in the standard library `iostream`.

In contrast, to write data to a file, we need to create our own output stream and associate it with the file. This can be done as follows:

```
ofstream ofs_pay("pay.txt");
```

This declaration creates an output stream `ofs_pay`, which is a value of type `ofstream` (for *output file stream*). The declaration also associates the stream with the file `pay.txt` and *opens* the file so the program can write to it. If the file does not exist, it is created. If the file already exists, it is overwritten. (We may learn later how to avoid this.)

The general form of the above declaration is

```
ofstream stream_name(file_name);
```

The stream name must be a valid variable name. The file name can be any string of characters, although some operating systems put some restrictions, such as not allowing blank spaces.

The type `ofstream` is defined in the standard library `fstream`. This library must be included in any program that uses `ofstream`. This type is also part of the namespace `std`, so it must be referenced as `std::ofstream`, unless the program includes the directive `using std::ofstream`.

In the above example, the name of the stream begins with the prefix `ofs`. This is to indicate that this variable is an output file stream. This is not required but it makes the code easier to understand.

```
int main() {
    std::ifstream ifs_hours("hours.txt");
    std::ofstream ofs_pay("pay.txt");

    int employee_number;
    ifs_hours >> employee_number;

    double num_hours;
    ifs_hours >> num_hours;

    double pay = num_hours * 20;
    ofs_pay << employee_number << ' ' << num_hours << ' ' << pay
            << '\n';

    return 0;
}
```

Figure 3.3: A first version of the pay calculator

Let's now turn to file input. Just as writing to a file is done through an output file stream, reading from a file is done through an *input file stream*, a value of type `ifstream`. For example, the declaration

```
ifstream ifs_hours("hours.txt");
```

creates the input stream `ifs_hours`, associates it with the file `hours.txt` and opens the file for reading. If the file does not exist, an error occurs and every subsequent attempt to read from the stream fails. (We will learn how to detect this type of error later in this chapter.)

The general form of the above declaration is

```
ifstream stream_name(file_name);
```

The type `ifstream` is also defined in the standard library `fstream` and is also part of the `std` namespace. Note the use of the prefix `ifs` in the above example.

Figure 3.3 shows a very preliminary first version of the pay calculator. This version computes the pay of only the first employee in the input file. This version also assumes that the input file is always called `hours.txt` and that the output file should always be called `pay.txt`

```
Enter employee number 0 when done.

Employee number: 12
Hours: 43

Employee number: 23
Hours: 37.5

Employee number: 0
```

Figure 3.4: A sample session of the program that creates `hours.txt`

Study Questions

3.2.1. What is the general form of an output file stream declaration?

3.2.2. When an output file stream is created and a file is opened, what happens if the file does not exist?

3.2.3. When an output file stream is declared as we saw in this section, if the file already exists, what happens to its original contents?

3.2.4. What is the general form of an input file stream declaration?

3.2.5. When an input file stream is created and a file is opened, what happens if the file does not exist?

3.2.6. In what library are the types `ofstream` and `ifstream` defined?

Exercises

3.2.7. Create a program that allows the user to generate the file `hours.txt` read by our pay calculator. Refer to Figure 3.1 for the format of the file. Your program should interact with the user as shown in Figure 3.4.

3.2.8. Modify the income tax program of Exercise 1.8.11 so that in addition to interacting with the user as shown in Figure 1.7, the program also writes the data to a file called `tax_return.txt`. That file should begin with the income followed by the number of dependents, the tax and the effective tax rate, as shown in Figure 3.5.

```
42000
2
4400
10.5
```

Figure 3.5: Sample tax return file for the program of Exercise 3.2.8

```
Income: $42000
Number of dependents: 2

Income tax: $4400
Effective tax rate: 10.5%
```

Figure 3.6: Sample output for the program of Exercise 3.2.9

3.2.9. Create a program that reads the tax return file produced by the program of the previous exercise and displays the tax return on the computer's screen, as shown in Figure 3.6.

3.3 Detecting the End of the File

In this section, we expand the program of the previous previous section so it computes the pay of all the employees in the input file, not just the first one. This requires a way of detecting when we have read all the employees that are present in the file. There are several ways in which this can be done.

The simplest way is probably to require that the file begin with the number of employees. The program can then start by reading this number and using a `for` loop to compute the pay of all the employees, as shown in Figure 3.7. (The declaration of the streams as well as the opening and closing of the files is omitted. It should be done as shown in Figure 3.3.)

An alternative that's somewhat more flexible is to have the last employee be followed by a special value called a **sentinel**. The sentinel cannot be a valid employee number so we can distinguish it from an employee number. The number 0 can be used in the pay calculator. Figure 3.8 shows a sample input file with a sentinel value. In other programs, the sentinel may need to be some other value.

```
int num_employees;
ifs_hours >> num_employees;

for (int i = 1; i <= num_employees; ++i) {
    int employee_number;
    ifs_hours >> employee_number;
    double num_hours;
    ifs_hours >> num_hours;

    double pay = num_hours * 20;
    ofs_pay << employee_number << ' ' << num_hours << ' ' << pay
            << '\n';
}
```

Figure 3.7: A version of the pay calculator that uses a count

```
12 43
23 37.5
37 40.33
0
```

Figure 3.8: A sample input file with a sentinel

```
int employee_number;
do {
    ifs_hours >> employee_number;

    if (employee_number != 0) {
        double num_hours;
        ifs_hours >> num_hours;

        double pay = num_hours * 20;
        ofs_pay << employee_number << ' ' << num_hours << ' ' << pay
                << '\n';
    }
} while (employee_number != 0);
```

Figure 3.9: A version of the pay calculator that uses a sentinel

Figure 3.9 shows a revised pay calculator that uses the sentinel to detect the end of the file.

The `do-while` loop of Figure 3.9 is somewhat messier than the `for` loop of Figure 3.7. The reason is that the test

```
employee_number != 0
```

occurs twice: once in the body of the loop and then again as the condition of the loop. Not only does this make the code look awkward, it is also inefficient since that test will be done twice for each employee.

Figure 3.10 shows a better way of setting up this loop. The number of the first employee is read *before* the loop. If that number is not 0, then the body of the loop reads the hours, computes the pay and reads the number of the next employee. This repeats until the number of the next employee turns out to be 0.

Note that the code of Figure 3.10 repeats the reading of the employee number. But this doesn't affect the execution time of the program since each employee number is read only once. In contrast, as we said earlier, the test `employee_number != 0` not only occurs twice in the code of Figure 3.9, it is also executed twice for each employee.

There is another way in which we can read the input file, one that is even more flexible because it doesn't require adding either a count or a sentinel value to the file.

```
int employee_number;
ifs_hours >> employee_number;

while (employee_number != 0) {
    double num_hours;
    ifs_hours >> num_hours;

    double pay = num_hours * 20;
    ofs_pay << employee_number << ' ' << num_hours << ' ' << pay
            << '\n';

    ifs_hours >> employee_number;
}
```

Figure 3.10: A simplified loop

Suppose that the file does not end in a sentinel value and that after computing the pay of the last employee, we go ahead and attempt to read another employee number. Then the reading operation will fail because the file does not contain another integer. And when a reading operation fails, the stream enters an *error state*.

There are several ways in which we can test if a stream is in an error state. The easiest way is to simply use the stream as a Boolean expression, as in

```
if (ifs_hours) ...
```

When used as a Boolean expression, a stream is considered `false` if it is in an error state, and `true` if it's not.

For example, consider the following code:

```
ifs_hours >> employee_number;
if (ifs_hours) ...
```

This reads an employee number. Then, if the reading is successful, the code represented by the three dots is executed. If the reading fails, that code is not executed.

Streams can also be used in the test of a loop. Here's an example:

```
int employee_number;
ifs_hours >> employee_number;

while (ifs_hours) {
    double num_hours;
    ifs_hours >> num_hours;

    double pay = num_hours * 20;
    ofs_pay << employee_number << ' ' << num_hours << ' ' << pay
        << '\n';

    ifs_hours >> employee_number;
}
```

Figure 3.11: A version of the pay calculator that detects the end of the file

```
ifs_hours >> employee_number;
while (ifs_hours) {
    ...
    ifs_hours >> employee_number;
}
```

This reads an employee number. If the reading is successful, the loop is executed. Otherwise, it is skipped entirely. At the end of each iteration of the loop, another employee number is read. If the reading succeeds, the test of the loop will be `true` and the loop will execute again. If instead the reading fails, then the test will be `false` and the loop will terminate. The net effect is that this loop will run if a first employee number can be read and as long as another employee number can be read.

Figure 3.11 shows how we can take advantage of this in the pay calculator. The loop will run as long as an employee number can be read from the file. This code works without the need to add either a count or a sentinel value to the file.

In Figure 3.11, the code that reads the employee number occurs twice. It is possible to eliminate this repetition. The key observation is that a reading operation such as

```
ifs_hours >> employee_number
```

evaluates to the stream (just as `2 + 3` evaluates to `5`). This implies that we can use a reading operation as the condition of an `if` statement or a loop.

For example, consider again the following code:

```
ifs_hours >> employee_number;
if (ifs_hours) ...
```

This code can be read as follows:

Read an employee number.
If the reading is successful, ...

By using the fact that the reading operation evaluates to the stream, we can combine these two statements into one:

```
if (ifs_hours >> employee_number) ...
```

The test of the `if` statement is evaluated first, which implies that the reading operation is executed first. Since this operation evaluates to the stream, it is the stream that is used as the test of the `if` statement. If the reading is successful, the stream is `true` and the code represented by the three dots is executed. If the reading fails, the stream is `false` and that code is not executed. The end result is the same as before: if the reading is successful, then the code represented by the three dots is executed.

Now, you may find the code

```
if (ifs_hours >> employee_number) ...
```

a little harder to read than the earlier two-statement version. In part, that's because the reading operation now serves two different purposes: it reads an employee number and it is used as the test of the loop. Here's one possible way to read this code:

If an employee number can be read from the file, ...

This makes sense and is, in fact, fairly natural as long as it is understood that if an employee number can be read, it will actually be read.

Reading operations can also be used in the test of a loop. For example, consider again the following code:

```
ifs_hours >> employee_number;
while (ifs_hours) {
    ...
    ifs_hours >> employee_number;
}
```

We know what this code does: the loop runs if a first employee number can be read and as long as another employee number can be read.

We can eliminate the repetition of the reading operation by using the reading operation directly in the test of the loop:

```
while (ifs_hours >> employee_number) {
    ...
}
```

When the test of this loop is evaluated, the reading operation executes and evaluates to the stream. If the reading is successful, the stream is `true` and the body of the loop is executed. If the reading fails, the stream is `false` and the body of the loop is not executed. And this keeps repeating. The end result is the same as before: the loop runs if a first employee number can be read and as long as another employee number can be read.

This new loop not only avoids the repetition of the reading operation, it also reads very well:

> *While an employee number can be read, ...*

In fact, the new loop reads much better than the earlier one since a direct reading of that loop would have been fairly awkward:

> *Read an employee number.*
> *While the reading is successful, execute the body of the loop and read another employee number.*

Figure 3.12 shows a revised pay calculator that uses the reading of the employee number as the test of the loop. This version is simpler than the earlier versions in two ways: one, it doesn't require that the input file contain a count or a sentinel value; two, it doesn't have to repeat either a test or the reading of the employee number.

As we just saw, the fact that a reading operation evaluates to the stream allows us to simplify loops. But it also has another advantage: it allows us to read multiple values with a single input statement, as in

```
int employee_number;
while (ifs_hours >> employee_number) {
    double num_hours;
    ifs_hours >> num_hours;

    double pay = num_hours * 20;
    ofs_pay << employee_number << ' ' << num_hours << ' ' << pay
        << '\n';
}
```

Figure 3.12: A more concise way version of the pay calculator that detects the end of the file

```
ifs_hours >> employee_number >> hours
```

What happens here is that this expression is evaluated from left to right, as if it had been written with parentheses:

```
(ifs_hours >> employee_number) >> hours
```

The first input operation reads an employee number and evaluates to `ifs_hours`. This value is then used as the left operand of the second input operation, which causes the hours to be read.

The same is true of output statements such as

```
ofs_pay << employee_number << ' ' << num_hours
```

This is evaluated from left to right. Each output operation evaluates to the stream `ofs_pay` and this value is used as the left operand of the next output operation.

Study Questions

3.3.1. What is a sentinel value?

3.3.2. What does an input or output operation evaluate to?

3.3.3. When used as a Boolean expression, what does an input or output stream evaluate to?

```
42000
28000
−1
2
2
70000
40000
8000
11.4
```

Figure 3.13: Sample tax return file for the program of Exercise 3.3.5

Exercises

3.3.4. Modify the latest version of the pay calculator program so that it prints to the screen the average number of hours worked by each employee and the total pay of all the employees.

3.3.5. Modify the income tax program of Exercise 2.3.3 so that in addition to interacting with the user, the program writes the data to a file called `tax_return.txt`. That file should begin with the various incomes followed by the sentinel value −1, the number of adults, the number of dependents, the total income, the taxable income, the tax and the effective tax rate. Figure 3.13 shows the file that corresponds to the sample session of Figure 2.11.

3.3.6. Create a program that reads the tax return file produced by the program of the previous exercise and displays the tax return on the computer's screen, as shown in Figure 3.14. The program should be able to handle any number of incomes.

3.3.7. Suppose that the file `temperatures.txt` contains the maximum temperature recorded in one location for each day of the year. Create a program that computes and prints to the computer's screen the average daily maximum for the year.

3.3.8. Create a program that computes the outcome of a referendum. The program reads a file called `results.txt`. In this file, each line contains a district number, a number of *yes* votes and a number of *no* votes.

```
Income 1: $42000
Income 2: $28000
Number of adults: 2
Number of dependents: 2

Total income: $70000
Taxable income: $40000
Income tax: $8000
Effective tax rate: 11.4%
```

Figure 3.14: Sample output for the program of Exercise 3.3.6

Those three numbers are separated by a single space. The program should compute the total number and percentage of *yes* and *no* votes, and prints those numbers in the following format:

```
Yes: 456 (53.4%)
No: 398 (46.6%)
```

3.4 Different Wages

Up until now, our pay calculator has been assuming that every employee gets paid $20 per hour. We now modify this to make the program more realistic. The program will read a file called `wages.txt` that contains one line per employee. Each line contains an employee number and the wage that employee earns, that is, the amount he or she should be paid for every hour of work. The employees occur in the same order in both files.

Figure 3.15 shows the revised pay calculator. The only novelty here is that the program reads from two input files at the same time.

Exercises

3.4.1. Modify the program of Exercise 3.3.7 so that instead of reading from one file `temperatures.txt`, the program reads from two files `potsdam.txt` and `cancun.txt`. The program should print the average daily maximum for both locations.

```
std::ifstream ifs_hours("hours.txt");
std::ifstream ifs_wages("wages.txt");
std::ofstream ofs_pay("pay.txt");

int employee_number;
while (ifs_hours >> employee_number) {
    double num_hours;
    ifs_hours >> num_hours;

    double wage;
    ifs_wages >> employee_number >> wage;

    double pay = num_hours * wage;
    ofs_pay << employee_number << ' ' << num_hours << ' ' << pay
            << '\n';
}
```

Figure 3.15: A version of the pay calculator in which employees earn different wages

3.5 More on Strings

In this section, we make our pay calculator more flexible by having the user specify the names of the input and output files. The program will interact with the user as follows:

```
Name of input file containing the hours: hours.txt
Name of output file for the pay: pay.txt
```

After asking the user for a file name, the program can store the string entered by the user in a variable of type `string`. For example,

```
string hours_file_name;
```

Strings can be read in at least two different ways. We are already familiar with the input operator:

```
cin >> hours_file_name
```

But this is probably not the best choice here because when reading a value of type `string`, the input operator works as follows: it skips white space and then reads characters until it sees another white space. White space means a blank space, a tab character (`\t`) or a new line character (`\n`). This implies that the input operator would not be able to read a file name that contains any blank spaces.

An alternative way of reading a `string` is to use the `getline` function:

```
getline(cin, hours_file_name)
```

This function takes two arguments, an input stream and a `string` variable. Starting at the current position in the stream, `getline` reads characters, including white space, all the way to the end of the current line and stores those characters in the string variable. The new line character is read but not stored in the `string`. In other words, while the input operator essentially reads words, the `getline` function reads lines.

Once a file name is stored in a `string` variable, we can use it to declare an input file stream and open the file as follows:[1]

```
std::ifstream ifs_hours(hours_file_name);
```

Figure 3.16 shows the revised pay calculator. (Portions of the program that did not change have been omitted.) The program ends by telling the user that the hours file has been read and that the pay file has been written.

Data types such as `int`, `double` and `char` are called *primitive* or *fundamental* because they are part of the C++ language itself. In contrast, data types such as `string`, `ifstream` and `ofstream` are not part of the language but are instead defined in its standard library.

The latest version of the pay calculator is available on the course web site under `Pay Calculator` as `pay_calculator_1_3_file_names.cpp`.

[1]Note that opening a file by directly using a `string` variable is possible only in the latest version of C++, which is called C++11 (because it was finalized in the year 2011). In older versions of C++, the value of type `string` first needs to be converted into another type of string called a *C string*:

```
std::ifstream ifs_hours(hours_file_name.c_str());
```

This is what must be done with compilers that still don't support the new features of C++11. Note that on some compilers it is necessary to set an option to turn on the new features of C++11.

```
cout << "Name of input file containing the hours: ";
string hours_file_name;
getline(cin, hours_file_name);

cout << "Name of output file for the pay: ";
string pay_file_name;
getline(cin, pay_file_name);

std::ifstream ifs_hours(hours_file_name);
std::ifstream ifs_wages("wages.txt");
std::ofstream ofs_pay(pay_file_name);

int employee_number;
while (ifs_hours >> employee_number) {
    ...
}

cout << "\nHours read from " << hours_file_name
     << " and pay written to " << pay_file_name << ".\n";
```

Figure 3.16: A version of the pay calculator that reads the file names from the user

Study Questions

3.5.1. When reading a `string` variable, when does the input operator stop reading?

3.5.2. What function can be used to read an entire line of text into a `string` variable?

Exercises

3.5.3. Modify the temperature program of Exercise 3.3.7 so the user can specify the name of the temperature file as follows:

```
Name of temperature file: potsdam.txt
```

3.5.4. Modify the program of the previous exercise as follows. The input file now specifies, for each day, not just the maximum temperature but also a description of the weather for that day. The description is one of the following four strings: `"sunny"`, `"mostly sunny"`, `"mostly cloudy"`, `"cloudy"`. (The quotes are not included in the file.) Each line in the file specifies the maximum temperature and weather description for one day, as in

```
73 mostly sunny
```

Revise the temperature program so that the output now includes the percentage of days to which each description applies:

```
Sunny: 22%
Mostly sunny: 26%
Mostly cloudy: 22%
Cloudy: 30%
```

3.6 Error Checking

In this section, we add some error checking to our pay calculator. We will consider only one possible type of error: the fact that a file may not open properly. In the case of an input file, this can occur because the file does not exist. In the case of an output file, this can occur because the file does not

exist and our program does not have permission to create a new file in the current directory/folder, or because the file exists but some other program has already opened it and has a *lock* on the file.

If a file does not open properly, this is considered an error and causes the stream to enter an error state, just like when a reading error occurs. And we already know that we can test for this by using the stream as a Boolean expression.

For example, the following code attempts to open the hours file and prints an error message in case of failure:

```
std::ifstream ifs_hours(hours_file_name);
if (!ifs_hours)
    cout << "Could not open file " << hours_file_name << ".\n";
```

As explained in Section 1.8, the symbol `!` represents the logical NOT operator. So the above code will print an error message if `ifs_hours` is `false`, that is, if the opening of the file failed.

Figure 3.17 shows how this error checking can be incorporated into the pay calculator. The wage file is opened first because there is no point asking the user for the names of the other files if the wage file won't open.

In case any of these files does not open, the program prints an error message and terminates the program by using the statement

```
return 1;
```

This is an example of a *return* statement. We will learn more about return statements later. In this context, that is, in the `main` function, a return statement causes the program to terminate. Recall that our programs normally end with

```
return 0;
```

The value `0` usually indicates that the program terminated normally, without errors. A positive value, such as `1`, usually indicates that an error occurred and that the program terminated abnormally.

The latest version of the pay calculator is available on the course web site under `Pay Calculator` as `pay_calculator_1_4_error_checking.cpp`.

Exercises

3.6.1. Add error checking to the programs you created for the last two exercises of Section 3.2.

```
std::ifstream ifs_wages("wages.txt");
if (!ifs_wages) {
    cout << "Could not open file wages.txt.\n";
    return 1;
}

cout << "Name of input file containing the hours: ";
string hours_file_name;
getline(cin, hours_file_name);
std::ifstream ifs_hours(hours_file_name);
if (!ifs_hours) {
    cout << "Could not open file " << hours_file_name << ".\n";
    return 1;
}

cout << "Name of output file for the pay: ";
string pay_file_name;
getline(cin, pay_file_name);
std::ofstream ofs_pay(pay_file_name);
if (!ofs_pay) {
    cout << "Could not open file " << pay_file_name << ".\n";
    return 1;
}
```

Figure 3.17: Error checking in the pay calculator

```
12 7 8 7.5 7.75 8.5 0 0
23 8 8 8.25 9 5 3 0
37 5 5.5 6 5 5 2 3
```

Figure 3.18: A revised input file

```
double num_hours = 0;
for (int i = 1; i <= 7; ++i) {
    double num_hours_one_day;
    ifs_hours >> num_hours_one_day;
    num_hours += num_hours_one_day;
}
```

Figure 3.19: Computing the total number of hours worked

3.7 Extending the Pay Calculator

Right now, the input file read by the pay calculator specifies, for each employee, the total number of hours worked by that employee for the entire pay period. In this section, we will expand the program so that it calculates those totals. In other words, the input file will specify, for each employee, the number of hours that employee worked on each day of the pay period, as illustrated in Figure 3.18.

This turns out to be an easy extension of our program. Right now, for each employee, the program reads the total number of hours worked with a simple input statement:

```
ifs_hours >> num_hours
```

We just need to replace this by a loop that reads the daily numbers and computes the total, as shown in Figure 3.19. This assumes that the pay period is a seven-day week. If that's not the case, then the number `7` in the condition of the loop should be replaced by the number of days in the pay period.

Note that we now have in our program two nested loops that are controlled in very different ways. The inner loop, which is shown in Figure 3.19, simply runs seven times. The outer loop (see Figure 3.15) runs while another employee number can be read from the file. This illustrates the fact that most non-trivial programs are created by combining a variety of different techniques, often in interesting and creative ways.

```
2
39227 75 83
42993 80 79
44371 92 87
```

Figure 3.20: Sample input file `gradesheet.txt` for Exercise 3.7.2

The latest version of the pay calculator is available on the course web site under `Pay Calculator` as `pay_calculator_2_0_daily_hours.cpp`.

Exercises

3.7.1. Modify the temperature program of Exercise 3.3.7 so it computes the average daily maximum for each month. The averages should be printed to the screen one per line, preceded by the month number:

```
1: 24
2: 27
3: 37
...
```

To keep things simple, assume that a year always consists of 12 months of exactly 30 days. (To test your program without having to create a file with 360 temperatures, you can create a version of the program that assumes that there are only two months with three days each.)

3.7.2. Create a program that computes course grades for all the students in a class. The program reads a file `gradesheet.txt` that contains the grades that students have earned on all the assessments in the class. The first line of the file contains the number of assessments that have been conducted. Each following line contains a student number followed by the grade that the student earned in each of the assessments. Figure 3.20 shows an example. A student's course grade is the average of the grades earned in those assessments. The program should create an output file `coursegrades.txt` that lists the course grades of all the students, as shown in Figure 3.21.

3.7.3. Modify the course grade program of Exercise 3.7.2 to use student names instead of student numbers. In the input file, the name of a student is

```
39227 79
42993 79.5
44371 89.5
```

Figure 3.21: Sample output file `coursegrades.txt` for Exercise 3.7.2

```
2
Tracy Brown
75 83
Erin Smith
80 79
John White
92 87
```

Figure 3.22: Sample input file `gradesheet.txt` for Exercise 3.7.3

given on one line and the grades of that student are given on the next line, as shown in Figure 3.22. The output file should list each student's name and course grade on the same line, as shown in Figure 3.23.

3.8 Compile-Time Constants

In an earlier section, we checked that the wages file opens properly by using the following code:

```
std::ifstream ifs_wages("wages.txt");
if (!ifs_wages) {
    cout << "Could not open file wages.txt.\n";
    return 1;
}
```

```
Tracy Brown 79
Erin Smith 79.5
John White 89.5
```

Figure 3.23: Sample output file `coursegrades.txt` for Exercise 3.7.3

Notice that the string `"wages.txt"` occurs twice in this code. And this is not great: if ever we needed to change the name of that file, we would have to remember to change both occurrences of the name.

A better approach is to store the file name in a variable and then use the variable instead of using the value directly. For example,

```
string wages_file_name = "wages.txt";
std::ifstream ifs_wages(wages_file_name);
if (!ifs_wages) {
    cout << "Could not open file " << wages_file_name << ".\n";
    return 1;
}
```

Then, if ever we needed to change the file name, we would need to change it in only one place.

Note that the value of the variable `wages_file_name` is set when the variable is declared and that this value is not supposed to change during the execution of the program. As a precaution, we can prevent that value from being changed by accident by declaring that the value of the variable is *constant*:

```
const string wages_file_name = "wages.txt";
```

In addition, the value of that variable is determined before the program runs. In fact, that value is determined before the program is even compiled. Such values, and the variables that hold them, are called **compile-time constants**.

Compile-time constants often play a special role in programs, as we will see later. For this reason, it is useful to distinguish them from other variables. One way is by using a different style for their names. In these notes, names of compile-time constants will begin with the prefix `k` followed by words in *mixed-case format*, that is, words separated only by the fact that their first letter is in uppercase. For example,

```
const string kWagesFileName = "wages.txt";
```

A value such as `"wages.txt"` that occurs explicitly in a program is called a *literal constant*. In general, it is better to give a name to each literal constant by storing it in a variable, even if the literal constant occurs only once in the program. It makes programs easier to understand.

```
double num_hours = 0;
for (int i = 1; i <= kLengthPayPeriod; ++i) {
    double num_hours_one_day;
    ifs_hours >> num_hours_one_day;
    num_hours += num_hours_one_day;
}
```

Figure 3.24: Computing the total number of hours worked

```
const int kLengthPayPeriod = 7;
const string kWagesFileName = "wages.txt";

int main() {
    ...
}
```

Figure 3.25: Global compile-time constants

For example, in our pay calculator, the loop that reads the daily number of hours for one employee is controlled by the literal constant `7` (see Figure 3.19). The condition of that loop becomes easier to understand if we use instead a named constant:

```
const int kLengthPayPeriod = 7;
```

The revised loop is shown in Figure 3.24.

It is also a good idea to place these named compile-time constants in an easy to find place, such as at the very beginning of the program. This makes the program easier to modify.

In fact, we can even place these constants *before* the `main` function, as shown in Figure 3.25. This not only makes the constants easy to find, it also gives them **global** scope, which means that they are visible anywhere in the program. In contrast, variables declared within `main` only exist within that function. It makes sense to make a constant global when its value is a parameter that affects the behavior of the entire program. Global compile-time constants play a more significant role in larger programs that consist of multiple functions. We will see examples later.

Study Questions

3.8.1. What is a compile-time constant?

3.8.2. Why is it better to use a different naming style for compile-time constants?

3.8.3. What is a literal constant?

3.8.4. Why is it better to name literal constants and place their declaration in an easy to find place?

3.8.5. What is a global constant?

Exercises

3.8.6. Modify the first version of the pay calculator, the one shown in Figure 3.3, by changing all literal constants into compile-time constants.

3.9 Formatting of Floating-Point Numbers

Up until now, we haven't worried about the exact appearance of the numbers produced by the program. As illustrated in Figure 3.2, the numbers may appear in various formats, with or without a decimal point, and with various numbers of digits after the decimal point.

This is perfectly fine if another program is going to read those numbers, but not if the output file is meant to be read by a person. In this section we will revise the program to ensure that the numbers of hours and the pay amounts always appear with exact two digits after the decimal point. In addition, we will make sure these numbers are lined up neatly in columns.

We have already seen how to line up numbers in columns by using the stream manipulator `setw`. So the main new thing we need to learn is how to control the appearance of numbers that are not integers. In programming, non-integers are usually called **floating-point numbers**. We already know that in C++, floating-point numbers are normally stored in variables of type `double`.

Floating-point numbers can be printed in either fixed-point or scientific format. The **fixed-point format** is the format we use in everyday life: at

```
ofs_pay << std::setw(6) << employee_number
        << std::fixed << std::setprecision(2)
        << std::setw(7) << num_hours
        << std::setw(9) << pay << '\n';
```

Figure 3.26: Output formatting in the pay calculator

least one digit possibly followed by a decimal point and other digits. For example, `12.3` and `0.0123`.

The **scientific format** is one digit followed possibly by a decimal point, more digits, the letter `e` and an integer. The number that follows the letter `e` is an exponent. For example, `1.23e+1` represents the number $1.23 \times 10^1 = 12.3$ and `1.23e-2` represents the number $1.23 \times 10^{-2} = 0.0123$.

The output format of floating-point numbers can be set to either fixed-point or scientific by using the stream manipulators `fixed` and `scientific`. For example,

```
cout << fixed;
```

sets to fixed-point the format of floating-point numbers printed to `cout`. This format will be in effect until it is changed.

By default, the format will vary from fixed-point to scientific depending on various factors. This is appropriate only if the format of the numbers doesn't matter. When the format of the numbers matters, it is best to set it explicitly to either fixed-point or scientific.

Now, in either the fixed-point or scientific formats, the number of digits that appears after the decimal point can be specified by the `setprecision` manipulator. Numbers are rounded and trailing 0's are added, as needed. For example, in fixed-point format, with the precision set to 3, 12.3 is printed as `12.300` and 0.0123 is printed as `0.012`.

Figure 3.26 shows how we can use these stream manipulators to format the output of the pay calculator. This produces three columns that separated by two spaces and are wide enough to accommodate employee numbers with 6 digits, numbers of hours up to 99.99 and pay amounts up to 9999.99. In addition, the numbers of hours and the pay amounts are printed in fixed format with exactly two digits after the decimal point.

Just like `setw`, the stream manipulators `setprecision`, `fixed` and `scientific` are defined in the library `iomanip` and are part of the namespace

`std`.

The latest version of the pay calculator is available on the course web site under `Pay Calculator` as `pay_calculator_2_1_formatting.cpp`.

Study Questions

3.9.1. What is the general form of a number in scientific format?

3.9.2. What is the general form of a number in fixed-point format?

3.9.3. What is the effect of `setprecision` in each of these two formats?

Exercises

3.9.4. Revise the program you wrote for Exercise 3.2.9 so that all numbers are printed in fixed-point format. The income and the tax should be printed as whole numbers while the tax rate should be printed with one digit after the decimal point.

Chapter 4

Functions

In this chapter, we will redesign the pay calculator of the previous chapter to make it modular. This has a number of advantages, which we will discuss. Our main tool will be the important concept of a function.

4.1 Introduction

In these notes, we have already seen two examples of functions. In the running pace calculator, we used the `round` function to round a number to its nearest integer (see Figure 1.6). In the pay calculator, we used the `getline` function to read file names from the user (see Figure 3.16).

As explained earlier, a function is a block of code that performs a particular task. Functions take arguments and may return a value. For example, `round` takes a number as argument and returns the value of that number rounded to the nearest integer. This means that a function call such as `round(x)` is an expression that evaluates to the rounded value of `x`.

There are a number of benefits to using functions in programs. In the case of library functions such as `round` and `getline`, the most obvious benefit is convenience: by using these functions, we didn't have to write that code ourselves. Instead, we were able to *reuse* code written by somebody else.

So the reuse of standard library components such as `round` and `getline` makes it easier to implement our programs. But it also makes them more reliable: since library code has already been used many times before, it is usually more reliable than new code we could write ourselves.

In this chapter, we will learn to write our own functions. This will allow

us to reuse not only library code but also our own code. For example, imagine that we had written our own `round` function and that we needed to round several numbers. Then instead of repeating the rounding code several times, we would call the rounding function whenever needed, in effect reusing the code of that function. This is not only more convenient, it also makes our programs easier to modify. That's because if the rounding code needed to be modified, then we would only need to modify the rounding function. In contrast, if the rounding code was repeated, we would need to track down and modify every occurrence of that code.

So because they allow us to reuse software, functions make our programs more reliable and easier to implement and modify. But there's more to it. Functions also isolate the details of different programming tasks within separate functions. For example, in the running pace calculator, the details of the rounding of a number are contained within the `round` function and isolated from the rest of the program. This makes the `main` function of the running pace calculator easier to understand. In general, when a program consists of several functions that perform different tasks, this allows us to focus on one task at a time when trying to understand how the program works.

This isolation of different tasks within different functions also makes programs easier to implement because it allows us to focus on one task at a time when writing the code. In addition, in team projects, the implementation of the various functions can be easily divided among the different programmers.

Functions also make programs easier to test because functions can be tested in isolation. This makes it easier to locate where problems are.

Finally, functions make programs easier to modify not only because they reduce repeated code, but also because a change to a program may affect only one or a few functions. Those functions can then be modified in isolation, minimizing the risk that the rest of the program is broken by accident.

To summarize, through reuse and isolation, the use of functions makes programs more reliable and easier to implement, understand, test and modify.

Study Questions

4.1.1. What are two advantages of software reuse?

4.1.2. What is a disadvantage of repeated code?

4.1.3. How do functions make programs easier to understand?

```
int round(double x)
{
    int result = x;
    x -= result;
    if (x >= 0.5) ++result;
    if (x <= -0.5) --result;
    return result;
}
```

Figure 4.1: The definition of `round`

4.1.4. How do functions make programs easier to implement?

4.1.5. How do functions make programs easier to test?

4.1.6. How do functions make programs easier to modify?

4.1.7. What are five advantages of the use of functions in programs?

4.2 A Rounding Function

In this section, we create our first function. It will be our own version of the library function `round`.

To create this function, we need to *define* it. Figure 4.1 shows a possible definition for `round`. A function definition consists of a *header* and a *body*. The header specifies the return type, the name and the arguments of the function, in that order. In this case, the header of `round` indicates that the function returns an integer and takes a single number of type `double` as argument.

The body of the function follows the header and is placed between braces. The body of a function is the code that is executed when the function is called. In other words, the body specifies how the function performs its task. The body of a function is often called its *implementation*. To implement a function means to write its body.

The last statement in the implementation of `round` is a `return` statement. A `return` statement has two effects: it terminates the execution of the function and it specifies the value returned by the function.

When a function has more than one argument, they should be separated by commas in the function header, as in

```
int round(double x);

int main()
{
    ...
}

int round(double x)
{
    ...
}
```

Figure 4.2: The function `round` implemented after `main`

```
double some_function(int x, string s)
```

Functions can also have no arguments. For example, consider the `main` function that must be included in every C++ program:

```
int main()
```

This function has no arguments, as indicated by the empty parentheses following the name of the function.

Note that a function must be declared before it is used. In the running pace calculator, this can be achieved by placing the definition of `round` before the definition of `main` since `round` is used within `main`. It is also possible to implement `round` after `main` but only if `round` is declared before `main`, as illustrated in Figure 4.2. A function declaration consists of a function header followed by a semicolon. Some programmers prefer to begin a program with a declaration of all the functions followed by the implementation of `main` and then the implementation of the functions used by `main`.

Study Questions

4.2.1. What three pieces of information does a function header always include?

4.2.2. What is the body of a function?

4.2.3. What does it mean to implement a function?

4.2.4. What does a `return` statement do?

Exercises

4.2.5. Create a function called `negative` that takes a number as argument and returns the same number but with the opposite sign. For example, `negative(3.7)` returns -3.7 and `negative(-4)` returns `4`. The argument of the function is of type `double`.

4.2.6. Create a function called `abs` that takes a number as argument and returns the absolute value of that number. For example, `abs(3.7)` returns `3.7` and `abs(-4)` returns `4`. The argument of the function is of type `double`.

4.2.7. Create a function called `sum` that takes two numbers as arguments and returns their sum. For example, `sum(3.7, -2.3)` returns `1.4`. The arguments of the function are of type `double`.

4.3 Functions in the Pay Calculator

We now turn to our pay calculator. We will redesign that program by breaking up `main` into several functions. This will not reduce repeated code but it will simplify `main` by isolating several of the tasks that the program performs into separate functions. As stated earlier in this chapter, this will make the program easier to understand and modify. If we had designed the program in this way from the beginning, it would have also been easier to implement and easier to test.

In this section, we will add four functions to the program. They are shown in Figure 4.3.

The first function is `compute_pay`. It receives a number of hours and a wage as arguments and returns the corresponding pay. The arguments of the function have the same names as variables that are declared in `main`. But note that `main`'s variables and `compute_pay`'s arguments are separate variables. In fact, `main`'s variables are local to `main` and cannot be accessed by `compute_pay` while `compute_pay`'s arguments are local to `compute_pay` and cannot be accessed by `main`. It is fairly common for the same name to be reused for the local variables or arguments of several different functions.

The function `print_pay` receives an employee number, a number of hours, a pay amount and an output stream as arguments and prints the data to the stream. Note that the output stream is of type `ostream`. This is more general

```
double compute_pay(double num_hours, double wage)
{
    return num_hours * wage;
}

void print_pay(int employee_number, double num_hours, double pay,
               std::ostream & out)
{
    out << std::setw(6) << employee_number
        << std::fixed << std::setprecision(2)
        << std::setw(7) << num_hours
        << std::setw(9) << pay << '\n';
}

double read_hours(std::istream & in)
{
    double total_hours = 0;
    for (int i = 1; i <= kLengthPayPeriod; ++i) {
        double num_hours_one_day;
        in >> num_hours_one_day;
        total_hours += num_hours_one_day;
    }
    return total_hours;
}

double read_wage(std::istream & in)
{
    int employee_number;
    int wage;
    in >> employee_number >> wage;
    return wage;
}
```

Figure 4.3: Functions for the pay calculator

```
int employee_number;
while (ifs_hours >> employee_number) {
    double num_hours = read_hours(ifs_hours);
    double wage = read_wage(ifs_wages);
    double pay = compute_pay(num_hours, wage);
    print_pay(employee_number, num_hours, pay, ofs_pay);
}
```

Figure 4.4: Using the functions of Figure 4.3 in the pay calculator

than `ofstream`, which allows the function to print not only to an output file stream but to any type of output stream, including `cout`, for example. Note also the ampersand (`&`) between the stream type and the argument name. This is required for all stream arguments. We will explain why later in this chapter.

The function `print_pay` does not return a value. This is indicated by the return type `void` in the function header. Such functions are called **void functions** in C++. Non-void functions are sometimes called **valued functions**. A void function returns after the last statement of the function's body is executed. If we need the function to return from somewhere else, this can be achieved with an empty `return` statement:

```
return;
```

The function `read_hours` reads the hours worked by an employee each day of the pay period and returns the total. The function `read_wage` is similar. Note that these two functions assume that the data that needs to be read is the next data that is available from the stream received as argument.

Figure 4.4 shows a portion of the `main` function of the pay calculator that uses those four functions. For comparison, Figure 4.5 shows the same code before the introduction of the functions. The version of `main` that uses the functions is clearly much simpler and much easier to read and understand.

Note that the function `read_hours` uses the global constant `kLengthPayPeriod`. This is possible only because this constant has global scope. If the constant was local to `main`, it would have to be passed as an argument to `read_hours`. Therefore, the fact that the constant has global scope simplifies the declaration and calling of `read_hours`.

But note that this idea should not be abused. In particular, it is a bad idea to declare *variables* to have global scope. In a program with no global

```
int employee_number;
while (ifs_hours >> employee_number) {
    double num_hours = 0;
    for (int i = 1; i <= kLengthPayPeriod; ++i) {
        double num_hours_one_day;
        ifs_hours >> num_hours_one_day;
        num_hours += num_hours_one_day;
    }

    double wage;
    ifs_wages >> employee_number >> wage;

    double pay = num_hours * wage;
    ofs_pay << std::setw(6) << employee_number
            << std::fixed << std::setprecision(2)
            << std::setw(7) << num_hours
            << std::setw(9) << pay << '\n';
}
```

Figure 4.5: The code of Figure 4.4 without functions

variables, the only variables that can be modified by a function are those that are passed as arguments to the function. But in a program that has global variables, a function can modify a global variable that is not mentioned in the function call. Such hidden side-effects make programs usually harder to understand. As a general rule, only constants should have global scope and it is best to reserve this for compile-time constants that truly affect the behavior of the entire program.

Study Questions

4.3.1. What does the return type `void` indicate?

4.3.2. Is every function required to include a `return` statement in its body?

4.3.3. Why is it a bad idea for variables to have global scope?

Exercises

4.3.4. Create a function called `print_date` that prints a date to an output stream. The arguments of the function are the date and the output stream. The date is passed as three separate integers, one for the month of the date, one for the day and one for the year, in that order. The date is printed in the format `m/d/y`.

4.3.5. Create a function called `println` that takes a stream and an integer as arguments and prints the integer to the stream followed by a new line character.

4.3.6. Modify the pay calculator as described below. For each part, modify only one of the program's functions.

a) Employees are paid time and a half for every hour worked beyond 40 hours.

b) The number of hours is no longer printed to the output file.

c) The wage file contains the name of the employee. More precisely, the data for each employee consists of three lines, one for the employee number, one for the employee name and one for the numbers of hours worked each day of the pay period.

```
int round(double x)
{
    int result = x;
    x -= result;
    if (x >= 0.5) ++result;
    if (x <= -0.5) --result;
    return result;
}
```

Figure 4.6: The function `round`

4.4 Reference Arguments

Suppose that `y` is a variable of type `double`. The function `round` we created earlier in this chapter (and shown again in Figure 4.6) can be used to easily print the rounded value of `y`:

```
cout << round(y);
```

Or to set another variable to the rounded value of `y`:

```
i = round(y);
```

The function can also be used to round the value of `y`, in the sense of setting `y` to its rounded value:

```
y = round(y);
```

This last application of `round` is not as elegant as the others because the name of `y` needs to be repeated. One solution is to create another function that instead of returning the rounded value of its argument, would set its argument to its rounded value. Let's call this function `make_rounded`. The function could then be used to round `y` with a simple function call:

```
make_rounded(y);
```

Figure 4.7 shows how `make_rounded` can be defined. The function is very similar to `round`. One difference is that instead of returning `result`, `make_rounded` sets its argument `x` to the value of `result`. The other difference is that the declaration of the argument includes an ampersand (`&`).

```
void make_rounded(double & x)
{
    int result = x;
    x -= result;
    if (x >= 0.5) ++result;
    if (x <= -0.5) --result;
    x = result;
}
```

Figure 4.7: The function `make_rounded`

Understanding what this does, and how this works, requires taking a closer look at how arguments are passed to functions.

We see in Figure 4.6 that the argument of `round` was declared without this ampersand. What this indicates is that the argument of `round` is passed **by value**. This means that the function receives the *value* of the expression that is provided in the function call. For example, suppose that the function is called as follows:

```
round(y)
```

Then the argument `x` of the function is set to the value of `y`. This implies that `x` is a *copy* of `y`. In particular, if `round` modifies the value of `x`, the value of `y` won't be affected.

In contrast, the ampersand in the declaration of the argument of `make_rounded` indicates that the argument of the function is passed **by reference**. This means that this argument will be set to *refer* to the expression that's provided in the function call. For example, suppose that the function is called as follows:

```
make_rounded(y)
```

Then the argument `x` of the function will refer to `y`. The main consequence is that when `make_rounded` modifies the value of `x`, then, at the same time, it also modifies the value of `y`.

So reference arguments allow a function to modify the value of a variable that belongs to the calling function. This avoids the need for awkward code such as

```
y = round(y);
```

```
string read_name()
{
    cout << "What is your name? ";
    string name;
    getline(cin, name);
    return name;
}
```

Figure 4.8: A function that reads a name

```
void read_name(string & name)
{
    cout << "What is your name? ";
    getline(cin, name);
}
```

Figure 4.9: A version of `read_name` that uses a reference argument

But reference arguments have other important benefits.

One is that reference arguments can improve the efficiency of programs. For example, suppose that we want to ask the user for his or her name and then store that name in variable `user_name`. Figure 4.8 shows a function that we could create for this purpose. The function would then be used as follows:

```
string user_name = read_name();
```

Figure 4.9 shows another version of this function, one that uses a reference argument. This version would be used as follows:

```
string user_name;
read_name(user_name);
```

In terms of convenience, there is little difference between these functions, although some programmers prefer one style over the other. But there could be a large difference in terms of efficiency. That's because with the first version, the returned string needs to be copied into the variable of the calling function (`user_name`). But with the second version, this copying is not needed: the reference argument essentially allows `read_name` to read the name straight

```
void read_name_and_hometown(string & name, string & hometown)
{
    cout << "What is your name? ";
    getline(cin, name);
    cout << "What is your hometown? ";
    getline(cin, hometown);
}
```

Figure 4.10: A function that reads a name and hometown

into the variable of its calling function. If the string is long, this may lead to significantly faster code.

To illustrate another benefit of reference arguments, suppose that we need to extend `read_name` to ask the user for his or her name and hometown. And let's say that we want these two strings stored separately. A valued version of the function would need to return two strings but this is not possible. The simplest solution to this problem is to use two reference arguments, as shown in Figure 4.10.

To summarize, reference arguments allow a function to directly modify the variables of its calling function. This has three main benefits: it can simplify the code of the calling function, it can improve the overall efficiency of the program, and it allows a function to pass back multiple values to its calling function.

We end this section with several additional notes about reference arguments. First, there is another way in which reference arguments can improve the efficiency of programs. Suppose that we'd like to create a function `println` that takes a string as argument and prints the string and moves to a new line. How should the string argument be passed to the function? If it is passed by value, then the string will be unnecessarily copied during the function call. So it is more efficient to pass the string by reference, as shown in Figure 4.11. This allows the function to directly print the string held by the calling function instead of first wasting time copying the string.

Note, however, that in this case, there is a disadvantage to passing the string by reference: while we don't want the `println` to modify the string held by its calling function, it could happen by accident. This is not possible if the argument is passed by value but it can happen when the argument is passed by reference. One way to prevent this while still avoiding the copying

```
void println(string & s)
{
    cout << s << '\n';
}
```

Figure 4.11: A function that prints a string and moves to the next line

```
void println(const string & s)
{
    cout << s << '\n';
}
```

Figure 4.12: A version of `println` in which the argument is passed by constant reference

of the string is to declare that the string argument is constant, as shown in Figure 4.12. We say that such an argument is passed **by constant reference**.

So we now know three ways in which arguments can be passed to a function: by value, by reference and by constant reference. Arguments passed by value or constant reference can only be used to send data to the function. This is the case with the arguments of `round` and `println`. We can say that these arguments are *inputs* to the function.

In contrast, reference arguments can be used by the function to send data back to its calling function by directly modifying a variable of the calling function. In some cases, a function can use its reference arguments to both receive data from its calling function and send data back to it. This is how the argument of `make_rounded` is used. In other cases, reference arguments are only used to send data back to the calling function. This is how the arguments of the two `read` functions are used. So we can say that a reference argument is an *output* of the function or, in some cases, both input and output.

We can chose between passing by value, reference or constant reference as follows. If the function needs to modify a variable held by its calling function, then the corresponding argument needs to be passed by reference. This was the case with the arguments of `make_rounded` and of the two `read` functions. If the function should not modify a variable held by its calling function, then it is safer to pass the corresponding argument by value or constant reference. This is how we passed the arguments of `round` and `println`. If the function

does not need a copy of the value that's passed to it and if copying that value can take a significant amount of time, then it is more efficient to pass the argument by constant reference. This is what we did in the case of `println` because `println` does not need a copy of the string and because `string` arguments can be large. Otherwise, the argument can be passed by value. This is how the argument of `round` is passed. In general, it is fine to pass `int`, `double` and `char` arguments by value.

Note that when an argument is passed by value or constant reference, then when the function is called, any expression (of the correct type) can be provided as argument. For example,

```
i = round(y + 1.2);
println("hello");
```

are all valid function calls. But when an argument is passed by reference, then a variable must be provided in the function call. For example, the following are not valid:

```
make_rounded(y + 1.2);
read_name("hello");
```

This makes sense because if an argument is passed by reference, then what the argument refers to must be able to change. So it has to be a variable.

In the previous section, we noted that stream arguments must always be passed by reference (but we didn't use this terminology at the time). This is simply because every time a stream variable is used to read from or write to a file, the value of the stream variable changes. Mostly because the variable keeps track of the location where the next operation should be performed in the file. But also because, in case of an error, the state of the stream variable will be set to *error*. This is what causes the stream variable to evaluate to `false` when as a Boolean expression.

Study Questions

4.4.1. What happens when an argument is passed by value?

4.4.2. What happens when an argument is passed by reference?

4.4.3. What does a reference argument allow a function to do?

4.4.4. What are three benefits of reference arguments?

4.4.5. When should an argument be passed by reference?

4.4.6. When should an argument be passed by value?

4.4.7. When should an argument be passed by constant reference?

4.4.8. Why should stream arguments always be passed by reference?

Exercises

4.4.9. Create a function called `negate` that takes a number as argument and reverses its sign. For example, `negate(x)` changes the value of `x` from `3.7` to -3.7, or from -4 to `4`. The argument of the function is of type `double`.

4.4.10. Create a function called `add` that takes three numbers as arguments and sets the third one to be the sum of the first two. For example, `add(3.7, -2.3, x)` sets the value of `x` to `1.4`. The arguments of the function are of type `double`.

4.4.11. Create a function called `read_date` that reads a date from an input stream. The arguments of the function are the input stream and three separate integers, one for the month of the date, one for the day and one for the year, in that order. In the stream, the date is assumed to be in the format `m/d/y`.

4.4.12. Create a function called `println` that takes a string and a stream as arguments and prints the string to the stream followed by a new line character.

4.5 Reference Arguments in the Pay Calculator

We return to our pay calculator. We will add one more function to the program, a function to which `main` will delegate the opening of the files. This will result in the `main` function shown in Figure 4.13. Not only is this version of `main` much simpler than before, at 29 lines it is also reasonably short. It is

```
int main() {
    std::ifstream ifs_wages;
    std::ifstream ifs_hours;
    std::ofstream ofs_pay;
    string hours_file_name;
    string pay_file_name;

    if (!open_files(ifs_wages, ifs_hours, ofs_pay, hours_file_name,
                    pay_file_name))
        return 1;

    int employee_number;
    while (ifs_hours >> employee_number) {
        double num_hours = read_hours(ifs_hours);
        double wage = read_wage(ifs_wages);
        double pay = compute_pay(num_hours, wage);
        print_pay(employee_number, num_hours, pay, ofs_pay);
    }

    cout << "\nHours read from " << hours_file_name
         << " and pay written to " << pay_file_name << ".\n";

    return 0;
}
```

Figure 4.13: The `main` function of the pay calculator

useful from a programmer's point of view when the body of a function is short enough that most of it can fit on a computer screen.

Figure 4.14 shows the definition of the `open_files` function. The function receives as arguments three file streams and two strings. The function's job is to open the wage, hours and pay files, in that order, and associate them with those streams. The names of the hours and pay files are obtained from the user and returned to `main` through the `string` arguments. In case any of the files fails to open, the function immediately prints an error message, closes all the files and returns the value `false`.

The implementation of `open_files` is straightforward but includes two novelties. One is the return type `bool`. This type consists of the two Boolean values `true` and `false`.

The second novelty is the way in which the files are opened. Up until now, every file has been opened at the same time that we declared the associated file stream variable. But here, the declaration of the file stream variables, which occurs in `main`, is separate from the opening of the files, which occurs in `open_files`. To open a file and associated it with a stream, `open_files` uses code of the following general form:

```
file_stream.open(file_name);
```

Later, to close a file, the following code is used:

```
file_stream.close();
```

The redesign of our pay calculator is complete. The program now consists of six functions that each have a well-defined task. Various aspects of the program are now isolated within separate functions, including the opening of the files (`open_files`), the reading of the hours and wage (`read_hours` and `read_wage`), the computing of the pay (`compute_pay`), the printing of the pay (`print_pay`), and the overall control of the program (`main`).

We end this section by addressing an issue of style. Some programmers prefer to write code that satisfies the following guidelines:

1. The name of a `void` function should be a verb phrase that describes the action performed by the function.

2. The name of a Boolean function (a function that returns a Boolean value, that is, a function with return type `bool`) should be a predicate (a verb phrase that is either true or false).

```
bool open_files(std::ifstream & ifs_wages,
                std::ifstream & ifs_hours,
                std::ofstream & ofs_pay,
                string & hours_file_name,
                string & pay_file_name)
{
    ifs_wages.open(kWagesFileName);
    if (!ifs_wages) {
        cout << "Could not open file " << kWagesFileName << ".\n";
        return false;
    }

    cout << "Name of input file containing the hours: ";
    getline(cin, hours_file_name);
    ifs_hours.open(hours_file_name);
    if (!ifs_hours) {
        cout << "Could not open file " << hours_file_name << ".\n";
        ifs_wages.close();
        return false;
    }

    cout << "Name of output file for the pay: ";
    getline(cin, pay_file_name);
    ofs_pay.open(pay_file_name);
    if (!ofs_pay) {
        cout << "Could not open file " << pay_file_name << ".\n";
        ifs_wages.close();
        ifs_hours.close();
        return false;
    }

    return true;
}
```

Figure 4.14: The function `open_file`

```
int main() {
    std::ifstream ifs_wages;
    std::ifstream ifs_hours;
    std::ofstream ofs_pay;
    string hours_file_name;
    string pay_file_name;

    if (!files_open_successfully(ifs_wages, ifs_hours, ofs_pay,
                                 hours_file_name, pay_file_name))
        return 1;

    int employee_number;
    while (ifs_hours >> employee_number) {
        double num_hours = hours_read_from(ifs_hours);
        double wage = wage_read_from(ifs_wages);
        double pay = pay_computed_from(num_hours, wage);
        print_pay(employee_number, num_hours, pay, ofs_pay);
    }

    cout << "\nHours read from " << hours_file_name
         << " and pay written to " << pay_file_name << ".\n";

    return 0;
}
```

Figure 4.15: The `main` function of the pay calculator

3. The name of any other valued function should be a noun phrase that describes the value returned by the function.

Mainly as an exercise, let's revise our pay calculator so it meets these guidelines. One way to achieve this is simply to rename some of the functions. The resulting `main` function is shown in Figure 4.15.

Now, if we wanted to keep the original function names, which were all non-predicate verb phrases, then we would need to redesign the program so that all the functions are `void`. Figures 4.16 to 4.18 show the result. All the previously valued functions now have an additional reference argument that they use to send back to `main` the value that they used to return. This lengthens the implementation of `main` and `open_files` but shortens that of

`read_hours` and `read_wage`. The difference is not significant but `main` is not as simple as it used to be.[1]

When a function has more than one argument, a useful guideline is to generally order the arguments as follows: arguments used only as input (that is, arguments passed by value or constant reference), arguments used as both input and output, arguments used only as output. Within these categories, more important arguments typically come first. For example, the `pay` argument of `compute_pay` was placed after the others because it is used as output. And the arguments of `open_files` were placed in decreasing order of importance with respect to the purpose of the function. But we made an exception when we placed the stream argument of the `read` functions at the end. In this case, we preferred to be consistent with the order of the arguments of `print_pay`.

4.6 Modularity and Abstraction

In this chapter, we redesigned our pay calculator by breaking it up into a collection of six separate functions. This has a number of benefits, as discussed earlier in this chapter.

A program that consists of multiple functions is often said to be modular. But modularity is a concept that's more general than just the use of functions. Understanding what modularity is helps us better understand the benefits of modularity.

A program is **modular** if it consists of components (or modules) that satisfy the following two conditions:

1. Each component performs a well-defined task.

2. The components are as independent as possible from each other, in the sense that a change to one component affects other components as little as possible.

Components are often functions but they can also be programmer-defined data types.

Note that modularity is a matter of degree since components can be more or less independent from each other. The goal is to design programs that are as modular as possible.

[1] You may have an opinion about which style you prefer but, in any case, it is good to be familiar and comfortable with code written in a variety of different styles.

```
void compute_pay(double num_hours, double wage, double & pay)
{
    pay = num_hours * wage;
}

void print_pay(int employee_number, double num_hours, double pay,
               std::ostream & out)
{
    out << std::setw(6) << employee_number
        << std::fixed << std::setprecision(2)
        << std::setw(7) << num_hours
        << std::setw(9) << pay << '\n';
}

void read_hours(double & num_hours, std::istream & in)
{
    num_hours = 0;
    for (int i = 1; i <= kLengthPayPeriod; ++i) {
        double num_hours_one_day;
        in >> num_hours_one_day;
        num_hours += num_hours_one_day;
    }
}

void read_wage(double & wage, std::istream & in)
{
    int employee_number;
    in >> employee_number >> wage;
}
```

Figure 4.16: Functions for the pay calculator

```
void open_files(std::ifstream & ifs_wages,
                std::ifstream & ifs_hours,
                std::ofstream & ofs_pay,
                string & hours_file_name,
                string & pay_file_name,
                bool & success)
{
    ifs_wages.open(kWagesFileName);
    if (!ifs_wages) {
        cout << "Could not open file " << kWagesFileName << ".\n";
        success = false;
        return;
    }

    cout << "Name of input file containing the hours: ";
    getline(cin, hours_file_name);
    ifs_hours.open(hours_file_name);
    if (!ifs_hours) {
        cout << "Could not open file " << hours_file_name << ".\n";
        ifs_wages.close();
        success = false;
        return;
    }

    cout << "Name of output file for the pay: ";
    getline(cin, pay_file_name);
    ofs_pay.open(pay_file_name);
    if (!ofs_pay) {
        cout << "Could not open file " << pay_file_name << ".\n";
        ifs_wages.close();
        ifs_hours.close();
        success = false;
        return;
    }

    success = true;
}
```

Figure 4.17: The function `open_file`

```
int main() {
    std::ifstream ifs_wages;
    std::ifstream ifs_hours;
    std::ofstream ofs_pay;
    string hours_file_name;
    string pay_file_name;

    bool success;
    open_files(ifs_wages, ifs_hours, ofs_pay, hours_file_name,
               pay_file_name, success);
    if (!success) return 1;

    int employee_number;
    while (ifs_hours >> employee_number) {
        double num_hours;
        read_hours(num_hours, ifs_hours);

        double wage;
        read_wage(wage, ifs_wages);

        double pay;
        compute_pay(num_hours, wage, pay);
        print_pay(employee_number, num_hours, pay, ofs_pay);
    }

    cout << "\nHours read from " << hours_file_name
         << " and pay written to " << pay_file_name << ".\n";

    return 0;
}
```

Figure 4.18: The `main` function of the pay calculator

Earlier in this chapter, we mentioned that the use of functions in programs has a number of advantages. But these advantages actually follow from the more general concept of modularity.

1. Modular programs are easier to understand because each of their components focuses on one well-defined task and because each component can be understood in isolation. In addition, modular programs contain less repeated code, which also helps make them easier to read.

2. Modular programs are easier to implement because components can be coded one at a time and because the implementation work can be divided among various programmers. In addition, modularity makes it easier to reuse components from other programs, which saves work.

3. Modular programs are easier to test because components can be tested in isolation. This simplifies locating and fixing errors.

4. Modular programs are easier to modify because they contain less repeated code and because changes to one component often only affect that component.

5. Modularity increases the reliability of programs by making it easier to reuse software components that often have already been extensively tested.

Independence between software components usually occurs when one component does not depend on some aspect of another component. We then say that this aspect of the second component is *hidden* from the first one. This is called **information hiding**.

With functions, a high degree of information hiding, and therefore independence, is essentially automatic. For example, consider the function `compute_pay` of the pay calculator. The `main` function depends on *what* `compute_pay` does, the fact that it computes the pay. But `main` does not depend on *how* `compute_pay` computes the pay. Those implementation details are hidden in the body of the function.

The previous example illustrates the usual way in which information hiding and independence are achieved: by having the users of a component depend on its purpose (what the component does) but not on its implementation (how the component does what it does). This is called **abstraction**. In the case of functions, we call it **procedural abstraction**.

Abstraction is one of the most important techniques for the design of modular programs. Abstraction is automatic with functions but it isn't with other kinds of software components such as data types. With those components, abstraction must be designed into the software. This is why it is important to understand what abstraction is and how it is achieved.[2]

Note that abstraction leads to two clearly separate perspectives on each software component. From the point of view of its users, a component can be seen as having a purpose but no implementation: it is an *abstract* component. This is the *outside* or *public* view of the component. This view concerns only *what* the component does and it includes the *interface* of the component, which is the information needed to use, or communicate with, the component. In the case of a function, this is the name of the function, as well as its arguments and return type.

The developer who's implementing a component has a different view: he or she also sees the implementation of the component. This is the *inside* or *private* view. It concerns *how* the component does what it does and it includes the *inner workings* of the component.

Study Questions

4.6.1. What exactly is a modular program?

4.6.2. What are five advantages of modularity?

4.6.3. What is abstraction?

4.7 Documentation

Many of the advantages of modularity we described in the previous section require that we have a clear understanding of what each component does. For example, if the implementation of the components is divided among various programmers, then the programmer who's implementing a component and the programmer who's using that component must agree precisely on what the component does. And if a component is going to be reused in another program, the developer of that program will have to know exactly what the

[2] At Clarkson, data abstraction is one of the main focuses of the course CS142 *Introduction to Computer Science II*.

component does. The best approach is to carefully write down the purpose of each component. This is one form of *documentation* that all programs should include.

The purpose of each function of a program can be described in a separate document or in the source code itself. If it is done in the source code, it is usually better to put all the documentation at the top of each source file. Figure 4.19 show one way of doing this.

The code of our schedule viewer program is contained in a single file. After the `include` directives and `using` declarations, the global constants and the functions are declared and documented. Recall that the declaration of a function is its header followed by a semicolon. After the declaration and documentation of all the functions, the source file of the program continues with the implementation (body) of those functions.

The latest version of the pay calculator is available on the course web site under `Pay Calculator` as `pay_calculator_3_2_documentation.cpp`.

Study Questions

4.7.1. Why is it important to document the purpose of each function?

```
// Computes the pay corresponding to the given number of hours and
// wage. The pay is returned.
double compute_pay(double num_hours, double wage);

// Opens the wage, hours and pay files, in that order. The names
// of the hours and pay files are obtained from the user. Those
// names are returned through the corresponding arguments. In case
// one of the files does not open, an error message is immediately
// printed, all the files are closed and the function returns
// false.
bool open_files(std::ifstream & ifs_wages,
                std::ifstream & ifs_hours,
                std::ofstream & ofs_pay,
                string & hours_file_name,
                string & pay_file_name);

// Prints, for each employee, the number of that employee, the
// number of hours worked and the pay earned.
void print_pay(int employee_number, double num_hours, double pay,
               std::ostream & out);

// Reads the number of hours worked each day by an employee and
// returns the total.  The next data in the stream is assumed to
// be the numbers of hours worked each day by the employee.  The
// total is returned.
double read_hours(std::istream & in);

// Reads the wage of the next employee.  The next data in the
// stream is assumed to be the number and wage of the employee.
double read_wage(std::istream & in);

// Runs the pay calculator.
int main();
```

Figure 4.19: Declaration and documentation of the pay calculator functions

Chapter 5

Vectors

In this chapter, we will learn how to use vectors to store large amounts of data.

5.1 A Simple File Viewer

We will create a simple file viewer that allows the user to view the contents of a text file. The file viewer is not an editor: the user can only *view* the contents of the file, not modify it.

Figure 5.1 shows what the interface of this program will look like. The program displays the name of the file that is currently being viewed, if any, followed by a certain number of lines from that file surrounded by a border. Below the text, a menu of commands is displayed followed by the prompt "`choice:`". The user types the first letter of a command, the command executes and everything is redisplayed.

The commands *next* and *previous* cause the file viewer to display the next or previous "pages". The command *open* causes the program to ask the user for the name of another file to open.

We will design and implement the file viewer later in this chapter. A main issue in the design of the program is the access to the file contents. Because the user can page forward and backward through the file, the program may have to display some lines multiple times. These lines would then also have to be read from the file multiple times. Since files are typically stored on a secondary storage device such as a disk, file input and output operations are relatively slow, much slower than operations that can be performed on the computer's main memory.

```
preface.txt
_____________________________________________________

  1  After a few computer science courses, students may start to
  2  get the feeling that programs can always be written to
  3  solve any computational problem.  Writing the program may
  4  be hard work.  For example, it may involve learning a
  5  difficult technique.  And many hours of debugging.  But
  6  with enough time and effort, the program can be written.
  7
  8  So it may come as a surprise that this is not the case:
  9  there are computational problems for which no program
 10  exists.  And these are not ill—defined problems (Can a
_____________________________________________________

  next  previous  open  quit
_______

command: o
file: introduction.txt
```

Figure 5.1: Sample interface of the file viewer

Therefore, a more efficient alternative is to store the contents of the file in main memory, that is, in the program's variables. How exactly could this be done? One option would be to read each line as a string and store those strings in separate variables. But this is not possible since we don't know ahead of time how many lines the file contains. Even if we were willing to put a limit on that number, say, 10,000 lines, it would totally impractical to declare and work with 10,000 separate variables.

Ideally, what we would like is a way to store a large number of strings in a single variable. This is what vectors will allow us to do.

5.2 Vector Basics

A **vector** is a *container* that holds a (finite) sequence of values called *elements*. These elements can be of pretty much any type but in an individual vector, all elements have to be of the same type. The type of element held by a particular vector is specified at the time that the vector is created. For example,

```
vector<int> v;
```

creates a vector `v` that holds integers.

Vectors can be created in many different ways. The above declaration produces an empty vector, one that contains no elements. On the other hand,

```
vector<int> v = {12, 37, 18};
```

creates a vector that contains the given integers.

Larger vectors can be created by specifying their size. For example,

```
vector<int> v(n, e);
```

creates a vector that contains `n` elements initialized to be copies of `e`. On the other hand,

```
vector<int> v(n);
```

creates a vector that contains `n` elements initialized to a default value determined by their type. For example, numbers are initialized to 0 and strings are initialized to be empty.

Vectors are *dynamic* in the sense that they can grow and shrink as needed. For example,

```
v.push_back(e);
```

adds a new element to the back of `v` and initializes that element be a copy of `e`. On the other hand,

```
v.pop_back();
```

deletes the last element of `v`. And

```
v.clear();
```

deletes all the elements of `v`, resulting in an empty vector.

Individual vector elements are numbered starting at 0. These numbers are called *indices* and can be used to access the elements of a vector. For example,

```
cout << v[i];
```

("c-out v-i") prints the element in `v` that has index `i`. The square brackets represent the *indexing* operator.

The indexing operator returns a reference to the requested element, which implies that the operator can be used not only to retrieve an element but also to modify that element. For example

```
v[i] = e;
```

assigns the value `e` to element `v[i]`. (In other words, the above code stores a copy of `e` at index `i` in `v`.)

The contents of a vector can be printed by using a **range-for** loop. For example,

```
for (int x : v) cout << x;
```

prints all the elements of `v`. This can be read as

For every integer `x` in `v`, print `x` to `cout`.

Range-for loops can also be used to modify the elements of a vector. For example,

```
for (int & x : v) ++x;
```

adds 1 to all the elements of `v`. Note that this requires that `x` be declared to be a reference to an integer. In fact, the element of a range-for loop is declared just like a function argument, which means that it can also be a constant reference.

The general form of a range-for loop is

```
for (element_declaration : container) statement
```

As in the case of other loops, the body of a range-for loop can also be a compound statement:

```
for (element_declaration : container) {
    statements
}
```

An alternative way of performing an operation on every element of a vector is to use an ordinary `for` loop controlled by an index. For example,

```
for (int i = 0; i < v.size(); ++i) cout << v[i];
```

prints all the elements of `v`. The code `v.size()` returns the number of elements in `v`. Note that the indices in a vector range from 0 to the size of the vector minus 1.

If an operation needs to be performed on every element of a vector, a range-for loop is more convenient. But if an operation needs to be performed on only a portion of a vector, a loop controlled by an index is both more convenient and more efficient. For example,

```
for (int i = 0; i < 10; ++i) cout << v[i];
```

prints the first 10 elements of `v`.

Note that the indexing operator is not safe. When using the operator to access vector elements, it is very important to make sure that the index is valid, that is, at least 0 and less than the size of the vector. Otherwise, the operator will access a memory location outside of the memory currently occupied by the vector, possibly causing the program to crash or the value of some other variable to change. This type of bug can be very difficult to find.

Table 5.1 shows the vector operations we discussed in this section. These are also the operations we will need for the file viewer program. Several additional vector operations will be presented later in this chapter. The type `vector` is defined in the library file `vector` and included in the `std` namespace.

Study Questions

5.2.1. What is a vector?

5.2.2. In what way are vectors dynamic?

5.2.3. What operator can be used to access individual vector elements?

5.2.4. What is the general form of a range-for loop?

5.2.5. What is an example of a situation where an ordinary loop is preferable to a range-for loop for working with a vector?

Exercises

5.2.6. Suppose that `v` is a vector of integers. Write code that replaces the contents of `v` with the numbers 10, 20, ..., 1000.

5.2.7. Suppose that `v` is a vector of strings. Write a range-for loop that prints the contents of the vector to `cout`, one string per line.

5.2.8. Suppose that `v` is a vector of integers. Write a loop that prints the last ten integers of the vector to `cout`. Assume that `v` contains at least ten integers.

`vector<T> v`
`vector<T> v(n)`
`vector<T> v(n, e)`
`vector<T> v = {elements}`
: Creates a vector `v` that can hold elements of type `T`. The vector is initialized to be empty, or to contain `n` elements of type `T`, or `n` copies of element `e`, or copies of the given `elements`.

`v.size()`
: Returns the number of elements contained in vector `v`.

`v[i]`
: Returns a reference to the element at index `i` in vector `v`.

`v.push_back(e)`
: Adds a copy of element `e` to the back of vector `v`.

`v.pop_back()`
: Deletes the last element of vector `v`.

`v.clear()`
: Deletes all the elements of vector `v`.

Table 5.1: Some basic vector operations

5.2.9. Create a function called `sum` that takes as argument a vector of integers `v` returns the sum of all the integers in `v`.

5.2.10. Create a function called `fill` that takes as arguments a vector of integers `v` and an integer `x` and replaces every element of `v` by a copy of `x`.

5.2.11. Create a function called `read` that takes as arguments an input stream `in`, a vector of integers `v` and an integer `n` and fills `v` with `n` integers read from `in`. The integers read from `in` are assumed to be separated by white space. The original contents of `v` is deleted.

5.2.12. Create a function called `read` that takes as arguments an input file stream `in` and a vector of integers `v` and fills `v` with integers read from `in`. Integers are read until the end of the file is reached. The integers read from `in` are assumed to be separated by white space. The original contents of `v` is deleted.

5.3 Object-Oriented Programming

In the previous section, we learned what vectors are and some basic operations that can be performed on them. Later in this chapter, we will use vectors to implement our file viewer program. And note that we will do that without knowing how vectors are implemented. In other words, to use vectors, all we need to know is the *purpose* of a vector, that is, what data it holds (a sequence of elements) and what operations it provides (e.g., `push_back` and the indexing operator). We don't need to know how the vector works, that is, how the data is stored and how the operations work. This is a great example of abstraction. And since vectors are a data type, not a function, this is actually an example of **data abstraction**. (Recall that in the case of functions, abstraction is called procedural abstraction. See Section 4.6.)[1]

So vectors are a great example of data abstraction. But they also are a good example of *object-oriented programming*. This section briefly discusses

[1]Note that even though code that uses vectors does not depend on the implementation of vectors, only on their purpose, it is still worthwhile to learn the techniques used in a typical implementation of vectors. Mainly because these techniques have many other applications but also because learning how vectors work gives us a deeper understanding of vectors, which can help us use them more effectively. At Clarkson, the implementation of vectors is covered in the course CS142 *Introduction to Computer Science II*.

the basic idea and rationale behind object-oriented programming. It also introduces some of the terminology. This is all useful for understanding why vectors are designed the way they are.

Techniques like modularity and abstraction have been developed to help in the creation of large programs. Object-oriented programming is another one of those techniques.

The way most people usually learn to program is called **imperative programming**. In imperative programming, a program is viewed as a sequence of instructions that tell the computer what to do. Imperative programming works well with small programs but it is not as effective with large programs.

In **object-oriented programming** (OOP), a program is viewed as a collection of **objects** that work together to accomplish the overall goal of the program. Each object has a certain set of responsibilities. Objects collaborate by requesting services from each other. They do so by sending **messages** to other objects. The **receiver** of a message responds by following a predetermined **method**. The set of messages understood by an object, as well as the methods used to respond to those messages, are determined by the type, or **class**, of the object.

For example, if vectors were designed in an imperative way, then they would come with a set of functions that allow us to perform various operations on vectors. For example,

```
push_back(v, e);
```

would add a copy of `e` to the back of vector `v`. But note how this code is usually interpreted: *we* are modifying the vector. The vector itself is passive; it just sits there waiting for us to perform operations on it.

In contrast, in the object-oriented view of programming, a vector is an object with responsibilities. When needed, we *ask* a vector to add an element to its back:

```
v.push_back(e);
```

On receiving the message `push_back` with argument `e`, `v` responds by executing the corresponding method, which is determined by the fact that `v` is of class `vector`.

Another aspect of object-oriented programming is that objects are always properly initialized as soon as they are created. For example,

```
vector<int> v;
```

creates a vector of integers and automatically initializes it to be empty. On the other hand,

```
vector<int> v(n);
```

causes the vector to be initialized to contain `n` integers. This is in contrast to a declaration such as

```
int x;
```

which leaves `x` with an arbitrary value.

The automatic initialization of an object is done by a special method called a **constructor**. Like other methods, constructors can have arguments. The declaration

```
vector<int> v(n);
```

automatically calls a constructor that takes a single integer as argument. The declaration

```
vector<int> v;
```

automatically calls a constructor that has no arguments. This constructor is called the **default constructor**. In general, the object constructed by the default constructor of a class is called the **default object** of that class.

It is also possible to initialize a vector to be a copy of another vector:

```
vector<int> v(v2);
```

The constructor used in this case is called the **copy constructor**. Note that the copy constructor can also be used with the following alternate syntax:

```
vector<int> v = v2;
```

In addition to being well-suited for thinking about large, complex programs, object-oriented programming naturally leads to a high degree of modularity. First, because it automatically produces a lot of data abstraction, which results in a high degree of independence between components. Second, OOP encourages software components to delegate as many tasks as possible to other components, just as we would when organizing the members of a group of people in the real world. This tends to lead to components that are

highly cohesive, that is, well focused on a particular task. Recall that independence and cohesion are the two defining characteristics of modularity (see Section 4.6).

In pure OOP, every component of a program is an object. With languages such as C++ and Java, we use a mix of object-oriented and imperative programming. Note that in the context of C++, methods are often called **member functions**.

Study Questions

5.3.1. In OOP, how is an object viewed differently from just a piece of data?

5.3.2. What is a message? What is a method? What is a receiver?

5.3.3. What is a constructor? What is a default constructor? What is a copy constructor?

5.3.4. What is the ultimate goal of both data abstraction and OOP?

5.4 Design and Implementation of the File Viewer

We now use vectors to create the file viewer program we outlined at the beginning of this chapter. We will build the program gradually, beginning with a first version that always displays the entire file. In particular, the only two commands available in this version of the program will be `open` and `quit`. Figure 5.2 shows the main function of this version of the program. Figures 5.3 and 5.4 show the functions `display` and `open_files`.

Figure 5.5 shows the main function of a second version of the program. (Some portions of the code are omitted to allow the function to fit on one page.) In this version, the program begins by asking the user for a window height, that is, the number of lines to be displayed at one time. In addition, the commands *next* and *previous* are now implemented. Note how the implementation of these commands guarantees that they do not move too far in either direction.

Figure 5.6 shows the `display` function of the second version of the program. (The `open_file` function is the same as in the first version.) In the implementation of `display`, the variable `ix_stop_line` is created for efficiency reasons, to avoid unnecessarily redoing the addition

```
// Runs the file viewer.
int main() {
    string file_name;
    vector<string> v_lines;

    bool done = false;
    while (!done) {
        display(file_name, v_lines);

        cout << "command: ";
        char command;
        cin >> command;
        cin.get();  // '\n'

        if (command == 'q') {
            done = true;
        }
        else if (command == 'o') {
            open_file(file_name, v_lines);
        }
    }

    return 0;
}
```

Figure 5.2: The main function of a first version of the file viewer

```
// Displays the file name, all the document lines stored in the
// vector and the menu of commands.
void display(const string & file_name,
             const vector<string> & v_lines)
{
    const string long_separator =
        "_____________________________________________________________";
    const string short_separator = "________";

    cout << '\n';

    if (file_name == "")
        cout << "<no file opened>\n";
    else
        cout << file_name << '\n';

    cout << long_separator << '\n';

    for (int i = 0; i < v_lines.size(); ++i) {
        cout << setw(6) << i+1 << "  " << v_lines[i] << '\n';
    }

    cout << long_separator << '\n'
         << "  open  quit\n";
    cout << short_separator << '\n';
}
```

Figure 5.3: The `display` function

```
// Asks the user for a new file name and reads the contents of the
// file into the vector. The name of the file is returned through
// the argument file_name.
void open_file(string & file_name, vector<string> & v_lines) {
    cout << "file: ";
    getline(cin, file_name);
    std::ifstream file(file_name);
    v_lines.clear();
    string line;
    while (getline(file, line)) v_lines.push_back(line);
}
```

Figure 5.4: The `open_files` function

```
ix_current_line + window_height
```

at every iteration of the loop. Note how the function takes care not to access lines that are not present in the document.

The last version of the program we will create in this section will perform some error-checking. When the user attempts to open a file, if the file fails to open, the program will print an error message. This will be accomplished as follows. The `main` function will hold a variable `error_message` that's initially empty. If a file fails to open, the `open_files` function will assign a value to that variable. The next time it is called, the `display` function will display the error message and reset it to empty.

Figure 5.7 shows the main function of this version of the program. Figures 5.8 and 5.9 show the `open_files` and `display` functions. In the implementation of the `open` command, note that the resetting of `ix_top_line` is now done by the `open_files` function. This is because we don't want the index to be reset if the file fails to open and it is simpler for `main` to delegate that task to `open_files`.

The three versions of the file viewer we created in this section are available on the course web site under `File Viewer`.

Exercises

5.4.1. Add a *jump* command to the file viewer. This command asks the user for a line number and redisplays the file with the requested line at the top. In

```
// Runs the file viewer.
int main() {
    cout << "Window height? ";
    int window_height;
    cin >> window_height;
    cin.get();  // \n
    cout << '\n';

    string file_name;
    vector<string> v_lines;
    int ix_top_line = 0;

    bool done = false;
    while (!done) {
        display(file_name, v_lines, ix_top_line, window_height);

        cout << "command: ";
        ...

        if (command == 'q') {
            ...
        }
        else if (command == 'n') {
            if (ix_top_line + window_height < v_lines.size())
                ix_top_line += window_height;
        }
        else if (command == 'p') {
            if (ix_top_line > 0) ix_top_line -= window_height;
        }
    }

    return 0;
}
```

Figure 5.5: The main function with *next* and *previous* commands

```
// Displays the file name, the required number of lines from the
// document stored in the vector, and the menu of commands.
void display(const string & file_name,
             const vector<string> & v_lines,
             int ix_top_line,
             int window_height)
{
    const string long_separator =
        "_____________________________________________________________";
    const string short_separator = "________";

    cout << '\n';

    if (file_name == "")
        cout << "<no file opened>\n";
    else
        cout << file_name << '\n';

    cout << long_separator << '\n';

    int ix_stop_line = ix_top_line + window_height;
    for (int i = ix_top_line; i < ix_stop_line; ++i) {
        if (i < v_lines.size())
            cout << setw(6) << i+1 << "  " << v_lines[i];
        cout << '\n';
    }

    cout << long_separator << '\n'
         << "  next  previous  open  quit\n";
    cout << short_separator << '\n';
}
```

Figure 5.6: A `display` function that no longer displays the whole file

```
// Runs the file viewer.
int main() {
    cout << "Window height? ";
    ...

    string file_name;
    vector<string> v_lines;
    int ix_top_line = 0;
    string error_message;

    bool done = false;
    while (!done) {
        display(error_message, file_name, v_lines, ix_top_line,
                window_height);

        cout << "command: ";
        ...

        if (command == 'q') {
            done = true;
        }
        else if (command == 'o') {
            open_file(file_name, v_lines, ix_top_line,
                      error_message);
        }
        ...
    }

    return 0;
}
```

Figure 5.7: The main function with error handling

```
// Asks the user for a new file name, reads the contents of the
// file into the vector and resets the index to the top line to 0.
// The name of the file is returned through the argument file_name.
// In case the file does not open, the file name, vector and index
// are left unchanged and the argument error_message is assigned a
// value.
void open_file(string & file_name, vector<string> & v_lines,
               int & ix_top_line, string & error_message)
{
    cout << "file: ";
    string new_file_name;
    getline(cin, new_file_name);
    std::ifstream file(new_file_name);
    if (!file) {
        error_message = "Could not open " + new_file_name;
    }
    else {
        v_lines.clear();
        string line;
        while (getline(file, line)) v_lines.push_back(line);
        file_name = new_file_name;
        ix_top_line = 0;
    }
}
```

Figure 5.8: An `open_files` function that performs error checking

```
// Displays an error message (if any), the file name, the required
// number of lines from the document stored in the vector and the
// menu of commands. The error message is reset to empty after
// being displayed.
void display(string & error_message,
             const string & file_name,
             const vector<string> & v_lines,
             int ix_top_line,
             int window_height)
{
    const string long_separator =
        "_____________________________________________________________";
    const string short_separator = "________";

    cout << '\n';

    if (error_message != "") {
        cout << "ERROR: " + error_message << '\n';
        error_message = "";
    }

    ...
}
```

Figure 5.9: A `display` function that displays an error message

case the user enters an invalid line number *N*, the program should print the error message `ERROR: N is not a valid line number`. Modify the program as little as possible. (Make sure that the *previous* command still works after a jump.)

5.5 Multiway Branches

The main function of the file viewer program (see Figures 5.5 and 5.7) uses a sequence of `if-else` statements to figure out which command was entered by the user, and to perform the appropriate action. This is somewhat inefficient since, in many cases, this requires multiple tests to be performed. This also obscures the fact that what we are really coding here is a multiway branch, where we want a different action to be performed for each of the four possible commands.

An alternative way of coding this multiway branch is to use a different type of conditional statement called a `switch` statement, as shown in Figure 5.10. When this `switch` statement executes, the value of `command` determines which of the cases is executed.

The general form of a `switch` statement is shown in Figure 5.11. The *`expression`* must evaluate to an integer or a character (that is, a value of type `int` or `char`). Depending on that value, the `switch` statement jumps to the corresponding case and executes the specified statements. If the value of the expression does not match any of the given cases, then the statements of the `default` case are executed.

Note that the `default` case is optional. When one is not specified, then nothing happens if the value of the expression does not match any of the cases.

The `break` statements at the end of each case are also optional. Without them, the `switch` statement jumps to the case that matches the value of the expression and then executes the statements of that case and every other case that follows, until either a `break` statement or the end of the `switch` statement is encountered.

The version of the file viewer with a `switch` statement is available on the course web site under `File Viewer`.

Study Questions

5.5.1. What is the general form of a `switch` statement?

```
int main() {
    ...

    bool done = false;
    while (!done) {
        display(error_message, file_name, v_lines, ix_top_line,
                window_height);

        cout << "command: ";
        char command;
        cin >> command;
        cin.get(); // '\n'

        switch (command) {
            case 'q': {
                done = true;
                break;
            }
            case 'o': {
                open_file(file_name, v_lines, ix_top_line,
                          error_message);
                break;
            }
            case 'n': {
                if (ix_top_line + window_height < v_lines.size())
                    ix_top_line += window_height;
                break;
            }
            case 'p': {
                if (ix_top_line > 0) ix_top_line -= window_height;
                break;
            }
        }
    }

    return 0;
}
```

Figure 5.10: Using a `switch` statement in the file viewer

```
switch (expression) {
    case value1: {
        statements
        break;
    }
    case value2: {
        statements
        break;
    }
    ...
    default: {
        statements
        break;
    }
}
```

Figure 5.11: The general form of a `switch` statement

Exercises

5.5.2. Redo Exercise 2.1.5 but this time use a `switch` statement.

5.6 More on Vectors

In Section 5.2, we learned several basic operations that can be performed on vectors. These operations were sufficient for the implementation of our file viewer program. But vectors support several additional operations, as shown in Tables 5.2 and 5.3. In fact, the class supports several more, as described in a reference such as [CPP].[2]

Note that in these tables, the vector operations are described by using an object-oriented perspective. For example, the description of `push_back` does not say that this operation adds an element to the back of the vector. Instead, it says that `push_back` asks the vector to add an element to its back.

The second constructor **value-initializes** the elements of the vector. This implies that if `T` is a class with at least one programmer-defined constructor,

[2]Some of these operations involve concepts that we have not covered, such as iterators, ranges and capacity. You can ignore these operations for now.

`vector<T> v`
`vector<T> v(n)`
`vector<T> v(n, e)`
`vector<T> v({elements})`
`vector<T> v(v2)`
: Creates a vector `v` that can hold elements of type `T`. The vector is initialized to be empty, or to contain `n` value-initialized elements of type `T`, or `n` copies of element `e`, or copies of the given `elements`, or copies of all the elements of vector `v2`.

`v.size()`
: Asks vector `v` for the number of elements it currently contains.

`v[i]`
: Returns a reference to the element at index `i` in vector `v`.

`v.front()`
`v.back()`
: Asks vector `v` for a reference to its front or back element.

`v.resize(n)`
`v.resize(n, e)`
: Asks vector `v` change its size to `n`. If `n` is smaller than the current size of `v`, the last elements of `v` are deleted. If `n` is larger than the current size, then `v` is padded with either copies of the default object of class `T` or with copies of element `e`.

`v.push_back(e)`
: Asks vector `v` to add a copy of element `e` to its back end.

`v.pop_back()`
: Asks vector `v` to delete its last element.

`v1 = v2`
`v1 = {elements}`
: Makes vector `v1` contain copies of all the elements of vector `v2`, or copies of the given `elements`. Returns a reference to `v1`.

Table 5.2: Some basic vector operations

`v.empty()`
: Asks vector `v` if it is empty.

`v.max_size()`
: Asks vector `v` for the maximum number of elements it can contain.

`v.clear()`
: Asks vector `v` to delete all its elements.

`v.assign(n, e)`
`v.assign({elements})`
: Asks vector `v` to change its contents to `n` copies of element `e`, or to copies of the given `elements`.

`v1.swap(v2)`
: Asks vector `v1` to swap contents with vector `v2` (without any elements being copied).

Table 5.3: Some additional vector operations

such as `string` and `vector`, then the elements are initialized with the default constructor of the class. On the other hand, if `T` is a primitive data type, then the elements are initialized to 0.

The argument of the fourth constructor is an **initializer list**, which is given by a so-called *brace list* of elements separated by commas. That constructor can also be used with the following syntax:

```
vector<T> v = {elements}
```

The fifth constructor is known as a **copy constructor**. It can also be used with the equal sign syntax:

```
vector<T> v = v2
```

Note that vectors have `push_back` and `pop_back` methods but no `push_front` or `pop_front`. The reason has to do with efficiency: it is possible to implement the operations at the back efficiently but doing so at the front is more difficult.

As mentioned earlier, vectors are dynamic in the sense that they can grow and shrink as needed. Even then, vector do have a maximum size, which

is related to the value of the largest integer that can be stored in an `int` variable. However, that maximum size, which can be retrieved by the method `max_size`, is typically much larger than what's needed in most applications. For example, for a vector of integers, a typical limit is approximately 1 billion elements.

Vectors are also said to be **generic** because they can hold elements of various types. In fact, vectors belong to a portion of the C++ standard library called the *Standard Template Library* (STL) and the word *template* in the name of this library refers to a C++ construct that allows the creation of generic software components. The STL includes several other generic components, including other containers as well as functions that implement useful algorithms.

Study Questions

5.6.1. In what way are vectors generic?

5.6.2. Why do STL vectors have no `push_front` or `pop_front` methods?

Exercises

5.6.3. Experiment with vectors by writing a test driver that creates more than one type of vector and uses all the methods shown in Tables 5.2 and 5.3.

5.7 More on Strings

We have already used strings many times in these notes. And we know a few operations that can be performed on strings. But it turns out that just like vectors, strings support a large number of operations. And just like the type `vector`, the type `string` is also a class, with constructors and methods.

Tables 5.4 to 5.7 show several `string` operations. Many more are described in a reference such as [CPP].[3]

The second constructor provides a way for converting literal strings into `string` objects. And this conversion is often done automatically: in any situation where the compiler expects a `string` object and a literal string

[3]Once again, some of these operations involve concepts we still have not covered, such as iterators, ranges, exceptions and capacity. You can ignore these operations for now.

`string s`
`string s(s2)`
`string s(s2, i, n)`
`string s(n, c)`
: Creates a string `s` and initializes it to be empty, or a copy of string `s2`, or a copy of the substring of `s2` that starts at index `i` and is of length `n`, or `n` copies of character `c`. The argument `s2` can also be a literal string.

`s.length()`
`s.size()`
: Asks string `s` for the number of characters it currently contains.

`s.empty()`
: Asks string `s` if it is empty.

`s.max_size()`
: Asks string `s` for the maximum number of characters it can contain.

`s[i]`
: Returns a reference to the character at index `i` in string `s`.

`s1 = s2`
: Makes string `s1` a copy of string `s2`. The right operand can also be a C string or a single character. Returns a reference to `s1`.

`s1.swap(s2)`
: Asks string `s1` to swap contents with string `s2`.

`s.clear()`
: Asks string `s` to delete all its characters.

`s1 op s2`
: Compares string `s1` with string `s2` where the operator `op` is one of ==, !=, <, >, <= or >=. Uses alphabetical order. Returns `true` or `false`. One of the operands must be a `string` object but the other can be a literal string.

Table 5.4: Some string operations (part 1 of 4)

`s1 + s2`
: Returns a string that consists of a copy of string `s1` followed by a copy of string `s2`. One of the operands must be a `string` object but the other can be a literal string or a single character.

`s1 += s2`
: Appends a copy of string `s2` to string `s1`. The right operand can also be a literal string or a single character. A reference to `s` is returned.

`s.resize(n)`
`s.resize(n, c)`
: Asks string `s` to change its size to `n`. If `n` is smaller than the current size of `s`, the last characters of `s` are erased. If `n` is larger, `s` is padded with the null character (`\0`) or with copies of character `c`.

`s.substr(i, m)`
`s.substr(i)`
: Asks string `s` for a copy of the substring that starts at index `i`, and is of length `m` or ends at the end of the string.

`s.insert(i, s2)`
: Asks string `s` to insert into itself, at index `i`, a copy of string `s2`. The second argument can be of any of the forms accepted by the constructors. A reference to `s` is returned.

`s.replace(i, m, s2)`
: Asks string `s` to replace the substring of length `m` that starts at index `i` by a copy of string `s2`. The third argument can be of any of the forms accepted by the constructors. A reference to `s` is returned.

`s.erase(i, m)`
`s.erase(i)`
: Asks string `s` to delete `m` characters starting at index `i`, or all the characters from index `i` to the end of the string. A reference to `s` is returned.

Table 5.5: Some string operations (part 2 of 4)

`s1.find(s2)`
`s1.find(s2, i)`
: Asks string `s1` for the index of the first occurrence of string `s2` as a substring. In the first version, the entire string is searched. In the other, the search starts at index `i`. The argument `s2` can also be a literal string or a single character. If the search is unsuccessful, the constant `string::npos` ("not a position") is returned.

`s1.rfind(s2)`
`s1.rfind(s2, i)`
: Similar to `find` except that the search is for the *last* occurrence and that the search *ends* at index `i`.

`s.c_str()`
: Asks string `s` for a C string that contains the same characters as `s`. (See text for explanation.)

`stoi(s)`
`stod(s)`
: Converts string `s` into a number. Starting from the beginning of the string, skips whitespace and then converts as many characters as possible into a number. The first version returns an `int` while the second version returns a `double`. The string is not modified.

`to_string(x)`
: Returns a string representing number `x`. The argument can be of any of the usual numeric types.

Table 5.6: Some string operations (part 3 of 4)

`stream << s` Outputs the characters of string `s`. Returns a reference to the stream.
`stream >> s` Reads characters into string `s`. Skips leading white space and stops reading at white space (blank, tab or newline). That terminating character is not read. Returns a reference to the stream.
`getline(stream, s)` Reads characters into string `s` until then end of the current line. The newline character is read but not included in `s`. Returns a reference to the stream.

Table 5.7: Some string operations (part 4 of 4)

is provided instead, the compiler will automatically use the constructor to convert the literal string into a `string` object. This is called an **implicit conversion**.

The `find` and `rfind` methods return the special value `string::npos` ("not a position") in case the search is unsuccessful. If `s` is a string, that constant can also be accessed as `s.npos`.

The `c_str` method converts a `string` object into a type of string called a *C string*. Without getting into the details, C strings are the standard way in which strings are stored in the older language C from which C++ is derived. C++ strings (the class `string`) are much more convenient and also much safer to use than C strings. But it is still possible to come across situations where it is necessary to use C strings. This is when the `c_str` method is useful.[4]

The functions `stoi`, `stod` and `to_string` can be used to easily perform

[4]For example, consider the file stream constructor that takes a file name as argument, as in

```
ifstream ifs(file_name);
```

Before C++11, the latest version of C++, the file name argument had to be a C string. So it was often necessary to convert the file name from a C++ string to a C string:

```
ifstream ifs(file_name.c_str());
```

This may still be necessary with compilers that don't fully support C++11.

```
#include <cstdlib>
#include <string>

int stoi(const std::string & s)
{
    return std::atoi(s.c_str());
}
```

Figure 5.12: An implementation of `stoi`

```
#include <sstream>
#include <string>

std::string to_string(int x)
{
    std::ostringstream oss;
    oss << x;
    return oss.str();
}
```

Figure 5.13: An implementation of `to_string`

conversions between numbers and their string representations. But note that some compilers do not properly support these functions.[5]

Some of the string operations presented in this section can be used to simplify the `display` function of the file viewer program. The revised code is shown in Figure 5.14. The new string operations that are used are one of the constructors and the `empty` method. (Compare with the code shown in Figures 5.6 and 5.9.)

The version of the file viewer with the new string operations is available

[5]These functions are new to C++11 but some compilers that support C++11 do not properly support these functions. (As of February 14, this was still the case with the version of the g++ compiler that comes with the Code::Blocks IDE.) In this case, the `stoi` function can be implemented by using the C string `atoi` function as shown in Figure 5.12. The `stod` function can be implemented by using `atof` in the same way. Figure 5.12 shows the easiest implementation of `to_string`. This implementation uses a *string stream*, a type of stream that, at Clarkson, is normally covered in the course CS142 *Introduction to Computer Science II*.

```
void display(string & error_message,
             const string & file_name,
             const vector<string> & v_lines,
             int ix_top_line,
             int window_height)
{
    const string long_separator(50, '-');
    const string short_separator(8, '-');

    cout << '\n';

    if (!error_message.empty()) {
        cout << "ERROR: " + error_message << '\n';
        error_message.clear();
    }

    if (file_name.empty())
        cout << "<no file opened>\n";
    else
        cout << file_name << '\n';

    ...
}
```

Figure 5.14: A version of `display` that uses more string operations

on the course web site under `File Viewer`.

Study Questions

5.7.1. How can we convert a literal string to a `string` object?

5.7.2. How can we convert a C++ string into a C string?

Exercises

5.7.3. Experiment with C++ strings by writing a test driver that uses all the string operations presented in this section.

5.7.4. Create a function `println(s)` that takes a C++ string as argument and behaves exactly as the code `cout << s << endl`. (But don't use this code in implementing the function.)

5.7.5. Write a code segment that starts with a string that contains a person's name in the format `"John Doe"`. Assume that the name contains a single blank space. Your code should produce another string that contains the same name but in the format `"Doe, John"`. *Hint*: Consider using the method `find` and the operator `+=`.

5.8 A Simple Text Editor

In this section, we will further illustrate the usefulness of vectors by expanding our file viewer into a simple text editor. This program will allow the user to add, remove and replace entire lines of text. It won't allow the user to edit the contents of the lines. For example, it won't be possible to insert a word directly in the middle of a line. The only way to accomplish this will be by replacing the entire line by a new one.

Here are more details on how the editor works. The editor has a *buffer* that contains lines of text, usually an edited copy of the contents of some file. The editor displays the name of the file, if any, followed by a certain number of lines from the buffer, surrounded by a border. A cursor indicates the position of the current line. Below the text, a menu of commands is displayed followed by the prompt "`choice:`". The user types the first letter of a command, the command executes and everything is redisplayed. Some commands prompt

```
co-op.txt
_____________________________________________________________

  1 List for Co-op
  2
  3 bread
  4 yogurt
> 5 cumin
  6 black beans
  7 chick pea flour
  8 toothpaste
  9


_____________________________________________________________

  next      jump     insert  open  quit
  previous  replace  delete  save
_______
choice: i
new line: ginger
```

Figure 5.15: Sample user interface of the text editor

the user for more information. Figure 5.15 shows what the user interface looks like. The available editor commands are described in Figure 5.16.

Note that the displayed contents of the buffer always includes an extra empty line we will call the *end line*. In Figure 5.15, the end line is the one numbered 9. That line is not really part of the buffer but the cursor can move there. This allows the user to insert a new line at the end of the buffer. In addition, in the case of an empty buffer, the end line gives the cursor something to point to.

As mentioned at the beginning of this section, the text editor can be created by extending the file viewer. In this section, we will highlight the new code.

Figure 5.17 shows some key portions of the `main` function of the program. In addition to keeping track of the index of the top line, the function also keeps track of the index of the current line. Both indices are initialized to 0 and passed to the `display` function.

The revised portions of the `display` function are shown in Figure 5.18. The loop that displays the lines was revised to print a cursor at the beginning

`next`	The next line becomes the current line.
`previous`	The previous line becomes the current line.
`jump`	Asks for a line number and makes that line become the current line.
`replace`	Asks for a new line and replaces the current line.
`insert`	Asks for a new line and inserts it before the current line.
`delete`	Deletes the current line.
`open`	Asks for a file name and reads that file into the buffer.
`save`	Asks for a file name and saves the contents of the buffer to that file.
`quit`	Stops the editor.

Figure 5.16: The commands of the text editor

of the current line and to print the end line, which is the line that has index equal to the size of the vector.

Figure 5.19 shows the `switch` statement of the `main` function of the program. In each case, an auxiliary function is called to perform the required task.

Figures 5.20 and 5.21 show the functions `move_to_next_line`, `move_to_previous_line` and `save_file`. This code is fairly straightforward.

Figure 5.22 shows the `insert_line` and `erase_line` functions. These functions use the vector methods `insert` and `erase`. The argument of `erase` and the first argument of `insert` is

```
v_lines.begin() + ix_current_line
```

This requires some explanation.

Both methods require an argument that indicates the position where the insertion or deletion should be performed. This argument could be an index, in which case the `insert` and `erase` methods could have been used simply as follows:

```
int main() {
    cout << "Window height? ";
    ...

    string file_name;
    vector<string> v_lines;
    int ix_top_line = 0;
    int ix_current_line = 0;
    string error_message;

    bool done = false;
    while (!done) {
        display(error_message, file_name, v_lines, ix_top_line,
                ix_current_line, window_height);

        cout << "command: ";
        ...

        switch (command) {
            ...
        }
    }

    return 0;
}
```

Figure 5.17: The `main` function of the text editor

```
void display(string & error_message,
             const string & file_name,
             const vector<string> & v_lines,
             int ix_top_line,
             int ix_current_line,
             int window_height)
{
    ...

    int ix_stop_line = ix_top_line + window_height;
    for (int i = ix_top_line; i < ix_stop_line; ++i) {
        if (i <= v_lines.size()) {
            if (i == ix_current_line)
                cout << '>';
            else
                cout << ' ';
            cout << setw(6) << i+1;
            if (i < v_lines.size())
                cout << "  " << v_lines[i];
        }
        cout << '\n';
    }

    cout << long_separator << '\n'
         << "  next      jump     insert  open  quit\n"
         << "  previous  replace  delete  save\n"
         << short_separator << '\n';
}
```

Figure 5.18: The `display` function of the text editor

```
switch (command) {
    case 'q': {
        done = true;
        break;
    }
    case 'o': {
        open_file(file_name, v_lines, ix_top_line,
                  ix_current_line, error_message);
        break;
    }
    case 'n': {
        move_to_next_line(v_lines, ix_current_line,
                          ix_top_line, window_height);
        break;
    }
    case 'p': {
        move_to_previous_line(ix_current_line,
                              ix_top_line);
        break;
    }
    case 'd': {
        erase_line(v_lines, ix_current_line);
        break;
    }
    case 'i': {
        insert_line(v_lines, ix_current_line, ix_top_line,
                    window_height);
        break;
    }
    case 's': {
        save_file(v_lines, file_name, error_message);
        break;
    }
}
```

Figure 5.19: The `switch` statement of the `main` function of the text editor

```
void move_to_next_line(const vector<string> & v_lines,
                       int & ix_current_line,
                       int & ix_top_line,
                       int window_height)
{
    if (ix_current_line < v_lines.size()) {
        ++ix_current_line;
        // check if window needs to be scrolled down
        if (ix_current_line >= ix_top_line + window_height)
            ++ix_top_line;
    }
}

void move_to_previous_line(int & ix_current_line,
                           int & ix_top_line)
{
    if (ix_current_line > 0) {
        --ix_current_line;
        // check if window needs to be scrolled up
        if (ix_current_line < ix_top_line)
            --ix_top_line;
    }
}
```

Figure 5.20: The `move_to_next_line` and `move_to_previous_line` functions

```
void save_file(const vector<string> & v_lines, string & file_name,
               string & error_message)
{
    cout << "file: ";
    string new_file_name;
    getline(cin, new_file_name);
    std::ofstream file(new_file_name);
    if (!file) {
        error_message = "Could not open " + new_file_name;
    }
    else {
        for (const string & s : v_lines) file << s << '\n';
        file_name = new_file_name;
    }
}
```

Figure 5.21: The `save_file` function

```
void insert_line(vector<string> & v_lines,
                 int & ix_current_line,
                 int & ix_top_line,
                 int window_height)
{
    cout << "new line: ";
    string new_line;
    getline(cin, new_line);
    v_lines.insert(v_lines.begin() + ix_current_line, new_line);
    move_to_next_line(v_lines, ix_current_line, ix_top_line,
                      window_height);
}

void erase_line(vector<string> & v_lines,
                int & ix_current_line)
{
    if (ix_current_line < v_lines.size())
        v_lines.erase(v_lines.begin() + ix_current_line);
}
```

Figure 5.22: The `insert_line` and `erase_line` functions

```
v_lines.insert(ix_current_line, new_line);
```

and

```
v_lines.erase(ix_current_line);
```

But the STL includes several other containers for which numerical indices are not an efficient way to specify positions. Therefore, for the sake of uniformity, the `insert` and `erase` methods use a different mechanism for specifying positions, a mechanism that can be used efficiently with all the different types of containers included in the STL.

This mechanism is that of an **iterator**. An iterator is a general concept and each type of STL container supplies its own type of iterator, implemented in a way that's appropriate for that container.

The main purpose of an iterator is simply to mark a position within a container. When an iterator marks the position of an element in a container, we say that the iterator *points* to that element.

In both the `insert_line` and `erase_line` functions, we need an iterator that points to the current line. Such an iterator is obtained by adding the index of the current line to the *begin* iterator of the vector, which is an iterator that points to the first element of the vector:

```
v_lines.begin() + ix_current_line
```

In general, `v.begin() + i` is an iterator that points to the element in `v` that has index `i`. In other words, `v.begin() + i` points to `v[i]`.

Table 5.8 shows some vector operations that involve iterators. (There are others.)

The complete source code of the text editor is available on the course web site under `TextEditor`.

Exercises

5.8.1. Modify the text editor as described below. Change the original program as little as possible. Make sure it remains modular. Document any new functions.

a) Implement the *replace* command. If the current line is the end line, the line entered by the user should be added at the end of the buffer, as if the user had inserted a line at that position.

`v.insert(itr, e)`
`v.insert(itr, {elements})`
Asks vector `v` to insert, at the position indicated by the iterator `itr`, a copy of element `e` or copies of the given `elements`.

`v.erase(itr)` Asks vector `v` to delete the element that `itr` points to.
`v.begin()` Asks vector `v` for a begin iterator.

Table 5.8: Some vector operations that involve iterators

b) Implement the *jump* command. This command asks the user for a line number and redisplays the buffer with the requested line at the top. The requested line also becomes the current line. In case the user enters an invalid line number *N*, the program should print the error message `ERROR: N is not a valid line number`.

c) Modify the *save* command so it uses the current file, if any, as a default value. The default value, if any, should be displayed as follows:

```
choice: s
file name: [co—op.txt]
```

If the user enters an empty file name, then the default is used.

d) Add a *clear* command that empties the buffer. The command also resets the file name so that the string <`no file opened`> is displayed just like when the program is started.

e) Add *Next* and *Previous* commands that cause the editor to display the next or previous "pages", just like the *next* and *previous* commands of the file viewer. After these commands are executed, the new top line becomes the current line.

f) Add an *Append* command that inserts a new line *after* the current one. If the current line is the end line, add a blank line *and* the new line to the end of the buffer. After the insertion, the new line becomes the current line.

g) The *quit* command currently stops the editor without ensuring that the buffer was saved to a file. Similarly, the *open* and *clear* commands delete the current contents of the buffer without ensuring that this contents has been saved. Fix this. In case the buffer has not been saved since it was last modified, *clear*, *open* and *quit* should ask the user if he or she wants to save the current contents of the buffer. If the user says yes, the *save* command should be executed. *Hint*: Add to the program a Boolean variable that indicates if the buffer was saved since the last modification.

5.9 Adding More Error-Checking to the File Viewer

Our file viewer program performs some error checking: when executing the *open* command, it checks it the file opens and prints an error message in case it doesn't. But the program behaves badly in case the user doesn't enter a number for the window height or a valid command when prompted for one. We will fix this in this section.

The reading of the window height and commands is done in the `main` function. In order not to clutter that function with error checking code, we will remove the reading of those values from `main` and delegate it to two new auxiliary functions `get_window_height` and `get_command`.

Figure 5.23 shows the revised `main` function. The `get_window_height` function keeps asking the user until he or she enters a valid window height. The `get_command` function, on the other hand, will prompt the user for a command only once and set the `error_message` argument in case the command entered by the user is not valid.

Figure 5.24 shows another way in which this version of the `main` function can be coded. In case the `get_command` function sets the `error_message`, a `continue` statement is executed. This has the effect of skipping the remainder of the body of the `while` loop and returning control to the top of the loop. In other words, a `continue` statement stops the execution of the current iteration of the loop and moves to the next iteration.

In general, `continue` statements are used to simplify code. In our case, the main advantage is that with a `continue` statement, we don't need to indent the `switch` statement and nest it within an `if` statement. In other

```
int main() {
    int window_height = get_window_height();

    ...

    bool done = false;
    while (!done) {
        display(error_message, file_name, v_lines, ix_top_line,
                window_height);

        char command = get_command(error_message);
        if (error_message == "") {
            switch (command) {
                ...
            }
        }
    }

    return 0;
}
```

Figure 5.23: A `main` function that delegates the reading of the window height and commands

```
int main() {
    int window_height = get_window_height();

    ...

    bool done = false;
    while (!done) {
        display(error_message, file_name, v_lines, ix_top_line,
                window_height);

        char command = get_command(error_message);
        if (error_message != "") continue;

        switch (command) {
            ...
        }
    }

    return 0;
}
```

Figure 5.24: A `main` function that uses a `continue` statement

words, the `continue` statement makes the error checking code less obtrusive.

Note that if a `continue` statement occurs in nested loops, the `continue` statement applies only to the innermost loop that contains the `continue` statement. In the case of `while` and `do-while` loops, after a `continue` statement is executed, the condition of the loop is evaluated. In the case of a `for` loop, the update statement is executed first.

Figure 5.25 shows an implementation of the `get_window_height` function. Recall that if the reading of the window height fails, the stream `cin` enters an error state, which causes `cin` to evaluate to `false`. Once a stream is in an error state, every subsequent reading operation on that stream will fail. To be able to read from the stream again, it is necessary to *clear* the error state by sending the message `clear` to the stream.

In case a valid window height is extracted from the stream, the `get_window_height` function then checks that the rest of the line is blank. This is done with a range-for loop that examines each character of the rest of the line. If one is found not to be blank, the `all_blank` variable is set to `false` and the execution of the range-for loop is terminated by using a `break` statement. If all the characters in the rest of the line are blank, then the `all_blank` variable will still be `true` when the execution of the range-for loop terminates.

A `break` statement is similar to a `continue` statement in that it only affects the innermost, enclosing loop. But while a `continue` statement only causes the execution of the current iteration of a loop to be terminated, a `break` statement terminates the execution of the entire loop.

The `get_window_height` can be simplified by making it delegate to another function the task of checking that the rest of the line is blank. The revised function is shown in Figure 5.26. Figure 5.27 shows an implementation of the `all_blank` function.

Figure 5.28 shows an implementation of the `get_command` function. This code is straightforward.

The latest version of the file viewer is available on the course web site under `File viewer` as `file_viewer_2_0.cpp`.

Study Questions

5.9.1. What is the effect of a `continue` statement?

5.9.2. What is the effect of a `break` statement?

```
int get_window_height()
{
    int window_height = 0;
    bool done = false;
    while (!done) {
        cout << "Window height? ";
        cin >> window_height;
        if (!cin || window_height <= 0) {
            cout << "Please enter a positive number.\n\n";
            cin.clear();
            // flush rest of line
            string rest_of_line;
            getline(cin, rest_of_line);
            continue;
        }

        // window height is good; check that rest of line is blank
        string rest_of_line;
        getline(cin, rest_of_line);
        bool all_blank = true;
        for (char c : rest_of_line) {
            if (c != ' ') {
                all_blank = false;
                break;
            }
        }
        if (!all_blank) {
            cout << "Please enter a positive number and "
                 << "nothing else.\n";
            continue;
        }

        // all good
        done = true;
    }
    return window_height;
}
```

Figure 5.25: The `get_window_height` function

```
int get_window_height()
{
    int window_height = 0;
    bool done = false;
    while (!done) {
        cout << "Window height? ";
        cin >> window_height;
        if (!cin || window_height <= 0) {
            cout << "Please enter a positive number.\n\n";
            cin.clear();
            // flush rest of line
            string rest_of_line;
            getline(cin, rest_of_line);
            continue;
        }

        // window height is good; check that rest of line is blank
        string rest_of_line;
        getline(cin, rest_of_line);
        if (!all_blank(rest_of_line)) {
            cout << "Please enter a positive number and nothing "
                 << "else.\n";
            continue;
        }

        // all good
        done = true;
    }
    return window_height;
}
```

Figure 5.26: A simplified `get_window_height` function

```
// Returns true if the argument contains only blank spaces.
bool all_blank(const string & s)
{
    for (char c : s) if (c != ' ') return false;
    return true;
}
```

Figure 5.27: The `all_blank` function

```
char get_command(string & error_message)
{
    cout << "command: ";
    string command;
    cin >> command;

    // check that rest of line is blank
    string rest_of_line;
    getline(cin, rest_of_line);
    if (!all_blank(rest_of_line)) {
        error_message = "Invalid command";
        return 'x';
    }

    // check that command is valid
    if (command == "n" || command == "next")
        return 'n';
    else if (command == "o" || command == "open")
        return 'o';
    else if (command == "p" || command == "previous")
        return 'p';
    else if (command == "q" || command == "quit")
        return 'q';

    // command is not valid
    error_message = "Invalid command";
    return 'x';
}
```

Figure 5.28: The `get_command` function

`T a[N]`
`T a[N] = {elements}`
`T a[] = {elements}`
Creates an array `a` that can hold elements of type `T`. The array is initialized to contain `N` default-initialized elements of type `T`, or copies of the given `elements`. In case the number of given `elements` is less than `N`, the remaining elements of `a` are value-initialized. `N` must be a compile-time constant.

`a[i]` Returns a reference to the element at index `i` in array `a`.

Table 5.9: Some basic array operations

Exercises

5.9.3. Add to the text editor the same error checking that we added to the file viewer in this section.

5.10 Arrays

In this chapter, we learned that C++ vectors can be used to store sequences of elements. But there is a simpler alternative: ordinary arrays. Table 5.9 shows how arrays can be created and how their elements can be accessed.

In the first form of array declaration, the elements of the array are **default-initialized**. This implies that if `T` is a class, then the elements are initialized with the default constructor of the class. On the other hand, if `T` is a primitive data type, then the elements are not initialized.

In the second form of array declaration, elements that are not given explicit initial values are value-initialized. Recall that this implies that if `T` is a class with at least one programmer-defined constructor, such as `vector` and `string`, then these elements are initialized with the default constructor of the class. On the other hand, if `T` is a primitive data type, then these elements are initialized to 0.

Note that the list of given elements can be empty:

```
T a[N] = {}
```

```
void display(const int a[10])
{
    for (int i = 0; i < 10; ++i) cout << a[i] << ' ';
    cout << '\n';
}
```

Figure 5.29: A function that displays the elements of an array of size 10

In that case, all the array elements are value-initialized.

The third form of array declaration produces an array that's just large enough to contain the given elements.

Note that in the last two forms of array declaration, the equal sign can be omitted:

```
T a[N]{elements}
T a[]{elements}
```

This is common practice especially when a size is given and the list of elements is empty:

```
T a[N]{}
```

Range-for loops can be used with arrays. For example,

```
for (int x : a) cout << x;
```

prints all the elements of `a`. However, range-for loops can only be used when the array is a local variable of the function that contains the loop. In other situations, an ordinary `for` loop controlled by indices must be used. For example, Figure 5.29 shows a function that displays the elements of an array of size 10. A range-for loop cannot be used here because the array is an argument of the function.

When the function in Figure 5.29 is called, its argument must be an array of size 10. This implies that the function can only display arrays of that size. It is possible to create a more flexible function by not specifying the size of the array argument, as shown in Figure 5.30. But note that it is then necessary to also pass the size of the array as an argument to the function. This is because there is usually no reliable way for a function to determine the size of an array argument.

```
void display(const int a[], int n)
{
    for (int i = 0; i < n; ++i) cout << a[i] << ' ';
    cout << '\n';
}
```

Figure 5.30: A function that displays the elements of an array of any size

Note that array arguments are always passed by reference. This is why the array arguments of the `display` functions were declared constant. Recall that this causes the arrays to be passed by constant reference and protects those arrays from accidental modification.

Ordinary C++ arrays have two advantages over vectors. One is that they are simpler to learn. The other is that in some cases, the array indexing operator runs slightly more quickly than the vector indexing operator.

However, ordinary arrays have several major weaknesses compared to vectors. In fact, vectors were developed precisely to address these weaknesses.

Perhaps the main weakness of ordinary C++ arrays is that they have a predetermined size. Predetermined here means determined at compile time, before the execution of the program. The size of an array is also fixed: it cannot change during the execution of the program. This is a significant limitation that can make a program fail in case an array is too small, or waste memory in case an array is unnecessarily large.

In contrast, as we know, vectors can be declared to be of any size and that size can be changed as needed during the execution of the program, by using methods such as `resize`, `push_back` and `assign`.

In C++, it is possible to create arrays that have a size that's determined at run time. However, this involves low-level techniques that are somewhat inconvenient and prone to errors.[6] In any case, all C++ arrays, no matter how they are created, have several other weaknesses.

First, arrays are not aware of their size. In particular, as we said earlier, there is usually no reliable way for a function to determine the size of an array argument. This is why we must typically pass the size of an array as a separate additional argument, as we did with the `display` function. One danger of this setup, however, is that nothing guarantees that the value of the size argument

[6] At Clarkson, these techniques are covered in the course CS142 *Introduction to Computer Science II*.

is correct. In contrast, vectors are aware of their size and whenever we need to know that size, we just have to ask by using the method `size`.

Second, the usual operators, such as =, == and <, don't work with arrays. (Actually, they can sometimes work but they don't do what you would expect.) In contrast, vectors support all the usual operators, including =, == and <.

Third, it is not easy to have a function return a copy of an array. In particular, the return type of a C++ function cannot be an array. It is possible to get around this problem but, once again, this involves low-level techniques that are inconvenient and prone to errors. In contrast, functions can return copies of vectors just as if they were a value of any of the primitive data types.[7]

To summarize, ordinary C++ arrays have the following four weaknesses: (1) they have a fixed, predetermined size, (2) they don't know their size, (3) they don't support the usual operators and (4) functions cannot easily return copies of arrays.

Study Questions

5.10.1. What are two advantages of arrays over vectors?

5.10.2. What are four advantages of vectors over ordinary arrays?

Exercises

5.10.3. Modify the file viewer by using an array to implement the `Buffer`. Use an array of size 100,000. In case a file called X is too large, the program should print the error message *ERROR: file X is too large* and redisplay the previous file.

5.10.4. Repeat the previous exercise but this time, in case the file is too large, have the program use the array to store part of the file. When another part is needed, the program reopens the file to read that part and store it in the array.

5.10.5. Suppose that `a` is a array of integers of size `N`, where `N` is a compile-time constant. Write code that replaces the contents of `a` with the numbers 10, 20, 30, ...

[7]Note, however, that this is something that is usually done only with small vectors. It is more efficient to avoid the copying of the vector by passing it to the function as a reference argument.

5.10.6. Create a function called `sum` that takes as arguments an array of integers `a` and its size `n` and returns the sum of all the integers in `a`.

5.10.7. Create a function called `fill` that takes as arguments an array of integers `a`, its size `n` and an integer `x` and replaces every element of `a` by a copy of `x`.

5.10.8. Create a function called `read` that takes as arguments an input stream `in`, an array of integers `a` and an integer `n` and fills `a` with `n` integers read from `in`. The integers read from `in` are assumed to be separated by white space. The function assumes that `a` is of size at least `n`. The first `n` elements of `a` are set by the function. The others are left unchanged.

Chapter 6

Structures

In this chapter, we will revisit two programs we created earlier in these notes. More precisely, we will extend the pay calculator and improve the design of the text editor. In the process, we will learn how to create structures.

6.1 Extending the Pay Calculator

In the latest version of the pay calculator, the input file read by the program specifies, for each employee, the number of hours that the employee worked on each day of the pay period. An example is shown in Figure 6.1. Each line in this file consists of an employee number followed by exactly seven numbers of hours.

In this section, we will extend the pay calculator so it includes the computation of the number of hours each employee works each day. In other words, the input file read by the program will now only specify the times when the employee started and stopped working each day, as illustrated in Figure 6.2.

Figure 6.3 shows the `main` function of the pay calculator. It should be clear that to extend the pay calculator as we just described, all we will need

```
12 7 8 7.5 7.75 8.5 0 0
23 8 8 8.25 9 5 3 0
37 5 5.5 6 5 5 2 3
```

Figure 6.1: An input file for the pay calculator

```
12
in 9:00
out 16:00
in 9:00
out 17:00
in 8:45
out 16:15
in 9:00
out 16:45
in 9:00
out 17:30
stop

23
in 9:00
out 17:00
in 9:00
out 17:00
in 9:00
out 17:15
in 9:00
out 18:00
in 9:00
out 15:00
in 14:15
out 17:15
stop
```

Figure 6.2: A revised input file

```
int main() {
    std::ifstream ifs_wages;
    std::ifstream ifs_hours;
    std::ofstream ofs_pay;
    string hours_file_name;
    string pay_file_name;

    if (!open_files(ifs_wages, ifs_hours, ofs_pay, hours_file_name,
                    pay_file_name))
        return 1;

    int employee_number;
    while (ifs_hours >> employee_number) {
        double num_hours = read_hours(ifs_hours);
        double wage = read_wage(ifs_wages);
        double pay = compute_pay(num_hours, wage);
        print_pay(employee_number, num_hours, pay, ofs_pay);
    }

    cout << "\nHours read from " << hours_file_name
         << " and pay written to " << pay_file_name << ".\n";

    return 0;
}
```

Figure 6.3: The `main` function of the pay calculator

to do is modify the implementation of the `read_hours` function. (The fact that only one function of the program will need to be modified to achieve this extension is a very good illustration of the benefits of modularity.)

Figure 6.4 shows the current implementation of the `read_hours` function. It reads the various numbers of hours and adds each one of them to a running total.

Figure 6.5 shows how the `read_hours` function can be modified to achieve the extension describe in this section. While it hasn't reached the keyword *stop*, the function reads a start time and a stop time, computes their difference, in hours, and adds that number to the running total. The reading of the times, as well as the computation of the difference between two times, is delegated

```
double read_hours(std::istream & in)
{
    double total_hours = 0;
    for (int i = 1; i <= kLengthPayPeriod; ++i) {
        double num_hours_one_day;
        in >> num_hours_one_day;
        total_hours += num_hours_one_day;
    }
    return total_hours;
}
```

Figure 6.4: The `read_hours` function

to two other functions.

The code of Figure 6.5 works but it is fairly awkward because the storage of each time requires the use of two separate variables, one for the hours and one for the minutes. All these variables must be declared and also passed to the other functions.

This code would be much easier to write and understand if we had a `Time` data type that represented an entire time. Each `Time` value would consist of a number of hours and a number of minutes but `read_hours` would be able to handle each `Time` as a single unit, as shown in Figure 6.6. That code is clearly much simpler.

Figure 6.7 shows how the `Time` data type can be created. This declaration specifies that each `Time` value is a **structure** that consists of two integers called `hours` and `minutes`. A structure is a construct that combines various other variables into a single unit. These variables are called the *members* of the structure. Individual `Time` structures can be declared and passed to functions as any other variable, as shown in Figure 6.6.

Figure 6.8 shows the `read` and `difference` functions used by `read_hours`, as well as an additional function that prints times. These functions can be viewed as performing operations on times. Note how they directly access the members `hours` and `minutes` of individual `Time` structures by using the *member selection operator*, which is represented by a single dot (`.`).

When a `Time` structure is created, it is automatically initialized to the value 99:99, as specified in Figure 6.7. This initial value was selected because there is no obvious default initial value for a time. The fact that 99:99 is not a

```
double read_hours(std::istream & in)
{
    double total_hours = 0;
    string keyword;
    in >> keyword;
    while (keyword != "stop") {
        int start_time_hours;
        int start_time_minutes;
        read(start_time_hours, start_time_minutes, in);

        in >> keyword; // "out"
        int stop_time_hours;
        int stop_time_minutes;
        read(stop_time_hours, stop_time_minutes, in);

        total_hours += difference(stop_time_hours,
                                  stop_time_minutes,
                                  start_time_hours,
                                  start_time_minutes);
        in >> keyword;
    }
    return total_hours;
}
```

Figure 6.5: A revised `read_hours` function

```
double read_hours(std::istream & in)
{
    double total_hours = 0;
    string keyword;
    in >> keyword;
    while (keyword != "stop") {
        Time start_time;
        read(start_time, in);

        in >> keyword; // "out"
        Time stop_time;
        read(stop_time, in);

        total_hours += difference(stop_time, start_time);
        in >> keyword;
    }
    return total_hours;
}
```

Figure 6.6: A `read_hours` function that uses a `Time` data type

```
struct Time
{
    int hours = 99;
    int minutes = 99;
};
```

Figure 6.7: The `Time` data type

```
void read(Time & t, std::istream & in)
{
    in >> t.hours;
    in.get(); // colon
    in >> t.minutes;
}

void print(const Time & t, std::ostream & out)
{
    out << t.hours << ':';
    if (t.minutes < 10) out << 0;
    out << t.minutes;
}

double difference(const Time & t1, const Time & t2)
{
    return (t1.hours + t1.minutes/60.0) -
        (t2.hours + t2.minutes/60.0);
}
```

Figure 6.8: The `Time` functions

valid time makes it easy to detect if we forget to later set a time to its proper value.

The functions `print` and `difference` receive their `Time` arguments by constant reference. This is because these functions do not need to modify those arguments and to avoid copying two integers whenever a `Time` is passed to these functions. For the sake of efficiency, it is safer to always pass structures that don't need to change by constant reference instead of by value.

Note that we chose for these functions short names such as `read` and `print` instead of the longer, more descriptive `read_time` and `print_time`. The shorter names are descriptive enough because the `Time` arguments of these functions make it clear that the functions work on times.

The version of the pay calculator we created in this section is available on the course web site under `Pay calculator` as `pay_calculator_4_0.cpp`.

Study Questions

6.1.1. What is a structure?

6.1.2. How can the members of a structure be accessed?

Exercises

6.1.3. Create a function `init(t, hours, minutes)` that initializes `Time t` to the given hours and minutes. The arguments `hours` and `minutes` are integers.

6.1.4. Create a type of structure called `Date`. Each `Date` structure represents a date such as January 22, 2014. Each date is stored as three integers, `month`, `day` and `year`. Use January 1, 2000 as a default initial value. In addition to the structure, create the following functions:

a) A function `init(d, month, day, year)` that initializes `date` to the given month, day and year. The arguments `month`, `day` and `year` are integers.

b) A function `read(d, in)` that reads `Date d` from input stream `in`. Dates are typed as `m/d/y` where `m`, `d` and `y` are integers. No error-checking is performed. The stream is returned.

c) A function `print(d, out)` that prints `Date d` to output stream `out`. Dates are printed in numerical format, as in `1/22/2014`.

d) A function `print_in_words(d, out)` that also prints `Date d` to output stream `out` but with the month in words, as in `January 22, 2014`.

6.1.5. Create a type of structure called `ThreeDVector`. Each `ThreeDVector` structure consists of a three-dimensional vector of real numbers, such as $(3.5, 2.64, -7)$. Use $(0, 0, 0)$ as a default initial value. In addition to the structure, create the following functions:

a) A function `init(v, x, y, z)` that initializes `ThreeDVector v` to `(x, y, z)`, respectively. The arguments `x`, `y` and `z` are of type `double`.

b) A function `read(v, in)` that reads `ThreeDVector v` from input stream `in`. Vectors are entered in the format `(x, y, z)` where `x`, `y` and `z` are real numbers. No error-checking is performed. The stream is returned.

c) A function `print(v, out)` that prints `ThreeDVector v` to output stream `out`. Vectors are printed in the format described for `read`.

d) A function `add(v1, v2)` that returns the sum of `ThreeDVector`'s `v1` and `v2`.

6.1.6. Create a type of structure called `Fraction`. Each `Fraction` structure represents a fraction, that is, a number of the form a/b, where a is an integer and b is a positive integer. Use $0/1$ as a default initial value. In addition to the structure, create the following functions:

a) Functions `init(r, a)` and `init(r, a, b)` that initialize `Fraction r` to `a` and `a/b`, respectively. The arguments `a` and `b` are integers and `b` is assumed to be positive. No error-checking is performed.

b) A function `read(r, in)` that reads `Fraction r` from input stream `in`. Fractions are entered in the format `a/b` where `a` is an integer and `b` is a positive integer. No error-checking is performed. The stream is returned.

c) A function `print(r, out)` that prints `Fraction r` to output stream `out`. Fractions are printed in the format described for `read`. Fractions are not reduced.

d) A function `print_mixed(r, out)` that prints `Fraction r` to output stream `out`. Fractions are printed in mixed form `n a/b` where `n` is an integer and `a/b` is an optional positive proper fraction (one in which the numerator is less than the denominator). For example, `-5 6/8`. The fraction is not reduced.

e) Functions `add(r, s)` and `multiply(r, s)` that return, respectively, the sum and product of `Fraction`'s `r` and `s`. For example, if `r` is 2/3 and `s` is 3/4, then `add(r, s)` returns a `Fraction` whose value is 17/12 and `multiply(r, s)` returns 6/12. The returned fractions are not reduced.

6.1.7. Create a function `add_minutes(t, num_minutes)` that adds to `Time t` the given number of minutes. The argument `num_minutes` is an integer that can be arbitrarily large and even negative. When a time goes forward past 23:59, it simply cycles back to 0:00. When a time goes backwards past 0:00, it cycles forward to 23:59.

6.2 Improving the Design of the Text Editor

Figure 6.9 shows a portion of the `main` function of the text editor we created earlier in these notes. This code is somewhat cluttered because it involves a relatively large number of variables that must be declared and then passed to most of the other functions. This makes the code hard to read and understand.

In this section, we will simplify the code of the text editor in the same way we simplified the code of the pay calculator in the previous section. We will do this by creating a type of structure that combines, into a single unit, most of the text editor variables.

One possibility is to create a type `Buffer` that combines all the variables that relate to the contents and displaying of a buffer. Figure 6.10 shows the declaration of this type.

Figure 6.11 shows a portion of a version of `main` that uses the `Buffer` type. By comparing with Figure 6.9, we see that the declaration of the variable `buffer` replaces the declaration of five different variables. And each function call now involves one or two arguments, instead of possibly six. The new

```
int main() {
    string file_name;
    vector<string> v_lines;
    int ix_top_line = 0;
    int ix_current_line = 0;
    int window_height;
    string error_message;

    ...

    bool done = false;
    while (!done) {
        display(error_message, file_name, v_lines, ix_top_line,
                ix_current_line, window_height);
        ...

        switch (command) {
            case 'o': {
                open_file(file_name, v_lines, ix_top_line,
                          ix_current_line, error_message);
                break;
            }
            case 'n': {
                move_to_next_line(v_lines, ix_current_line,
                                  ix_top_line, window_height);
                break;
            }
            case 'i': {
                insert_line(v_lines, ix_current_line, ix_top_line,
                            window_height);
                break;
            }
            ...
        }
    }

    return 0;
}
```

Figure 6.9: A portion of the `main` function of the text editor

```
struct Buffer
{
    string file_name;
    vector<string> v_lines;
    int ix_top_line = 0;
    int ix_current_line = 0;
    int window_height = 0;
};
```

Figure 6.10: The `Buffer` data type

version of `main` is clearly simpler and easier to understand than the earlier version.

Figure 6.12 shows the `move_to_next_line` function before and after the introduction of the `Buffer` type into the program. The main advantage of the new version is that the four arguments are reduced to one. A disadvantage is that accessing the various elements of the buffer (the vector, the indices and the window height) now requires using the member selection operator, as in

```
b.ix_current_line
```

Another disadvantage is that in the earlier version, it was possible to declare the vector constant, to prevent it from being modified by accident. In the new version, the whole `Buffer` needs to be non-constant because some of its members need to be modified. This is a bit less safe. Overall, however, the drastic simplification of the `main` function probably outweighs these disadvantages.

The version of the text editor we created in this section is available on the course web site under `Text editor` as `text_editor_2_0.cpp`.

Exercises

6.2.1. Modify the text editor to allow the user to work on more than one buffer during a single session. Add a *buffer* command that produces a numbered list of all the existing buffers and allows the user to choose one. There should also be an option to create a new buffer. All the other commands should operate of the current buffer, that is, the one currently displayed.

```
int main() {
    Buffer buffer;
    string error_message;

    ...

    bool done = false;
    while (!done) {
        display(error_message, buffer);
        ...

        switch (command) {
            case 'o': {
                open_file(buffer, error_message);
                break;
            }
            case 'n': {
                move_to_next_line(buffer);
                break;
            }
            case 'i': {
                insert_line(buffer);
                break;
            }
        }
    }

    return 0;
}
```

Figure 6.11: A portion of a version of `main` that uses the `Buffer` data type

```
void move_to_next_line(const vector<string> & v_lines,
                       int & ix_current_line,
                       int & ix_top_line,
                       int window_height)
{
    if (ix_current_line < v_lines.size()) {
        ++ix_current_line;
        // check if window needs to be scrolled down
        if (ix_current_line >= ix_top_line + window_height)
            ++ix_top_line;
    }
}

void move_to_next_line(Buffer & b)
{
    if (b.ix_current_line < b.v_lines.size()) {
        ++b.ix_current_line;
        // check if window needs to be scrolled down
        if (b.ix_current_line >= b.ix_top_line + b.window_height)
            ++b.ix_top_line;
    }
}
```

Figure 6.12: The two versions of the `move_to_next_line` function

Chapter 7

Algorithms and Generic Programming

In this chapter, we will learn several simple but useful algorithms. We will learn to implement these algorithms in a generic way and we will also learn how to use the implementations that are available in the Standard Template Library.

7.1 Introduction

An algorithm is simply a sequence of steps for solving a computational problem. This informal definition is fairly broad. In fact, every program we write can be considered an algorithm. But the term algorithm is usually reserved for the solution to a single, well-defined and somewhat limited problem. This in contrast to the multiple tasks typically performed by most programs. Here are three problems that illustrate what we mean by a single well-defined problem:

1. Computing the maximum of two values.
2. Counting the number of occurrences of an element in a vector.
3. Sorting the elements of a vector into some order.

Algorithms have been developed for many such problems. Some of these algorithms are trivial. For example, to compute the maximum of two values, it is often sufficient to compare them by using the less-than operator (<). Other algorithms are nontrivial but simple. A counting algorithm is one such

```
if (x < y)
    y is the maximum
else
    x is the maximum
```

Figure 7.1: An algorithm that computes the maximum of two values

example. But some algorithms are fairly complex. This includes efficient algorithms for sorting vectors.[1]

In this section, we will learn several simple but important algorithms. We will also learn how to use the wide variety of algorithms that are available in a portion of the C++ standard library called the *Standard Template Library (STL)*. For example, the STL includes functions that implement algorithms for the three problems mentioned above. The function

```
max(a, b)
```

returns the maximum of `a` and `b`. The number of occurrences of element `e` in vector `v` can be computed as follows:

```
count(v.begin(), v.end(), e)
```

And

```
sort(v.begin(), v.end())
```

rearranges the elements of vector `v` in increasing order. These functions are called *STL algorithms*.

7.2 Generic Programming

Consider the problem of computing the maximum of two values. As mentioned earlier, in the case of values that can be compared by using the less-than operator, this problem can be solved by the trivial algorithm shown in Figure 7.1. This assumes that the two values to be compared are `x` and `y`.

[1]Examples are mergesort, heapsort and quicksort. At Clarkson, these algorithms are normally covered in the courses *CS142 Introduction to Computer Science II* and *CS344 Algorithms and Data Structures*.

```
int max(const int & x, const int & y)
{
    if (x < y)
        return y;
    else
        return x;
}
```

Figure 7.2: A function that computes the maximum of two integers

```
string max(const string & x, const string & y)
{
    if (x < y)
        return y;
    else
        return x;
}
```

Figure 7.3: A function that computes the maximum of two strings

Since computing the maximum of two values is a problem that is likely to occur often, it is convenient to create a function that implements the above algorithm. For example, Figure 7.2 shows a function that returns the maximum of two integers.

Now, suppose that we need to compute the maximum of two strings, defined as the string that occurs later in the usual alphabetical order. The algorithm of Figure 7.1 can be used on strings since the class `string` provides a less-than operator. But the function of Figure 7.2 only works for integers. Therefore, we need to create a new function, as shown in Figure 7.3.

What we have here is that the maximum algorithm of Figure 7.1 is **generic**, in the sense that it can be used on more than one type of value. But the functions of Figures 7.2 and 7.3 are not generic: they each work on only one type of argument.

The process of creating a new function that implements a generic algorithm on a different type of value is inconvenient and prone to errors. To simplify this process and reduce the risk of errors, we could explicitly identify the type that must be changed to produce a new version of the algorithm, as shown in

```
// template: replace T by the desired type
T max(const T & x, const T & y)
{
    if (x < y)
        return y;
    else
        return x;
}
```

Figure 7.4: A template to generate functions that implement the `max` algorithm

Figure 7.4. The result is a *template* in which the generic name `T` is used for the type of value being compared. Then, whenever a version of `max` is needed for a particular type of value, all we have to do is copy the template and replace every occurrence of `T` by the desired type.

What we just described is a fairly mechanical process, which means that it's a good candidate for automation. And, in fact, C++ compilers can do just that. The key is to define a **function template**, as shown in Figure 7.5. The template definition begins with the keyword `template` followed by a declaration that identifies `T` as a **template parameter**. Then, for example, when the function `max` is called with integer arguments, the compiler looks for a `max` function that can take two integers as argument. When that fails, the compiler then looks for a template that can be used to generate such a function. The template of Figure 7.5 will work if `T` is set to `int`. The process of generating a function out of a template is called **template instantiation**.[2]

Note how the documentation of the function template `max` clearly states a condition that the type `T` must meet. It is important to clearly identify and document such requirements when creating templates.

Strictly speaking, a function template is not a function but we can think of it as a **generic function**, that is, a function that can work on more than one type of argument. The creation of generic functions is an example of **generic programming**, the writing of code that can be used on more than one type of data. In C++, classes can also be generic. The class `vector` is an example

[2]The template declaration of Figure 7.5 uses the keyword `typename` to declare the template parameter `T`. An alternative is to use the keyword `class`. This is allowed even if the type may not be a class. Although the keyword `typename` is more accurate, the use of the keyword `class` is widespread.

```
// Returns the maximum of x and y.
// Requirement on T: values can be compared by using the <
// operator.
template <typename T>
T max(const T & x, const T & y)
{
    if (x < y)
        return y;
    else
        return x;
}
```

Figure 7.5: The function template `max`

```
namespace my {

... (your code)

} // namespace my
```

Figure 7.6: A namespace

of a generic class.[3]

The exercises of this section ask you to create several generic functions. Some of these functions have the same names as functions that are available in the C++ standard library. To avoid possible conflicts, and to allow you to choose which function you want to use in a particular piece of code, it is best to place your functions in your own namespace, as shown in Figure 7.6. Then, to use your own implementation of `max`, for example, you would specify the namespace when calling the function (`my::max`) or you would include a `using` declaration in your code (`using my::max`). Note that specifying a namespace when calling a function is a way to override any other `using` declarations (such

[3]Bjarne Stroustrup, the original designer of C++, describes the language as a better C that supports data abstraction, object-oriented programming and generic programming [Str]. We briefly mentioned the concepts of data abstraction and object-oriented programming in Section 5.3. At Clarkson, all of these concepts are covered in more detail in the courses *CS142 Introduction to Computer Science II* and *CS242 Advanced Programming Concepts*.

as `using std::max`).[4]

Study Questions

7.2.1. What is an algorithm?

7.2.2. What is a generic function?

7.2.3. What C++ construct allows us to implement generic functions?

7.2.4. What is generic programming?

Exercises

7.2.5. Create the following generic functions. In each case, clearly document any requirements on the template arguments.

a) A generic function `min(x,y)` that returns the minimum of its two arguments.

b) A generic function `average(x,y)` that returns the average of its two arguments.

7.3 Some Simple Algorithms

In this section, we will learn several basic but useful algorithms. But first, we will improve the `max` generic function of the previous section in two ways.

Figure 7.7 shows the final result. One difference is the return value of the function. By returning a constant reference to one of the arguments, the function is more efficient, especially when the arguments are large. This can be the case with strings, for example. In general, when designing generic functions, it is usually safer to avoiding copying values of an unknown type.

The second difference is that this implementation of `max` uses the *conditional operator*. The general form of this operator is

[4]Note that some library implementations contain `using` declarations that may be inadvertently included in your programs. When this happens, then to use your own function, it becomes necessary to specify your namespace. For example, if the declaration `using std::max` ends up being included in your program, then the only way to call your own `max` would be as `my::max`.

```
// Returns the maximum of x and y.
// Requirement on T: values can be compared by using the <
// operator.
template <typename T>
const T & max(const T & x, const T & y)
{
    return (x < y ? y : x);
}
```

Figure 7.7: An improved generic `max` algorithm

```
condition ? expression : expression
```

If the condition is true, then the operator evaluates to the value of the first expression. Otherwise, the operator evaluates to the value of the second expression.[5]

As with an `if` statement or a loop, the condition must be a Boolean expression. As for the two expressions, they must evaluate to values that are appropriate for the context in which the operator is used. In the implementation of the generic `max`, these expressions evaluate to values of type `T`.

The conditional operator does not make the `max` function more efficient. But it results in more concise code. Note that not everybody would agree that this improves readability.

We now turn to algorithms for other problems. Consider the problem of swapping the values of two variables. Figure 7.8 shows a simple generic algorithm that solves this problem.

The other problems we will consider in this section all concern vectors. Figure 7.9 shows a generic algorithm that counts the number of occurrences of an element in a vector.

Figure 7.10 shows a more flexible version of this algorithm. It counts the number of occurrences of an element in the portion of a vector specified by two indices. Note that in C++, the standard way of specifying a range of positions with two indices is to interpret the indices as the endpoints of an interval that is closed on the left and open on the right. In other words, the first index specifies the first position we're interested in while the second index specifies the first position we're *not* interested in.

[5]The conditional operator is sometimes called the *ternary* or *ternary conditional* operator. It is the only C++ operator that has three operands.

```
// Swaps the values of x and y.
template <typename T>
void swap(T & x, T & y)
{
    T original_x = x;
    x = y;
    y = original_x;
}
```

Figure 7.8: A generic `swap` algorithm

```
// Returns the number of occurrences of e in v.
// Requirement on T: values can be compared by using the ==
// operator.
template <typename T>
int count(const vector<T> & v, const T & e)
{
    int n = 0;
    for (int i = 0; i < v.size(); ++i)
        if (v[i] == e) ++n;
    return n;
}
```

Figure 7.9: A generic `count` algorithm

```
// Returns the number of occurrences of e in the range
// [start, stop) of v.
// Requirement on T: values can be compared by using the ==
// operator.
template <typename T>
int count(const vector<T> & v, int start, int stop, const T & e)
{
    int n = 0;
    for (int i = start; i < stop; ++i)
        if (v[i] == e) ++n;
    return n;
}
```

Figure 7.10: A more flexible `count` algorithm

```
// Returns the index of the first occurrence of e in the range
// [start, stop) of v. Returns stop if e is not found.
// Requirement on T: values can be compared by using the ==
// operator.
template <typename T>
int find(const vector<T> & v, int start, int stop, const T & e)
{
    for (int i = start; i < stop; ++i)
        if (v[i] == e) return i;
    return stop;
}
```

Figure 7.11: A generic `find` algorithm

Figure 7.11 shows a generic algorithm that finds the first occurrence of an element in a range of positions within a vector. The vector is scanned from left to right (in increasing order of indices). This algorithm performs what is called a **sequential search** of the vector. In case the element is not found within the range of positions, the index that marks the right end of the range is returned. This is a convenient (and standard) way of indicating failure.

Figure 7.12 shows a generic algorithm that copies elements from a vector to another. The elements to be copied from the first vector are specified by a range of positions. The destination of the copying, that is, the range of positions where the elements should be copied to in the second vector, is specified by a single index that marks the beginning of that range. The function returns an index that marks the end of the destination range.

Note that the `copy` algorithm can be used to copy elements from a range of positions to another within the same vector, as in

```
copy(v, 0, 10, v, 20)
```

But since the algorithm copies forward, from `start` to `stop`, it is typically not useful when `dest` falls within the range `[start,stop)`. (An exercise asks you to explain why.)

Finally, Figure 7.13 shows a generic algorithm that returns the index of the maximum element in a range of positions within a vector. The algorithm scans the range of positions from left to right and stores in the variable `ix_max` the index of the largest element it has seen so far. The variable `ix_max` is initialized to the first index of the range.

```
// Copies the elements in the range [start,stop) in v1 to a range
// of positions that begins at index dest in v2. Returns an index
// that marks the end of the destination range. Copies forward,
// from start to stop.
// Precondition: assumes that the destination range is large enough.
template <typename T>
int copy(const vector<T> & v1, int start, int stop,
         vector<T> & v2, int dest)
{
    for (int i = start; i < stop; ++i) {
        v2[dest] = v1[i];
        ++dest;
    }
    return dest;
}
```

Figure 7.12: A generic `copy` algorithm

```
// Returns the index of the largest element in the range
// [start,stop) in v.
// Precondition: assumes that the range is not empty.
// Requirement on T: values can be compared by using the <
// operator.
template <typename T>
int max_element(const vector<T> & v, int start, int stop)
{
    int ix_max = start;
    for (int i = start + 1; i < stop; ++i)
        if (v[ix_max] < v[i]) ix_max = i;
    return ix_max;
}
```

Figure 7.13: A generic `max_element` algorithm

Exercises

7.3.1. Explain why the generic algorithm `copy` is typically not useful when `dest_begin` falls within the range `[start,stop)`?

7.3.2. Create the following generic functions. In each case, clearly document any requirements on the template arguments.

a) A generic function `println(v, start, stop)` that prints to `cout` the elements in the range `[start,stop)` in `v`. The elements are printed on a single line with consecutive elements separated by a single space.

b) A generic function `sum(v, start, stop)` that returns the sum of the elements in the range `[start,stop)` in `v`.

c) A generic function `average(v, start, stop)` that returns the average of the elements in the range `[start,stop)` in `v`.

d) A generic function `fill(v, start, stop, e)` that sets to `e` every element in the range `[start,stop)` in `v`.

e) A generic function `replace(v, start, stop, x, y)` that replaces by `y` every occurrence of element `x` in the range `[start,stop)` in `v`.

f) A generic function `min_element(v, start, stop)` that returns the index of the smallest element in the range `[start,stop)` in `v`.

g) A generic function `reverse(v, start, stop)` that reverses the order of the elements in the range `[start,stop)` in `v`.

h) A generic function

```
copy_backward(v1, start, stop, v2, dest)
```

that copies the elements in the range `[start,stop)` in `v1` to a range of positions that *ends* at index `dest` in `v2`. Returns an index that marks the beginning of the destination range. Copies backward, from `stop` to `start`.

7.4 Algorithms in the STL

As mentioned earlier in this chapter, the STL includes functions that implement a wide variety of standard algorithms. This section presents several of

`swap(x, y)` Swaps the values of `x` and `y`.

`max(x, y)`
`min(x, y)`
`max({elements})`
`min({elements})`
Returns the maximum or minimum of the given elements. Uses $<$ to compare elements.

Table 7.1: Some generic algorithms

these algorithms. Many more are described in a reference such as [CPP].

As the word *template* in the name of the STL indicates, most components in this library are templates. This includes the class `vector`, as well as the functions we will describe in this section. For example, Table 7.1 describes the STL algorithms `swap`, `max` and `min`. These functions are generic so they can be used on arguments of any type that satisfy certain requirements. (These requirements are usually obvious.)

In the previous section, we designed several algorithms that work on vectors. For example,

```
count(v, start, stop, e)
```

returns the number of occurrences of `e` in the range of positions `[start, stop)` within `v`. The functions we created are generic because they each work on more than one type of vector. But note that these functions only work on vectors.

The corresponding STL algorithms are even more generic because they work not only on multiple types of vectors but also on other types of containers. This has an important consequence. As mentioned in Section 5.8, numerical indices are not an efficient way of specifying positions in some of these other types of containers. Therefore, instead of indices, the STL algorithms use *iterators* to specify ranges of positions.

Recall that the main purpose of an iterator is simply to mark a position within a container. Each type of STL container supplies its own type of iterator, implemented in a way that's appropriate for that container.

For example, the STL `count` algorithm has the following form:

```
count(start, stop, e)
```

This function returns the number of occurrences of `e` in the range of positions `[start, stop)`. The arguments `start` and `stop` could be vector iterators or iterators that point to elements in some other type of container.

Table 7.2 lists some STL generic algorithms that are designed to be used on sequences such as vectors. Another example of a sequence is the STL container `list`.[6]

Using the algorithms of Table 7.2 on vectors requires only a very basic understanding of iterators. That's in part because it is easy to convert between vector indices and vector iterators. If `i` is an index, then `v.begin() + i` gives an iterator that points to the element in `v` that has index `i`. So, for example,

```
count(v.begin() + i, v.begin() + j, e)
```

returns the number of occurrences of `e` in the range of positions `[i, j)` in `v`.

On the other hand, if `itr` is a vector iterator, then `itr - v.begin()` gives the index of the element that `itr` points to. So, for example,

```
find(v.begin() + i, v.begin() + j, e) - v.begin()
```

gives the index of the first occurrence of `e` in the range of positions `[i, j)` in `v`.

It is possible to apply the STL algorithms to entire vectors. For example,

```
count(v.begin() + 0, v.begin() + v.size(), e)
```

returns the number of occurrences of `e` in all of `v`. But the following is much more convenient:

```
count(v.begin(), v.end(), e)
```

This works because `v.begin()` gives an iterator that points the first element of `v` and `v.end()` gives an iterator that points to a position that's just past the last element of `v`.

The STL generic algorithm `find` performs a sequential search of the given range. An alternative is to perform a *binary search*. This algorithm executes much more quickly as long as two conditions are met: the elements in the range are sorted and the iterators that specify the range meet certain conditions.

[6] At Clarkson, STL lists are covered in detail in the course *CS142 Introduction to Computer Science II*.

`count(start, stop, e)`
: Returns the number of occurrences of element `e` in the range `[start,stop)`. Uses == on elements.

`find(start, stop, e)`
: Returns an iterator to the first occurrence of element `e` in the range `[start,stop)`. Returns `stop` if `e` is not found. Uses == on elements.

`max_element(start, stop)`
`min_element(start, stop)`
: Returns an iterator that points to the maximum or minimum element in the range `[start,stop)`. Return `stop` if the range is empty. Uses < on elements.

`copy(start, stop, dest_begin)`
: Copies the elements in the range `[start,stop)` to a range of positions that begins at `dest_begin`. If n is the number of elements that are copied, returns an iterator that points n positions to the right of `dest_begin`. Copies forward, from `start` to `stop`. Typically not useful when `dest_begin` falls within the range `[start,stop)`.

`copy_backward(start, stop, dest_end)`
: Copies the elements in the range `[start,stop)` to a range of positions immediately to the left of `dest_end`. If n is the number of elements that are copied, returns an iterator that points n positions to the left of `dest_end`. Copies backward, from `stop` to `start`. Use instead of `copy` when the destination range begins within `[start,stop)`.

`fill(start, stop, e)`
: Sets all the elements in the range `[start,stop)` to be copies of element `e`.

`replace(start, stop, x, y)`
: Replaces all occurrences of element `x` by a copy of element `y` in the range `[start,stop)`. Uses == on elements.

`reverse(start, stop)`
: Reverses the order of the elements in the range `[start,stop)`.

Table 7.2: Some generic sequence algorithms

`binary_search(start, stop, e)`
`lower_bound(start, stop, e)`
`upper_bound(start, stop, e)`
Performs a binary search in the range `[start,stop)`. Assumes that the range is sorted with respect to <. Uses < on elements. The first version returns `true` if element `e` is present in the range. Otherwise, it returns false. The second version returns an iterator that points to the first position where `e` could be inserted in the range while preserving the order. The third version returns an iterator that points to the last such position.

`sort(start, stop)` Sorts the elements in the range `[start,stop)` using the < operator. Requires random-access iterators.

Table 7.3: Some generic algorithms for searching and sorting

Without going into details, iterators that meet those particular conditions are called *random-access iterators*. Vector iterators are an example of random-access iterators. (List iterators are not.)[7]

The STL includes several versions of the binary search algorithm. These are described in Table 7.3. This table also includes a description of a generic sorting algorithm.

The generic algorithm `swap` is defined in the library file `utility`. The other STL generic algorithms mentioned in this section are defined in the library file `algorithm`. All of them are part of the `std` namespace.

Study Questions

7.4.1. What is an iterator?

7.4.2. How can we convert between vector indices and vector iterators?

7.4.3. Where do the iterators `v.begin()` and `v.end()` point to?

[7] At Clarkson, the binary search algorithm, as well as iterator categories such as random-access iterators, are covered in detail in the course *CS142 Introduction to Computer Science II*.

Exercises

7.4.4. Experiment with the STL searching and sorting algorithms by writing a test driver that uses all the algorithms shown in Table 7.3. To see the difference between `lower_bound` and `upper_bound`, make sure you search for an element that occurs multiple times in the sequence.

Bibliography

[CPP] cppreference.com. Web. Last accessed January 2014. http://cppreference.com.

[Laz] Ed Lazowska. Exponentials R us: Seven computer science game-changers from the 2000s. *Xconomy*, 2009. Web. Last accessed January 2014. http://www.xconomy.com/seattle/2009/12/24/exponentials-r-us-seven-computer-science-game-changers-from-the-2000's-and-seven-more-to-come.

[Str] Bjarne Stroustrup. The C++ programming language. Web. Last accessed October 2013. http://www.stroustrup.com/C++.html.

Index