C++ Programming for Games
Module I

e-Institute Publishing, Inc.

Editor: Susan Nguyen
Cover Design: Adam Hoult

E-INSTITUTE PUBLISHING INC
www.gameinstitute.com

Frank Luna, Game Institute Faculty, *C++ Programming for Games*

Table of Contents

Chapter 1

Introducing C++

Introduction

C++ is a powerful language that unifies high-level programming paradigms, such as object oriented programming, with low-level efficiencies, such as the ability to directly manipulate memory. For these reasons, C++ has been embraced as the language of choice among game developers. C++ fulfills the need for high-level language constructs which aid in the organization of building complex virtual worlds, but is also able to perform low-level optimizations in order to squeeze out extra performance for such things as sophisticated special effects, realistic physics, and complex artificial intelligence.

Chapter Objectives

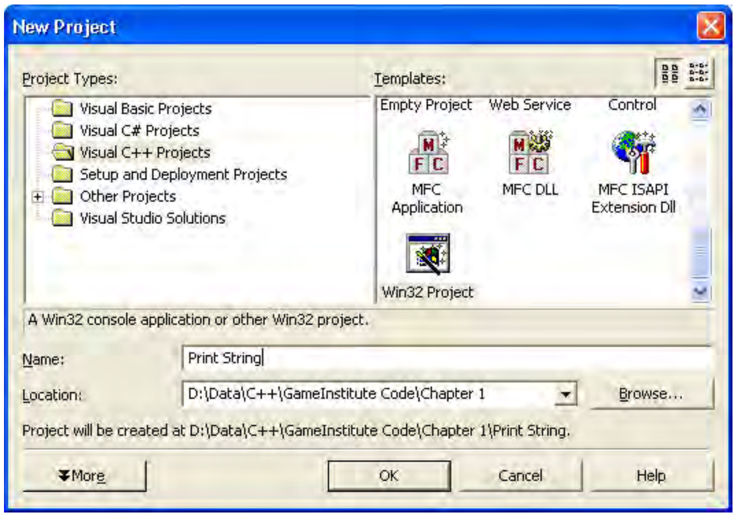
- Create, compile, link and execute C++ programs.
- Find out how C++ code is transformed into machine code.
- Learn some of the basic C++ features necessary for every C++ program.
- Discover how to output and input text information to and from the user.
- Understand the concept of variables.
- Perform simple arithmetic operations in C++.

1.1 Getting Started—Your First C++ Program

A **program** is a list of instructions that directs the computer to perform a series of operations. An operation could be adding two numbers together or outputting some data to the screen. In this section you will learn, step-by-step, how to create, compile, link and execute a C++ program using Visual C++ .NET (either the 2002 or 2003 edition). We recommend that you actually perform these steps as you read them in order to fully understand and absorb all that you are learning. If you are not using Visual C++ .NET, consult your particular C++ development tool's documentation for information on how to create, compile, link and execute a C++ program.

1.1.1 Creating the Project

After you launch Visual C++ .NET, go to the menu and select *File->New->Project*. The following dialog box appears:



A Win32 console application or other Win32 project.

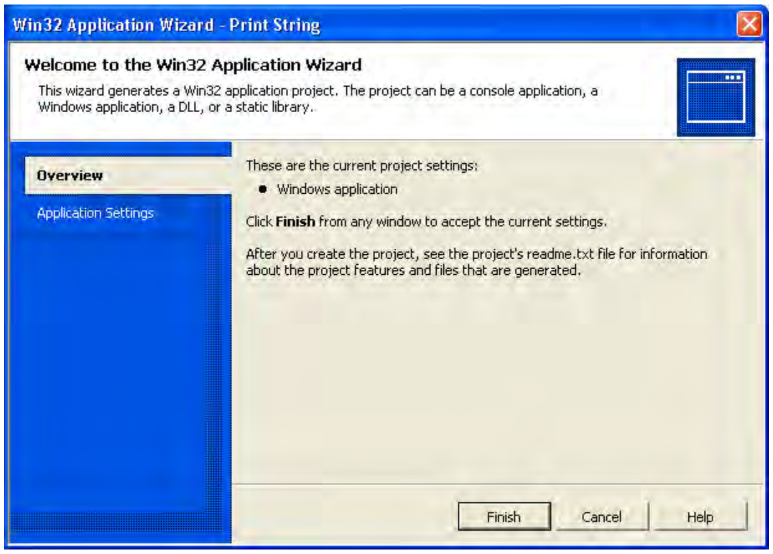
Name: Print String

Location: D:\Data\C++\GameInstitute Code\Chapter 1

Project will be created at D:\Data\C++\GameInstitute Code\Chapter 1\Print String.

Figure 1.1: The “New Project” dialog box.

Enter a name of your choosing for the project and the location to which you wish to save it on your hard drive. Then press the *OK* button. A new *Overview* dialog box now appears, as seen in Figure 1.2. Selecting the *Application Settings* button displays the dialog box shown in Figure 1.3



Welcome to the Win32 Application Wizard

This wizard generates a Win32 application project. The project can be a console application, a Windows application, a DLL, or a static library.

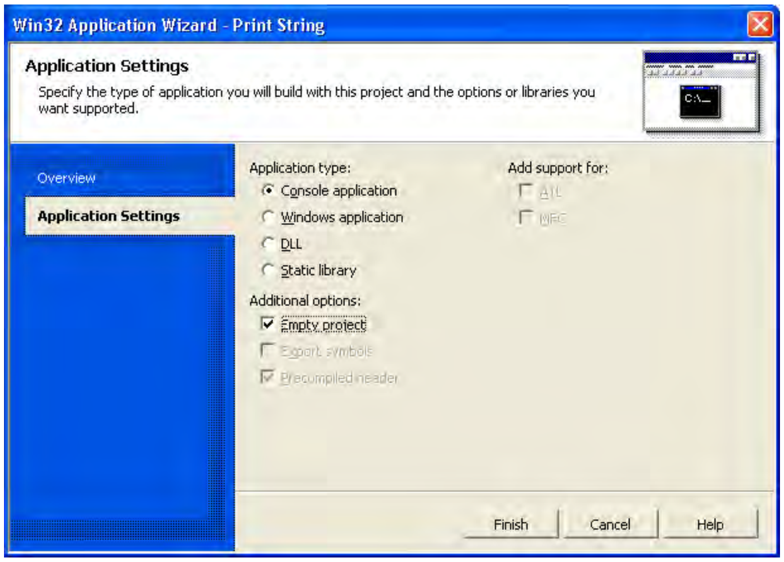
These are the current project settings:

- Windows application

Click **Finish** from any window to accept the current settings.

After you create the project, see the project's readme.txt file for information about the project features and files that are generated.

Figure 1.2: The “Overview” dialog box. Select the "Application Settings" button from the blue column on the left.


Win32 Application Wizard - Print String

Application Settings

Specify the type of application you will build with this project and the options or libraries you want supported.

Application type:
(● Console application) Windows application / DLL / Static library

Add support for:
☐ ATL
☐ MFC

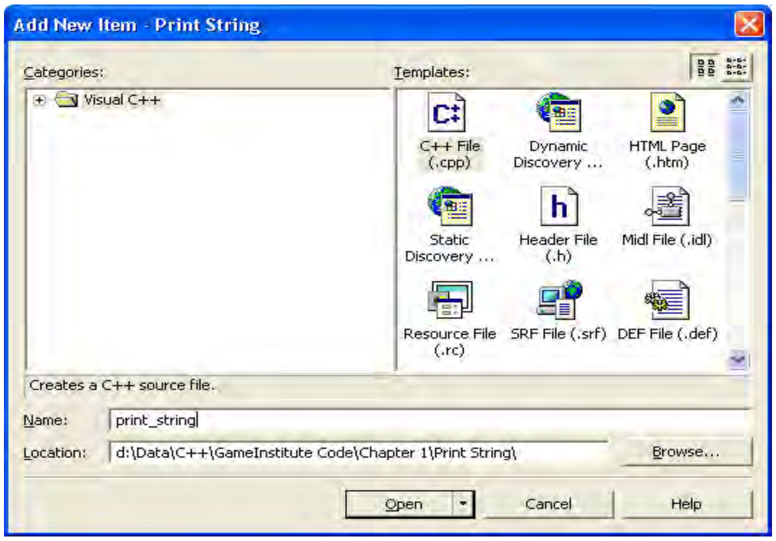
Additional options:
☑ Empty project
☐ Export symbols
☑ Precompiled header

Figure 1.3: The "Application Settings" dialog box. Be sure to select "Console application" and check "Empty project." Afterward, press the "Finish" button.

Once you have selected *Console application* for the *Application type* setting, and have checked *Empty project* for the *Additional options* setting, press the *Finish* button. At this point, we have successfully created a C++ project. The next step is to add a C++ source code file (.CPP) to the project; that is, a file in which we will actually write our C++ code.

1.1.2 Adding A .CPP File to the Project

To add a .CPP file to your project, go to the menu and select *Project->Add New Item…* An *Add New Item* dialog box appears (Figure 1.4). From the right category, *Templates*, select a *C++ File (.cpp)*. Give the .CPP file a name, and then select *Open*. A blank .CPP file should automatically be opened in Visual C++ .NET.



Creates a C++ source file.

Name: print_string

Location: d:\Data\C++\GameInstitute Code\Chapter 1\Print String\

Figure 1.4: The "Add New Item" dialog box. Select the file type you wish to add to the project. In this case we want to add a C++ File (.cpp).

1.1.3 Writing the Code

You should now see a blank .CPP file where we can begin to write our code. Type or copy the following C++ code, exactly as it is, into your .CPP file.

Program 1.1: Print String.

```
//==================================================================
// print_string.cpp By Frank Luna
//==================================================================

#include <iostream>
#include <string>

int main()
{
     std::string firstName = "";

     std::cout << "Enter your first name and press Enter: ";

     std::cin >> firstName;

     std::cout << std::endl;

     std::cout << "Hello, " << firstName << std::endl << std::endl;
}
```

1.1.4 Compiling, Linking, and Executing

After the C++ code is completed, it must be translated into a language the computer understands—that is, machine language—in order that it can be executed. There are two steps to this translation process:

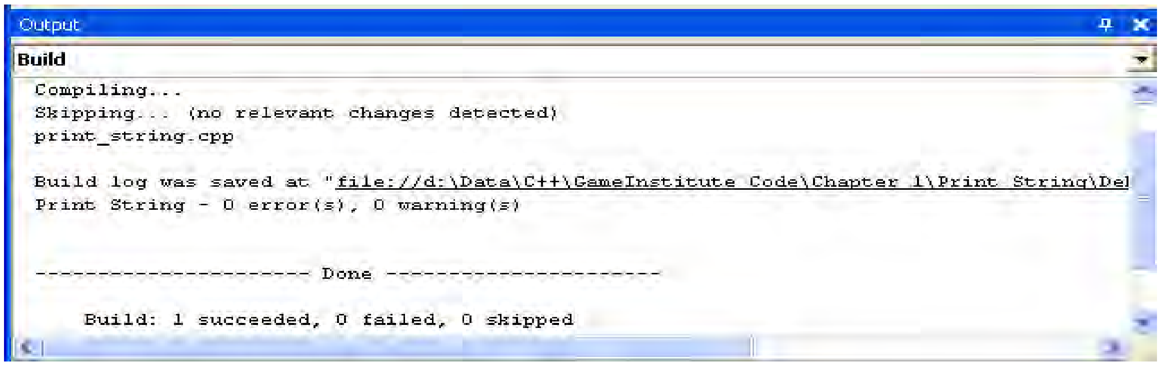
1) Compilation
2) Linking

In the compilation step, the compiler **compiles** each source code file (.CPP) in your project (more complex projects will contain more than one source code file) and generates an **object file** (.OBJ) for each one. An object file is said to contain **object code**.

In the next step, called **linking**, the **linker** combines all the object files, as well as any **library files** (.LIB), to produce an executable file. It is the executable file which will run on your platform.

> **Note:** A library file is a set of object code that usually stores the object code of many object files in one compact file. In this way, users do not have to link numerous object files but can merely link one library file.

The first step towards generating an .EXE is to compile the program. To compile the program, go to the menu and select *Build->Compile*. At the bottom of VC++ .NET, the results of your compilation should be displayed in the *Output* window—see Figure 1.5.

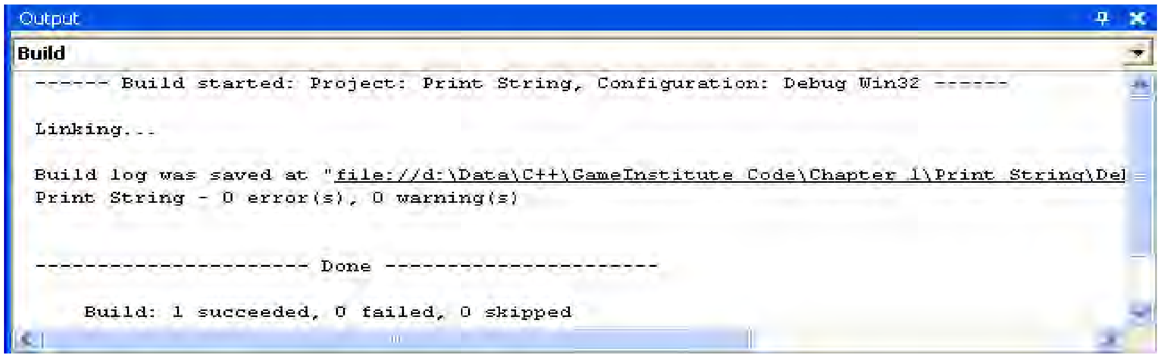


```
Compiling...
Skipping... (no relevant changes detected)
print_string.cpp

Build log was saved at "file://d:\Data\C++\GameInstitute Code\Chapter 1\Print String\Del
Print String - 0 error(s), 0 warning(s)


---------------------- Done ----------------------

    Build: 1 succeeded, 0 failed, 0 skipped
```

Figure 1.5: Compilation Output.

Observe that we have zero errors and zero warnings; this means we have written legal C++ code, and therefore our program has compiled successfully. If we had written any illegal C++ code (e.g., we made typos or used incorrect C++ punctuation), the compiler would let us know by displaying various errors or warnings. For example, if you removed one of the ending semicolons from Program 1.1 and tried to compile it, you would get the following error message: "error C2146: syntax error: missing ';' before identifier 'cout'". You can use these error messages to help pinpoint the problems in your code that are preventing a successful compile.

Once we have fixed any compiler errors and there are zero errors or warnings, we can proceed to the next step—the **build step** (also called linking). To build the program, select the *Build->Build Solution* item from the menu. Similarly to the compilation process, the results of your build should be displayed in the *Output* window—see Figure 1.6.



```
------ Build started: Project: Print String, Configuration: Debug Win32 ------

Linking...

Build log was saved at "file://d:\Data\C++\GameInstitute Code\Chapter 1\Print String\Del
Print String - 0 error(s), 0 warning(s)


---------------------- Done ----------------------

    Build: 1 succeeded, 0 failed, 0 skipped
```

Figure 1.6: Build Output.

Observe that we have zero errors and zero warnings; this means we have written legal C++ code, and therefore our program has linked successfully. As with compilation, there would be error messages if we had written any illegal C++ code.

At post-build we have an executable file generated for a program. We can now execute the program from VC++ .NET by going to the menu and selecting *Debug->Start Without Debugging*. Doing so launches our console application, which outputs the following:

```
Enter your first name and press Enter:
```

Doing as the application instructs, you will input your first name and press enter. (Note that we bold input text in this book so that it is clear which text is entered as input and which text is output.) The program then displays the following:

Program 1.1: Output.

```
Enter your first name and press Enter: Frank

Hello, Frank

Press any key to continue
```

Note that by choosing *Start without Debugging*, the compiler automatically adds the “Press any key to continue” functionality.

Before continuing with this chapter, spend some time studying the ouput of Program 1.1 and the code used to create it. Based on the output, can you guess what each line of code does?

> **Note:** If your program has not been compiled or built, you can still go directly to the “Start without Debugging” menu command, and VC++ .NET will automatically compile, build, and execute the program in one step.

1.2 The “Print String” Program Explained

The following subsections explain Program 1.1 line-by-line.

1.2.1 Comments

```
//=========================================================
// print_string.cpp By Frank Luna
//=========================================================
```

The first three lines in Program 1.1 are *comments*. A single lined comment is designated in C++ with the double forward slashes ‘//’—everything on the same line that follows the ‘//’ is part of the comment. C++ also supports a multi-line comment, where everything between a ‘/*…*/’ pair is part of the comment. For example:

```
/*  This  is
  a multi-

  line comment */
```

Note also that by using the ‘/*…*/’ style comment, you can comment out parts of a line, whereas ‘//’ comments always comment the entire line. For example,

```
cout << "Hello, " << /*print firstName*/ firstName << endl;
```

Here we have inserted a comment (/*print firstName*/) in the middle of a code line. In general, ‘/*…*/’ style comments comment out only what is between them. Observe that comments are highlighted in green in VC++ .NET.

Comments are strings that are ignored by the compiler. That is, when the compiler compiles the code, it skips over any comments. Their main purpose is to make notes in a program. For example, at some point you may write some tricky code that is difficult to follow. With comments, you can write a clear English (or some other natural language) explanation of the code.

Writing clear comments becomes especially important when working in teams, where other programmers will need to read and modify your code. For instance, you might have a piece of code that expects a certain kind of input, which may not be obvious to others. By writing a comment that explains the kind of input that is expected, you may prevent your coworker from using your code incorrectly and wasting time. Furthermore, the code that was obvious to you three months ago might not be so obvious today. So maintaining useful comments can be beneficial to you as well.

Note that throughout the above discussion, we state that the goal is writing “clear” and “useful” comments, as opposed to “bad” comments. “Good” comments are clear, concise, and easy to understand. “Bad” comments are inconsistent, out of date, ambiguous, vague, or superfluous and are of no use to anyone. In fact, bad comments can even increase the confusion of the reader. As Bjarne Stroustrup, the inventor of the C++ language, states in his text, *The C++ Programming Language: Special Edition*: “Writing good comments can be as difficult as writing the program itself. It is an art well worth cultivating.”

1.2.2 White Space

Between the comments and the #include lines, we have a blank line. Blank lines and spaces are called *white space*, for obvious reasons. The important fact about white space is that the compiler ignores it (although there are some exceptions). We very well could have multiple line spaces, multiple spaces between words, and multiple C++ statements all on the same line. From the compiler’s point of view, the following code is equivalent to Program 1.1 from Section 1.1.3:

Program 1.2: Playing with the white space.

```
//=================================================================
// print_string.cpp By Frank Luna
//=================================================================

#include <iostream>

#include <string>
int
main(){
   std::string
           firstName =
           "";

std::cout << "Enter your first name and press Enter: ";

   std::cin >> firstName;  std::cout << std::endl;


   std::cout<<"Hello, "<<firstName<<std::endl<<std::endl;   }
```

You should try to compile the above program to verify that it still works correctly.

Note that spaces inside a string (e.g., “s p a c e s”) are not ignored, as they are actually considered space characters. Additionally, C++ keywords cannot simply be broken up and expected to mean the same thing. For example, you cannot write “`firstName`” as

```
     fir  st  Name
```

Symbols that have actual meaning must be kept intact as they are defined.

Finally, the #include directives, which we discuss in the next section, are special kinds of statements and they must be listed on their own line.

1.2.2 Include Directives

In the programs we write, we will need to use some code that we did not write ourselves. For this course, most of this "other" code will be found in the C++ **standard library**, which is a set of C++ utility code that ships with your C++ compiler. Some examples of useful code which the C++ standard library includes are:

- Code for outputting and inputting data to and from the console window
- Functions for computing various math operations such as sine and cosine
- Random number generators
- Code for saving and loading files to and from the hard drive
- Code to work with strings

There is, of course, much more functionality to the standard library than just described, and we will become more familiar with it as we progress through this course. (Note that there are entire volumes dedicated to just the C++ standard library functionality.)

We know that we will be using code from the C++ standard library, but in order to do so, we must *include* some standard library code into our code. To do this, we use an **include directive** (`#include` <*file*>). In our first example program, Program 1, we invoked two include directives:

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

The first include directive instructs the compiler to take all the code in the specified file (the file in between the angle brackets), "iostream," and include it (i.e., copy and paste it) into our .CPP file. Similarly, the second include directive instructs the compiler to take all the code in "string" and include it into our .CPP file. *iostream* and *string* are C++ standard library **header** files, which contain C++ standard library code. Note that in addition to including C++ standard library header files, your program links with standard library files as well; however, this is done automatically—the C++ standard libraries are linked by default when you create a new C++ project.

1.2.3 Namespaces

Namespaces are to code as folders are to files. That is to say, as folders are used to organize groups of related files and prevent file name clashes, namespaces are used to organize groups of related code and prevent code name clashes. For instance, the entire C++ standard library is organized in the standard (`std`) namespace. That is why we had to prefix most of our code in Program 1.1. Essentially, the `std::` prefix tells the compiler to search the standard namespace (folder) to look for the standard library code we need.

Of course, prefixing all of your code library with `std::`, or some other namespace can become cumbersome. With the `using namespace X` clause, you can move code in some particular namespace to the **global namespace**, where `X` is some namespace. Think of the global namespace as the "working folder". You do not need to specify the folder path of a file that exists in the folder you are currently working in, and likewise, you do not need to specify a namespace when accessing code in the global namespace. The following revision of Program 1.1 includes a `using namespace std` clause, which moves the code in the `std` namespace to the global namespace, and as such, we no longer need to prefix our standard library code with the `std::` prefix:

Program 1.3: Print String with "using namespace std" clause.

```
//=================================================================
// print_string.cpp By Frank Luna
//=================================================================

#include <iostream>
#include <string>

using namespace std;

int main()
{
     string firstName = "";

     cout << "Enter your first name and press Enter: ";

     cin >> firstName;

     cout << endl;

     cout << "Hello, " << firstName << endl << endl;
}
```

Note that you can "use" more than one namespace. For example, if you had defined another namespace called `math`, you could write:

```
     using namespace math;
     using namespace std;
```

The namespace/folder analogy, prefixing Standard Library code, and the `using namespace X` clause are all you need to know about namespaces for now. Note, however, that there is much more to namespaces than we covered here, such as creating your own namespaces, but we will defer a more detailed discussion until Chapter 5.

1.2.4 The `main{...}` Function

The `main{...}` function is special because it is the entry point for every C++ program; that is, every C++ program starts executing at `main`. ***Consequently, every C++ program must have a `main` function.*** The code inside a function's corresponding curly braces is called the **function body** or **function definition**. We will discuss in more detail what we mean by the term **function** in Chapter 3. For now, just understand that the first instruction in main's function body is the first instruction executed by the program.

In C++, the curly braces {} are used to define the beginning and the end of a logical code unit. For example, the code inside the braces of `main` defines the unit (function) `main`. The opening brace '{' denotes the beginning of a code block, and the closing brace '}' denotes the end of the code block. Braces must always be paired together; a beginning brace must always have an ending brace.

1.2.5 std::string

`std::string` is a C++ variable type (more on variables in Section 1.3) that represents a string of characters (i.e., text used for constructing words and sentences). For example, "hello," and "this is a string" are both strings. As you can tell by the `std::` prefix, `std::string` is part of the C++ standard library, and in order to use it we must write the include directive `#include <string>`.

In our example program, we declare a `std::string` variable called `firstName` (the variable name can be almost anything we want with a few exceptions—see Section 1.3.2) and define it to be an empty string (i.e. ""). This `std::string` variable `firstName` will be used to store (i.e., save) the first name the user enters into the program. `std::string` is a powerful variable type, and we will return to it in detail in Chapter 6, but for now it suffices to know that it is used to store text strings.

1.2.6 Input and Output with `std::cin` and `std::cout`

Program 1.1 is able to output and input data to and from the console window. By examining the code of Program 1.1 and its corresponding output, we might guess that "`std::cout <<`" outputs data to the console window and "`std::cin >>`" inputs information from the console window. This guess is, in fact, correct. Observe that `std::cout` and `std::cin` exist in the `std` namespace, and to use them we must write the include directive `#include <iostream>`.

For example, in Program 1.1 we display the text "Enter your first name and press Enter:" to the console window with the following line:

```
std::cout << "Enter your first name and press Enter: ";
```

We prompt the user to enter his/her name with the following line:

```
std::cin >> firstName;
```

The statement "`std::cout << std::endl;`" instructs the computer to output a new line. As a result, the program will move the cursor to the next line for either input or output.

Finally, we can chain outputs together with separate insertions per line, as we do with this call:

```
std::cout << "Hello, " << firstName << std::endl << std::endl;
```

Here we output "Hello, " followed by the string value stored in the variable `firstName`, followed by two "new line" commands. We can also chain inputs together, but this will be discussed later.

Note: The symbol '<<' is called the *insertion operator*. And the symbol '>>' is called the *extraction operator*. These names make sense when we consider that '<<' is used to insert output into an outbound stream of data, and '>>' is used to extract input from an inbound stream of data.

Important Note!
So far, almost every line of code in Program 1.1 has ended with a semicolon. These are examples of **statements**. A statement in C++ instructs the computer to do a specific action, such as create a variable, perform some arithmetic, or output some text. Of particular importance is that a statement ends with a semicolon (not a new line). A semicolon ends a C++ statement much like a period ends an English sentence. We could very well have statements that span multiple lines, but for readability purposes, this usually is not done.

1.3 Variables

In Program 1.1 we ask the user to enter his/her name. The program then says "Hello" to that name. The computer knows what name to say "Hello" to because we *saved* the name the user entered with the line:

```
std::cin >> firstName;
```

The command `std::cin >>` prompted the user to enter some text and when he entered it, the program saved what was entered in a string variable called `firstName`. Because the string has been saved, we can output the string with the following line:

```
std::cout << "Hello, " << firstName << std::endl << std::endl;
```

A **variable** occupies a region of physical system memory and stores a value of some type. There are several built-in C++ variable types which allow you to store different types of values, and as you will learn in later chapters, you can even make your own variable types. The following table summarizes the C++ variable types:

Table1.1: C++ Types.

Variable Type	Description
`std::string`	Used to store string variables. Note that `std::string` is not part of the core language, but part of the standard library.
`char`	Used to store single character variables such as ‘a’, ‘b’, ‘c’, etc. Usually this is an 8-bit value so that it can store up to 256 values—enough to represent the Extended ASCII Character Set. See Appendix A for a table of the Extended ASCII Character Set. Observe from the ASCII table, that characters are actually represented with integers, thus a `char` type is really an integer, but interpreted as a character.
`int`	The primary type used to store an integer value.
`short`	Typically stores a shorter range of integers than int.
`long`	A signed integer type that typically can store a longer range of integers than int.
`float`	Used to store floating point numbers; that is, numbers with decimals like 1.2345 and –32.985.
`double`	Similar to a `float`, but typically stores floating point numbers with greater precision than `float`.
`bool`	Used to store truth-values; that is, `true` or `false`. Note that `true` and `false` are C++ keywords. Also note that, in C++, zero is also considered false, and any non-zero value, negative or positive, is considered to be true.

The exact range of values each type can hold or the amount of memory that each type occupies is not noted in the table because these values are largely platform-dependent. A char may be 8-bits (1 byte) on some platforms, but may be 32-bits on another platform. Thus, you would usually like to avoid making assumptions about the C++ types in your code, in order that your code remain *portable* (i.e., works on other platforms). For example, if you assume in your code that `chars` are 8-bits, and then move development to a platform with 32-bit chars, you will have errors which will need to be fixed.

Note that you can use `std::cout` and `std::cin` to output and input these other types as well. For example, consider the following program.

Program 1.4: Type Input/Output.

```cpp
// Program asks user to enter different types of values, the
// program then echoes these values back to the console window.

#include <iostream>
#include <string>

using namespace std;

int main()
{
	// Declare variables and define them to some default value.
	char letter = 'A';
	int integer = 0;
	float dec   = 0.0f;

	cout << "Enter a letter: ";  // Output
	cin >> letter;               // Get input

	cout << "Enter an integer: ";// Output
	cin >> integer;              // Get input

	cout << "Enter a float number: "; // Output
	cin >> dec;                       // Get input

	cout << endl; // Output another new line

	// Now output back the values the user entered.
	cout << "Letter: "  << letter  << endl;
	cout << "Integer: " << integer << endl;
	cout << "Float: "   << dec     << endl;
}
```

Program 1.4: Output.

```
Enter a letter: F
Enter an integer: 100
Enter a float number: -5.987123

Letter: F
Integer: 100
Float: -5.98712
Press any key to continue
```

1.3.1 Variable Declarations and Definitions

When we write the following code, we say that we are **declaring** a variable:

```
	int myVar; // Variable Declaration.
```

In particular, we declare a variable called `myVar` of type `int`. Although `myVar` has been declared, it is **undefined**; that is, the value it stores is unknown. Consequently, it is common to say that the variable contains **garbage**. Try compiling and executing this small program to see what value `myVar` contains:

```
#include <iostream>
using namespace std;
int main()
{
	int myVar;
	cout << myVar << endl;
}
```

When we *assign* a value to a variable we are **defining** or **initializing** the variable:

```
	myVar = 10; // Variable Definition.
```

The '=' symbol, in C++, is called the **assignment operator**. Note that the assignment operator assigns values to variables—it says nothing about equality in a purely comparative sense. This is often confusing to new C++ students so be sure to try to remember this.

We can also declare and define a variable at the same time:

```
	int myVar = 10; // Variable Declaration and Definition.
```

Because variables that contain garbage are useless and prone to introducing errors, it is advisable to *always* define your variables at the time of declaration to some default value. Typically, zero is a good default value for numeric values, and an empty string, "", is a good default value for strings.

It is possible to make multiple declarations and/or definitions in one statement using the **comma operator** as this next code snippet shows:

```
	int x, y, z;
	x = 1, y = 2, z = 3;
	float i = 1.0f, j = 2.0f, k = 3.0f;
```

Despite being more compact, it is advised for readability purposes, to make one declaration or definition per line.

Finally, we can also chain assignments together (assigned values flow from right to left):

```
float num = 0.0f;
float a, b, c, d;
a = b = c = d = num;
```

1.3.2 Variable Names

As shown in Program 1.3, when we declare/define a variable we must give it a name (**identifier**) so that we can refer to that variable within the program. Variable names can be almost anything, with some exceptions:

1. Variable names must begin with a letter. The variable name `2VarName` is illegal. However, the underscore is considered a letter, and therefore, the identifier `_myVar` is legal.

2. Variable names can include the underscore ('_'), letters, and numbers, but no other symbols. For instance, you cannot use symbols like '!', '@', '#', '$', '%' in your variable names.

3. Variable names cannot be C++ keywords (Appendix B). For instance, you cannot name a variable "float" since that is a C++ keyword that specifies the C++ type `float`.

4. Variable names cannot have spaces between them.

   Note: `C++` is case sensitive. For example, these identifiers are all unique because they differ by case: `hello`, `Hello`, `HELLO`, `heLLo`.

1.3.3 The `sizeof` Operator

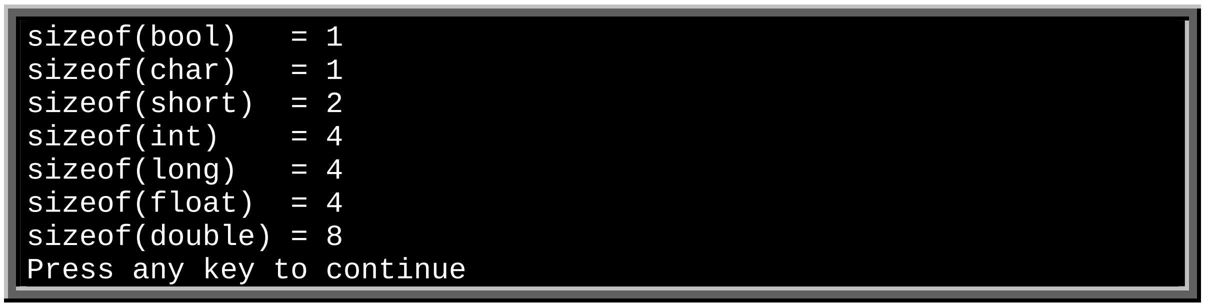
As stated in Section 1.3, the range of values of the C++ types and the amount of memory they occupy is platform-dependent. In order to get the size of a type, in bytes, on the current platform, you use the `sizeof` operator. Consider the following program, written on a 32-bit Windows platform:

Program 1.5: The "sizeof" Operator.

```
// Program outputs the size of various types.

#include <iostream>
using namespace std;
int main()
{
     cout << "sizeof(bool)   = " << sizeof(bool)   << endl;
     cout << "sizeof(char)   = " << sizeof(char)   << endl;
     cout << "sizeof(short)  = " << sizeof(short)  << endl;
     cout << "sizeof(int)    = " << sizeof(int)    << endl;
```

```
	cout << "sizeof(long)   = " << sizeof(long)   << endl;
	cout << "sizeof(float)  = " << sizeof(float)  << endl;
	cout << "sizeof(double) = " << sizeof(double) << endl;
}
```

Program 1.5: Output.



```
sizeof(bool)   = 1
sizeof(char)   = 1
sizeof(short)  = 2
sizeof(int)    = 4
sizeof(long)   = 4
sizeof(float)  = 4
sizeof(double) = 8
Press any key to continue
```

Note that these results will be specific to the platform on which the program is being executed. In fact, from these results we can infer the following value ranges and bytes required for these C++ types on the 32-bit Windows platform:

Type	Range	Bytes Required
`Char`	[–128, 127]	1
`Short`	[–32768, 32767]	2
`Int`	[–2147483648, 2147483647]	4
`Long`	[–2147483648, 2147483647]	4
`Float`	$\pm[1.2\times10^{-38}, 3.4\times10^{38}]$	4
`Double`	$\pm[2.2\times10^{-308}, 1.8\times10^{308}]$	8

Note that there is no difference in range or memory requirements between an `int` and a `long`.

1.3.4 The `unsigned` Keyword

An `int` supports the range [–2147483648, 2147483647]. However, if you only need to work with positive values, then it is possible to take the memory reserved for representing negative numbers and use it to represent additional positive numbers so that our range of positive values increases. This would be at the cost of sacrificing the ability to represent negative integers. We can do this by prefixing our integer types with the `unsigned` keyword. On 32-bit Windows we have the following:

Type	Range	Bytes Required
`unsigned char`	[0, 255]	1
`unsigned short`	[0, 65535]	2
`unsigned int`	[0, 4294967295]	4
`unsigned long`	[0, 4294967295]	4

Only integer types can be `unsigned`. By using unsigned types we have not gained a larger range, we have merely transformed our range. That is, an `int` type holds 4294967296 unique values (counting zero) whether we use the range [–2147483648, 2147483647] or the range [0, 4294967295].

1.3.5 Literal Assignments

Literals are values that are not variables. For example, *10* is literally the number ten. Likewise, the string *hello world* is literally the string “hello world.” The following code illustrates some literal assignments:

```
    bool b                 = true;     // boolean literal.
    char letter            = 'Z';      // character literal.
    std::string str        = "Hello";  // string literal.
    int num                = 5;        // integer literal.
    unsigned int uint      = 5U;       // unsigned integer literal
    long longNum           = 10L;      // long literal.
    unsigned long ulong    = 50UL;     // unsigned long literal.
    float floatNum         = 123.987f; // float literal.
    double dblNum          = 567.432;  // double literal.

    // Note that the literal type suffixes are not case
    // sensitive.  Thus, the following also works:

    unsigned long ulong2 = 50ul;     // unsigned long literal.
    float floatNum2      = 123.987F; // float literal.
```

There is some ambiguity with floating point numbers. For example, it is unclear whether `123.987` should be treated as a `float` or a `double`. To avoid this ambiguity, C++ allows us to attach a type suffix to the literal. If the literal has an ‘f’ suffix as in `123.987f` then it is treated as a `float`; otherwise it is treated as a `double`. A similar problem arises with longs and unsigned types. Observe `uint`, `longNum`, and `ulong`, in the above code, where we use the U, L, or combination UL to denote the literal type.

1.3.6 Type Conversions

It is possible to make variable assignments between various types. For instance, we can assign an `int` variable to a `float` variable and vice versa. However, there are some caveats. Program 1.6 illustrates this.

Program 1.6: Type Conversions.

```
// Demonstrates some type conversions.

#include <iostream>

using namespace std;

int main()
{
     // Case 1: Convert from a less precise type
     // to a more precise type:

     char  c = 10;
     short s = c;
     cout << "char to short: " << s << endl;

     // Case 2: Convert from a more precise integer
     // to a less precise integer:

     unsigned char uc = 256;
     cout << "int to uchar: " << (int)uc << endl;

     // Case 3: Convert from a float to an int,
     // assuming the int can store the float's value.

     int i = 496512.546f;
     cout << "float to int: " << i << endl;

     // Case 4: Convert from a float to a short, this
     // time the int can't store the float:

     s = 496512.987123f;
     cout << "float to short: " << s << endl;
}
```

Program 1.6: Output.

```
char to short: 10
int to uchar: 0
float to int: 496512
float to short: -27776
Press any key to continue
```

- Case 1: Here we convert from a less precise type to a more precise type. Since the more precise type can fully represent the less precise type, there is no conversion problem and everything works out as expected.

- Case 2: Here we convert from a more precise integer to a less precise integer. However, an `unsigned char` cannot represent the value 256. What happens is called *wrapping*. The `unsigned char` cannot store values over 255 so it wraps back to zero. Thus 256 becomes zero, 257 would become 1, 258 would become 2, and so on. The values "wrap" back around. Wrapping also occurs in the opposite direction. For instance, if we assigned –1 to an `unsigned char`, the value would wrap around in the other direction and become 255.

- Case 3: Here we assign a `float` to an `int`. Observe that the `float` is *truncated*—the `float` loses its decimal.

- Case 4: Here we assign a `float` to a `short`, but the `short` cannot store the whole number part of the `float`. Thus we observe a wrapping scenario.

  Note: Integer wrapping is a serious problem that can lead to hard-to-find bugs. Thus, you should always ensure that you are working with values that your types can correctly store.

The C++ compiler does these type conversions implicitly; that is, automatically. However, sometimes you need to explicitly tell the compiler to treat a type as a different type. We actually see this in Case 2 of Program 1.6. Specifically, the line:

```
cout << "int to uchar: " << (int)uc << endl;
```

The `(int)uc` syntax tells the compiler to treat `uc` as an `int`, not as a char. This is because `cout` will output the character representation of `uc` since it is a `char`. But we want it to output the integer representation of the `char`. Thus we must perform a *type cast* (conversion between types) and explicitly tell the compiler to treat `uc` as an `int`. In general, this type of casting can be done with either of the following syntaxes:

```
result = static_cast<typeToConvertTo>(valueToConvert);
result = (typeToConvertTo) valueToConvert;
```

Examples:

```
int         x = 5;
float result = static_cast<float>(x);
int         y = (int)result;
```

1.3.7 `Typedef`s

At some point, you might find a C++ type that is too long. For example, `unsigned int` is a lot to write out repeatedly. C++ allows you to define an alternate name (synonym) via the `typedef` keyword. For example, we might define the following shorter names for the unsigned types:

```
typedef unsigned char uchar;
typedef unsigned short ushort;
typedef unsigned int uint;
typedef unsigned long ulong;
```

Thus, for example, instead of having to write:

```
unsigned int x = 19;
```

We can simply write:

```
uint x = 19;
```

1.3.8 Const Variables

Sometimes we will want to define a variable that cannot change. Such variables are constant. For example, we may want to define a constant variable `pi` to represent the mathematical constant $\pi \approx 3.14$. To do this we use the **const** modifying keyword:

```
const float pi = 3.14f;
```

If the programmer tries to change `pi`, an error will result:

```
pi = 12.53f; // error, cannot redefine constant pi.
```

Constant variables are often used for notational convenience. It is clearer to read the symbolic name "pi" than it is to read the number 3.14; a programmer may not immediately connect 3.14 to π.

1.3.9 Macros

Sometimes we want to create a symbol name (identifier) that stands for some set of code. We can do this by defining a **macro**. For example, the following line of code defines a macro `PRINTHELLO`, which when written, executes the statement: `cout << "Hello" << endl;`.

```
// Define a macro PRINTHELLO that means
// "cout << 'Hello' << endl;"
#define PRINTHELLO cout << "Hello" << endl;
```

Using this macro we could write a program that outputs “Hello” several times like so:

Program 1.7: Macros

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

// Define a macro PRINTHELLO that means
// "cout << 'Hello' << endl;"
#define PRINTHELLO cout << "Hello" << endl;

int main()
{
      PRINTHELLO
      PRINTHELLO
      PRINTHELLO
      PRINTHELLO
}
```

Note that the semi-colon is included in the macro definition and thus we did not need to place one at the end of each line.

Program 1.7 Output

```
Hello
Hello
Hello
Hello
Press any key to continue
```

When the compiler compiles the source code and encounters a macro, it internally replaces the macro symbol with the code for which it stands. Program 1.7 is expanded internally as:

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

1.4 Arithmetic Operations

In addition to declaring, defining, inputting, and outputting variables of various types, we can perform basic arithmetic operations between them.

1.4.1 Unary Arithmetic Operations

A *unary operation* is an operation that acts on one variable (operand). Table 1.2 summarizes three unary operators:

Table 1.2: Unary Operations.

Unary Operator	Description	Example
Negation Operator: -	Negates the operand.	int x = 5; int y = -x; // y = -5.

Increment Operator: ++	Increments the operand by one. Note that the increment operator can prefix or postfix the operator.	int x = 5;
		++x; // x = 6 (prefix)
		x++; // x = 7 (postfix)

Decrement Operator: --	Decrements the operand by one. Note that the decrement operator can prefix or postfix the operator.	int x = 5;
		--x; // x = 4 (prefix)
		x--; // x = 3 (postfix)

Observe that the increment and decrement operators can be written as a prefix or postfix. The technical difference between prefix and postfix is determined by where the increment/decrement occurs in a statement. The following program illustrates:

Program 1.8: Prefix versus Postfix.

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

int main()
{
	int k = 7;
	cout << "++k = " << ++k << endl;
	cout << "k++ = " << k++ << endl;

	cout << k << endl;
}
```

Program 1.8: Output.

```
++k = 8
k++ = 8
```

```
9
Press any key to continue
```

Do you see the difference between the prefix and postfix increment/decrement operator? In the first call to `cout <<`, `k` is incremented first, before being displayed. Hence the number 8 is displayed. However, in the second call to `cout <<`, `k` is first displayed, and then is incremented. Hence the number 8 is displayed again. Finally, we output `k` a third time to show that it was indeed incremented, but only after it was displayed the second time. Therefore, in a complex expression, when using the prefix form, the value is incremented/decremented first, before it is used. Conversely, when using the postfix form, the value is incremented/decremented last, after it is used.

1.4.2 Binary Arithmetic Operations

C++ contains five binary arithmetic operators:

1) Addition operator (+),
2) Subtraction operator (-),
3) Multiplication operator (*),
4) Division operator (/)
5) Modulus operator (%).

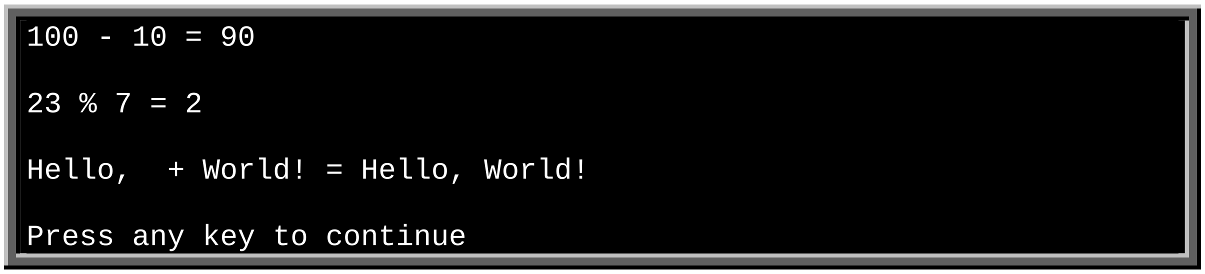
Addition, subtraction, multiplication and division are defined for all numeric types. The modulus operator is an integer operation only. The only arithmetic operation defined for `std::string` is the addition operator. The following program illustrates the arithmetic operations:

Program 1.9: Arithmetic Operations.

```
// Program demonstrates some arithmetic operations.

#include <iostream>
#include <string>

using namespace std;

int main()
{
	//=========================
	// Do some math operations:

	float f1 = 10.0f * 10.0f;
	float f2 = f1 / 10.0f;
	float fDif = f1 - f2;

	cout << f1 << " - " << f2 << " = " << fDif;
	cout << endl << endl;
```

```
      //=============================
      // Do some integer operations:

      int i1 = 19 + 4;
      int i2 = 10 - 3;

      int remainder = i1 % i2;

      cout << i1 << " % " << i2 << " = " << remainder;
      cout << endl << endl;

      //============================
      // Do some string operations:

      string s1 = "Hello, ";
      string s2 = "World!";

      string stringSum = s1 + s2;

      cout << s1 << " + " << s2 << " = " << stringSum;
    cout << endl << endl;
}
```

Program 1.9: Output.



```
100 - 10 = 90

23 % 7 = 2

Hello,  + World! = Hello, World!

Press any key to continue
```

1.4.3 The Modulus Operator

The modulus operator returns the remainder of an integer division. For example,

$$\frac{23}{7} = 3 + \frac{2}{7}.$$

Here we call the numerator 2, in $2/7$, the *remainder*—it is the remaining part that cannot be divided evenly by seven. (We will say two integers *divide evenly* if and only if the division results in an integer; i.e., not a fraction.)
Consider the example, $5/13$. In this case, the remainder is five; that is, 13 divides into 5 zero times, and so the remaining part that cannot be evenly divided by 13 is 5.

1.4.4 Compound Arithmetic Operations

C++ defines the following “shortcut” operators that really perform two operations, namely an arithmetic operation and an assignment operation. The following table summarizes:

Table 1.3: Compound Arithmetic Operations.

Compound Arithmetic Operation	Equivalent Meaning
x += y	x = x + y
x -= y	x = x – y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y

The following program illustrates how they are used in a C++ program:

Program 1.10: Compound Arithmetic Operators.

```
#include <iostream>

using namespace std;

int main()
{
	int x = 0;
	int y = 0;

	cout << "Enter an integer: ";
	cin >> x;

	cout << "Enter an integer: ";
	cin >> y;

	// Save to separate variables so each operation is
	// independent of each other.

	int a = x;
	int b = x;
	int c = x;
	int d = x;
	int e = x;

	a += y;
	b -= y;
	c *= y;
	d /= y;
	e %= y;

	cout << "x += y = " << a << endl;
```

```
     cout << "x -= y = " << b << endl;
     cout << "x *= y = " << c << endl;
     cout << "x /= y = " << d << endl;
     cout << "x %= y = " << e << endl;
}
```

Program 1.10: Output.

```
Enter an integer: 50
Enter an integer: 12
x += y = 62
x -= y = 38
x *= y = 600
x /= y = 4
x %= y = 2
Press any key to continue
```

Note: The output of Program 1.9 brings up an important point. Namely, 50 / 12 is not 4, but approximately 4.1667. What happened? The decimal portion of the answer is lost because integers are being used, and they are unable to represent decimals. Bear this truncation in mind when doing division with integers and ask yourself whether or not this is a problem for your particular circumstances.

1.4.5 Operator Precedence

Consider the following statement:

```
int x = 5 + 3 * 8;
```

In which order will the compiler perform the various arithmetic operations? Each operator has a defined precedence level, and operators are evaluated in the order of greatest precedence to least precedence. Multiplication, division and the modulus operations have the same precedence level. Similarly, addition and subtraction have the same precedence level. However, the precedence level of multiplication, division, and modulation is greater than that of addition and subtraction. Therefore, multiplication, division, and modulation operations always occur before addition and subtraction operations. Thus the above expression is evaluated like so:

```
int x = 5 + 3 * 8;
      = 5 + 24
      = 29
```

Note that operators with the same precedence level are evaluated left to right.

Sometimes you want to force an operation to occur first. For example, you may have actually wanted the addition to take place before the multiplication. You can give greatest precedence to an operation by surrounding it with a pair of parentheses, much like you do in mathematical notation. We can force the addition to come before multiplication by using the following parentheses:

```
int x = (5 + 3) * 8;
      = 8 * 8
      = 64
```

Parentheses can also be nested so that you can explicitly specify the second, third, etc. operation that should occur. In an expression with nested parentheses, the operations are evaluated in the order from the innermost parentheses to the outermost parentheses. The following example illustrates:

```
int x = (((5 + 3) - 2) * 6) + 5;
      = ((8 – 2) * 6) + 5
      = (6 * 6) + 5
      = 36 + 5
      = 41
```

Note: Observe how arithmetic operations evaluate to a numeric value. The terminology for something that evaluates to a number is called a **numeric expression.** More generally, something that evaluates to something else is considered an **expression.** As you will see in the next chapter, there exists logical expressions (expressions with logical operators), which evaluate to truth-values.

1.5 Summary

1. A C++ compiler translates C++ code to object code. The linker then combines the object code, from several object files, to produce an executable program that can be run on the operating system (i.e., machine language code).

2. White space consists of blank lines and spaces, which are ignored by the compiler. Use white space to format your code in a more readable way.

3. The C++ standard library includes code for outputting and inputting data to and from the console window, functions for computing various math operations such as sine and cosine, random number generators, code for saving and loading files to and from the hard drive, code to work with strings, and much, much more.

4. Every C++ program must have a `main` function, which defines the program entry point.

5. Namespaces are used to organize code into logical groupings and to prevent name clashes.

6. A variable occupies a region of physical system memory and stores a value of some type. Variable names can consist of letters and numbers, but cannot begin with a number (the underscore is considered a letter). Recall that C++ is case sensitive.

7. C++ can implicitly convert between its intrinsic types; however, one must be alert for decimal truncation and integer wrapping. It is good practice to try and avoid type conversions when practical.

8. The rules of operator precedence define the order in which the compiler performs a series of operations. Use parentheses to explicitly define the order in which the operations should be performed. Consequently, this also makes your code clearer.

1.6 Exercises

1.6.1 Arithmetic Operators

Write a program that asks the user to input two real numbers, n_1 and n_2. Compute the sum $n_1 + n_2$, the difference $n_1 - n_2$, the product $n_1 \cdot n_2$, and the quotient n_1 / n_2 (assume $n_2 \neq 0$), and output the results. Your program output should be formatted as follows:

```
Enter a real number n1: 64.67
Enter a real number n2: -14.2
64.67 + -14.2 = 50.47
64.67 - -14.2 = 78.87
64.67 * -14.2 = -918.314
64.67 / -14.2 = -4.55423
Press any key to continue
```

1.6.2 Cin/Cout

Rewrite the program example given in Section 1.1.3. This time, ask the user to enter his first *and* last names, separated by a space, on one line (i.e., use one "`cin >>`" operation to read both the first and last name in one pass). Does a problem occur? If so, describe it in complete sentences. Then try and find a temporary "workaround" to the problem, by any means possible.

1.6.3 Cube

Write a program that asks the user to input a real number *n*. Compute n^3 and output the result. Your program output should be formatted as follows:

```
Enter a real number: 7.12
7.12^3 = 360.944
Press any key to continue
```

1.6.4 Area/Circumference

Write a program that asks the user to input the radius *r* of a circle. Compute the area *A* and circumference *C* of the circle using the formulas $A = \pi \cdot r^2$ and $C = 2 \cdot \pi \cdot r$ respectively, and output the results. Note that for this exercise you can make the approximation $\pi \approx 3.14$. Your program output should be formatted as follows:

```
Enter the radius of a circle: 10
The area A of a circle with radius 10 = 314
The circumference C of a circle with radius 10 = 62.8
Press any key to continue
```

1.6.5 Average

Write a program that asks the user to input five real numbers. Compute the average of these numbers and output the results to the user. Your program output should be formatted as follows:

```
Enter a0: 5
Enter a1: 10
Enter a2: -2
Enter a3: 2.7
Enter a4: 0
The average of the five inputs a0...a4 = 3.14
Press any key to continue
```

1.6.6 Bug Fixing

The following program contains several bugs. Enter the program into your C++ editor and try to compile it. What error/warning messages do you get? In complete sentences, describe what you think each error/warning message means. Afterwards, fix the errors so that it compiles correctly.

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

int mian()
{
      string str = "Hello World!"

      cout << str << endl;

      cout << float x = 5.0f * str << end;

      int 65Num = 65;

      cout << "65Num = " < 65Num << endl;
}
```

Chapter 2

Logic, Conditionals, Loops and Arrays

Introduction

The previous chapter covered the creation of trivial C++ programs, but we are still very limited in terms of the ideas we can express using C++. In this chapter we expand our C++ "vocabulary" and "grammar" in order to program more interesting actions such as, "If the player's hitpoints are less than zero then the player is dead and the game is over." In addition, we will learn how to execute blocks of code repeatedly. This is particularly useful in games where we need to *repeatedly* check user input and update the game world accordingly. Finally, we will learn how to store a collection of variables in a logical container. In a casino game example, this would be useful to represent a deck of cards programmatically; you could keep a container of 52 `ints`, where each element in the container represents a card in the deck.

Chapter Objectives:

- Understand and evaluate logical expressions.
- Form and apply conditional, if…then, statements.
- Discover how to execute a block of code repeatedly using the various kinds of loop statements C++ exposes.
- Learn how to create containers of variables and how to manipulate the individual elements in those containers.

2.1 The Relational Operators

The relational operators are fairly straightforward because you are probably already familiar with their concepts from your high school algebra classes. The relational operators allow us to determine some elementary relationships between variables. For example, given the following two variables:

```
int var1 = 15;
int var2 = 7;
```

What can we say about them? One thing we can determine is that they are not equal. Moreover, we can say `var1` is greater than `var2`, or `var2` is less than `var1`. The C++ relational operators allow us to express these types of ideas in code. Note that relational operations are a type of boolean expression. A boolean expression is one that evaluates to a truth-value—true or false. For example, when relating two objects, it can be said that they are either equal (equality is true) or not equal (equality is false).

Table 2.1 summarizes the C++ relational operators:

Table 2.1: The Relational Operators.

Relational Operator	Description
Equals operators ==	Recall that the single equal sign '=' is the assignment operator. Consequently, C++ uses a double equal sign '==' to test

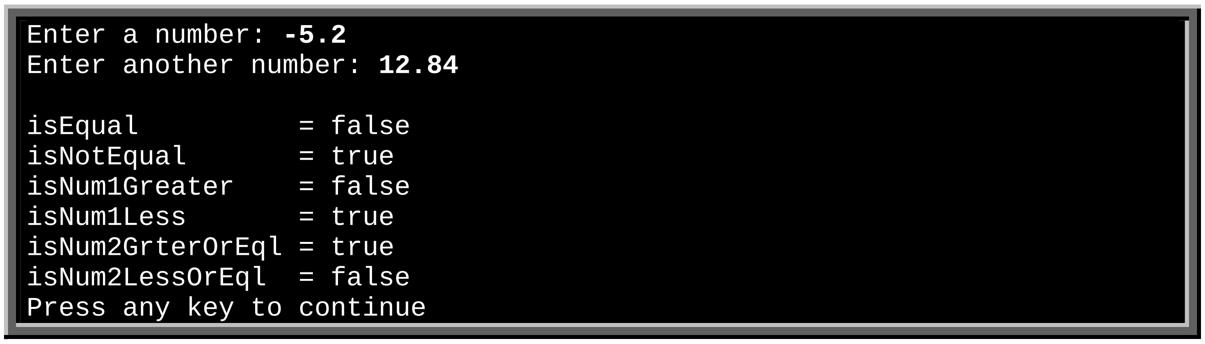
	equality. This operator returns `true` if the two operands are equal, otherwise it returns `false`.
Not equals operator `!=`	The not equals operator returns `true` if the two operands are *not* equal, otherwise it returns `false`.
Less than operator `<`	The less than operator returns `true` if the left hand operand is less than the right hand operator, otherwise it returns `false`.
Greater than operator `>`	The greater than operator returns `true` if the left hand operand is greater than the right hand operator, otherwise it returns `false`.
Less than or equals operator `<=`	The less than or equals operator returns `true` if the left hand operand is less than or equals the right hand operator, otherwise it returns `false`.
Greater than or equals operator `>=`	The greater than or equals operator returns `true` if the left hand operand is greater than or equals the right hand operator, otherwise it returns `false`.

To see how these operators can be used in a C++ program, consider the following program:

Program 2.1: Relational operations.

```
// Program demonstrates how the relational operators
// are evaluated.

#include <iostream>
using namespace std;

int main()
{
      // Set output flag so that 'bool' variables are
      // output as 'true' and 'false' instead of '1'
      // and '0', respectively.
      cout.setf( ios_base::boolalpha );

      float num1 = 0.0f;
      float num2 = 0.0f;

      cout << "Enter a number: ";
      cin >> num1;

      cout << "Enter another number: ";
      cin >> num2;

      bool isEqual    = num1 == num2;
      bool isNotEqual = num1 != num2;

      bool isNum1Greater = num1 > num2;
      bool isNum1Less    = num1 < num2;

      bool isNum2GrterOrEql = num2 >= num1;
      bool isNum2LessOrEql  = num2 <= num1;

      cout << endl;
      cout << "isEqual          = " << isEqual          << endl;
      cout << "isNotEqual       = " << isNotEqual       << endl;
      cout << "isNum1Greater    = " << isNum1Greater    << endl;
      cout << "isNum1Less       = " << isNum1Less       << endl;
      cout << "isNum2GrterOrEql = " << isNum2GrterOrEql << endl;
      cout << "isNum2LessOrEql  = " << isNum2LessOrEql  << endl;
}
```

Output 2.1



```
Enter a number: -5.2
Enter another number: 12.84

isEqual          = false
isNotEqual       = true
isNum1Greater    = false
isNum1Less       = true
isNum2GrterOrEql = true
isNum2LessOrEql  = false
Press any key to continue
```

Observe that we store the result of a relational expression in a `bool` value. Recall that a `bool` variable type is used to store truth-values—`true` or `false`.

> **Note:** In Program 2.1 we use a line that we have not seen before; that is,
>
> `cout.setf( ios_base::boolalpha );`
>
> This line is used to set formatting flags which control the way `cout` outputs information. In this case we pass a flag `ios_base::boolalpha`, which tells `cout` to output `bool` values as "true" or "false." If we did not include this line, `cout` would output true values as "1" and false values as "0." We will discuss other various formatting flags as we across them in this course.

2.2 The Logical Operators

When working with boolean values (true or false values), there are three other elementary types of things we would like to say. Consider two `bool` variables, A and B. Eventually, we will need to make statements such as these: "If A *and* B are both true then this," or "While either A *or* B is true then this," or "If A and *not* B are both true then this." To express these ideas in code, C++ provides a logical **and** operator (&&), a logical **inclusive or** operator (`||`), and a logical **not** operator (`!`).
The '&&' and '`||`' operators are binary operators; that is, they act on two operands. Conversely, the '`!`' operator is a unary operator which acts on one operand. Note that the truth-value of these logical operations depends on the truth-value of the operand(s). The following truth tables, where 'T' denotes true and 'F' denotes false, show the truth-values of these logical operators for different truth combinations of A and B.

Table 2.1: The logical AND (&&) truth table. Observe from the four possible combinations that an AND operation is true if and only if both of its operands are true.

A	B	A && B
T	T	T
T	F	F
F	T	F
F	F	F

Table 2.2: Logical OR (||) truth table. Observe from the four possible combinations that an OR operation is true if and only if at least one of its operands is true.

A	B	A || B
T	T	T
T	F	T
F	T	T
F	F	F

Table 2.3: Logical NOT (!) truth table. The not operator simply negates the truth-value of a boolean operand—true becomes false when negated and false becomes true when negated.

A	!A
T	F
F	T
---	---
---	---

> **Note:** An **inclusive or**, `A || B`, is true if A is true, or B is true, or both are true. An **exclusive or** is true if A is true or B is true, *but not both*. Observe that we do not need an *exclusive or* operator because we can formulate an *exclusive or* out of our three existing operators. For instance, the following logical expression evaluates to true if either A is true or B is true, but not A and B (not both):
>
> *Exclusive OR:* `(A || B) && !(A && B)`

Before looking at a sample program, let us evaluate some boolean expressions with the logical operators by hand first. Suppose `A` is true, `B` is false, and `C` is true.

Example 1: Evaluate `A && !B`.

Substituting the known truth-values (using 'T' for true and 'F' for false) into the expression and evaluating step-by-step yields:

```
  A && !B
= T && !F
= T && T
= T
```

Example 2: Evaluate `!B || C`.

Substituting our known truth-values (using ‘T’ for true and ‘F’ for false) into the expression and evaluating step-by-step yields:

```
  !B || C
= !F || T
=  T || T
=  T
```

Example 3: Evaluate `(A || C) && !(A && C)`.

Substituting our known truth-values (using ‘T’ for true and ‘F’ for false) into the expression and evaluating step-by-step yields:

```
  (A || C) && !(A && C)
= (T || T) && !(T && T)
=  T && !T
=  T && F
=  F
```

Although logical operators are mostly used in conditional statements, Program 2.2 shows how the logical expressions are evaluated.

Program 2.2: Logical expressions evaluated.

```
// Program demonstrates how the logical operators
// are evaluated.

#include <iostream>
using namespace std;

int main()
{
      cout.setf( ios_base::boolalpha );

      bool B0 = false;
      bool B1 = false;
      bool B2 = false;

      cout << "Enter 0 for false or 1 for true: ";
      cin >> B0;
      cout << "Enter 0 for false or 1 for true: ";
      cin >> B1;

      cout << "Enter 0 for false or 1 for true: ";
      cin >> B2;

      bool notB0 = !B0;
      bool notB1 = !B1;
      bool notB2 = !B2;

      bool isB0AndB1      = B0 && B1;
      bool isB0AndB1AndB2 = B0 && B1 && B2;

      bool isB0OrB1       = B0 || B1;
      bool isB1OrB2       = B1 || B2;

      // Exclusive OR...
      bool isB0ExclOrB1   = (B0 || B1) && !(B0 && B1);

      // Complex logical expression...
      bool isComplex = (B0 && (B1 || B2)) &&
                      !((B0 && B1) || (B0 && B2));

      cout << "B0             = " << B0             << endl;
      cout << "B1             = " << B1             << endl;
      cout << "B2             = " << B2             << endl;
      cout << "notB0          = " << notB0          << endl;
      cout << "notB1          = " << notB1          << endl;
      cout << "notB2          = " << notB2          << endl;
      cout << "isB0AndB1      = " << isB0AndB1      << endl;
      cout << "isB0AndB1AndB2 = " << isB0AndB1AndB2 << endl;
      cout << "isB0OrB1       = " << isB0OrB1       << endl;
      cout << "isB1OrB2       = " << isB1OrB2       << endl;
      cout << "isB0ExclOrB1   = " << isB0ExclOrB1   << endl;
      cout << "isComplex      = " << isComplex      << endl;
}
```

Output 2.2

```
Enter 0 for false or 1 for true: 0
Enter 0 for false or 1 for true: 1
Enter 0 for false or 1 for true: 0
B0             = false
B1             = true
B2             = false
notB0          = true
notB1          = false
notB2          = true
isB0AndB1      = false
isB0AndB1AndB2 = false
isB0OrB1       = true
isB1OrB2       = true
isB0ExclOrB1   = true
isComplex      = false
Press any key to continue
```

For the given program, input `B0 = 0 = false`, `B1 = 1 = true`, `B2 = 0 = false`[1], it is important that each logical expression from this program be evaluated manually (as was done in the preceding three examples) and the results verified with the program's output. Do your results match the program's output?

[1] Recall that C++ treats zero as false and any non-zero number, positive or negative as true.

Also try the program using different inputs, and again verify the resulting program output by first evaluating the logical operations yourself, either mentally or on paper. It is important that you are able to evaluate logical expressions mentally and quickly, and some of the exercises of this chapter will help you in building this skill.

The "hardest" logical expression Program 2.2 performs is the "complex" one. Therefore, we will walk through the evaluation of this expression step-by-step for the input given in the sample run. For this particular sample run, we entered 0, 1, 0, respectively, and therefore:

```
B0 = false
B1 = true
B2 = false
```

Recall that the logical expression was:

```
isComplex = (B0 &&(B1 || B2)) && !((B0 && B1) || (B0 && B2));
```

We will use 'T' to denote true and 'F' to denote false.

```
isComplex = (B0 && (B1 || B2)) && !((B0 && B1) || (B0 && B2));
          = (F  && (T  || F )) && !((F  && T ) || (F  && F ))
          = (F  &&     (T)   ) && !(    (F)    ||     (F)   )
          = (F && T) && !(F || F)
          =    (F)   &&    !(F)
          = F && T
          = F
```

Not coincidentally, this is the same result which Program 2.2 calculated.

Finally, observe that like the arithmetic operators, we can use parentheses to control the order in which the logical operators are evaluated.

2.3 Conditional Statements: If, If…Else

Now that we can form boolean expressions with both the relational and logical operators, we can begin to form **conditional statements.** In general, a conditional statement takes on the form: "If A is true then P follows," where A is some boolean expression (i.e., evaluates to either true or false) and P is a statement (or statements) that follow on the condition that A is true (i.e., A implies P). The statement after the "if" and before the "then" is called the **antecedent** and the statement(s) which follow(s) the "then" is called the **consequent**. For example, in the statement "If the player's hitpoints are less than zero then he is dead," the antecedent would be "the player's hitpoints are less than zero" and the consequent would be "he is dead."

Clearly, conditional statements are the key to any computer program, as the program must be able to make various decisions based on current conditions and react to user input. For example, in a game we need to be able to test whether game objects have collided (e.g., if collided then execute collision physics), or whether the player still has enough magic points to cast a spell (e.g., if magic points are greater than zero then cast magic missile), or whether the user has pressed a key or mouse button (e.g., if left arrow key pressed then strafe left; if right mouse button pressed then fire secondary weapon).

2.3.1 The If Statement

In the context of C++, we use conditional statements to control program flow. If the antecedent is true then execute the consequent C++ statement(s). For example, consider the following short program:

Program 2.3: Program demonstrates the 'if' statement.

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

int main()
{
	float num = 0.0f;
	cout << "Enter a real number: ";
	cin >> num;

	cout << endl;
	char cube = 0;
	cout << "Cube " << num << " ??? (y)es / (n)o: ";
	cin >> cube;

	// Did the user enter a 'y' or 'Y' for yes--test both uppercase
	// and lowercase version.
	if( cube == 'y' || cube == 'Y' )
		num = num * num * num;

	cout << endl;
	cout << "num = " << num << endl;
}
```

Output 2.3a.

```
Enter a real number: 2
Cube 2 ??? (y)es / (n)o: Y
num = 8
Press any key to continue
```

Output 2.3b.

```
Enter a real number: 2
Cube 2 ??? (y)es / (n)o: n
num = 2
Press any key to continue
```

If `(cube == 'y' || cube == 'Y')` evaluates to `true` then the code "`num = num * num * num;`" is executed, otherwise it is not executed. Thus an "if" statement can be used to control which code is to be executed based on the condition.

Program 2.3 executes one statement if the condition,`(cube == 'y' || cube == 'Y'),` is true. More than one statement can be executed in consequence but to do so we must use a **compound statement,** which is a set of statements enclosed in curly braces. Program 2.4 illustrates this (new lines have been bolded).

Program 2.4: Compound Statements.

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

int main()
{
	float num = 0.0f;
	cout << "Enter a real number: ";
	cin >> num;

	char cube = 0;
	cout << "Cube " << num << " ??? (y)es / (n)o: ";
	cin >> cube;

	if( cube == 'y' || cube == 'Y' )
	{
		cout << "Cubing num..." << endl;
		num = num * num * num;
		cout << "Done cubing..." << endl;
	}

	cout << "num = " << num << endl;
}
```

Output 2.4.

```
Enter a real number: 4
Cube 4 ??? (y)es / (n)o: y
Cubing num...
Done cubing...
num = 64
Press any key to continue
```

Note: The indentation of the line that follows the `if( boolExpression )` line is made purely for readability purposes in order to identify the code that is to be executed if `boolExpression` is true. Remember, white space is ignored.

2.3.2 The Else Clause

To give more control over the program execution flow, C++ provides an `else` clause. The `else` clause allows us to say things like “If A is true then P follows, else Q follows.” Program 2.5 demonstrates how the `else` clause can be used.

Program 2.5: The else clause.

```
// Program demonstrates the 'else' clause.

#include <iostream>
using namespace std;

int main()
{
      int num = 0;
      cout << "Enter an integer number: ";
      cin >> num;

      // Test whether the entered number is >= to zero.
      if( num >= 0 )
      {
            // If num >= 0 is true, then execute this code:
            cout << num << " is greater than or equal to zero.";
            cout << endl;
      }
      else // num < 0
      {
            // If num >= 0 is *not* true then execute this code:
            cout << num << " is less than zero." << endl;
      }
}
```

Output 2.5a.

```
Enter an integer number: -45
-45 is less than zero.
Press any key to continue
```

Output 2.5b.

```
Enter an integer number: 255
255 is greater than or equal to zero.
Press any key to continue
```

Program 2.5 asks the user to enter an integer number *n* and then outputs some information based on the relationship between *n* and the number zero. As you can see, due to the conditional statements, the code that is executed varies depending on the input.

2.3.3 Nested If...Else Statements

An if…else statement can branch execution along two separate paths—one path if the condition is true or a second path if the condition is not true. However, what if you need more than two execution paths? For instance, what if you wanted to say: "if A then B else, if C then D else, if E then F else G." This can be achieved by using several nested if…else statements. Program 2.6 shows how this might be used to initialize a player's character class in a fantasy role-playing-type game based on the character the user selected.

Program 2.6: Program illustrates creating multiple execution paths using nested if…else statements.

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

int main()
{
      // Output some text asking the user to make a selection.
      cout << "Welcome to Text-RPG 1.0" << endl;
      cout << "Please select a character class number..."<< endl;
      cout << "1)Fighter 2)Wizard 3)Cleric 4)Thief : ";

      // Prompt the user to make a selection.
      int characterNum = 1;
      cin >> characterNum;

      // Initialize character variables to default value.
      int numHitpoints       = 0;
      int numMagicPoints     = 0;
      string weaponName = "";
      string className  = "";

      if( characterNum == 1 ) // Fighter selected?
      {
            numHitpoints   = 10;
            numMagicPoints = 4;
            weaponName     = "Sword";
            className      = "Fighter";
      }
      else if( characterNum == 2 ) // Wizard selected?
      {
            numHitpoints   = 4;
            numMagicPoints = 10;
            weaponName     = "Magic Staff";
            className      = "Wizard";
      }
      else if( characterNum == 3 ) // Cleric selected?
      {
            numHitpoints   = 7;
            numMagicPoints = 7;
            weaponName     = "Magic Staff";
            className      = "Cleric";
      }
      else // Not 1, 2, or 3, so select thief.
      {
            numHitpoints   = 8;
            numMagicPoints = 6;
            weaponName     = "Short Sword";
            className      = "Thief";
      }

      cout << endl;
      cout << "Character properties:" << endl;
      cout << "Class name  = " << className      << endl;
      cout << "Hitpoints   = " << numHitpoints   << endl;
      cout << "Magicpoints = " << numMagicPoints << endl;
      cout << "Weapon      = " << weaponName     << endl;
}
```

Output 2.6.

```
Welcome to Text-RPG 1.0
Please select a character class number...
1)Fighter 2)Wizard 3)Cleric 4)Thief : 2

Character properties:
Class name  = Wizard
Hitpoints   = 4
Magicpoints = 10
Weapon      = Magic Staff
Press any key to continue
```

Here we provide four different execution paths based on whether the user chose a “Fighter,” “Wizard,” “Cleric,” or “Thief.” Adding more execution paths is trivial. You need only add more “else if” statements in the pattern shown.

2.3.4 The Switch Statement

The switch statement is essentially a cleaner alternative to nested if…else statements. It is best explained by example, so consider the following:

Program 2.7: Program illustrates creating multiple execution paths using the switch statement.

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

int main()
{
      int num = 0;
      cout << "Enter an even integer in the range [2, 8]: ";
      cin >> num;

      switch( num )
      {
      case 2:
            cout << "Case 2 executed!" << endl;
            break;
      case 4:
            cout << "Case 4 executed!" << endl;
            break;
      case 6:
            cout << "Case 6 executed!" << endl;
            break;
      case 8:
            cout << "Case 8 executed!" << endl;
            break;
      default:
            cout << "Default case executed implies you do not ";
            cout << "enter a 2, 4, 6, or 8." << endl;
            break;
      }
}
```

Output 2.7.

```
Enter an even integer in the range [2, 8]: 4
Case 4 executed!
Press any key to continue
```

Here `num` is the value to test against several possible *cases*. The code first compares `num` against the first `case 2`. If `num` equals 2 then the code following `case 2` is executed; otherwise, the code jumps to the next case—`case 4`. If `num` equals 4 then the code following `case 4` is executed; otherwise, the code jumps to the next case, and so on. The `default` case is used to handle any other case not specifically handled in the switch statement. For example, if `num` equals 5, there is no `case` statement to handle the case where `num` is 5, so therefore the `default` case handles it.

An important fact about the switch statement is that a `break` statement is necessary following your case handling code. When the execution flows into a case handler and there is not an ending `break` statement, the program flow automatically falls to the next case handler, and then the next and so on until the end of the default case handler, or until a `break` is encountered. The `break` statement essentially exits out of the switch statement, which is typically desired after a particular case was handled. To illustrate, Program 2.8 shows what happens if you do not include a `break` statement.

Program 2.8: Program demonstrates a switch statement with no "breaks."

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

int main()
{
      int num = 0;
      cout << "Enter an even integer in the range [2, 8]: ";
      cin >> num;

      switch( num )
```

```
        {
        case 2:
                cout << "Case 2 executed!" << endl;
        case 4:
                cout << "Case 4 executed!" << endl;
        case 6:
                cout << "Case 6 executed!" << endl;
        case 8:
                cout << "Case 8 executed!" << endl;
        default:
                cout << "Default case executed implies you do not ";
                cout << "enter a 2, 4, 6, or 8." << endl;
        }
}
```

Output 2.8.

```
Enter an even integer in the range [2, 8]: 4
Case 4 executed!
Case 6 executed!
Case 8 executed!
Default case executed implies you do not enter a 2, 4, 6, or 8.
Press any key to continue
```

On the other hand, this execution fall-through may be used for your own purposes. For example, you might have a situation where you want to execute the same code for several cases. This can be implemented like so:

```
        case 0: // Fall through to case 1
        case 1: // Fall through to case 2
        case 2:
                ... // Execute same code for 0, 1, and 2.
                break;
        case 3: // Fall through to case 4
        case 4: // Fall through to case 5
        case 5:
                ... // Execute same code for 3, 4, and 5.
                break;
```

2.3.5 The Ternary Operator

The ternary operator is a compact notation to represent a basic if…else statement. It is the only operator in C++ that takes three operands. The general syntax is this:

Ternary Operator: `(boolExpression ? value1 : value2)`

The ternary operator may be read as follows. If `boolExpression` is true then the ternary operation evaluates to `value1`, else it evaluates to `value2`. Consider this specific example, where `B` is of type `bool`:

```
int x = B ? 10 : -5;
```

If `B` is true then the expression evaluates to 10, which is then assigned to `x`. However, if `B` is not true then the expression evaluates to –5, which is then assigned to `x`. Notice that this is equivalent to:

```
int x;
if( B )
	x = 10;
else
	x = -5;
```

Finally, it is worth mentioning that many programmers dislike the ternary operator because of its cryptic syntax.

2.4 Repetition

The ability to repeat C++ statements is an important one. For instance, to make nontrivial programs we will need to be able to say things like "For each game character, test whether or not any were hit by an enemy projectile" or "While the player is not dead, continue game play." C++ facilitates the need for repetition via **loops.** C++ provides three different loop styles; these variations are for convenience only—you could use only one of these styles and forever ignore the other two. However, by providing three different styles, you can pick the style that is most natural to the type of repetition needed.

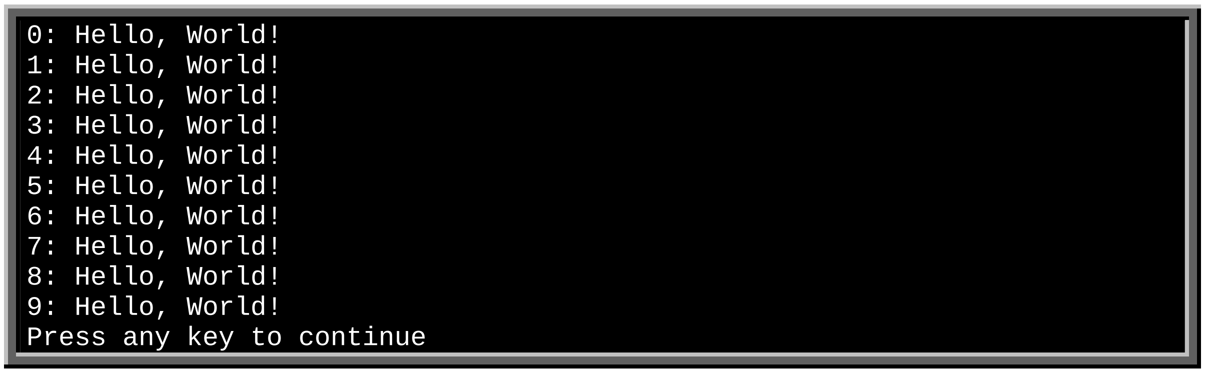
2.4.1 The for-loop

The for-loop is commonly used when you need to repeat some statement(s) a known number of times. The following program executes an output statement ten times.

Program 2.9: Program demonstrates the for-loop.

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

int main()
{
	for(int cnt = 0; cnt < 10; ++cnt)
	{
		cout << cnt << ": Hello, World!" << endl;
	}
}
```

Output 2.9.



```
0: Hello, World!
1: Hello, World!
2: Hello, World!
3: Hello, World!
4: Hello, World!
5: Hello, World!
6: Hello, World!
7: Hello, World!
8: Hello, World!
9: Hello, World!
Press any key to continue
```

The syntax of the for-loop is simple. There are essentially four parts to a "for loop."

```
for(Part 1; Part 2; Part 3)
{
     Part 4;
}
```

- Part 1: This can be any C++ statement(s). However, it is usually used to initialize a **counting variable;** that is, a variable that counts the loop cycles. The code of Part 1 is executed first and only executed once. Program 2.9 declares and initializes a counting variable called `cnt` to zero; that is, "`int cnt = 0`."

- Part 2: This is the conditional part; that is, the loop continues to loop only so long as this condition is true. This condition is tested in every loop cycle. Program 2.9 makes the condition that the program should continue to loop as long as the counting variable is less than ten; that is, "`cnt < 10`."

- Part 3: This can be any C++ statement(s). However, it is usually used to modify the counting variable in some way. The statement(s) of Part 3 are executed for every loop cycle. In Program 2.9, we increment the counter variable so that `cnt` is increased by one for every loop cycle. Because `cnt` is initialized to zero and it is incremented by one for every loop cycle, it follows that Program 2.9's for-loop will repeat exactly ten times.

- Part 4: This part contains the statement(s) which you want to execute for every cycle of the loop. Just as in an "if" statement, the curly braces are optional for one statement. However, if you need to execute several statements per cycle then you need the curly braces to form a compound statement.

To show the flexibility of the for-loop, the following code is functionally equivalent to the for-loop of Program 2.9:

```
int cnt = 0;
```

```
for( ; cnt < 10; )
{
	cout << cnt << ": Hello, World!" << endl;
	++cnt;
}
```

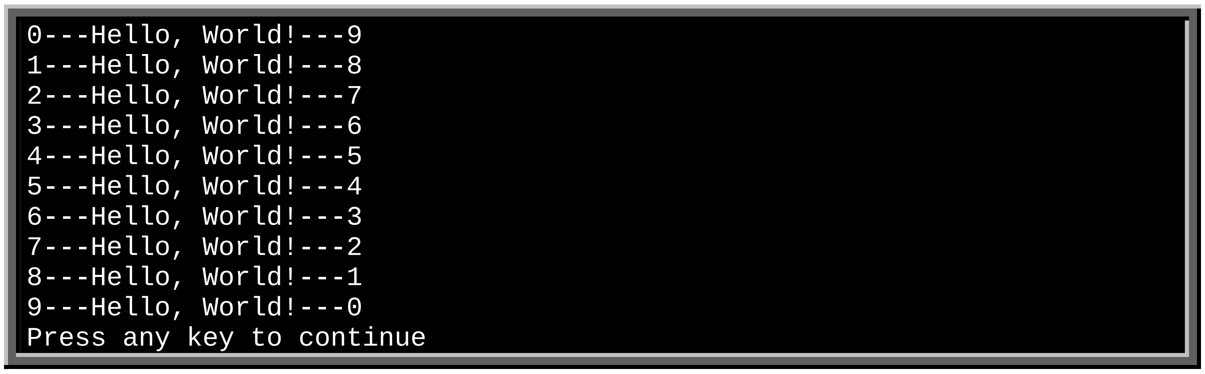
What we have done here is moved the counter initialization outside the loop and replaced Part 1 with an empty statement, which is perfectly legal since Part 1 can be “any C++ statement(s)”. Second, we have moved the counter increment from Part 3 to Part 4, and we replaced Part 3 with an empty statement. Again, this is perfectly legal since Part 3 can be “any C++ statement(s)”. Convince yourself that this alternate for-loop is functionally equivalent to the for-loop of Program 2.9.

Finally, Part 1 and Part 3 of the for-loop can contain multiple statements. For example:

Program 2.10: Program demonstrates the for-loop.

```
#include <iostream>
using namespace std;
int main()
{
	for(int cnt1 = 0, int cnt2 = 9; cnt1 < 10; ++cnt1, --cnt2)
	{
		cout << cnt1 << "---Hello, World!---" << cnt2 << endl;
	}
}
```

Output 2.10.



```
0---Hello, World!---9
1---Hello, World!---8
2---Hello, World!---7
3---Hello, World!---6
4---Hello, World!---5
5---Hello, World!---4
6---Hello, World!---3
7---Hello, World!---2
8---Hello, World!---1
9---Hello, World!---0
Press any key to continue
```

This time there are two counter variables (separated by commas), which are initialized to 0 and 9. Moreover, one is incremented and the other is decremented. Consequently, as shown from the output, one counts forward and one counts backwards. Part 2—the condition—remains the same; that is, it still specifies that we loop ten times.

2.4.2 The `while` Loop

The while-loop is commonly used when you need to repeat some statements an unknown number of times. For example, in a poker game, after every hand, the program might ask the user if he wants to play again. Based on this user input, the program will decide whether to "repeat" and play again, or to exit the loop.

The following program illustrates a while-loop that terminates based on user input.

Program 2.11: Program demonstrates the while-loop.

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

int main()
{
      // Boolean value, true if we want to quit, false otherwise.
      bool quit = false;

      // Keep looping so long as quit is not true.
      while( !quit )
      {
            // Ask the user if they want to quit or not.
            char inputChar = 'n';
            cout << "Continue to play? (y)es/(n)o...";
            cin >> inputChar;

            // Test for both uppercase or lowercase.
            if( inputChar == 'n' || inputChar == 'N')
            {
                  cout << "Exiting..." << endl;
                  quit = true;
            }
            else
                  cout << "Playing game..." << endl;
      }
}
```

Output 2.11.

```
Continue to play? (y)es/(n)o...y
Playing game...
Continue to play? (y)es/(n)o...Y
Playing game...
Continue to play? (y)es/(n)o...y
Playing game...
Continue to play? (y)es/(n)o...n
Exiting...
Press any key to continue
```

Program 2.11 is a useful example because, in addition to the while-loop, it demonstrates many of the other topics of this chapter; namely, relational operators (e.g., `inputChar == 'n'`), logical operators (e.g., `!quit`) and conditional statements (`if…else`).

As Program 2.11 implies, the while-loop takes on the following general syntax:

```
while( condition is true )
        Execute this C++ statement(s);
```

The condition used in Program 2.11 is the boolean expression `!quit` (not quit), which instructs the program to keep looping so long as `!quit` is true. If the condition is true, we execute the statements in the loop body. Inside the loop body, the program asks the user if he wishes to continue. If the player chooses "no" then the program assigns true to quit, thereby making `!quit` false. This will cause the while-loop to terminate on the next cycle when the condition `!quit` is tested again. Observe that this loop will repeat an unknown amount of times and its termination depends on user input.

2.4.3 The do…while Loop

The do…while loop is similar to the while-loop. However, a do…while is guaranteed to execute at least once. Essentially it says: "Do these statements at least once regardless of the condition, then while the condition holds, continue to do these statements." Program 2.12 shows the do…while syntax.

Program 2.12: Program demonstrates the do…while loop.

```cpp
#include <iostream>
using namespace std;
int main()
{
	bool condition = false;

	do
	{
		cout << "Enter a 0 to quit or 1 to continue: ";
		cin >> condition;
	}
	while( condition );
}
```

Output 2.12.

```
Enter a 0 to quit or 1 to continue: 1
Enter a 0 to quit or 1 to continue: 1
Enter a 0 to quit or 1 to continue: 0
Press any key to continue
```

As you can see from Program 2.12, despite `condition` being initialized to false, we still enter the loop body. Inside the body, the program assigns the truth-value the user entered to `condition`. Then at the end, the condition is tested to see if we will loop again. By moving the loop condition to the end, we are guaranteed the loop body statements will be executed at least once.

Note that it does not take too much imagination to see how a do…while loop could be rewritten using a while-loop. Consequently, in practice, the do…while loop is not encountered very often.

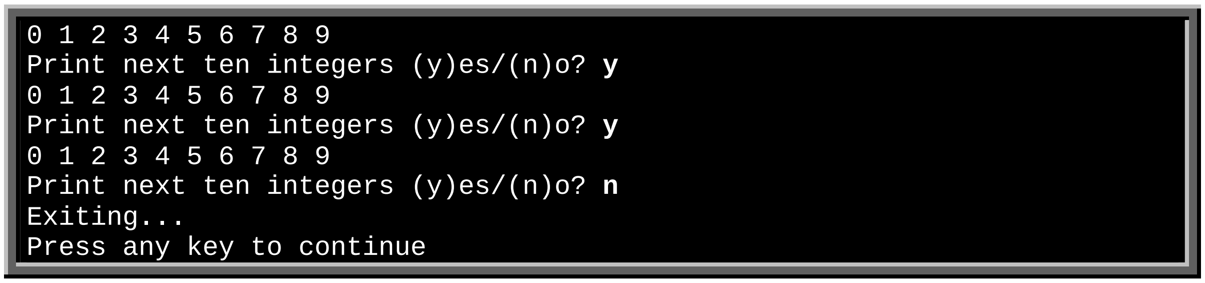
2.4.4 Nesting Loops

Just as if…else statements can be nested, loops can be nested; that is, a loop inside a loop. Consider the following program, which nests a for-loop inside a while-loop:

Program 2.13: Program demonstrates nested loops; that is, loops inside loops.

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

int main()
{
      bool quit = false;

      while( !quit )
      {
            for(int cnt = 0; cnt < 10; ++cnt)
                  cout << cnt << " ";

            cout << endl;

            char inputChar = 'n';
            cout << "Print next ten integers (y)es/(n)o? ";
            cin >> inputChar;

            if(inputChar == 'n' || inputChar == 'N')
            {
                  cout << "Exiting..." << endl;
                  quit = true;
            }
      }
}
```

Output 2.13.



```
0 1 2 3 4 5 6 7 8 9
Print next ten integers (y)es/(n)o? y
0 1 2 3 4 5 6 7 8 9
Print next ten integers (y)es/(n)o? y
0 1 2 3 4 5 6 7 8 9
Print next ten integers (y)es/(n)o? n
Exiting...
Press any key to continue
```

Here the outermost while-loop executes the innermost for-loop. The inner for-loop outputs the integers 0-9, and the outer while-loop continues to loop as long as the user specifies to continue. Thus, rows of integers 0-9 will continue to be output as long as the user answers “yes” to the question.

2.4.5 Break and Continue Keywords

Sometimes special cases inside a loop need to be handled which terminate the loop early. For example, you may iterate (loop) over a set of game items until you find a specific one. However, once you find it, you can stop your search and exit the loop. To facilitate this case, C++ provides the `break` keyword, which breaks out of the current loop. To illustrate, Program 2.14 rewrites Program 2.11, this time using an **infinite loop;** that is, a loop that is always true (and as such, will loop infinite times). This time, if the player chooses not to continue, a break statement is executed to exit the infinite loop.

Program 2.14: Program demonstrates the ‘break’ keyword. (Note that Program 2.14 has the same functionality as Program 2.11 and therefore will have same output for the same input.)

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

int main()
{
      // Loop forever.
      while( true )
      {
            // Ask the user if they want to quit or not.
            char inputChar = 'n';
            cout << "Continue to play? (y)es/(n)o...";
            cin >> inputChar;

            // Test for both uppercase or lowercase.
            if( inputChar == 'n' || inputChar == 'N')
            {
                  cout << "Exiting..." << endl;
                  break; //<--This will exit out of the infinite loop.
            }
            else
                  cout << "Playing game..." << endl;
      }
}
```

In addition to breaking out of a loop early, there may be special cases that allow us to decide early on whether we can jump straight to the next loop cycle. The following program sums the numbers from zero to fifteen. However, it ignores the numbers three, seven, and thirteen. It does this by testing for these numbers, and if it finds one of them, then the program simply continues (jumps) to the next loop cycle, thereby ignoring them (i.e., not including them in the sum).

Program 2.15: Program demonstrates the ‘continue’ keyword. (Note that Program 2.15 does not output anything.)

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

```
int main()
{
	int sum = 0;
	for(int cnt = 0; cnt <= 15; ++cnt)
	{
		if( cnt == 3 || cnt == 7 || cnt == 13)
			continue; // <--jump straight to next loop cycle.

		// Otherwise, add the number to the sum.
		sum += cnt;
	}
}
```

> **Note:** It does not take too much effort to realize that the functionality of the `break` and `continue` statements can be implemented with if…else statements. However, in some cases, the `break` and `continue` keywords provide a much cleaner, shorter, and intuitive way of expressing the idea than a more complex if…else statement would.

2.5 Arrays

Thus far, when variables were needed, we simply declared them like so:

```
std::string str = “”;
float f = 0.0f;
int n   = 0;
```

However, what if you need 100 variables or an even greater amount? It is not practical to actually declare so many variables and give them all unique names. We can, however, declare a set of variables of the same type in a compact way using an **array.** An array is a contiguous block of memory that contains *n* number of variables. We call the individual variables in an array the **elements** of the array, and we call the number of elements an array holds the **array size.** To declare an array we use the following general syntax:

```
type identifier[n];
```

Where *n* is an integer constant that specifies the number of elements the array contains. Here are some specific examples of array declarations:

```
float fArray[16];       // Array with 16 float elements.
int n[200];             // Array with 200 integer elements.
std::string str[5000]; // Array with 5000 string elements.
```

A specific element in an array can be accessed using the **bracket operator** and supplying an array **index,** which identifies the element:

```
fArray[0]  = 1.0f; // Assign 1.0f to element zero.
```

```
fArray[1]  = 2.0f; // Assign 2.0f to element one.
fArray[12] = 13.0f;// Assign 13.0f to element twelve.
//fArray[20] = 21.0f;// !! BAD, OUT OF BOUNDS INDEX !!
```

Important:

Observe that arrays are zero-based; that is, the first element in the array is identified with an index of zero. And therefore, the last element is identified with an index of *n-1*, where *n* is the total number of elements.

Supplying an out of bounds index results in code that compiles—the compiler will not force you to correct it. However, it is strongly advised not to do so as you are essentially accessing memory that does not belong to you. In the former example, the array `fArray` was declared to store 16 elements. Thus, an index of 20 is an out of bounds index.

2.5.1 Array Initialization

To initialize the elements of an array, each element could be accessed one by one and assigned an initial value like so:

```
int intArray[8];
intArray[0] = -4;
intArray[1] = 6;
intArray[2] = -2;
intArray[3] = 0;
intArray[4] = 33;
intArray[5] = 78;
intArray[6] = 0;
intArray[7] = 4;
```

In addition, C++ provides an alternative syntax:

```
int intArray[8] = {-4, 6, -2, 0, 33, 78, 0, 4};
```

Here the first entry in the curly brace list corresponds to element [0], the second entry to [1], and so on. By using this curly brace notation, where each element is explicitly initialized, the compiler can deduce how many elements the array needs, and therefore, the array size, 8, is not explicitly required. That is, this next statement is equivalent:

```
int intArray[] = {-4, 6, -2, 0, 33, 78, 0, 4};
```

In actuality, all the preceding array initialization syntaxes are only practical for small arrays. Manually initializing an array with a thousand elements, for example, is clearly impractical.

2.5.2 Iterating Over an Array

Typically, when a large array of items are stored, the elements are somewhat related to a larger whole. For example, in 3D computer graphics we usually represent a 3D object with an array of polygons. These polygons form a larger whole, namely the 3D object. If all of the polygon parts of the 3D object move in the same direction then the 3D object itself moves in that direction. As such, we normally do not modify the elements of arrays individually. Rather, we usually want to apply the same operation to every element (e.g., initialize all elements to zero, add all corresponding elements of two arrays into a third array). A loop is perfect for doing this. Program 2.16 iterates through every element in two arrays of the same size and adds their corresponding elements together and stores the sum in a third array, which is also of the same size.

Program 2.16: Program demonstrates iterating over arrays.

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

int main()
{
      int array0[7] = {1, 2, 3, 4, 5, 6, 7};
      int array1[7] = {-9, -8, -7, -6, -5, -4, -3};

      int sum[7];//<--Stores the addition result.
      for(int i = 0; i < 7; ++i)
      {
            // Add the corresponding i-th elements and store in sum.
            sum[i] = array0[i] + array1[i];

            cout << array0[i] << " + " << array1[i] << " = ";
            cout << sum[i] << endl;
      }
}
```

Output 2.16.

```
1 + -9 = -8
2 + -8 = -6
3 + -7 = -4
4 + -6 = -2
5 + -5 = 0
6 + -4 = 2
7 + -3 = 4
Press any key to continue
```

Loops and arrays go hand-in-hand and many problems are solved using by them together. Note how the counter variable of the for-loop naturally becomes an index into the i-th element of the array.

2.5.3 Multidimensional Arrays

It is possible to have an array of arrays; that is, an array where each individual element is also an array. One can be declared using the following double bracket syntax:

```
type identifier[m][n];
```

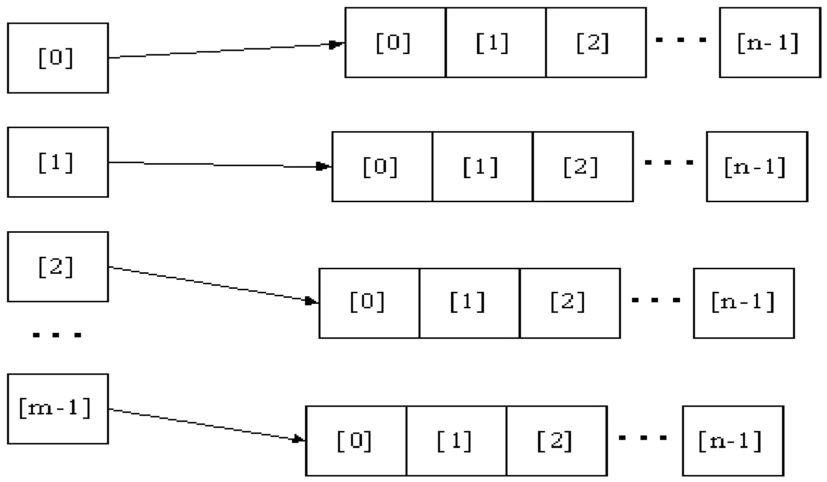


Figure 2.1: An array or arrays, where each element contains an array. If we say the array of arrays goes from top to bottom and each element of this array of arrays contains an array going left to right then we have a table layout.

As Figure 2.1 shows, an array of arrays forms a rectangular table or matrix. The constant *m* specifies the number of rows and the constant *n* specifies the number of columns. Such a matrix is of size (or dimension) $m \times n$. Here are some specific examples of matrix declarations:

```
float fMatrix[16][12];     // Matrix with 16x12 elements.
int n[20][20];             // Matrix with 20x20 elements.
std::string str[500][100]; // Matrix with 500x100 elements.
```

We can access a specific element in a matrix using the double bracket syntax and supplying two array indices, which identifies the row and column of the element:

```
 fMatrix [0][0]   = 1.0f; // Assign 1.0f.
 fMatrix [1][2]   = 2.0f; // Assign 2.0f.
 fMatrix [12][10] = 13.0f;// Assign 13.0f.
//fMatrix[16][3] = 21.0f;// !! BAD, OUT OF BOUNDS INDEX !!
```

In addition to performing individual element initializations, you can initialize matrices using curly brace syntax as shown here:

```
// Matrix with four rows of three columns.
int matrix[4][3] =
{
      {1, 2, 3},   // Row 1
      {4, 5, 6},   // Row 2
      {7, 8, 9},   // Row 3
      {10, 11, 12}// Row 4
};
```

Because of the extra dimension, iteration over a matrix is done with a double nested loop—one that iterates over the rows and one that iterates over the columns. The following double for-loop iterates over the preceding matrix and outputs its elements to the console window:

```
// Loop over rows.
for(int i = 0; i < 4; ++i)
{
      // Loop over columns.
      for(int j = 0; j < 3; ++j)
      {
            cout << matrix[i][j] << " ";
      }
      cout << endl; // New line for each row.
}
```

You can create arrays of arrays of arrays and so on by following the same general pattern. That is, a 3D array would be declared as:

```
type identifier[m][n][p];
```

But, as a rule of thumb, it is advised that you not use a multidimensional array greater than a 3D array to avoid overly complex situations.

2.6 Summary

1. The relational operators allow us to determine certain elementary relationships between variables, such as equality and inequality.

2. We use three logical operators:

   a. The logical AND operator (&&)
   b. The logical OR operator (||)
   c. The logical NOT operator (!).

These logical operators allow us to form more complex boolean expressions and therefore enable us to make more useful conditional statements.

3. Conditionals are the key to any program that needs to make decisions at runtime. They allow us to make certain that a block of code will only be executed if a particular condition is satisfied (i.e., if this condition is true then execute this code).

4. Loops enable us to repeat a block of code a variable number of times.

5. An array is a contiguous block of memory that contains *n* amount of variables. We call the individual variables in an array the **elements** of the array, and we call the number of elements an array holds, the **array size.**

2.7 Exercises

2.7.1 Logical Operator Evaluation

Assume A is true, B is true, C is false and D is false. What Boolean value do the following expressions evaluate to?

a. A || B || C || D

b. A && B && C && D

c. !C && !D

d. !((A && B) || (B && !C)) || !(C && D)

e. !(!((A && !D) && (B || C)))

2.7.2 Navigator

Write a program that displays a menu which allows the user to move one unit north, east, south or west. Each time the player enters a selection, update the coordinates of the user and output the current position. Start the player at the origin of the coordinate system. Your program's output should look like the following:

```
Current Position = (0, 0)
Move (N)orth, (E)ast, (S)outh, (W)est (Q)uit? n
Current Position = (0, 1)
Move (N)orth, (E)ast, (S)outh, (W)est (Q)uit? e
```

```
Current Position = (1, 1)
Move (N)orth, (E)ast, (S)outh, (W)est (Q)uit? s
Current Position = (1, 0)
Move (N)orth, (E)ast, (S)outh, (W)est (Q)uit? w
Current Position = (0, 0)
Move (N)orth, (E)ast, (S)outh, (W)est (Q)uit? q
Exiting...
Press any key to continue
```

2.7.3 Average

Write an averaging program. The program must prompt the user to enter a positive integer that specifies the number of values *n* the user wishes to average. The program must then ask the user to input these *n* values. The program should then compute the average of the values inputted and output it to the user—use the following average formula:

$$avg = \frac{a_0 + a_1 + ... + a_{n-1}}{n}.$$

Your program's output should look like the following:

```
Enter the number of values to average: 5
[0] = 1
[1] = 2
[2] = 3
[3] = 4
[4] = 5
Average = 3
Press any key to continue
```

2.7.4 Factorial

The factorial of a positive integer *n*, denoted *n*!, is defined as follows:

$$n! = n(n-1)(n-2)...(3)(2)(1).$$

By agreement, 0! = 1. To ensure that you are comfortable with the factorial operation, let us write out a couple of examples.

- Evaluate 4!.

$4! = 4(3)(2)(1) = 24.$

- Evaluate 6!.
  $6! = 6(5)(4)(3)(2)(1) = 720.$

Write a program that asks the user to enter in a positive integer *n*. Compute the factorial of *n* and output the result to the user. Your program's output should look like the following:

```
Enter a positive integer to compute the factorial of: 5
5! = 120
Press any key to continue
```

2.7.5 Matrix Addition

The sum of two matrices is found by adding the corresponding elements of the two matrices. Consider the following two matrices **A** and **B.**

$$\mathbf{A} = \begin{bmatrix} -5 & 2 & 8 \\ 1 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 3 & -6 \end{bmatrix}$$

The sum, found by adding corresponding elements, is:

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} -5 & 2 & 8 \\ 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 2 \\ 0 & 3 & -6 \end{bmatrix} = \begin{bmatrix} -5+1 & 2+0 & 8+2 \\ 1+0 & 0+3 & 0-6 \end{bmatrix} = \begin{bmatrix} -4 & 2 & 10 \\ 1 & 3 & -6 \end{bmatrix}$$

Write a program that creates three matrix variables of dimension 2×3, two of which are to be initialized in such a way that they are identical to **A** and **B**; the third will be used to store the sum of **A** and **B.** Using a double for-loop to iterate over the matrix elements, compute the sum of each component and store the result in the third matrix (e.g., C[i][j] = A[i][j] + B[i][j]). Finally, output the matrix sum. Your program's output should look like the following:

```
A =
-5 2 8
1 0 0

B =
1 0 2
0 3 -6

A + B =
-4 2 10
1 3 -6

Press any key to continue
```

2.7.6 ASCII

Write a program that outputs every character in the extended ASCII character set in the range [33, 255]. Note that we omit characters from [0, 32] since they are special command characters. (Hint: Recall that characters are represented by the `char` and `unsigned char` types, so simply loop through each possible value—33-255—and output it.) Your program's output should look like the following:

```
33: ! 34: " 35: # 36: $ 37: % 38: & 39: ' 40: ( 41: ) 42: *
43: + 44: , 45: - 46: . 47: / 48: 0 49: 1 50: 2 51: 3 52: 4
53: 5 54: 6 55: 7 56: 8 57: 9 58: : 59: ; 60: < 61: = 62: >
63: ? 64: @ 65: A 66: B 67: C 68: D 69: E 70: F 71: G 72: H
73: I 74: J 75: K 76: L 77: M 78: N 79: O 80: P 81: Q 82: R
83: S 84: T 85: U 86: V 87: W 88: X 89: Y 90: Z 91: [ 92: \
93: ] 94: ^ 95: _ 96: ` 97: a 98: b 99: c 100: d 101: e 102: f
103: g 104: h 105: i 106: j 107: k 108: l 109: m 110: n 111: o 112: p
113: q 114: r 115: s 116: t 117: u 118: v 119: w 120: x 121: y 122: z
123: { 124: | 125: } 126: ~ 127: ⌂ 128: Ç 129: ü 130: é 131: â 132: ä
133: à 134: å 135: ç 136: ê 137: ë 138: è 139: ï 140: î 141: ì 142: Ä
143: Å 144: É 145: æ 146: Æ 147: ô 148: ö 149: ò 150: û 151: ù 152: ÿ
153: Ö 154: Ü 155: ¢ 156: £ 157: ¥ 158: ₧ 159: ƒ 160: á 161: í 162: ó
163: ú 164: ñ 165: Ñ 166: ª 167: º 168: ¿ 169: ⌐ 170: ¬ 171: ½ 172: ¼
173: ¡ 174: « 175: » 176: ░ 177: ▒ 178: ▓ 179: │ 180: ┤ 181: ╡ 182: ╢
183: ╖ 184: ╕ 185: ╣ 186: ║ 187: ╗ 188: ╝ 189: ╜ 190: ╛ 191: ┐ 192: └
193: ┴ 194: ┬ 195: ├ 196: ─ 197: ┼ 198: ╞ 199: ╟ 200: ╚ 201: ╔ 202: ╩
203: ╦ 204: ╠ 205: ═ 206: ╬ 207: ╧ 208: ╨ 209: ╤ 210: ╥ 211: ╙ 212: ╘
213: ╒ 214: ╓ 215: ╫ 216: ╪ 217: ┘ 218: ┌ 219: █ 220: ▄ 221: ▌ 222: ▐
223: ▀ 224: α 225: ß 226: Γ 227: π 228: Σ 229: σ 230: µ 231: τ 232: Φ
233: Θ 234: Ω 235: δ 236: ∞ 237: φ 238: ε 239: ∩ 240: ≡ 241: ± 242: ≥
243: ≤ 244: ⌠ 245: ⌡ 246: ÷ 247: ≈ 248: ° 249: ∙ 250: · 251: √ 252: ⁿ
253: ² 254: ■ 255:
Press any key to continue
```

2.7.7 Linear Search

Background Information

In writing computer programs you will often need to search a set of data for values that have some particular properties. For example, imagine you have some fantasy role-playing game and you want to organize a global dataset of "players" into subsets based on the character classes; that is, a subset of warriors, a subset of wizards, a subset of clerics, and etc. In order to do this, you will need to *search* the global player list for the members of each class and copy those members into their corresponding class subset.

As another example, suppose a business issues identification numbers to its customers. Given a particular identification number, the business would like to *search* its customer database for the customer information (e.g., name, address, order history, etc.) that corresponds with the given identification number. Note that the value used for the search, which in this example is an identification number, is called a **search key.** If the business searched for a customer's information using the customer's last name, then the last name would be the search key.

One method of searching a set of data is called a **linear search.** The linear search simply scans the dataset element-by-element until it finds the elements it is looking for. Clearly, this search can be fast if the particular element is found early on (not guaranteed), or can be slow if the particular element is one of the last few elements to be scanned.

Example: Find the array position of the value '9' in the following dataset: {7, 3, 32, 2, 55, 34, 6, 13, 29, 22, 11, 9, 1, 5, 42, 39, 8}.

Solution: We use the linear search method; that is, reading the array left to right and element-by-element we find that the value '9' is located in the eleventh position (e.g., index [11]) of the array (recall that array positions start at 0 and not 1).

In general, we have the following linear search algorithm:

Linear Search.

```
Let x[n] = x[0],…,x[n-1] be an array of given integers to search.
Let Position be an integer to store the array index of the item we are searching for.
Let Value be the value we are searching for.

For $i = 0$ to $n - 1$, do the following:
    If( x[i] = Value )
        Position = i;
        Break;
    Else
        Continue;
```

Exercise

Hardcode the following integer array: {7, 3, 32, 2, 55, 34, 6, 13, 29, 22, 11, 9, 1, 5, 42, 39, 8} into your program. Display this array to the user. Then ask the user to input an integer to search for. Your program should then search the array for the integer the user entered and output its array position (i.e., its array index). Your output should look similar to the following:

```
List = 7, 3, 32, 2, 55, 34, 6, 13, 29, 22, 11, 9, 1, 5, 42, 39, 8
Enter an integer in the list to search for: 55
Item found at index [4]
Press any key to continue
```

2.7.8 Selection Sort

Background Information

A frequent task that occurs in writing computer programs is the need to sort a set of data. For example, you might want a list of integers sorted in ascending or descending order; you may want to sort records of customers by age, zip code, and/or gender; or in 3D computer graphics you might want to sort 3D geometry (e.g., polygons) based on their distance from the viewer. Because sorting is so important, we will end up spending several lab projects working on the different sorting techniques that have been devised. However, to begin with, we will start with one of the simplest (and also one of the least efficient) sorting methods, called the **selection sort.**

The selection sort algorithm is as follows:

Ascending order Selection Sort.

```
Let x[n] = x[0],...,x[n-1] be an array of given integers to sort.
Let p be an integer to store an array index.

For $i = 0$ to $n - 2$, do the following:

    1. Find the array index of the smallest value in the subarray x[i],…,x[n-1] and store
       it in p.
    2. Swap x[i] and x[p].
    3. Increment i.
```

To be sure that you understand what this algorithm specifies, consider the following example:

Example:

Consider the following integer array:

x[5] = { 7, 4, 1, 8, 3 };

When $i = 0$ we scan through the subarray x[0],…,x[n-1] and search for the smallest number. We find that the smallest integer 1 is located at index p = 2. The algorithm then tells us to swap x[i] and x[p], which currently means to swap x[0] and x[2]. Doing so yields the new array configuration:

1, 4, 7, 8, 3

Incrementing *i*, we now have $i = 1$. Scanning through the subarray x[1],…,x[n-1] and searching for the smallest integer we find 3 at index p = 4. Swapping x[1] and x[4] yields:

1, 3, 7, 8, 4

Incrementing *i*, we now have $i = 2$. Scanning through the subarray x[2],…,x[n-1] and searching for the smallest integer we find 4 at index p = 4. Swapping x[2] and x[4] yields:

1, 3, 4, 8, 7

Incrementing *i*, we now have $i = 3$. Scanning through the subarray x[3],…,x[n-1] and searching for the smallest integer we find 7 at index p = 4. Swapping x[3] and x[4] yields:

1, 3, 4, 7, 8

We have successfully sorted the array in ascending order. Observe that we make $n - 2$ passes, since the last pass correctly sorts both the x[n-2] and x[n-1] elements.

As you can see, this algorithm moves one element into its correct sorted position per pass. That is, for each cycle it finds the smallest element in the subarray x[i],…,x[n-1] and moves it to its correct sorted position, which is the front of the subarray: x[i]. This makes sense because, if we are sorting in ascending order then the smallest element in the subarray x[i],…,x[n-1] should be first. Because each pass correctly sorts one element, we do not need to consider that element anymore. We effectively ignore that element by shortening the subarray by one element (incrementing *i*). We then repeat the process with our new shortened subarray by searching for the next smallest element and placing it in its sorted position, and so on, until we have sorted the entire array.

Exercise

Ask the user to input ten random (non-sorted) integers and store them in an array. Sort these integers in ascending order using the **selection sort** and output the sorted array to the user. Your output should look similar to the following:

```
Enter ten unsorted integers...
[0] = 5
[1] = -3
[2] = 2
[3] = 1
[4] = 7
[5] = -9
[6] = 4
[7] = -5
[8] = 6
[9] = -12

Unsorted List = 5, -3, 2, 1, 7, -9, 4, -5, 6, -12,
Sorting...
Sorted List = -12, -9, -5, -3, 1, 2, 4, 5, 6, 7,
Press any key to continue
```

Chapter 3

Functions

Introduction

A **function** is a unit of code designed to perform a certain task. In order to perform its task, a function typically inputs some information and/or returns some information. The concept is somewhat similar to mathematical functions. Consider the trigonometric function $\sin(x)$, which takes a parameter x and evaluates (returns) some value, namely the sine of x. For example, the sine of 45° is approximately 0.707, $\sin(45°) = 0.707$; that is to say, given 45° as a *parameter*, the function $\sin(x)$ works to compute the sine of the given angle, and then *returns* or *evaluates* to the result 0.707.

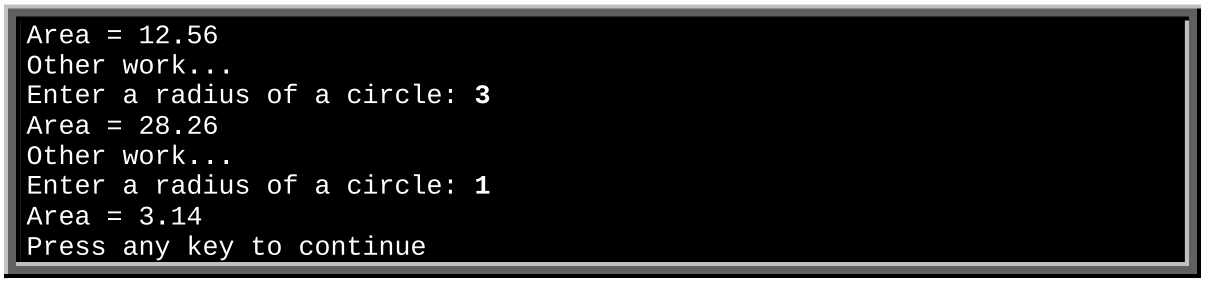
The utility of functions can be be easily demonstrated if we first consider a program without them. To illustrate, suppose we have a program that must input the radius of a circle from the user and compute the area throughout the program. (Recall the area of a circle is given by $A = \pi \cdot r^2$, where r is the radius of the given circle.) As a first attempt we might do the following:

Program 3.1: Program without Functions.

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

int main()
{
	float PI = 3.14f;

	// Input a radius and output the circle area.
	float radius = 0.0f;
	cout << "Enter a radius of a circle: ";
	cin >> radius;
	float area = PI*radius*radius;
	cout << "Area = " << area << endl;

	// Do some other work...
	cout << "Other work..." << endl;

	// Input another radius and output the circle area.
	cout << "Enter a radius of a circle: ";
	cin >> radius;
	area = PI*radius*radius;
	cout << "Area = " << area << endl;

	// Do some other work...
	cout << "Other work..." << endl;

	// Input another radius and output the circle area.
	cout << "Enter a radius of a circle: ";
	cin >> radius;
	area = PI*radius*radius;
	cout << "Area = " << area << endl;

	// and so on...
}
```

Output 3.1.

```
Enter a radius of a circle: 2
```



```
Area = 12.56
Other work...
Enter a radius of a circle: 3
Area = 28.26
Other work...
Enter a radius of a circle: 1
Area = 3.14
Press any key to continue
```

The main problem which Program 3.1 suffers is code duplication—there is duplicate code which essentially performs the same task. Besides bloating the code, programs with duplicated code are hard to maintain because if changes or corrections need to be made, it would be necessary to make the change or correction in every duplicated instance. In a large real world program, going through each source code file and making changes is not only a waste of time, but it is prone to error.

The problems of Program 3.1 could be resolved using a function whose sole task is to input the radius from the user, compute the area and return the result. For the sake of discussion, let us assume such a function exists and call it `Area`; moreover, assume the task of `Area`—that is inputting the radius from the user, computing the area and returning the result—is executed by simply writing "`Area()`" in a C++ program. (When we execute a function we say that we **call** it or **invoke** it.) Program 3.1 can now be rewritten like so:

Program 3.2: Revision of Program 3.1 using an `Area` function. Note that this program will not compile yet because the function `Area` is not actually defined. Note that we have bolded the calls to the area function.

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

int main()
{
      // Input a radius and output the circle area.
      cout << "Area = " << Area() << endl;

      // Do some other work...
      cout << "Other work..." << endl;

      // Input another radius and output the circle area.
      cout << "Area = " << Area() << endl;

      // Do some other work...
      cout << "Other work..." << endl;

      // Input another radius and output the circle area.
      cout << "Area = " << Area() << endl;

      // and so on...
}
```

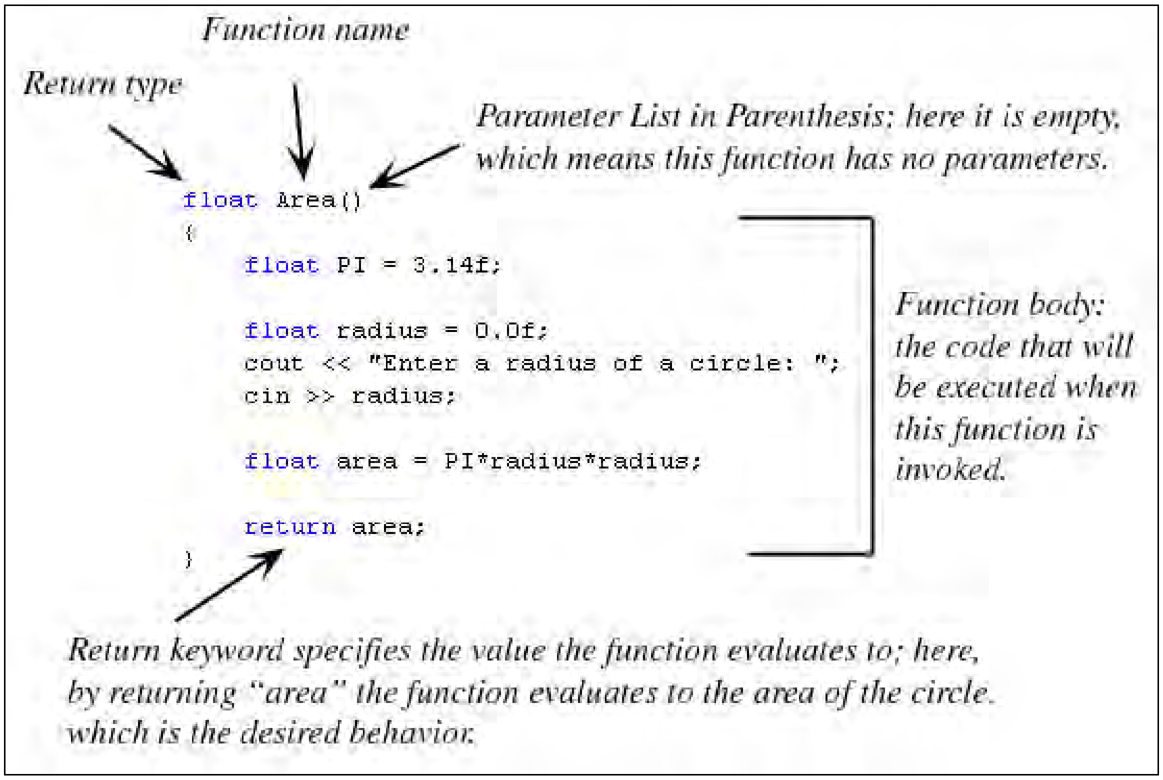
Program 3.2 is much cleaner and more compact. There is no longer duplicate code for the input code and calculation code—we simply write "`Area()`" wherever necessary to input a radius and compute a circle area. Moreover, if a change or correction needs to be made to the code `Area` executes, it would only necessitate modification of the `Area` function. With that, let us see how `Area` works.

3.1 User Defined Functions

Now that we understand the benefits of functions, let us examine the actual syntactic details of making and using them. To **define a function** (i.e., create) the following needs to be specified:

- return type (i.e., the type of value the function evaluates to)
- name (i.e., what you want to refer to it as)
- parameter list (i.e., what values does it take as input, if any)
- body (i.e., the code to be executed when the function is invoked)

Figure 3.1 shows the syntax of how the `Area` function, from the preceding discussion, would be implemented.


Function name
Return type
Parameter List in Parenthesis: here it is empty, which means this function has no parameters.
Function body: the code that will be executed when this function is invoked.
Return keyword specifies the value the function evaluates to; here, by returning "area" the function evaluates to the area of the circle, which is the desired behavior.

```
float Area()
{
    float PI = 3.14f;

    float radius = 0.0f;
    cout << "Enter a radius of a circle: ";
    cin >> radius;

    float area = PI*radius*radius;

    return area;
}
```

Figure 3.1: Function definition.

Program 3.3 rewrites Program 3.2, this time defining `Area` so that the program will compile and run.

Program 3.3: Revision of Program 3.2, this time with the `Area` function defined.

```
#include <iostream>
```

```
using namespace std;

float Area()
{
	float PI = 3.14f;

	float radius = 0.0f;
	cout << "Enter a radius of a circle: ";
	cin >> radius;

	float area = PI*radius*radius;

	return area;
}

int main()
{
	// Input a radius and output the circle area.
	cout << "Area = " << Area() << endl;

	// Do some other work...
	cout << "Other work..." << endl;

	// Input another radius and output the circle area.
	cout << "Area = " << Area() << endl;

	// Do some other work...
	cout << "Other work..." << endl;

	// Input another radius and output the circle area.
	cout << "Area = " << Area() << endl;

	// and so on...
}
```

Observe from Program 3.3 that the function `Area` is defined before any calls to that function. A function must be either declared or defined before it is called, as the compiler must recognize the function before you call it. A **function declaration** (also called a **function prototype**) consists of the return type, function name, and parameter list followed by a semicolon—there is no body in a function declaration. Once a function is declared, it can be defined elsewhere in the program. However, once declared, the function definition can come even after you call the function. Program 3.4 rewrites Program 3.3 using a function declaration.

Program 3.4: Revision of Program 3.3, this time using a function declaration.

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

// Function declaration.  The function declaration just tells the
// compiler the function exists (it will be defined later), and
// its name, return type and parameters.
float Area();

int main()
```

```
{
	// Input a radius and output the circle area.
	cout << "Area = " << Area() << endl;

	// Do some other work...
	cout << "Other work..." << endl;

	// Input another radius and output the circle area.
	cout << "Area = " << Area() << endl;

	// Do some other work...
	cout << "Other work..." << endl;

	// Input another radius and output the circle area.
	cout << "Area = " << Area() << endl;

	// and so on...
}

// Function definition.  The function definition contains the
// function body and consists of the code that specifies what
// the function actually does.
float Area()
{
	float PI = 3.14f;

	float radius = 0.0f;
	cout << "Enter a radius of a circle: ";
	cin >> radius;

	float area = PI*radius*radius;

	return area;
}
```

Note: It is illegal syntax to define a function inside another function. This includes `main` because `main` is a function, itself.

3.1.2 Functions with One Parameter

The `Area` function did not have a parameter. But let us look at an example which does have a parameter. A useful function might be one that cubes (x^3) the given input, as follows:

Program 3.5: Function with a parameter. We have bolded the function calls to `Cube`.

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

// Declare a function called 'Cube' which has a parameter
// of type 'float' called 'x', and which returns a value
// of type 'float'.
```

```
float Cube(float x);

int main()
{
	float input0 = 0.0f;
	cout << "Enter a real number: ";
	cin >> input0;

	cout << input0 << "^3 = " << Cube(input0) << endl;

	float input1 = 0.0f;
	cout << "Enter another real number: ";
	cin >> input1;

	cout << input1 << "^3 = " << Cube(input1) << endl;
}

// Provide the definition of Cube--it computes x^3.
float Cube(float x)
{
	float result = x * x * x; // x^3 = x * x * x.

	return result;
}
```

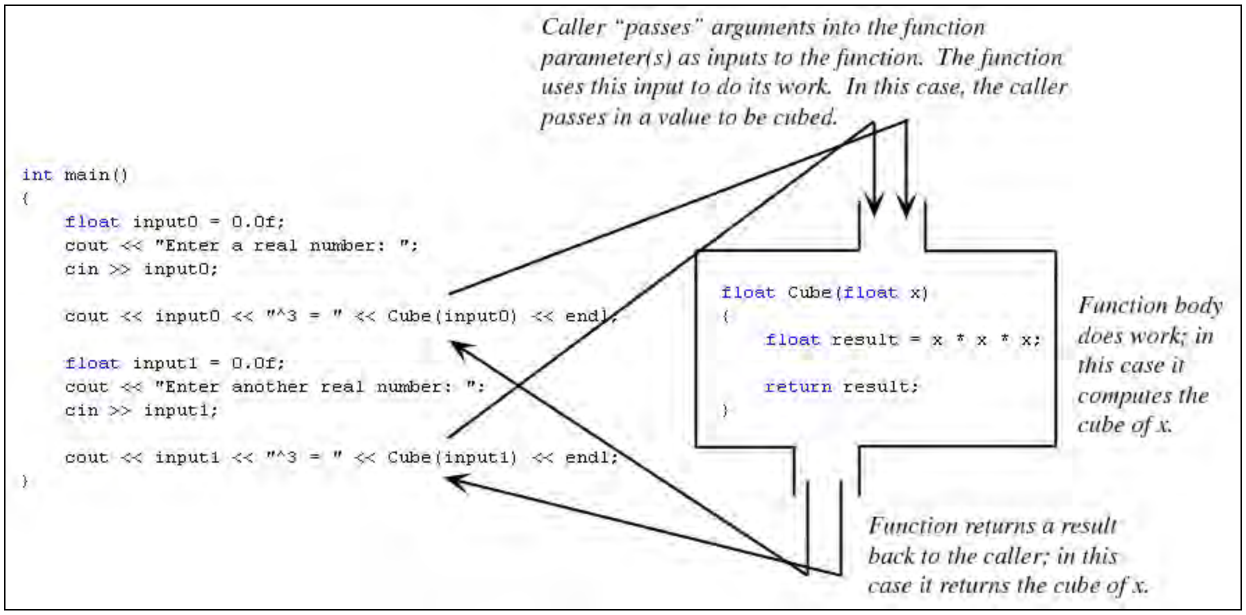
Program 3.5 Output

```
Enter a real number: 2
2^3 = 8
Enter another real number: 3
3^3 = 27
Press any key to continue
```

The `Cube` function is similar to the `Area` function except that it takes a parameter (i.e., its parameter list inside the parentheses of the function declaration/definition is not empty). A parameter is not a value in and of itself, but rather a value placeholder (variable). That is, the function caller will “pass in” or “input” a value into this placeholder variable for the function to use. The actual value passed into a particular function call is called an **argument.**

For example, in program 3.5, we write `Cube(input0)`. Here `input0` is the argument—it stores a specific value which is input into the `Cube` function; that is, the value stored in `input0` is *copied* into the `Cube` parameter `x`. The word “copied” is important, as `input0` and `x` are not the same variables but will contain copies of the same value when the function is called. This copying of argument value to parameter is called **passing by value.** Once the argument is copied to the parameter, the code inside the body of `Cube` can execute, where `x` contains the value that was passed into it.

Figure 3.2 shows how you can think of function code relative to the calling program code. Think of a function as a separate “machine” where you feed data into it (copy arguments into the parameters), it does something with those parameters (function body), and it outputs a result (returns something back to you). Again, functions are useful primarily because they prevent code duplication and provide a level of code organization; that is, breaking up programs into more manageable parts, where each part does a specific task.



```
int main()
{
    float input0 = 0.0f;
    cout << "Enter a real number: ";
    cin >> input0;

    cout << input0 << "^3 = " << Cube(input0) << endl;

    float input1 = 0.0f;
    cout << "Enter another real number: ";
    cin >> input1;

    cout << input1 << "^3 = " << Cube(input1) << endl;
}
```

```
float Cube(float x)
{
    float result = x * x * x;

    return result;
}
```

Caller "passes" arguments into the function parameter(s) as inputs to the function. The function uses this input to do its work. In this case, the caller passes in a value to be cubed.

Function body does work; in this case it computes the cube of x.

Function returns a result back to the caller; in this case it returns the cube of x.

Figure 3.2: Calling functions with parameters and returning results.

3.1.3 Functions with Several Parameters

Functions are not limited to zero or one parameter, but can have several parameters. The following program uses a function named `PrintPoint`, which takes three parameters: one each for the x-coordinate, y-coordinate and z-coordinate of a point. The function then outputs the coordinate data in a convenient point format.

Program 3.6: Functions with several parameters.

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

void PrintPoint(float x, float y, float z);

int main()
{
      PrintPoint(1.0f, 2.0f, 3.0f);

      PrintPoint(-5.0f, 3.5f, 1.2f);

      PrintPoint(-12.0f, 2.3f, -4.0f);

      PrintPoint(9.0f, 8.0f, -7.0f);
}

void PrintPoint(float x, float y, float z)
```

```
{
      cout << "<" << x << ", " << y << ", " << z << ">" << endl;
}
```

Program 3.6 Output

```
<1, 2, 3>
<-5, 3.5, 1.2>
<-12, 2.3, -4>
<9, 8, -7>
Press any key to continue
```

As Program 3.6 shows, additional parameters could be added and separated with the comma operator. When a function is called, an argument for each parameter is provided.

Another thing to notice about the `PrintPoint` function is that because its sole task is to output a point to the console window, it does not need to return a value. It is said that the function returns `void`, and as such, `void` is specified for the return type. Observe that you do not need to write a `return` statement for a function that returns `void`.

3.2 Variable Scope

As Figure 3.2 implies, by using functions our programs are broken up into code parts. Moreover, functions themselves can contain other code units such as if statements and loops. Furthermore, these code units can be nested. This brings up the topic of **variable scope.** Variable scope refers to what variables a code unit can "see" or "know about". A variable defined inside a particular code unit is said to be a **local variable** relative to that code unit. Additionally, a variable defined outside a particular code unit is a **global variable** relative to that code unit. (A subunit of code is not considered to be "outside" the unit of code that contains the subunit.) A unit of code can "see" variables that are global and local, relative to it. Let us look at a couple of examples now to see how this vocabulary is used.

3.2.1 Example 1

Program 3.7: Variable scope example 1.

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

float gPI = 3.14f;

float SphereVolume(float radius);

int main()
```

```
{
	cout << "PI = " << gPI << endl;
	cout << endl;

	float input0 = 0.0f;
	cout << "Enter a sphere radius: ";
	cin >> input0;

	float V = SphereVolume(input0);
	cout << "V = " << V << endl;
}

float SphereVolume(float radius)
{
	float V = (4.0f/3.0f)*gPI*radius*radius*radius;

	return V;
}
```

Program 3.7 Output

```
PI = 3.14

Enter a sphere radius: 3
V = 113.04
Press any key to continue
```

The first variable defined is `gPI`. Because `gPI` is outside the code units of `main` and `SphereVolume`, it is global relative to both of them, and both functions can "see", use, and modify `gPI`.

The next set of variables occurs inside the `main` function unit of code. These variables, `input0` and `V`, are local to `main` and global to no code unit. Therefore, `main` is the only code unit that can "see" them. For example, `SphereVolume` cannot "see" `input0`, and if you try to use `input0` in `SphereVolume`, you will get a compiler error. Note, however, that you can define a variable with the same variable name as a variable in another code unit, as we do with `V`. Because the variables `V` are defined in separate code units they are completely independent of each other.

Finally, as already mentioned, `SphereVolume` defines its own separate version of `V`. This `V` is local to `SphereVolume` and therefore `SphereVolume` is the only code unit that can "see" this `V`. Additionally, the parameter variable `radius` is local to `SphereVolume`. When the argument `input0` is passed into the function, the value stored in `input0` is copied to the variable `radius`. It is important to understand that this copy takes place and that `input0` and `radius` are separate variables in memory.

> **Important:** Variables declared in a code unit are destroyed when the program exits that code unit. For example, when `SphereVolume` is invoked, the program will create memory for the variable `V`. After the function ends (after V is returned) the memory for V is deleted.

3.2.2 Example 2

Program 3.8: Variable scope example 2.

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

int main()
{
      for(int i = 0; i < 5; ++i)
      {
            int cnt;
            cout << "Hello, World!" << endl;
            ++cnt;
      }

      cout << "cnt = " << cnt << endl;
}
```

This short program fails to compile. In particular, we get the error “error C2065: 'cnt' : undeclared identifier.” This error is caused because `cnt` is local to the for-loop code unit. The variable `cnt` is neither local nor global to `main`, and as such, `main` cannot “see” it. Thus, the program reports that it is “undeclared” when `main` attempts to access it.

Besides the compilation error, Program 3.8 also has a logic error. Namely, the counting variable `cnt` is not keeping track of the number of loop cycles. Recall the “important” note from the Section 3.2.1. Variables declared in a code unit are destroyed when the program exits that code unit. Every time the for-loop repeats, `cnt` is re-created and re-destroyed after that loop cycle, and therefore, the value does not persist. The program needs to be rewritten as follows:

Program 3.9: Revision of Program 3.8.

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

int main()
{
      int cnt;
      for(int i = 0; i < 5; ++i)
      {
            cout << "Hello, World!" << endl;
            ++cnt;
      }

      cout << "cnt = " << cnt << endl;
}
```

Note that `cnt` is global to the for-loop, and will not be created and destroyed every time the loop repeats. In this way, it will be able to actually count the loop cycles. Furthermore, `cnt` is now local to main, and so fixes the “'cnt' : undeclared identifier” error.

3.2.3 Example 3

Program 3.10: Variable scope example 3.

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

int main()
{
      float var = 5.0f;

      if( var > 0.0f )
      {
            float var = 2.0f;

            cout << "var = " << var << endl;
      }

      cout << "var = " << var << endl;
}
```

Program 3.10 Output

```
var = 2
var = 5
Press any key to continue
```

Recall that we can create variables of the same name if they exist in different code units. This is straightforward with separate functions but can be tricky when using if/loop statements. Program 3.10 declares a variable called var local to main and assigns 5.0 to it. The program then asks if `var` is greater than zero. It is, and the program executes the 'if' statement consequent. The program declares a new variable also called `var` (which is legal since the 'if' statement is a separate code unit) and assigns 2.0 to it. The program then proceeds to output `var` using `cout`. However, this presents a dilemma: Which `var` is used in the `cout` statement: the one local to the 'if' statement or the one local to `main`? As a rule, C++ always uses the variable "closest" to the working code unit. In this example, the program chooses to output the variable `var` that is local to the 'if' statement because it is "closer" to the 'if' statement than the variable `var` that is global to the 'if' statement (local to `main`). The program output verifies this—the `cout` statement inside the 'if' statement printed out the local version of `var`, which contained the value 2.

3.3 Math Library Functions

The C++ standard library provides functions for many of the elementary math functions and operations, such as trigonometric, logarithmic, and exponential functions, as well as square root and absolute value functions. To use the standard math functions, the `<cmath>` header file must be included. The following table summarizes some of the most commonly used math functions:

Table 3.1: Some Standard Library Math Functions.

Function Declaration	**Description**
`float cosf(float x);`	Returns $\cos(x)$.
`float sinf(float x);`	Returns $\sin(x)$.
`float tanf(float x);`	Returns $\tan(x)$.
`float acosf(float x);`	Returns $\cos^{-1}(x)$.
`float asinf(float x);`	Returns $\sin^{-1}(x)$.
`float atanf(float x);`	Returns $\tan^{-1}(x)$.
`float sqrtf(float x);`	Returns $\sqrt{x}$.
`float logf(float x);`	Returns $\ln(x)$.
`float expf(float x);`	Returns e^x.
`float powf(float x, float y);`	Returns x^y.
`float fabsf(float x);`	Returns $\lvert x\rvert$.
`float floorf(float x);`	Returns the largest integer $\leq x$.
`float ceilf(float x);`	Returns the smallest integer $\geq x$.

Remark 1: The trigonometric functions work in radians and not degrees. A number *x* can be converted from radians to degrees by multiplying it by $180°/\pi$. For example: $x = 2\pi = 2\pi \cdot 180°/\pi = 2 \cdot 180° = 360°$. Likewise, a number *x* can be converted from degrees to radians by multiplying it by $\pi/180°$. For example: $360° = 360° \cdot \pi/180° = 2\pi$.

Remark 2: The functions above work with `floats`, hence the 'f' suffixes. The standard math library also provides versions that work with `doubles`. The double versions are the same except that the 'f' suffix is omitted. For example, the double version of the cosine function would be `double cos(double x)`. In real-time 3D computer game graphics floats are typically used, which is the reason why the float versions were given in the above table.

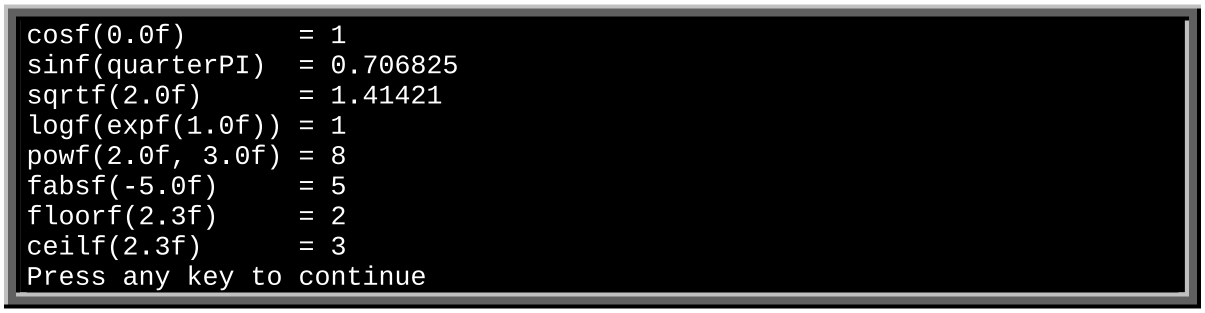
The following program shows how to call some of these "calculator" functions. The results can be verified by performing the computations on a calculator.

Program 3.11: Examples of using the standard math library functions.

```
#include <iostream>
```

```
#include <cmath>

using namespace std;

int main()
{
      float PI        = 3.14f;
      float quarterPI = PI / 4.0f;

      cout << "cosf(0.0f)       = " << cosf(0.0f)       << endl;
      cout << "sinf(quarterPI)  = " << sinf(quarterPI)  << endl;
      cout << "sqrtf(2.0f)      = " << sqrtf(2.0f)      << endl;
      cout << "logf(expf(1.0f)) = " << logf(expf(1.0f)) << endl;
      cout << "powf(2.0f, 3.0f) = " << powf(2.0f, 3.0f) << endl;
      cout << "fabsf(-5.0f)     = " << fabsf(-5.0f)     << endl;
      cout << "floorf(2.3f)     = " << floorf(2.3f)     << endl;
      cout << "ceilf(2.3f)      = " << ceilf(2.3f)      << endl;
}
```

Program 3.11 Output



```
cosf(0.0f)       = 1
sinf(quarterPI)  = 0.706825
sqrtf(2.0f)      = 1.41421
logf(expf(1.0f)) = 1
powf(2.0f, 3.0f) = 8
fabsf(-5.0f)     = 5
floorf(2.3f)     = 2
ceilf(2.3f)      = 3
Press any key to continue
```

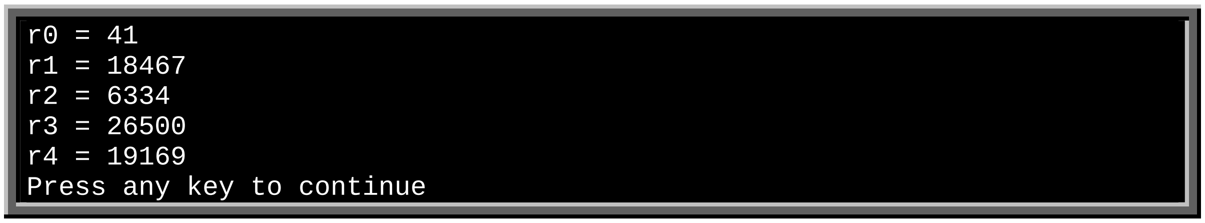
3.4 Random Number Library Functions

The C++ standard library provides a function called `rand` (include `<cstdlib>`), which can be used to generate a pseudorandom number. This function returns a random integer in the range `[0, RAND_MAX]`, where `RAND_MAX` is some predefined constant. Consider the following example:

Program 3.12: Random numbers without seeding.

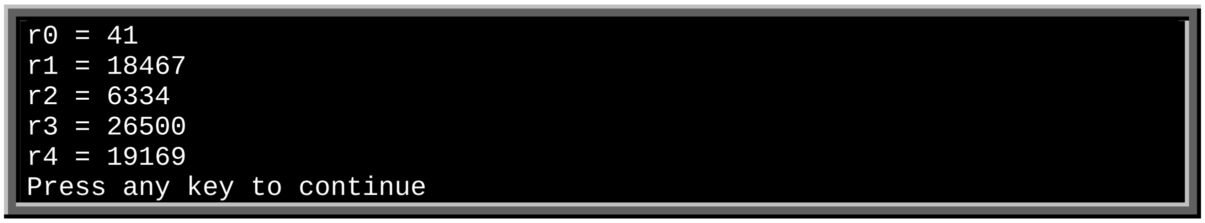
```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
      int r0 = rand();
      int r1 = rand();
      int r2 = rand();
      int r3 = rand();
      int r4 = rand();
      cout << "r0 = " << r0 << endl;
      cout << "r1 = " << r1 << endl;
      cout << "r2 = " << r2 << endl;
      cout << "r3 = " << r3 << endl;
      cout << "r4 = " << r4 << endl;
}
```

Program 3.12 Output after first run



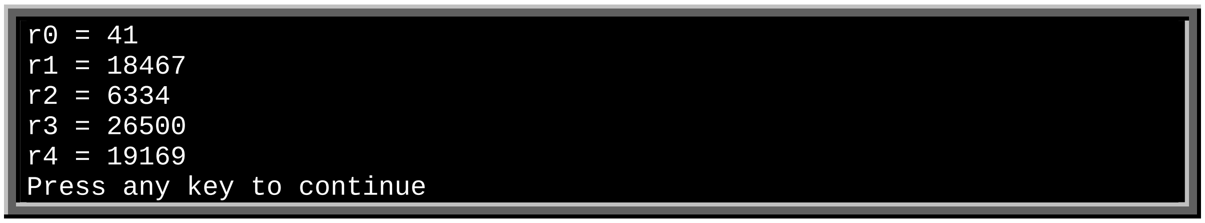
```
r0 = 41
r1 = 18467
r2 = 6334
r3 = 26500
r4 = 19169
Press any key to continue
```

Program 3.12 Output after second run



```
r0 = 41
r1 = 18467
r2 = 6334
r3 = 26500
r4 = 19169
Press any key to continue
```

Program 3.12 Output after third run



```
r0 = 41
r1 = 18467
r2 = 6334
r3 = 26500
r4 = 19169
Press any key to continue
```

This program is executed several times with the same output every time—that is not very random.

This is because pseudorandom numbers are not really random but generated using a complex mathematical algorithm. The algorithm has a starting point which it uses to start generating pseudorandom numbers. If the starting point is always the same then the sequence of generated pseudorandom numbers will also always be the same. The solution to this problem is to set the algorithm's starting point at the beginning of the application to a value that is different than the previous starting points used. An easy way to achieve this is to use the current system time as a starting point, since the system time will be different every time the program runs. Because a different starting point is used every time the application runs, the random numbers generated will be different each time the application runs.

To get the system time we use the `time` function (include `<ctime>`). The MSDN (Microsoft Developers Network) library states: "The `time` function returns the number of seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC), according to the system clock."

The pseudorandom number generator's starting point is set using the `srand` function. (This is called **seeding** the pseudorandom number generator.) The following code snippet sets the starting point to that of the current time:

```
        srand( time(0) );
```

The value returned from `time` is passed to `srand`. Note that the pseudorandom number generator is seeded once per application before making any calls to `rand`.

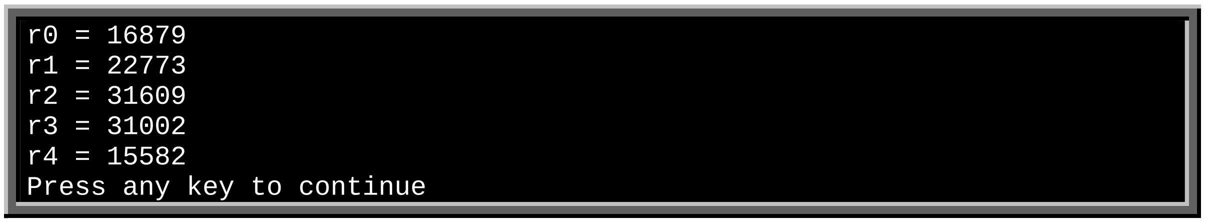
Let us now rewrite Program 3.12 with seeding:

Program 3.13: Random numbers with seeding.

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

int main()
{
        srand( time(0) );

        int r0 = rand();
        int r1 = rand();
        int r2 = rand();
        int r3 = rand();
        int r4 = rand();

        cout << "r0 = " << r0 << endl;
        cout << "r1 = " << r1 << endl;
        cout << "r2 = " << r2 << endl;
        cout << "r3 = " << r3 << endl;
        cout << "r4 = " << r4 << endl;
}
```

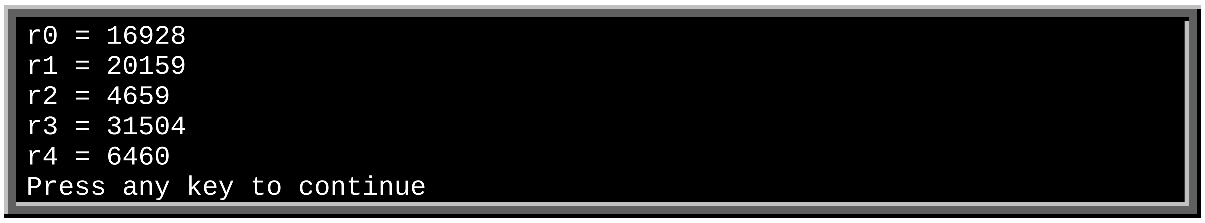
As the output shows, we get different pseudorandom numbers every time the program runs.

Program 3.13 Output after first run



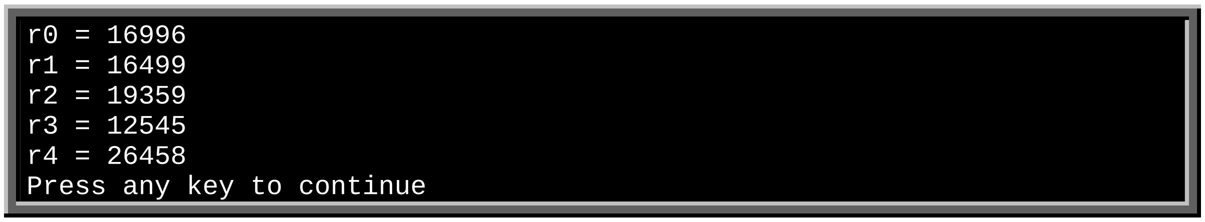
```
r0 = 16879
r1 = 22773
r2 = 31609
r3 = 31002
r4 = 15582
Press any key to continue
```

Program 3.14 Output after second run



```
r0 = 16928
r1 = 20159
r2 = 4659
r3 = 31504
r4 = 6460
Press any key to continue
```

Program 3.14 Output after third run



```
r0 = 16996
r1 = 16499
r2 = 19359
r3 = 12545
r4 = 26458
Press any key to continue
```

3.4.1 Specifying the Range

Usually, a random number in the range `[0, RAND_MAX]` is not desired; rather, we would like to specify the range. A random number in the range [0, n-1] can be computed, where n is an integer we specify, by using the modulus (%) operator. For example, to compute a random integer in the range [0, 24] we would write:

```
int num = rand();
int val = num % 25;
```

This is more compactly written as:

```
int num = rand() % 25;
```

It is not difficult to see how this works when you consider the fact that the remainder of a number divided by *n* must be in the range [0, *n-1*]. This is because if the remainder was greater than or equal to *n*, then the divisor could divide into the dividend again. Remember, a remainder is the remaining part that cannot be evenly divided by *n* (i.e., is not divisible by *n*).

We also want ranges that do not start at zero. For example, we may want a range like [2, 10]. This is easily formulated. First, we start with

```
rand() % 11;
```

This gives a random number in the range [0, 10]. To get the two, we can shift our random range over by adding two like so:

```
2 + rand() % 11;
```

But now our shifted range is [2, 12]. This is remedied by subtracting two from the right side modulus operand. That is:

```
2 + rand() % (11 – 2);
```

This yields a random number in the range [2, 10].

Using the same method of the preceding example, a random number can be generalized and generated in the range [*a*, *b*] using the formula:

```
a + rand() % ((b + 1) – a )
```

3.5 Function Overloading

Sometimes, two or more versions of some function are needed. For example, suppose that we want an `Area` function, like the one in Section 3.1, which prompts the user to enter a radius and then proceeds to return the computed area, but, in addition, we also want an `Area` function, which simply returns the computed area given the radius as a parameter. Here are the two versions:

Area Version 1

```
float Area()
{
      float PI = 3.14f;

      float radius = 0.0f;
      cout << "Enter a radius of a circle: ";
      cin >> radius;

      float area = PI*radius*radius;

      return area;
}
```

Area Version 2

```
float Area(float radius)
{
      float PI = 3.14f;
      return PI * radius * radius;
}
```

Because both versions of our area function were named "`Area`", the compiler would be expected to object to the redefinition of `Area`, and even if it did not, it is assumed that the compiler would not be able to distinguish between them. This is not the case, however, because these two versions of `Area` are not ambiguous due to their differing **function signature**. The signature of a function includes the function name and parameter listings. If either the function name or the parameter listings between several functions varies, then the compiler can deduce which function was called. Parameter listings vary by quantity, type or both. Observe that the return type is *not* part of the function signature. The act of defining several different versions—which differ in signature—of a function is called **function overloading.**

Let us look at another example. Recall that the `PrintPoint` function from Section 3.1.3 was implemented like so:

```
void PrintPoint(float x, float y, float z)
{
      cout << "<" << x << ", " << y << ", " << z << ">" << endl;
}
```

In addition to passing the coordinates x, y, and z, a programmer may want to pass in a 3-element array, where element [0] corresponds to the x-coordinate, [1] to the y-coordinate, and [2] to the z-coordinate. To facilitate this, we overload `PrintPoint` like so:

```
void PrintPoint(float p[3])
{
    cout << "<" << p[0] << ", " << p[1] << ", " << p[2] << ">" << endl;
}
```

A better way to implement our new `PrintPoint` is in terms of the other `PrintPoint` function. That is:

```
void PrintPoint(float p[3])
{
      PrintPoint(p[0], p[1], p[2]);
}
```

Program 3.15 revises Program 3.6, this time using both versions of `PrintPoint`.

Program 3.15: Function Overloading.

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

void PrintPoint(float x, float y, float z);
void PrintPoint(float p[3]);

int main()
{
      PrintPoint(1.0f, 2.0f, 3.0f);

      PrintPoint(-5.0f, 3.5f, 1.2f);

      float point1[3] = {-12.0f, 2.3f, -4.0f};

      float point2[3] = {9.0f, 8.0f, -7.0f};

      PrintPoint(point1); // use array version
      PrintPoint(point2); // use array version
}

void PrintPoint(float x, float y, float z)
{
      cout << "<" << x << ", " << y << ", " << z << ">" << endl;
}

void PrintPoint(float p[3])
{
      PrintPoint(p[0], p[1], p[2]);
}
```

Program 3.15 Output

```
<1, 2, 3>
<-5, 3.5, 1.2>
<-12, 2.3, -4>
<9, 8, -7>
Press any key to continue
```

The client (user of the functions) is now provided with two options. The client can either pass in the individual coordinates of the point, or pass in a 3-element array that represents the point. Providing several equivalent functions with different parameters is convenient because, depending on the data representation with which the client is working, the client can call the most suitable function version.

3.5.1 Default Parameters

A default parameter is a parameter where the caller has the option of specifying an argument for it. If the function caller chooses not to specify an argument for it then the function uses a specified default value called the “default” parameter. To illustrate, consider the following:

Program 3.16: Default Parameters.

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

// Function declaration with default parameters.  Default parameters
// take the syntax '= value' following the parameter name.
void PrintSomethingLoop(string text = "Default", int n = 5);

int main()
{
      // Specify an argument for both parameters.
      PrintSomethingLoop("Hello, World", 2);
      // Specify an argument for the first parameter.
      PrintSomethingLoop("Hello, C++");

      // Use defaults for both parameters.
      PrintSomethingLoop();
}

void PrintSomethingLoop(string text, int n)
{
       for(int i = 0; i < n; ++i)
             cout << text << endl;
       cout << endl;
}
```

Program 3.16 Output

```
Hello, World
Hello, World

Hello, C++
Hello, C++
Hello, C++
Hello, C++
Hello, C++

Default
Default
Default
Default
Default

Press any key to continue
```

Admittedly, this is a contrived and impractical function. Still, it illustrates the concept of default parameters. First of all, to specify a default value for a parameter, the syntax "= value" is appended to the parameter name. Also ***observe that the default parameter syntax ("= value") is only specified in the function declaration and not in the definition.*** The function `PrintSomethingLoop` simply outputs the passed-in `text` parameter `n` number of times. The first time this function is called, we specify arguments for both parameters. The second time this function is called, we specify the `text` parameter only and leave `n` to the default value—5. Note that it would be impossible to do the opposite (to specify `n` but leave `text` to its default value) because if we did, we would have something like `PrintSomethingLoop(10);` and the compiler would think we were trying to pass in 10 for `text`. Thus, default parameters must be the right-most parameters. For example, if you had parameters listed left to right, *a* through *d*, *d* would be considered the default parameter because it is the right-most parameter. If *d* was specified, then you could also include *c* as a default parameter (the next right-most parameter). However, you could not set *a* as a default parameter unless *b*, *c*, and *d* were also default parameters. You could not set *b* as a default parameter unless *c* and *d* were also default parameters, and so on. The third time this function is called we use the default values for both and specify no arguments. The corresponding program output verifies that, indeed, the default values were used.

3.6 Summary

1. A function is a unit of code designed to perform a certain task. By dividing our program into several different functions, we organize our code into more easily manageable parts. Moreover, functions help avoid code duplication. To define a function you must specify its return type, its name, its parameter list, and a body (i.e., the code to be executed when the function is invoked).

2. Variable scope refers to what variables a part of the program can "see" or "know" about. A variable defined in a particular code unit is said to be *local* relative to that code unit. A variable defined outside a particular code unit is said to be *global* relative to that code unit. A unit of code can "see" variables that are global and local, relative to it.

3. The C++ standard library provides functions for many of the elementary math functions and operations, such as trigonometric, logarithmic, exponential functions, as well as square root and absolute value functions. To use the standard math functions, the `<cmath>` header file must be included.

4. Use the `rand` function to generate random numbers. Remember to first seed the random number generator with the system time (and only once per program) using the `srand` function, so that your programs generate different random numbers every time they run. To use the random number functions the `<cstdlib>` header file must be included.

5. Function overloading allows implementation of several versions of a function with the same name as long as the function signature is still different. The signature of a function includes the function name and parameter listings. Therefore, to implement several versions of a function with the same name the parameter listings must vary between the different versions so that the signatures differ. Vary parameter listings in quantity, in type, or both. Note that a function signature does *not* include the return type.

6. A default parameter is a parameter where the caller has the option of specifying an argument for it. If the function caller chooses not to specify an argument for it then the function uses a specified default value which is called the "default" parameter. To make a parameter a default parameter, we append the syntax "= value" to the parameter name (e.g., `void Func(int x = 5)`). Note that the default parameter syntax ("= value") is only specified in the function declaration and not in the definition.

3.7 Exercises

3.7.1 Factorial

Rewrite the factorial program (Section 2.8.4) using a function. That is, implement a function called `Factorial` that inputs (i.e., has a parameter) a positive integer `n`. The function should then compute the factorial of `n`, and return the result. The function is to have the following prototype:

```
      int Factorial(int n);
```

After you have implemented this function, test your function by calculating 5!, 0!, 9!, and 3!, and output the results to the console window. The output should be formatted like so:

```
5! = 120
0! = 1
9! = 362880
3! = 6
Press any key to continue
```

3.7.2 ToUpper; ToLower

Recall that characters (i.e., `char` types) are represented internally with an integer value. The following chart shows an abridged listing of an abridged ASCII table.

```
33: ! 34: " 35: # 36: $ 37: % 38: & 39: ' 40: ( 41: ) 42: *
43: + 44: , 45: - 46: . 47: / 48: 0 49: 1 50: 2 51: 3 52: 4
53: 5 54: 6 55: 7 56: 8 57: 9 58: : 59: ; 60: < 61: = 62: >
63: ? 64: @ 65: A 66: B 67: C 68: D 69: E 70: F 71: G 72: H
73: I 74: J 75: K 76: L 77: M 78: N 79: O 80: P 81: Q 82: R
83: S 84: T 85: U 86: V 87: W 88: X 89: Y 90: Z 91: [ 92: \
93: ] 94: ^ 95: _ 96: ` 97: a 98: b 99: c 100: d 101: e 102: f
103: g 104: h 105: i 106: j 107: k 108: l 109: m 110: n 111: o 112: p
113: q 114: r 115: s 116: t 117: u 118: v 119: w 120: x 121: y 122: z
123: { 124: | 125: } 126: ~ 127: ⌂
```

Observe the sequential ordering of the letters (e.g., integers 65, 66, 67, correspond with letters 'A', 'B', and 'C', respectively). This sequential ordering makes it easy to setup a loop that iterates through each letter in the alphabet.

Using the preceding abridged ASCII table, implement a function called `ToUpperCase`, which inputs a single character variable and returns the uppercase form of that character. For example, if the letter 'a' is input, the function should return 'A'. If the input character is already in uppercase form, then the function should return that uppercase form (e.g., if 'A' is the input, then return 'A'). If the input character is not a letter character (i.e., not a member of the alphabet) then return a null character (‘/0’). The function is to have the following prototype:

```
char ToUpperCase(char input);
```

Additionally, implement a complementary function called `ToLowerCase`, which inputs a single char variable and returns the lowercase form of that character. For example, if the letter ‘A’ is input, the function should return ‘a’. If the input character is already in lower case form, then the function should return that lowercase form (e.g., if ‘a’ is the input then return ‘a’). If the input character is not a letter character (i.e., not a member of the alphabet) then return a null character (‘/0’). The function is to have the following prototype:

```
char ToLowerCase(char input);
```

After you have implemented these functions and verfied them to work, then you are to do the following: Create a loop that iterates over every lowercase letter in the alphabet. For each letter, call `ToUpperCase` and output the result to the console screen. The output for this part of the exercise should be the uppercase letterforms of every letter in the alphabet.

Additionally, create a loop that iterates over every uppercase letter in the alphabet. For each letter, call `ToLowerCase` and output the result to the console screen. The output for this part of the exercise should be the lowercase letterforms of every letter in the alphabet.

3.7.3 3D Distance

Frequently in a 3D game, you will need to know the distances between objects in the game. The distance between two points $\mathbf{u} = (u_x,\ u_y,\ u_z)$ and $\mathbf{v} = (v_x,\ v_y,\ v_z)$ in 3-space is given by the formula:

$$d = \sqrt{(v_x - u_x)^2 + (v_y - u_y)^2 + (v_z - u_z)^2}\ .$$

Implement a function that inputs two points and returns the distance between them. The function is to have the following prototype:

```
float dist3(float ux, float uy, float uz,
            float vx, float vy, float vz);
```

Test this function by using it to compute the distance between these points:

a. $(1,\ 2,\ 3)$ and $(0,\ 0,\ 0)$.
b. $(1,\ 2,\ 3)$ and $(1,\ 2,\ 3)$.
c. $(1,\ 2,\ 3)$ and $(7,\ -4,\ 5)$.

The output should look like this:

```
Distance between (1, 2, 3) and (0, 0, 0) = 3.74166
Distance between (1, 2, 3) and (1, 2, 3) = 0
Distance between (1, 2, 3) and (7, -4, 5) = 8.7178
Press any key to continue
```

(Tip: You may want to write a separate function to print 3D points to make the output of points less cumbersome.)

3.7.4 Arc Tangent 2

Background Information

Consider $\tan(\theta) = y/x$, where *y* and *x* are coordinates of some point in the 2D plane. Solving for theta we obtain $\theta = \tan^{-1}(y/x)$. But, recall from your studies of trigonometry that the inverse tangent function has some problems; in particular, its range is $[-90°, 90°]$, which means we cannot get angles outside quadrants 1 and 4. However, $(x,\quad y)$ can be in any quadrant. Clearly we have a problem, but making some observations easily solves it. Let us work with a concrete example to make this a bit easier.

Let $x = -4$ and $y = 4$. Clearly $(-4,\quad 4)$ lives in quadrant 2[2] and makes a 135° angle with the positive x-axis (sketch it out on paper). Because the point lies in quadrant 2, we know the inverse tangent will *not* return the correct angle since quadrant 2 is not in the inverse tangent's range. Are we stuck? Not yet. Let us calculate the inverse tangent just to see what happens: $\theta = \tan^{-1}(y/x) = \tan^{-1}(4/-4) = -45°$. Here we observe that if we add 180° to the inverse tangent result then we obtain the correct angle 135°; that is,
-45° + 180° = 135°. In fact, if the angle θ falls in quadrant 2 or 3 (which we can determine by examining the signs of *x* and *y*), we will always be able to get the correct angle by adding 180°. In summary:

> If θ is in quadrants 1 or 4 then $\theta = \tan^{-1}(y/x)$.
> Else if θ is in quadrant 2 or 3 then $\theta = \tan^{-1}(y/x)$ + 180°.

Exercise

Using `atanf`, write a function with the following prototype:

```
float MyArcTangent(float y, float x);
```

[2] We know this by examining the signs of x and y: Since x is negative it has to be to the left of the y-axis, and since y is positive it must be above the x-axis. Therefore, the point lies in quadrant 2.

This function should examine the signs of the coordinates of the point $(x, \quad y)$ and return the correct angle based on what quadrant the point lies in as described in the background readings for this lab project. Test your function with the following points: (2, 4), (-1, 5), (-6, -4), (4, -6). You should get the following results:

```
MyArcTangent( 4,  2) = 63.4671
MyArcTangent( 5, -1) = 101.27
MyArcTangent(-4, -6) = 213.707
MyArcTangent(-6,  4) = -56.3385
Press any key to continue
```

Now that you have written the function yourself, you should know that the C++ standard math library already includes a function that does exactly what your function does. Its prototype is:

```
      float atanf2(float y, float x);
```

3.7.5 Calculator Program

To get some practice using the standard math functions you will write a simple calculator program. The program should display the following menu:

```
1) cos(x), 2) sin(x), 3) tan(x), 4) atan2(y, x), 5) sqrt(x), 6) x^y, 7) ln(x), 8)
e^x, 9) |x|, 10) floor(x), 11) ceil(x), 12) Exit.
```

The program should prompt the user to make a selection from the menu. After the user has made their selection the program should ask for the input values; note that some functions only need one input value, whilst others need two. After the user enters the input values, the program should perform the calculation and output the result. The program should then loop back and again prompt the user to make a selection. The program should continue this process until the user quits:

```
1) cos(x), 2) sin(x), 3) tan(x), 4) atan2(y, x), 5) sqrt(x), 6) x^y
7) ln(x), 8) e^x, 9) |x|, 10) floor(x), 11) ceil(x), 12) Exit.1
Enter x: 3.14
cos(x) = -0.999999
1) cos(x), 2) sin(x), 3) tan(x), 4) atan2(y, x), 5) sqrt(x), 6) x^y
7) ln(x), 8) e^x, 9) |x|, 10) floor(x), 11) ceil(x), 12) Exit.4
Enter x: 2
Enter y: 4
atan2(y, x) = 1.10715
1) cos(x), 2) sin(x), 3) tan(x), 4) atan2(y, x), 5) sqrt(x), 6) x^y
7) ln(x), 8) e^x, 9) |x|, 10) floor(x), 11) ceil(x), 12) Exit.7
Enter x: 2
ln(x) = 0.693147
1) cos(x), 2) sin(x), 3) tan(x), 4) atan2(y, x), 5) sqrt(x), 6) x^y
7) ln(x), 8) e^x, 9) |x|, 10) floor(x), 11) ceil(x), 12) Exit.9
Enter x: -5
|x| = 5
1) cos(x), 2) sin(x), 3) tan(x), 4) atan2(y, x), 5) sqrt(x), 6) x^y
```

```
7) ln(x), 8) e^x, 9) |x|, 10) floor(x), 11) ceil(x), 12) Exit.11
Enter x: 11.2
ceil(x) = 12
1) cos(x), 2) sin(x), 3) tan(x), 4) atan2(y, x), 5) sqrt(x), 6) x^y
7) ln(x), 8) e^x, 9) |x|, 10) floor(x), 11) ceil(x), 12) Exit.12
Exiting...
Press any key to continue
```

3.7.6 Slot Machine

Implement a function that returns a random integer in the range `[low, high]`, where `low` and `high` are input parameters. The function prototype should look like this:

```
	int Random(int low, int high);
```

Be sure to verify that your function implementation works by testing it.

Using your `Random` function, write a virtual slot machine program. The program should start the player off with $1000.00, and should display a menu like this:

```
Player’s chips: $1000
1) Play slot. 2) Exit.
```

If the player enters “1”, the program should ask the user to enter in his or her bet. The program needs to verify that a legal bet was placed; that is, a bet greater than zero and less than or equal to the amount of money the player has. After the player has input his or her bet, the program must calculate three random numbers in the range [2, 7] and output them neatly to the screen. If all three numbers are sevens, then the player wins ten times their betting money; else if, the three numbers are all the same, but not sevens, then the player wins five times their betting money; else if, two out of the three numbers are the same then the player wins three times their betting money; else, the player loses his or her bet. At this point, calculate the player’s new chip amount and redisplay the menu. If at any point the player loses all of his or her chips, a message should be displayed to the player and the program should exit. Also, if the player enters “2” from the menu then the program should exit. Here is an example of what the output should look like:

```
Player’s chips: $1000
1) Play slot. 2) Exit. 1
Enter your bet: 1500
You did not enter a valid bet.
Enter your bet: 1000
3 3 7
You win!
Player’s chips: $3000
1) Play slot. 2) Exit. 2
Exiting…
```

3.7.7 Binary Search

Background Information

In the exercises of the previous chapter we examined the linear search. In the worst cast scenario, the linear search must scan every single element in the given array. For large datasets this can be problematic (i.e., too slow/inefficient). A faster search method is the **binary search.** However, the binary search requires that the dataset be already sorted in some way.

To illustrate the binary search, let us examine a concrete example. Consider the following array of integers, which have already been sorted in ascending order:

1, 4, 5, 6, 9, 14, 21, 23, 28, 31, 35

Suppose that you want to find integer 21. Instead of starting the search at the beginning of the array, in the binary search we start at the middle. The value stored in the middle array element is 14. Since 21 is greater than 14 and the array is sorted in ascending order, we are guaranteed that the item we are searching for lies in the upper half of the array. Thus, we do not need to check any elements in the lower half of the array. We have, with one test, eliminated half of the elements we would potentially have to scan. We now consider the upper half of the previous working dataset, which we have bolded:

1, 4, 5, 6, 9, 14, **21, 23, 28, 31, 35**

Again we start at the middle of our new dataset. The value stored in the middle is 28. Since the value 21 is less than 28 and the array is sorted in ascending order, we are guaranteed that the item we are searching for lies in the lower half of our working subarray. Thus, we do not need to check any elements in the upper half of our working subarray. We now consider the lower half of the previous working dataset, which we have bolded:

1, 4, 5, 6, 9, 14, **21, 23**, 28, 31, 35

In this case there is not an exact middle, so we will arbitrarily choose the lower element as the “middle.” And that element is the value we are searching for, located at position 6 in the array.

The key idea of the binary search is this: Because the data is sorted, we can quickly eliminate half of your data set with each scan. This is the beauty of the binary search. To make this result more profound, imagine that you had a sorted array of ten thousand integers. Using a linear search you very well might have to scan all 10,000 integers to find the one you want. Conversely, the binary search eliminates half of its data set after each scan. After just one test, the binary search has narrowed the search down to 5, 000 integers, after the second test the binary search is down to 2,500 integers, and so on.

Exercise

Write a function that searches, using the binary search algorithm, an integer array and returns the array position of the "found" integer that matches the search key. The function should be prototyped as follows:

```
int BinSearch(int data[],int numElements, int searchKey);
```

Use the following array for test purposes:

{1, 4, 5, 6, 9, 14, 21, 23, 28, 31, 35, 42, 46, 50, 53, 57, 62, 63, 65, 74, 79, 89, 95}

Your output should be similar to the following:

```
{1, 4, 5, 6, 9, 14, 21, 23, 28, 31, 35, 42, 46, 50, 53, 57, 62, 63, 65, 74, 79, 89,
95}
Enter search key (or 'x' to exit): 21
21 is in position 6.
Enter search key (or 'x' to exit): 50
50 is in position 13.
Enter search key (or 'x' to exit): 0
0 not found.
Enter search key (or 'x' to exit): x
Exiting…
```

3.7.8 Bubble Sort

Background Information

The **bubble sort** algorithm is similar to the selection sort in that it also makes several passes over the array, and after each pass the array becomes more and more sorted. However, the bubble sort has a couple of advantages over the selection sort. First, besides moving one element to its sorted position per pass, the other elements "bubble" closer to their correct positions. Second, we can "skip" a pass if it is unnecessary; that is, if the next element is already in its sorted position then we do not need to do any work and we can skip to the next element.

The bubble sort algorithm is best explained by looking at an example. Consider the following array of integers:

x[6] = x[0],…,[5] = {12, 5, 21, 1, 15, 17}.

And suppose that we wish to sort this array in *ascending order*.

Pass 1:

On the first pass, our working subarray is x[0],…,x[5] (the entire array). We start at x[0] and compare it to its right-next-door neighbor x[1]. Because 12 is greater than 5, we swap the two values. This yields:

```
5, 12, 21, 1, 15, 17
```

Next, we compare x[1] and x[2]. Because 12 is less than 21, we do *not* swap the two values. Thus the array remains unchanged. Continuing the pattern by comparing x[2] and x[3], we have 21 is greater than 1, so we swap the two values:

```
5, 12, 1, 21, 15, 17
```

Repeating this process yields:

```
5, 12, 1, 15, 21, 17      // comparing x[3] and x[4]
5, 12, 1, 15, 17, 21      // comparing x[4] and x[5]
```

From this first pass we observe the following rule:

- Rule 1: If x[i] is greater than its right-next-door neighbor x[i+1] then swap x[i] and x[i+1].

In consequence to this rule, we are guaranteed, for each pass, that the greatest value in the working subarray will be placed in its sorted position. And indeed, observe that the greatest value, 21, is in its sorted position (the top of the array).

Pass 2:

We are guaranteed from the previous pass that the greatest value is correctly positioned at the top of the working subarray. Hence, for the second pass we need only consider the subarray x[0],…,x[4]. We start at x[0] and compare it to its right-next-door neighbor x[1]. Because 5 is less than 12 we do nothing. Comparing x[1] and x[2] we have 12 is greater than 1, which indicates a swap operation must take place. Swapping x[1] and x[2] yields:

```
5, 1, 12, 15, 17, 21
```

Continuing this pattern results in the following:

```
5, 1, 12, 15, 17, 21      // comparing x[2] and x[3] results in no change.
5, 1, 12, 15, 17, 21      // comparing x[3] and x[4] results in no change.
```

Observe that the last element to be involved in a swap operation is x[2]. Moreover, observe that the subarray x[2],…,x[5] is sorted. This is no coincidence and brings us to a new rule:

- Rule 2: Suppose we are working with a zero-based array of *n* elements. For a particular bubble sort pass, if the last element to be involved in a swap operation is x[k] then we can conclude that the subarray x[k],…,x[n-1] is sorted.

Pass 3:

The previous pass told us that the last swap occurred at x[2]. Thus, for this pass we are only concerned with the subarray x[0],…,x[1]. We start at x[0] and compare it to its right-next-door neighbor x[1]. Since 5 is greater than 1 we swap the values. This yields:

1, 5, 12, 15, 17, 21

We are now done with the third pass. Because the last index involved in the last swap was x[1], we can conclude, from Rule 2, that the subarray x[1],…,x[5] is sorted. But if x[1],…,x[5] is sorted then the last element x[0] must also be in its sorted position, and from inspection we observe it is. Thus the entire array has been sorted in ascending order.

The following algorithm outline summarizes the bubble sort:

Table 3: Ascending order Bubble Sort.

```
Let x[n] = x[0],...,x[n-1] be an array of given integers to sort.
Let SubArrayEnd be an integer to store the last index of the working subarray.
Let nextEnd be an integer used to help compute the end of the next pass’ subarray.

Initialize SubArrayEnd = n – 1.

While SubArrayEnd > 0, do the following:

    1. Initialize nextEnd = 0;
    2. For j = 0 to SubArrayEnd - 1, do the following:
        a. If x[j] > x[j+1] then
            i. swap x[j] and x[j+1].
            ii. nextEnd = j;
        b. Increment j.

    3. SubArrayEnd = nextEnd.
```

Exercise

Ask the user to input ten random (non-sorted) integers and store them in an array. Sort these integers in ascending order using a **bubble sort** function and output the sorted array to the user. Your function should have the following prototype:

```
void BubbleSort(int data[], int n);
```

where `data` is the array parameter, and `n` is the number of elements in the array (i.e., the size of the array).

Your output should look similar to the following:

```
Enter ten unsorted integers...
[0] = 5
[1] = -3
[2] = 2
[3] = 1
[4] = 7
[5] = -9
[6] = 4
[7] = -5
[8] = 6
[9] = -12

Unsorted List = 5, -3, 2, 1, 7, -9, 4, -5, 6, -12,
Sorting...
Sorted List = -12, -9, -5, -3, 1, 2, 4, 5, 6, 7,
Press any key to continue
```

Chapter 4

References and Pointers

Introduction

We are at the point now where we can write some useful programs. Our programs can make decisions based on input and the program's status using conditional statements. We can execute blocks of code repeatedly with loop statements. We can organize our programs into multiple parts with functions, each designed for a specific task. However, there are still some outstanding problems.

First, recall that when passing arguments into functions, the argument is copied into the parameter. But what if you are passing in an array? An array can potentially be very large and copying every element value from the argument to the parameter would be very inefficient.

Second, we learned in the last chapter that a function could return or evaluate to some value. But what if we want to return more than one value?

Finally, so far in every program we have written, when we needed memory (variables) we declared them in the program code. But declaring the variables in the program implies that we know, ahead of time, all the memory the program will need. But what if the amount of memory needed is variable? For example, in a massive multiplayer online game, you may use an array to store all of the game players. Because players are constantly entering and leaving online play, the array may need to resize accordingly.

All of these problems can be solved with references or pointers.

Chapter Objectives

- Become familiar with reference and pointer syntax.
- Understand how C++ passes array arguments into functions.
- Discover how to return multiple return values from a function.
- Learn how to create and destroy memory at runtime (i.e., while the program is running).

4.1 References

A **reference** is essentially an alias for a variable. Given a reference *R* to a variable *A*, we can directly access *A* with *R* since *R refers* to *A*. Do not worry about why this is useful, as we will discuss that further in coming sections. For now, just focus on learning the syntax of references.

To create a reference you must:

- specify the type of variable the reference will refer to
- follow with the unary **address of operator** (&)

- follow with the name of the reference
- follow with an initialization, which specifies the variable the reference refers to

For example, to create a reference, called `valueRef`, to an integer called `value` we would write:

```
int value = 0; //<-- Create a variable called 'value'.
int& valueRef = value; //<--Create a reference to 'value'.
```

Here are some other examples using various types:

```
// Variables:
float pi    = 3.14f;
char letter = 'B';
bool truth  = false;
double e    = exp(1.0);

// References to those variables:
float& piRef    = pi;
char& letterRef = letter;
bool& truthRef  = truth;
double& eRef    = e;
```

We can access the variable a reference refers to through the reference. This is because a reference is just an alias to that variable. Program 4.1 verifies this:

Program 4.1: Accessing a variable via a reference to it.

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

int main()
{
      // Create variable.
      int value = 10;

      // Create reference to 'value'.
      int& valueRef = value;

      // Print the number stored in 'value'.
      cout << "value = " << value << endl;

      // Also print the value referenced by 'valueRef'.
      // Because 'valueRef' is an alias for 'value' it
      // should print the same number stored in 'value'.
      cout << "valueRef = " << valueRef << endl;

      // Modify the reference.  However since the reference is
      // just an alias for 'value', modifying 'valueRef' modifies
      // the number stored in 'value'.
      valueRef = 500;

      // Print the number stored in 'value' to prove that modifying
      // the reference modifies the variable it refers to.
      cout << "value = " << value << endl;
```

```
        // And print 'valueRef' again.
        cout << "valueRef = " << valueRef << endl;
}
```

Program 4.1 Output

```
value = 10
valueRef = 10
value = 500
valueRef = 500
Press any key to continue
```

We have two different names (`value` and `valueRef`), which both refer to the same variable—that is, the same unit of memory—and as such, they both can access and modify that variable.

> **Important:** References must be initialized when they are declared. You cannot have a reference that does not refer to anything. This is illegal:
>
> ```
> int& valueRef; //<--Error uninitialized reference.
> ```

4.1.1 Constant References

Suppose we try and write the following:

```
int& valueRef = 1;
```

If we try and compile this we will get an error; namely, "error C2440: 'initializing' : cannot convert from 'int' to 'int &' ." This should not be surprising since a reference is an alias to a variable and a literal is not a variable. Still, sometimes we will want to be able to assign literals to references. We note, however, that such an alias to a literal should not be able to change the literal through that alias; this restriction simply follows from the fact that it does not make sense to change a literal—a literal is literally that value.

To facilitate literal assignments to references we must use **constant references:**

```
const int& valueRef = 1;
```

If we try to change a constant reference like so:

```
valueRef = 20;
```

we get the error "error C2166: l-value specifies const object."

A constant reference is actually implemented by the compiler as follows:

```
const int temp = 1;
const int& valueRef = temp;
```

It creates a temporary `const` variable to store the literal, and then has the reference refer to this temporary variable. The temporary will stay alive in memory as long as the reference to it stays alive in memory.

4.2 Pointers

4.2.1 Computer Memory Primer

In a computer, each byte[3] of memory has a unique **memory address.** Figure 4.1 shows a conceptual example of a segment of computer memory.

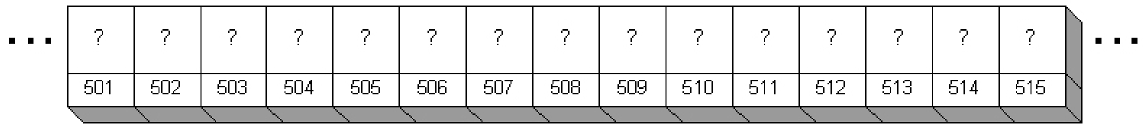


Figure 4.1: A segment of memory. Each upper square represents a byte of memory; the question marks denote that we do not know what value is stored in these bytes. Each bottom rectangle represents a unique memory address, which corresponds to a byte in memory.

We learned in the first chapter that the various C++ intrinsic types require different amounts of memory; recall that in 32-bit Windows, a `short` variable is two bytes and a `float` variable is four bytes. Figure 4.2 shows how a variable `varShort` of type `short` and a variable `varFloat` of type `float` would be stored in memory.

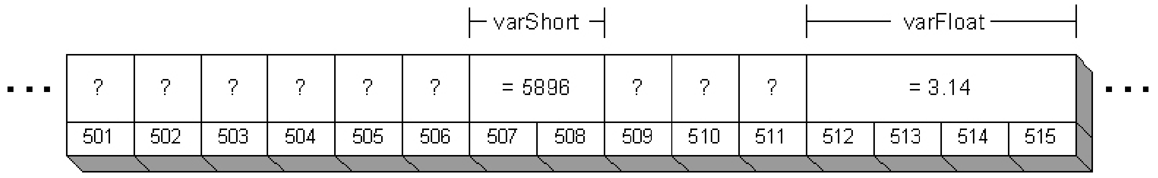


Figure 4.2: Variables stored in memory spanning multiple bytes.

Since these two variables require more than one byte apiece, they span over several bytes in the memory layout. That being the case, it is natural to ask what the address of `varShort` and `varFloat` is. C++ considers the address of multi-byte types to be the address of the "lowest" byte the variable spans. Thus, from Figure 4.2, the address of `varShort` is 507 and the address of `varFloat` is 512.

[3] A byte is the smallest addressable piece of memory.

4.4.2 Pointer Initialization

A **pointer** is a special variable type that can store the memory address of another variable. For example, suppose that a pointer exists called `varFloatPtr`, which stores the address of `varFloat`. Figure 4.3 illustrates this relation.

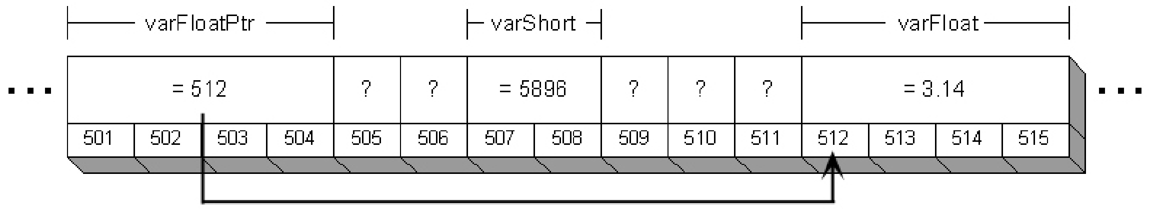


Figure 4.3: Here we have added a pointer variable `varFloatPtr`, which stores the address of another variable, namely `varFloat`. Observe that the pointer occupies four bytes; this is because we are assuming a 32-bit system where pointers are 32-bits.

You can see why they call these types of variables "pointers"—by storing the address of a variable they essentially 'point to' that variable. Furthermore, given the address of a variable (a pointer), the actual variable which is being pointed at can be accessed and modifed; this process is called **dereferencing.** Thus, like references, the same variable can be accessed via several different pointers that point to it. However, as it turns out, pointers can do more than references. In fact, the C++ reference mechanism is generally understood to be implemented using pointers "underneath the hood."

To declare a pointer, the type of variable the pointer will point to must be specified, followed by the unary **indirection operator** (*), followed by the name of the pointer. Example declarations:

```
bool*  boolPtr;
int*   intPtr;
float* floatPtr;
```

Note: The operator (*) is not ambiguous because the compiler can determine through context whether to interpret it as the unary indirection operator or as the binary multiplication operator.

Unlike references, pointers do not have to be initialized, but they should always be initialized for the same reason all variables should always be initialized to something—the program is easier to debug when you are able to recognize a default value. When a variable is filled with garbage, it is not so easy to recognize that it contains a bad value, and therefore, you may think it contains valid information. Moreover, unlike references, pointers can be assigned a **null** value. A null pointer is a pointer that points to nothing. So a good default value for pointers, if you wish to postpone initialization, is null. The null value in C++ is simply zero. Rewriting the preceding pointer declarations with initialization to null yields:

```
bool*  boolPtr  = 0;
int*   intPtr   = 0;
float* floatPtr = 0;
```

Note: Some programmers like to define a macro called `NULL`, which is equal to zero. That is,

```
#define NULL 0
```

This is so that they can nullify pointers by writing:

```
bool*  boolPtr  = NULL;
int*   intPtr   = NULL;
float* floatPtr = NULL;
```

We do not use `NULL` in this book, but we bring it up because you may see this `NULL` used in other code, such as Microsoft Windows code.

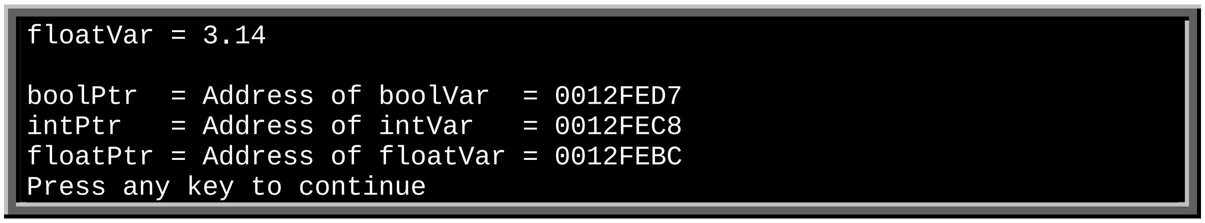
Initializing pointers to null is not very interesting. Pointers are variables that store the addresses of other variables, so what we want to be doing is assigning variable addresses to our pointers. To assign the address of a variable to a pointer, we need a way of getting the address of a variable. We can do that with the unary **address of** operator (&)—the same one used with references. The following examples illustrate:

Program 4.2: Initializing pointers and displaying memory addresses.

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

int main()
{
      bool boolVar   = true;
      int  intVar    = 50;
      float floatVar = 3.14f;

      // Initialize pointers to the addresses of the
      // corresponding variables.
      bool*  boolPtr  = &boolVar;
      int*   intPtr   = &intVar;
      float* floatPtr = &floatVar;

      // Print normal variable values.
      cout << "boolVar  = " << boolVar  << endl;
      cout << "intVar   = " << intVar   << endl;
      cout << "floatVar = " << floatVar << endl;

      cout << endl;

      // Print the addresses the pointers store.
      cout << "boolPtr  = Address of boolVar  = " << boolPtr  << endl;
      cout << "intPtr   = Address of intVar   = " << intPtr   << endl;
      cout << "floatPtr = Address of floatVar = " << floatPtr << endl;
}
```

Program 4.2 Output

```
boolVar  = 1
intVar   = 50
```



```
floatVar = 3.14

boolPtr  = Address of boolVar  = 0012FED7
intPtr   = Address of intVar   = 0012FEC8
floatPtr = Address of floatVar = 0012FEBC
Press any key to continue
```

The syntax '`&`' followed by the variable name evaluates to the memory address of the variable name. So the following, for example, `float* floatPtr = &floatVar`, reads like so: Declare a pointer to a `float` called `floatPtr` and assign to it the address of a `float` variable called `floatVar`.

The strange output for the pointers (e.g., `0012FED7`) is a 32-bit hexadecimal number, which is how `cout` outputs pointers by default. Hexadecimal is simply another numbering system that is useful when analyzing memory; we will postpone discussing hexadecimal until Chapter 12 in the next module. If you want to see the integer address representation then you can cast the pointer to an `int` before outputting it:

```
cout << "boolPtr  = Address of boolVar  = " << (int)boolPtr  << endl;
cout << "intPtr   = Address of intVar   = " << (int)intPtr   << endl;
cout << "floatPtr = Address of floatVar = " << (int)floatPtr << endl;
```

You will get output similar to this:

```
boolPtr  = Address of boolVar  = 1244887
intPtr   = Address of intVar   = 1244872
floatPtr = Address of floatVar = 1244860
```

Note: Pointers can only store variable addresses; if we try to assign a value to a pointer like this: `float* floatPtr = floatVar`, we get the error: "error C2440: 'initializing' : cannot convert from 'float' to 'float *'."

4.4.3 Dereferencing

Given the address of a variable (i.e. a pointer) we can access and modify the actual variable pointed to by *dereferencing* the pointer. To dereference a pointer, we prefix the pointer name with the indirection operator (*). For example, given the initialized pointer `float* floatPtr = &floatVar`, we can dereference `floatPtr` with the syntax `*floatPtr`, which evaluates to the variable being pointed to; that is, `floatVar`. Figure 4.4 shows the relationship between a pointer and the address, and a dereferenced pointer and the variable.

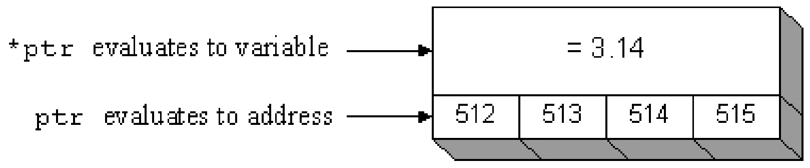


Figure 4.4: A pointer stores an address. By dereferencing a pointer we obtain the variable at the address pointed to.

With this pointer-variable relationship it follows that the variable whose address is stored in the pointer can be accessed, read, and modifed by dereferencing the pointer. This is similar to references; that is, with references, a variable can be accessed via references that refer to a variable, and similarly, with pointers, a variable can be accessed via pointers that point to it. The following program illustrates:

Program 4.3: Accessing a variable via a pointer to it.

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

int main()
{
      // Create variable.
      int value = 10;

      // Create a pointer to the address of 'value'.
      int* valuePtr = &value;

      // Print:
      cout << "value     = " << value     << endl;
      cout << "valuePtr  = " << valuePtr  << endl;
      cout << "*valuePtr = " << *valuePtr << endl;

      // Modify 'value' via the pointer by dereferencing it.
      *valuePtr = 500;

      // Print again to show changes:
      cout << "value     = " << value     << endl;
      cout << "valuePtr  = " << valuePtr  << endl;
      cout << "*valuePtr = " << *valuePtr << endl;
}
```

Program 4.3 Output

```
value     = 10
valuePtr  = 0012FED4
*valuePtr = 10
value     = 500
valuePtr  = 0012FED4
*valuePtr = 500
Press any key to continue
```

Admittedly, the following paragraph is hard to follow since the use of a pointer introduces a confusing layer of indirection. Read the paragraph slowly and refer to Figure 4.5.

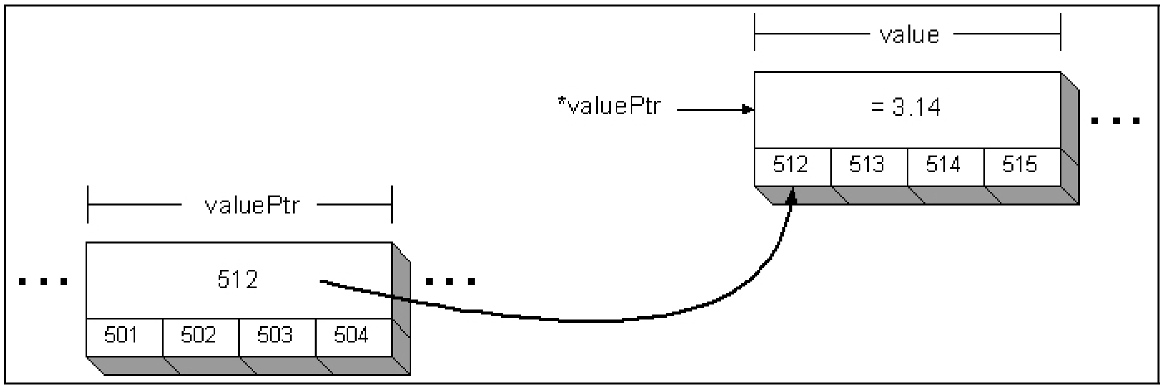


Figure 4.5: `valuePtr` stores the address of `value`. We can get the variable (`value`) at that address through the pointer `valuePtr` by dereferencing it (`*valuePtr`). Thus we can read and write to the variable `value` indirectly via the pointer `valuePtr`.

Program 4.3 is similar to Program 4.1, but instead of references we use pointers to indirectly modify the value stored in a variable. The first key operation Program 4.3 does is create a pointer to `value`: `int* valuePtr = &value`. The program then prints the value stored in `value` and the address stored in `valuePtr` (address of `value`). Then the program prints the value of the variable `valuePtr` currently points to (remember that it stores the address of this variable) by dereferencing the pointer (`*valuePtr`). Since `valuePtr` points to `value`, this output should be the same as just printing `value` directly, and based on the program output, it is; that is, `value == 10 == *valuePtr`.

Next the program makes an assignment to the dereferenced pointer: `*valuePtr = 500`. Because `*valuePtr` refers to the variable and `valuePtr` stores the address of (`value`), we expect this assignment to modify `value` as well. So finally, we print all the values out again to verify that making the assignment, `*valuePtr = 500` modified the variable that is pointed to (i.e `value`). And indeed it was modified: `value == 500 == *valuePtr`.

In summary, a pointer gives us *indirect* access to a variable. Given the address of a variable, we can get to that variable in the same way a letter can get to a house given the house address. By going through the pointer, you get the address of a variable, and then by dereferencing the pointer you get access to the variable. Of course, it is still not clear why this is even useful. Why take an indirect step when you can just access the variable directly? The following sections of this chapter show the benefits and real-world uses of references and pointers.

Note: Just as we can have constants for non-pointer variables and constant references, we can have constant pointers. There are four syntaxes we need to consider:

```
(i)     float* const constFloatPtr;
(ii)    float const* constFloat0;
(iii)   const float* constFloat1;
(iv)    const float* const constFloatConstFloatPtr;
```

Form (i) means that the pointer is constant; that is, the pointer variable itself cannot change; however, the variable pointed to can change. Form (ii) and (iii) are different syntaxes for the same idea; that being, the pointer is *not* constant but the variable pointed to is constant. Finally, form (iv) combines both; it says the pointer is constant and the variable pointed to is also constant.

Stroustrup suggests to read these declarations right-to-left; for example, (i) would read "constFloatPtr is a constant pointer to type float" and (iii) would read "constFloat1 is a pointer to type const float."

4.3 Arrays Revisited

4.3.1 Pointer to the Beginning of an Array

With our new understanding of the idea of pointers, it is now time to take a closer look at them. In C++, an array name can be converted to a pointer to the first element in the array. Consider the following array:

```
short arrayName[8] = {1, 2, 3, 4, 5, 6, 7, 8};
```

A pointer to the first element can be acquired by writing:

```
short* firstPtr = arrayName;
```

The claim is that `firstPtr` is a pointer to the first element in the array, which would be `arrayName[0]`—Figure 4.6.

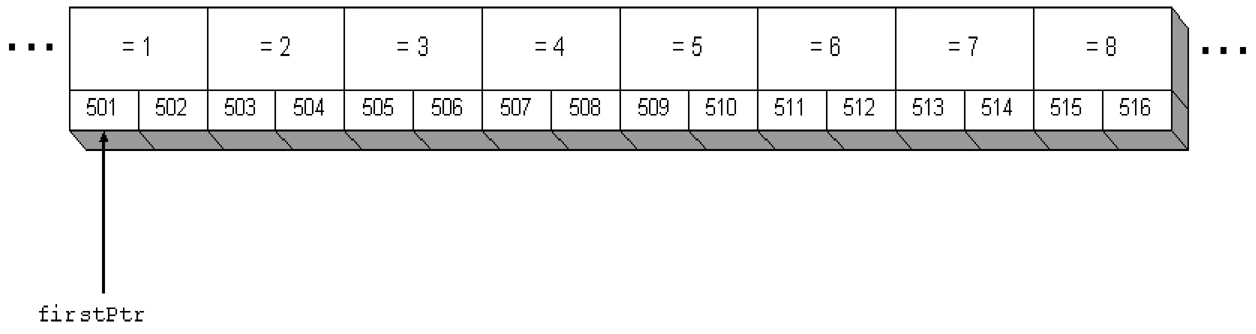


Figure 4.6: An array name gives a pointer to the first element of the array.

If this is true then dereferencing `arrayName` ought to yield the value of element [0]. Program 4.4 shows that this is, in fact, the case.

Program 4.4: Verification that we can get a pointer to the first element in an array via the array's name.

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

int main()
{
      short arrayName[8] = {1, 2, 3, 4, 5, 6, 7, 8};
```

```
       // Use array name to get pointer to the first element.
       short * firstPtr = arrayName;

       cout << "arrayName[0] = " << arrayName[0] << endl;
       cout << "*firstPtr = " << *firstPtr << endl;
}
```

Program 4.4 Output

```
arrayName[0] = 1
*firstPtr  = 1
Press any key to continue
```

The output of Program 4.4 verifies the claim; namely, `arrayName` is a pointer to the first element in the array.

4.3.2 Pointer Arithmetic

Given a pointer to the first element in an array, how do we access the other elements of the array? The C++ compiler knows the variable type of each element in the array (it knows because the type is specified when you declare the array) and it knows how many bytes need to be offset to get to the next element. Thus, C++ provides a way to navigate the elements of an array by offsetting the array pointer. To perform the actual offset operation we add an integer to the pointer. For example, we may write:

```
       firstPtr + 1;
       firstPtr + 5;
       firstPtr + 4;
```

The integer added indicates how many elements to offset. Figure 4.7 illustrates this.

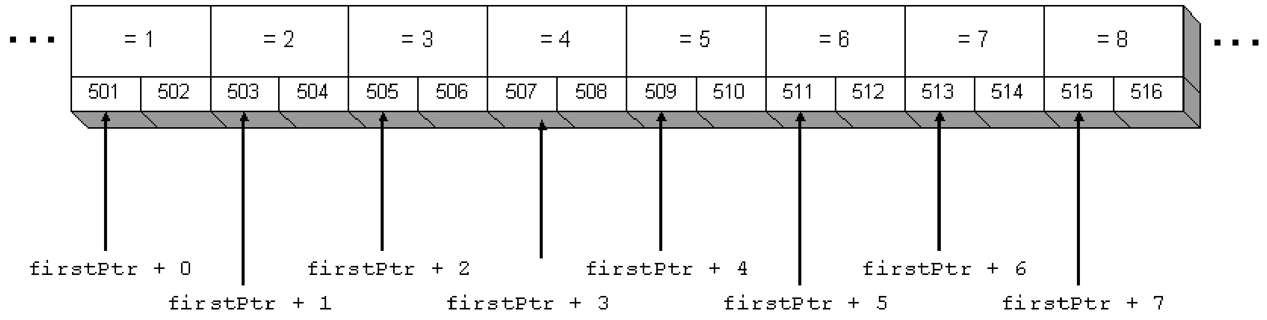


Figure 4.7: Pointer arithmetic.

Note that offsetting the pointer like this simply evaluates to a new pointer, which points to the offset element. In order to get the value at that new address the pointer needs to be dereferenced.

In addition to adding with the binary addition operator (+) a pointer can be "incremented", and also, compound assignments can be performed on it. Examples:

```
firstPtr++;   // Point to next element.
++firstPtr;   // Point to next element after that.
firstPtr += 4;// Point to element 4 array slots away
```

Note that the increment and compound assignment operators make assignments to the pointer and thus change the address it points to. In contrast:

```
firstPtr + 1;
```

This expression makes no assignment to `firstPtr` and does not actually change the address to which it points. Rather `firstPtr + 1` evaluates to a new pointer to the next element.

As well as addition type operations, it is possible to move backwards along the array using decrements and subtraction operations. However, these types of pointer arithmetic are seldom encountered. All of these pointer operations are referred to as **pointer arithmetic.** Note that multiplication and division is not defined for pointers.

> **Note:** You must be careful not to use pointer arithmetic to offset into a memory address that is not part of the array. For instance, the array in the preceding example contained ten elements. The offset `firstElementPtr + 12` goes outside the array boundary and into memory we do not "own." Accessing this memory, which may be used for something else, can be destructive.

The following program iterates over a small array using two different styles of pointer arithmetic and prints each element:

Program 4.5: Pointer arithmetic.

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

int main()
{
      short arrayName[8] = {1, 2, 3, 4, 5, 6, 7, 8};

      // Use array name to get pointer to the first element.
      short* firstPtr = arrayName;

      cout << "Style 1: Addition operator." << endl;
      for(int i = 0; i < 8; ++i)
      {
            cout << *(firstPtr + i) << " ";
      }
      cout << endl;
      cout << "Style 2: Increment operator." << endl;

      for(int i = 0; i < 8; ++i)
      {
            cout << *firstPtr << " ";
            ++firstPtr; // Move pointer to next element.
      }
      cout << endl;
}
```

Program 4.5 Output

```
Style 1: Addition operator.
1 2 3 4 5 6 7 8
Style 2: Increment operator.
1 2 3 4 5 6 7 8
Press any key to continue
```

In the first style the `ptr + integer` style is used, in particular: `firstPtr + i`. Because '*i*' is incremented for each loop cycle, each element is iterated over using this style. In the second style, the pointer for each loop cycle is incremented, which again effectively iterates over each element. Observe that in both cases the pointer must be dereferenced to get the actual value pointed to when we want to output it.

In the past we have always used the bracket operators [ ] to navigate the elements of an array. In reality the bracket operator is shorthand for the pointer arithmetic just discussed, plus a dereference; that is, the following are equivalent operations:

```
*(firstPtr + 1) == firstPtr[0]
*(firstPtr + 2) == firstPtr[1]
*(firstPtr + 3) == firstPtr[2]
...
```

To summarize, all of the elements of an array are accessible and navigable given a pointer to the first element in the array.

4.3.1 Passing Arrays into Functions

When passing arguments into functions, the argument is copied into the parameter. But what if you are passing in an array? An array can potentially be very large and copying every element value from the argument array to the parameter array would be very inefficient. C++ handles this problem by copying a pointer argument to the first element of the array into the parameter. The following example program verifies this:

Program 4.6: Array parameters.

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

void PrintArray(int array[20])
{
      // Output the size, in bytes, of the parameter.
```

```
	cout << "sizeof(array) = " << sizeof(array) << endl;

	// Print the array.
	for(int i = 0; i < 20; ++i)
		cout << array[i] << " ";

	cout << endl;
}

int main()
{
	// Seed the random number generator.
	srand( time(0) );

	// Array of 20 integers.
	int randomArray[20];

	// Fill each element with a random number in the range [0, 100].
	for(int i = 0; i < 20; ++i)
		randomArray[i] = rand() % 101;

	// Output the size, in bytes, of the array.
	cout << "sizeof(randomArray) = " << sizeof(randomArray) << endl;

	PrintArray( randomArray );
}
```

Program 4.6 Output

```
sizeof(randomArray) = 80
sizeof(array) = 4
23 5 20 41 5 32 90 13 49 8 98 39 39 80 1 6 79 60 98 66
Press any key to continue
```

From the output, we observe that the size of the array parameter is 4 bytes. Thus 20*4 bytes are *not* copied into the function, but rather only 4 bytes are—the size of a 32-bit pointer—thus showing a pointer was copied and not the entire array. From the preceding sections, we know that we can navigate over all the elements of an array given a pointer to the first element in the array. Hence, by passing a pointer for efficiency, we are not limited in what we can do with that array. The function `PrintArray` shows this as it proceeds to print every element in the array. Note that we use the bracket [] notation because it is clearer to read, but we could have used pointer arithmetic plus a dereference.

Note: Since we pass arrays with a pointer to the first element, we can also write the function signature like this:

```
void PrintArray(int* array);
```

Or like this:

```
void PrintArray(int array[]);
```

The efficiency gained by passing pointers to arrays into functions is the first advantage of pointers. Later, when we learn how to develop larger variable types (think of complex types built out of several intrinsic types), we will see that we can gain the same efficiency by passing pointers to these larger types instead of copies.

4.4 Returning Multiple Return Values

Suppose you have a function that needs to return multiple values back to the function caller. How would you do that? The usual `return` keyword approach restricts you to only one return value, so we must look for an alternative method. This alternative method relies on pointers or references.

4.4.1 Returning Multiple Return Values with Pointers

Suppose that we require a function called `GetMousePos`, which needs to return the x- and y-coordinates of the mouse position relative to the screen. Such a function is useful when you need a game (or any program) to react to mouse input. The following program illustrates how to implement this function so that it can return two parameters. (Note that, for illustration purposes, we return random numbers for the mouse's x- and y-coordinates since we do not know yet how to actually get the current mouse position.)

Program 4.7: Returning multiple return values with pointers.

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

void GetMousePos(int* outX, int* outY)
{
	// Pretend to return the mouse's current position.
	*outX = rand() % 801;
	*outY = rand() % 601;
}

int main()
{
	// Seed the random number generator.
	srand( time(0) );

	// Initialize two variables that will receive the
	// mouse position.
	int x = 0;
	int y = 0;

	// Output before x and y before receiving mouse position.
	cout << "Before GetMousePos(...)" << endl;
	cout << "x = " << x << endl;
	cout << "y = " << y << endl;
```

```
      GetMousePos( &x, &y );

      cout << "After GetMousePos(...)" << endl;
      cout << "x = " << x << endl;
      cout << "y = " << y << endl;
}
```

Program 4.7 Output

```
Before GetMousePos(...)
x = 0
y = 0
After GetMousePos(...)
x = 597
y = 353
Press any key to continue
```

From the output we verify that `GetMousePos` did indeed modify both `x` and `y`, thereby "returning" more than one value. How does it work? When we call a function with parameters, C++ does its normal processing; that is, it copies the argument value to the parameter. But in this case, the argument value is a memory address (`&x` and `&y`)—so it copies the address of `x` and the address of `y` into `outX` and `outY`, respectively. This means that the parameters `outX` and `outY` now point to the variables `x` and `y`. From what we studied previously, given a pointer to a variable, we can access that variable. Thus, we can modify `x` and `y` from inside the function. Figure 4.8 shows the relationship between `x` and `y`, and `outX` and `outY`, visually.

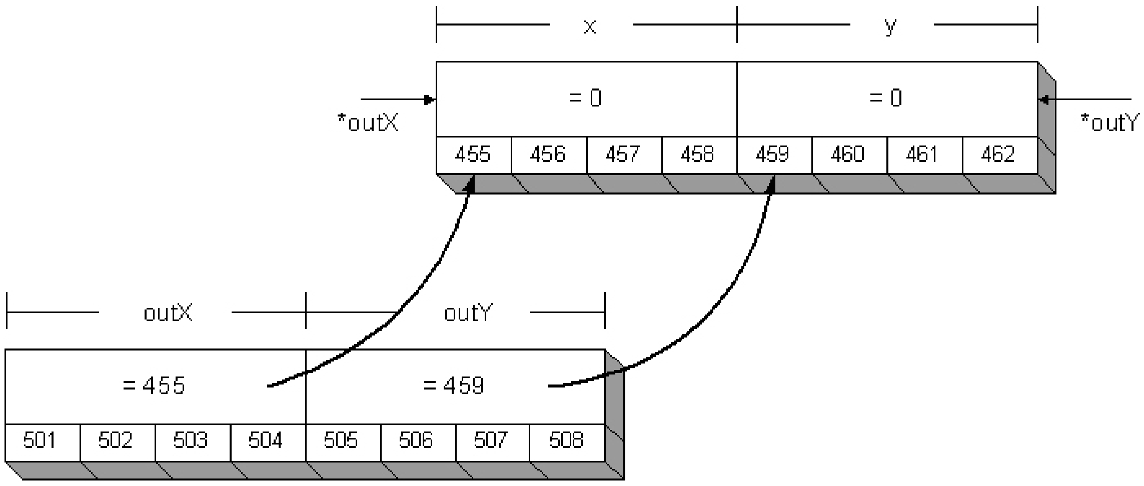


Figure 4.8: Observe that the parameters `outX` and `outY` point to the variables `x` and `y`, respectively.

In this way, we can modify multiple "outside" variables from "inside" a function, thereby "returning" multiple return values through these pointer parameters. The ability to return multiple values through pointer parameters is the second advantage of pointers.

4.4.2 Returning Multiple Return Values with References

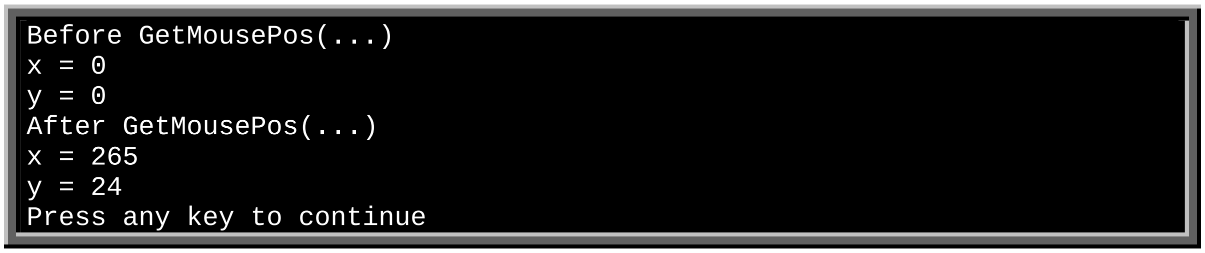
We can also return multiple return values with references. This section follows the same line or reasoning as the previous section, with the distinction being that it replaces pointer syntax with reference syntax.

Suppose that we require a function called `GetMousePos`, which needs to return the x- and y-coordinates of the mouse position relative to the screen. Such a function is useful when you need a game (or any program) to react to mouse input. The following program illustrates how to implement this function so that it can return two parameters. (Note that, for illustration purposes, we return random numbers for the mouse's x- and y-coordinates since we do not know yet how to actually get the current mouse position.)

Program 4.8: Returning multiple return values with references.

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

void GetMousePos(int& outX, int& outY)
{
      // Pretend to return the mouse's current position.
      outX = rand() % 801;
      outY = rand() % 601;
}

int main()
{
      // Seed the random number generator.
      srand( time(0) );

      // Initialize two variables that will receive the
      // mouse position.
      int x = 0;
      int y = 0;

      // Output before x and y before receiving mouse position.
      cout << "Before GetMousePos(...)" << endl;
      cout << "x = " << x << endl;
      cout << "y = " << y << endl;

      GetMousePos( x, y );

      cout << "After GetMousePos(...)" << endl;
      cout << "x = " << x << endl;
      cout << "y = " << y << endl;
}
```

Program 4.8 Output



```
Before GetMousePos(...)
x = 0
y = 0
After GetMousePos(...)
x = 265
y = 24
Press any key to continue
```

From the output, we verify that `GetMousePos` did indeed modify both `x` and `y`, thereby "returning" more than one value. So how does it work? When we call a function with parameters, C++ does its normal processing; that is, it copies the argument value to the parameter. But in this case, the parameter is a reference, so the parameters become references to the arguments. Thus the parameters, `outX` and `outY`, now *refer* to the variables `x` and `y`. From what we studied previously, given a pointer to a variable we can access that variable. Thus, we can modify `x` and `y` from inside the function.

4.5 Dynamic Memory

One of the drawbacks of the arrays covered thus far is that their size is fixed and their size must be specified in advance (i.e., at compile time). For instance, this is not legal:

```
	int n = 0;
	cout << "Enter an array size: ";
	cin >> n;

	float array[n];
```

We cannot create a variable-sized array like this. This presents a problem because it is not hard to imagine a case where the number of array items will vary. For example, in a massive multiplayer online game, you may use an array to store all of the game players. Because players are constantly entering and leaving online play, the array may need to resize. One possibility is to define an array with "maximum size" that can handle a maximum amount of players. However, it is unlikely that the array will be filled to the maximum value at all times, and therefore, memory will be wasted.

To remedy the problem of fixed-size arrays, C++ provides the concept of **dynamic memory**. The word "dynamic" is used in the sense that we can create and destroy memory at **runtime** (while the program is running). It is used in contrast to **static memory,** which is memory fixed at compile time.

How do we incorporate dynamic memory into our applications? First of all, we must use pointers. This is because the C++ operator which we will use to allocate additional memory returns a pointer to that memory. Therefore, we are forced into using pointers with dynamic memory. Dynamic memory is a third important function of pointers.

4.5.1 Allocating Memory

To create memory dynamically we use the `new` operator. The `new` operator returns a pointer to the newly allocated memory. Here are some examples:

```
// Allocate memory.
bool*  dynBool  = new bool;
int*   dynInt   = new int;
float* dynFloat = new float;

char* dynCharArray     = new char[80];     // dynamic array
long* dynLongArray     = new long[256];    // dynamic array
double* dynDoubleArray = new double[5000]; // dynamic array

// Initialize values.
*dynBool  = true;
*dynInt   = 5;
*dynFloat = 6.28f;
```

Note that when using the `new` operator to allocate arrays, the operator returns a pointer to the first element in the array. Therefore, accessing the elements of dynamic arrays is done in the same way as with regular arrays—using the bracket operator [] or pointer arithmetic.

Additionally, the non-array values can be initialized during allocation using parentheses syntax:

```
// Allocation and value initialization.
bool*  dynBool  = new bool(true);
int*   dynInt   = new int(5);
float* dynFloat = new float(6.28f);
```

The key benefit of dynamic memory is that variable amounts of memory can be allocated at runtime. For example, we can now rewrite the earlier problem with a code snippet like so:

```
int n = 0;
cout << "Enter an array size: ";
cin >> n;

float* array = new float[n];
```

Using dynamic memory, this *is* legal. Thus, we can allocate (and destroy) memory to meet the current needs of the program at runtime. Moreover with dynamic memory, we can allocate memory only if it is needed. For example, we can do the following:

```
if( memoryNeeded )
      dynMemory = new float[5000];
```

In this way, the array of 5000 `floats` will only be allocated if the program actually needs it. Clearly dynamic memory gives us a great deal of control over memory and this turns out to be one of the more powerful features of pointers (and C++ in general).

4.5.2 Deleting Memory

One of the key differences between static and dynamic memory is that the creation and deletion of static memory is handled automatically, whereas the programmer must create and delete dynamic memory manually.

To delete memory that was allocated with `new`, the `delete` operator must be invoked on the pointer to the memory. Here are some examples to match the previous section's `new` operations:

```
delete dynBool;
dynBool = 0;
delete dynInt;
dynInt = 0;
delete dynFloat;
dynFloat = 0;

delete[] dynCharArray;
dynCharArray = 0;
delete[] dynLongArray;
dynLongArray = 0;
delete[] dynDoubleArray;
dynDoubleArray = 0;
```

Observe that when you delete an array, you need to use a special delete operator; the `delete[]` operator. Also note that the pointers are set to null after deletion. This is good programming practice because it ensures that the pointer does not point to any memory after it is deleted. Consequently, programs are easier to debug because it will be clear if the program is trying to use a null pointer. Conversely, if the pointer was deleted but not nullified, then it would be harder to determine if the pointer was actually valid or not. By nullifying the pointer after its deletion, we explicitly state that it is now an invalid (null) pointer, and there is no ambiguity about its validity.

Rule: For every `new` operation invoked, a corresponding `delete` operator must be invoked when you are done with the memory. If deleting an array, be sure to use the `delete[]` syntax.

4.5.3 Memory Leaks

Memory leaks refer to the case where a pointer to memory which was allocated with `new` is lost before that memory was deleted. The following snippet illustrates a common example:

```
void MemLeaker()
{
      float* arrayOfFloats = new float[100];

      cout << "MemLeaker() called!" << endl;
}
```

```
int main()
{
      for(int i = 0; i < 500; ++i)
            MemLeaker();
}
```

Here an array of 100 floats is allocated in the function `MemLeaker`. When the function returns, all local variables are deleted automatically, which includes the pointer `arrayOfFloats`. Remember that a pointer is a variable too, it just points to other memory. What this amounts to is that we have lost our pointer to the dynamic memory we have allocated. Because the corresponding `delete` operator for that dynamic memory was never called, the memory will not be deleted—thereby causing a memory leak. Moreover, since the pointer to that memory was lost, it is impossible to ever delete it since we have no idea where that memory lives anymore—access to it was lost when the pointer (i.e. the address) was lost.

The memory loss problem is compounded by the fact that `MemLeaker` is called hundreds of times inside `main`. Because `MemLeaker` leaks a certain amount of memory each time it is called, if this function is called often enough, it is possible that it will drain all of the available memory, which will be problematic as your machine may eventually run out of available memory in which to perform operations or allocate new variables. It is imperative to always delete any memory which you have allocated with new when you are done using that memory. For reference, the function should be written like this:

```
void MemLeaker()
{
      float* arrayOfFloats = new float[100];

      cout << "MemLeaker() called!" << endl;

      delete[] arrayOfFloats;
      arrayOfFloats = 0;
}
```

4.5.4 Sample Program

The following sample program is the largest we have seen so far in this course. Each part has been commented with explanations, so be sure to read the comments carefully. The program allows the user to resize an array at runtime and set the values of the various elements in the array via a menu interface. In this way we can see dynamic memory in use. Memory will be allocated and destroyed while the program is running.

Program 4.9: Dynamic memory.

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

```
//================================================================
// Desc:  Function iterates through each element in the given
//        array.
// array - pointer to the first element of an integer array.
// size  - the number of elements in the array.
//================================================================
void PrintArray(int* array, int size)
{
      // If the array is of size zero than call it a null array.
      if( size == 0 )
      {
            cout << "NULL Array" << endl;
      }
      else
      {
            // size not zero so loop through each element and
            // print it.
            cout << "{";
            for(int i = 0; i < size; ++i)
                  cout << array[i] << " ";

            cout << "}" << endl;
      }
}

//================================================================
// Desc:  Function returns a new array given the new size.
// array   - pointer to the first element of an integer array.
// oldSize - the number of elements currently in 'array'.
// newSize - the number of elements we want in the new array.
//================================================================
int* ResizeArray(int* array, int oldSize, int newSize)
{
      // Create an array with the new size.
      int* newArray = new int[newSize];

      // New array is a greater size than old array.
      if( newSize >= oldSize )
      {
            // Copy old elements to new array.
            for(int i = 0; i < oldSize; ++i)
                  newArray[i] = array[i];
      }
      // New array is a lesser size than old array.
      else // newSize < oldSize
      {
            // Copy as many old elements to new array as can fit.
            for(int i = 0; i < newSize; ++i)
                  newArray[i] = array[i];
      }

      // Delete the old array.
      delete[] array;

      // Return a pointer to the new array.
      return newArray;
}
```

```
int main()
{
      // Our main array pointer and a variable to keep track
      // of the array size.
      int* array     = 0;
      int  arraySize = 0;

      // Boolean variable to let us know when the user wants
      // to quit, so that we can terminate the loop.
      bool done = false;
      while( !done )
      {
            // Every loop cycle print the array.
            PrintArray(array, arraySize);

            // Print a menu giving the user a list of options.
            cout <<
                  "1) Set Element "
                  "2) Resize Array "
                  "3) Quit ";

            // Input the users selection.
            int selection = 1;
            cin >> selection;

            // Some variables that will receive additional input
            // depending on the users selection.
            int index   = -1;
            int value   = 0;
            int newSize = 0;

            // Find out what menu option the user selected.
            switch( selection )
            {
            // Case 1: Set Element
            case 1:
                  // Ask for the index of the element the user wants
                  // to set.
                  cout << "Index = ";
                  cin >> index;

                  // Make sure index is "in array bounds."
                  if( index < 0 || index >= arraySize )
                  {
                        cout << "Bad Index!" << endl;
                  }
                  else
                  {
                        // Ask the user to input the value the user
                        // wants to assign to element 'index'.
                        cout << "[" << index << "] = ";
                        cin >> value;

                        // Set the value the user entered to the index
                        // the user specified.
                        array[index] = value;
                  }
                  break;
            // Case 2: Resize Array
            case 2:
                  // Ask the user to enter the size of the new array.
                  cout << "Size = ";
                  cin >> newSize;

                  // Call the resize function.  Recall that this
                  // function returns a pointer to the newly resized
                  // array.
                  array = ResizeArray(array, arraySize, newSize);

                  // Update the array size.
                  arraySize = newSize;
                  break;
            // Quit...
            default:
                  // Cause the loop to terminate.
                  done = true;
                  break;
            }
      }

      delete [] array;
      array = 0;
}
```

Program 4.9 Output

```
NULL Array
1) Set Element 2) Resize Array 3) Quit 2
Size = 4
{-842150451 -842150451 -842150451 -842150451 }
1) Set Element 2) Resize Array 3) Quit 1
Index = 3
[3] = 4
{-842150451 -842150451 -842150451 4 }
1) Set Element 2) Resize Array 3) Quit 1
Index = 2
[2] = 3
{-842150451 -842150451 3 4 }
1) Set Element 2) Resize Array 3) Quit 1
Index = 0
[0] = 1
{1 -842150451 3 4 }
1) Set Element 2) Resize Array 3) Quit 1
Index = 1
[1] = 2
{1 2 3 4 }
1) Set Element 2) Resize Array 3) Quit 2
Size = 7
{1 2 3 4 -842150451 -842150451 -842150451 }
1) Set Element 2) Resize Array 3) Quit 1
```

```
Index = 4
[4] = 5
{1 2 3 4 5 -842150451 -842150451 }
1) Set Element 2) Resize Array 3) Quit 2
Size = 5
{1 2 3 4 5 }
1) Set Element 2) Resize Array 3) Quit 3
Press any key to continue
```

Note: Notice the trick in this code:

```
cout <<
                  "1) Set Element "
                  "2) Resize Array "
                  "3) Quit ";
```

"Adjacent" strings like this will be put into one string. That is, the above is equal to:

```
cout << "1) Set Element 2) Resize Array 3) Quit ";
```

In this way, we can break long strings up over several lines to make the code more readable.

4.6 std::vector

Although dynamic memory is a powerful tool, the risk of memory leaks requires extreme caution. In fact, some other programming languages, such as Java, have deemed pointers to be too dangerous and error prone, and therefore they do not expose pointers to the programmer. Although not having pointers and low-level memory access can make programming simpler, the loss of this memory control can lead to inefficiencies.

The key theme of Program 4.9 was resizing an array, and it turns out that this is a common operation in non-trivial programs. In order to resize the array, dynamic memory was used. It would be worthwhile if there were a way to "wrap up" the code that resizes an array into a package of code. Once this resizing code was verified to work and that it contained no memory leaks, this code "package" could be used throughout our programs with the confidence that all the memory management was being done correctly behind the scenes. Fortunately for us, such a package exists and is part of the standard library.

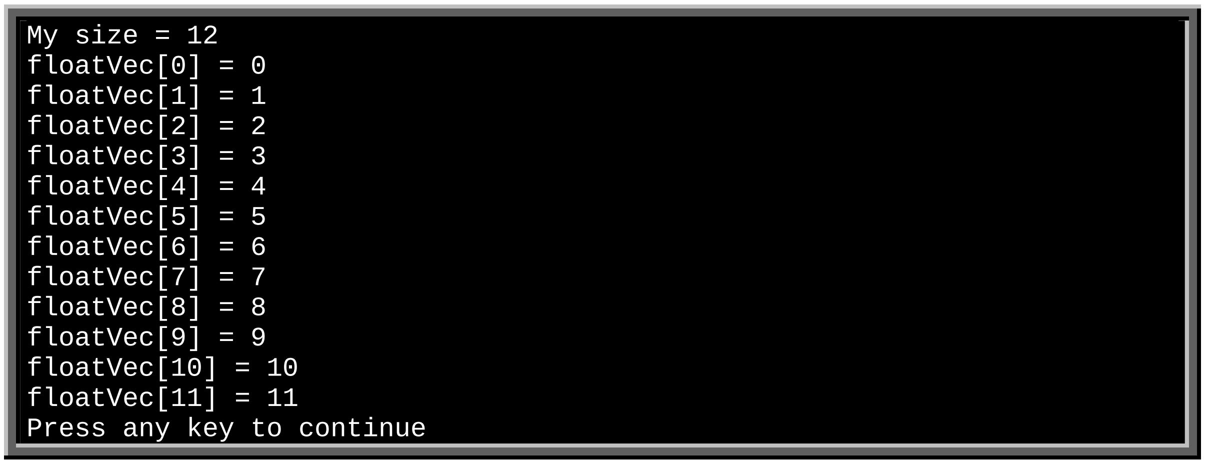
The code package is a special type called `std::vector` (include `<vector>`). A vector in this context is simply an array which can dynamically resize itself. The following program shows a simple example:

Program 4.10: Using std::vector as a resizable array.

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

```
int main()
{
      vector<float> floatVec;

      floatVec.resize(12);

      cout << "My size = " << floatVec.size() << endl;

      for(int i = 0; i < 12; ++i)
            floatVec[i] = i;

      for(int i = 0; i < 12; ++i)
            cout << "floatVec[" << i << "] = " << floatVec[i] << endl;

}
```

Program 4.10 Output



```
My size = 12
floatVec[0] = 0
floatVec[1] = 1
floatVec[2] = 2
floatVec[3] = 3
floatVec[4] = 4
floatVec[5] = 5
floatVec[6] = 6
floatVec[7] = 7
floatVec[8] = 8
floatVec[9] = 9
floatVec[10] = 10
floatVec[11] = 11
Press any key to continue
```

Program 4.10 demonstrates some syntax not yet discussed. First a variable is declared called `floatVec` of type `vector`, and inside the angle brackets <> a type is specified, which indicates what type of elements the vector stores. We store `float`s in the vector.

In the next statement, the vector variable name is followed by a dot and something called `resize(12)`. This syntax calls an operator, called `resize`, particular to `vector` that instructs the vector to resize itself to size 12. Also observe that a `vector` keeps track of its size—its size can be accessed using its `.size` operator.

Finally, the elements in `floatVec` can be accessed in the same way elements in an array are accessed, by using the bracket operator. Note that `vector` has many more operators than `resize`, but they will be discussed later in this course. For now, just think of `vector` as a resizable array.

Program 4.9 is now rewritten using `vector`. Note that by using `vector`, less code needs to be written. More importantly, observe that memory management does not need to be done as all the memory management is “wrapped up” inside `vector`.

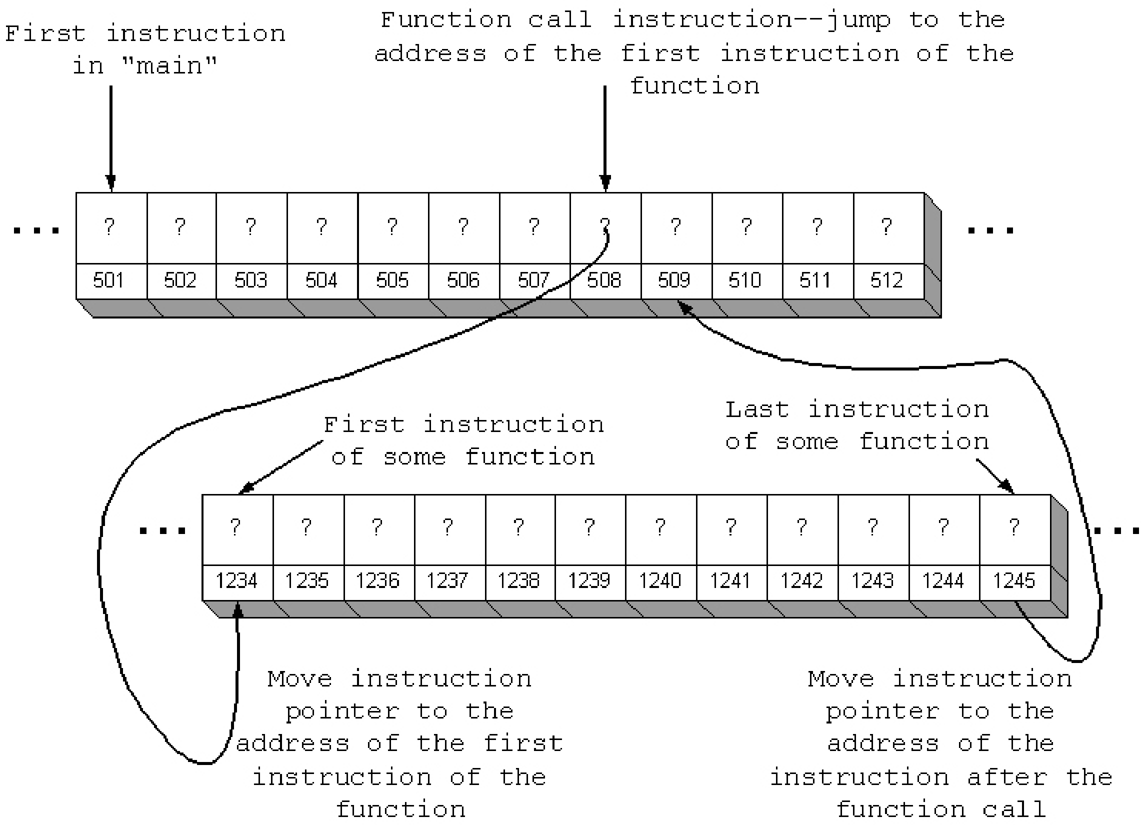
Program 4.11: Dynamic memory "wrapped up" in std::vector.

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

void PrintVector(vector<int>& v)
{
      if( v.size() == 0 )
      {
            cout << "NULL Array" << endl;
      }
      else
      {
            cout << "{";
            for(int i = 0; i < v.size(); ++i)
                  cout << v[i] << " ";

            cout << "}" << endl;
      }
}

int main()
{
      vector<int> array;

      // Boolean variable to let us know when the user wants
      // to quit, so that we can terminate the loop.
      bool done = false;
      while( !done )
      {
            // Every loop cycle print the array.
            PrintVector(array);

            // Print a menu giving the user a list of options.
            cout <<
                  "1) Set Element "
                  "2) Resize Array "
                  "3) Quit ";

            // Input the users selection.
            int selection = 1;
            cin >> selection;

            // Some variables that will receive additional input
            // depending on the users selection.
            int index   = -1;
            int value   = 0;
            int newSize = 0;

            // Find out what menu option the user selected.
            switch( selection )
            {
            // Case 1: Set Element
            case 1:
                  // Ask for the index of the element the user wants
                  // to set.
                  cout << "Index = ";
                  cin >> index;

                  // Make sure index is "in array bounds."
                  if( index < 0 || index >= array.size() )
                  {
                        cout << "Bad Index!" << endl;
                  }
                  else
                  {
                        // Ask the user to input the value the user
                        // wants to assign to element 'index'.
                        cout << "[" << index << "] = ";
                        cin >> value;

                        // Set the value the user entered to the index
                        // the user specified.
                        array[index] = value;
                  }
                  break;
            // Case 2: Resize Array
            case 2:
                  // Ask the user to enter the size of the new array.
                  cout << "Size = ";
                  cin >> newSize;

                  // Call the resize operator.  Recall that this
                  // function returns a pointer to the newly resized
                  // array.
                  array.resize(newSize);
                  break;
            // Quit...
            default:
                  // Cause the loop to terminate.
                  done = true;
                  break;
            }
      }
}
```

4.7 Function Pointers

Variables are not the only types of objects that live in computer memory. A program's instructions (machine code) are also loaded into memory. This is necessary because these instructions will need to be loaded in and out of the CPU's **instruction register** (a register that stores the current instruction being executed), in order for the computer to execute the instructions. What happens when a function is called? Ignoring arguments and parameters for the moment, to start executing the code of a function, the execution flow needs to jump from the current execution path to the beginning of the execution path of the function we wish to call. Since a function has a memory address, this is not a difficult task.

Simply set the **instruction pointer** (a CPU register that points to the memory address of the next machine instruction to execute) to point to the address of the first instruction of the function. Figure 4.9 shows an example of this concept.



First instruction in "main"
Function call instruction--jump to the address of the first instruction of the function
? ? ? ? ? ? ? ? ? ? ? ?
501 502 503 504 505 506 507 508 509 510 511 512
First instruction of some function
Last instruction of some function
1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245
Move instruction pointer to the address of the first instruction of the function
Move instruction pointer to the address of the instruction after the function call

Figure 4.9: CPU instructions in memory. The question marks simply indicate that we do not know what the actual bits of these instructions look like, but we assume they are instructions as marked in the figure. We see that a few instructions, after the first instruction of main, we come to a function call instruction, which modifies the instruction pointer to point to the first instruction of some function also in memory. The flow of execution thus flows into this function, and the function code is executed. The last instruction of the function modifies the instruction pointer again, this time setting it back to the instruction that followed the original function call instruction. Thus we observe the flow of execution into a function and then back out of a function.

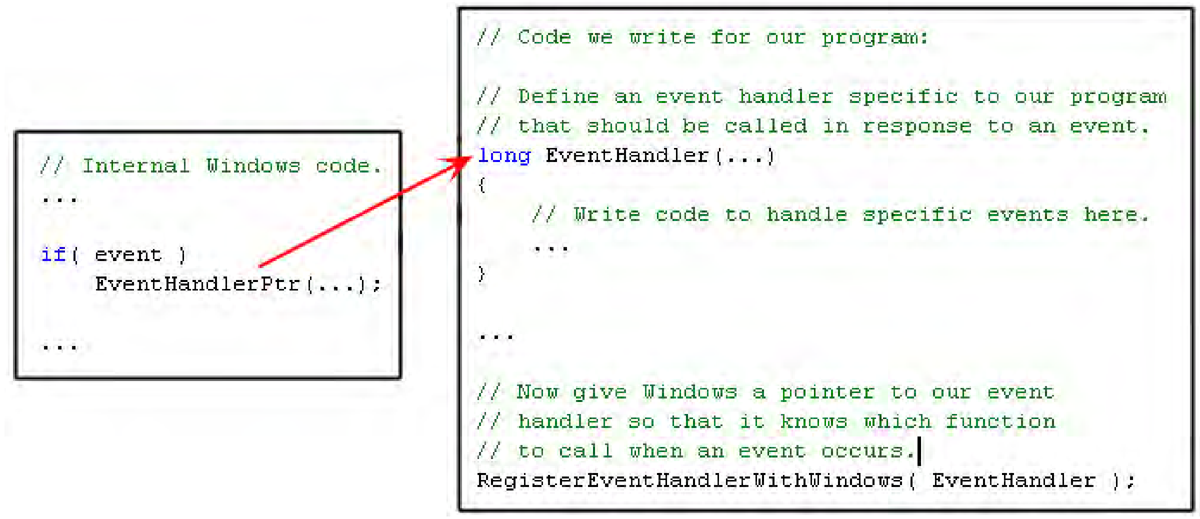
The preceding paragraph gave a simplified explanation of what occurs "behind the scenes." However, a more elaborate explanation is best left to a course on computer architecture and machine language. The key point to remember is that a function lives in memory and thus has a memory address. Therefore, we can have pointers to functions.

4.7.1 The Uses of Function Pointers

Like regular pointers, it may not be obvious why we need an additional level of indirection instead of calling the function directly, so let us look at an example.

Later in this course, we will introduce Windows programming; that is, creating programs with menus, dialog boxes, etc—no more console window! One of the things that we will learn is that Windows programming is fundamentally different from how we currently write our programs. In particular, Windows programs are **event driven.** A Windows program constantly scans for events such as mouse clicks, key presses, menu selections, and so on. When an event occurs, the program needs to respond to it. Each different Windows program generally responds differently to a specific event. This is one of the features that makes the programs different from each other. For example, a game responds to keyboard input much differently than a word processor does.

Windows (the operating system) is constantly checking for events. When an event occurs, Windows needs to call a function to handle the event, called an **event handler.** An event handler is a function that contains the code that should be executed in response to an event. Because each program generally handles an event differently, the event handler will generally be different for each program. Therefore, each Windows program defines its own event handler function, and then registers a pointer to this function with Windows. Consequently, Windows can call this event handler function, via the function pointer, when an event occurs. Figure 4.10 demonstrates this concept.



```
// Internal Windows code.
...

if( event )
    EventHandlerPtr(...);

...
```

```
// Code we write for our program:

// Define an event handler specific to our program
// that should be called in response to an event.
long EventHandler(...)
{
    // Write code to handle specific events here.
    ...
}

...

// Now give Windows a pointer to our event
// handler so that it knows which function
// to call when an event occurs.
RegisterEventHandlerWithWindows( EventHandler );
```

Figure 4.10: Defining an event handler and registering a pointer to it with Windows. In this way, the internal Windows code can call this event handling function, via the pointer, when an event occurs.

4.7.2 Function Pointer Syntax

To declare a function pointer variable, the following syntax is used:

```
returnType (*PointerName)(paramType paramName, ...)
```

The return type and parameter listing of the function pointer must match that of the function whose address is being assigned to the function pointer. Consider this example:

```
float Square(float x)
{
    return x * x;
}

int main()
{
      // Get a pointer to Square.  Note address of operator (&)
      // is not necessary for function pointers.
      float (*squarePtr)(float x) = Square;

      // Call Square via pointer:

      cout << "squarePtr(2.0f) = " << squarePtr(2.0f) << endl;
}
```

Note that a function pointer can be invoked without dereferencing.

4.8 Summary

1. A reference is essentially an alias for a variable. Given a reference *R* to a variable *A*, we can directly access *A* with *R* since *R refers* to *A*. Using references we can, for example, return multiple return values from a function.

2. A pointer is a special variable type that can store the memory address of another variable. Given a pointer to a variable, the actual variable to which it points can be accessed and modified by dereferencing the pointer. By using pointers multiple return values can be returned from a function, arrays can be efficiently passed to functions, and dynamic memory can be used.

3. In C++, the array name can be converted to a pointer to the first element in the array. Given a pointer to its first element, an array can be navigated by using pointer arithmetic.

4. Dynamic memory allows the creation and destruction of memory at runtime (while the program is running) in order to meet the current needs of the program. To allocate memory, use the C++ `new` operator and to destroy it, use either the `delete` operator for non-array pointers or the `delete[]` operator for array pointers. Remember that to avoid memory leaks, every `new` operation should eventually have a corresponding `delete/delete[]` operation.

5. If a resizable array is required, use `std::vector` instead of dynamic memory. `std::vector` handles the dynamic memory for you, thus preventing accidental memory leaks. Moreover, by using `std::vector,` less code is required and the program becomes easier to manage and maintain.

6. Function pointers are useful when a third party section of code needs to call a section of your code. For example, Windows may need to call your event handler function.

4.9 Exercises

4.9.1 Essay Questions

1. Explain in your own words and using complete sentences what the following terms are:
    a) References:
    b) Constant References:
    c) Pointers:
    d) Constant Pointers:
    e) Pointer arithmetic:
    f) Static Memory:
    g) Dynamic Memory
    h) Runtime
2. State three benefits of pointers.
3. What is the symbol for the “address of” operator and what does it do?
4. What is the symbol of the “indirection operator” and when is it used?
5. How are references probably implemented “behind the scenes?”
6. Why is dynamic memory useful?
7. Explain how arrays are passed into functions.
8. What is a safer (i.e., avoids memory leaks) and easier alternative to dynamic memory?
9. Explain a situation where function pointers might be useful.

4.9.2 Dice Function

Write a dice rolling function; that is, a function that returns two random numbers, both in the range [1, 6]. Implement the function two times: once using references, and a second time using pointers. Your function declarations should like this:

```
void Dice(int& die1, int& die2);
void Dice(int* die1, int* die2);
```

After you have implemented and tested this function, write a small craps-like gambling game that allows the user to place bets on the dice roll outcomes. You are free to make up your own game rules.

4.9.3 Array Fill

Write a function called `RandomArrayFill` that inputs (i.e., takes a parameter) an integer array, and also that inputs (i.e., takes a parameter) an integer, which contains the size of the array. The function is then to write a random number, in the range [0, 100], to each element of the array. The function declaration of this function should look like so:

```
void RandomArrayFill(int* array, int size);
```

Now write a program that asks the user to input the size of an integer array. The program then needs to create an array of exactly this size. Next, the program must pass this created array to the `RandomArrayFill` function, so that a random number is assigned to each element of the array. After which, the program must output every array element to the console window. Your program output should look similar to this:

```
Enter the size of an array to create: 6
Creating array and filling it with random numbers...
Array = {57, 23, 34, 66, 2, 96}
Press any key to continue
```

After you finish implementing the above function, rewrite it again, but this time using `std::vector`. The function declaration of this new function should look like so:

```
void RandomArrayFill(std::vector& vec);
```

4.9.4 Quadratic Equation

Background Info

Recall that the standard form of a quadratic equation is given by:

$$y = ax^2 + bx + c\ .$$

Here, *a*, *b*, and *c* are called the coefficients of the quadratic equation. Geometrically, this describes a parabola—Figure 4.11. Often we want to find the values of *x* where the parabola intersects the *x*-axis; that is, find *x* such that $y = ax^2 + bx + c = 0$. These values of *x* are called the **roots** of the quadratic equation.

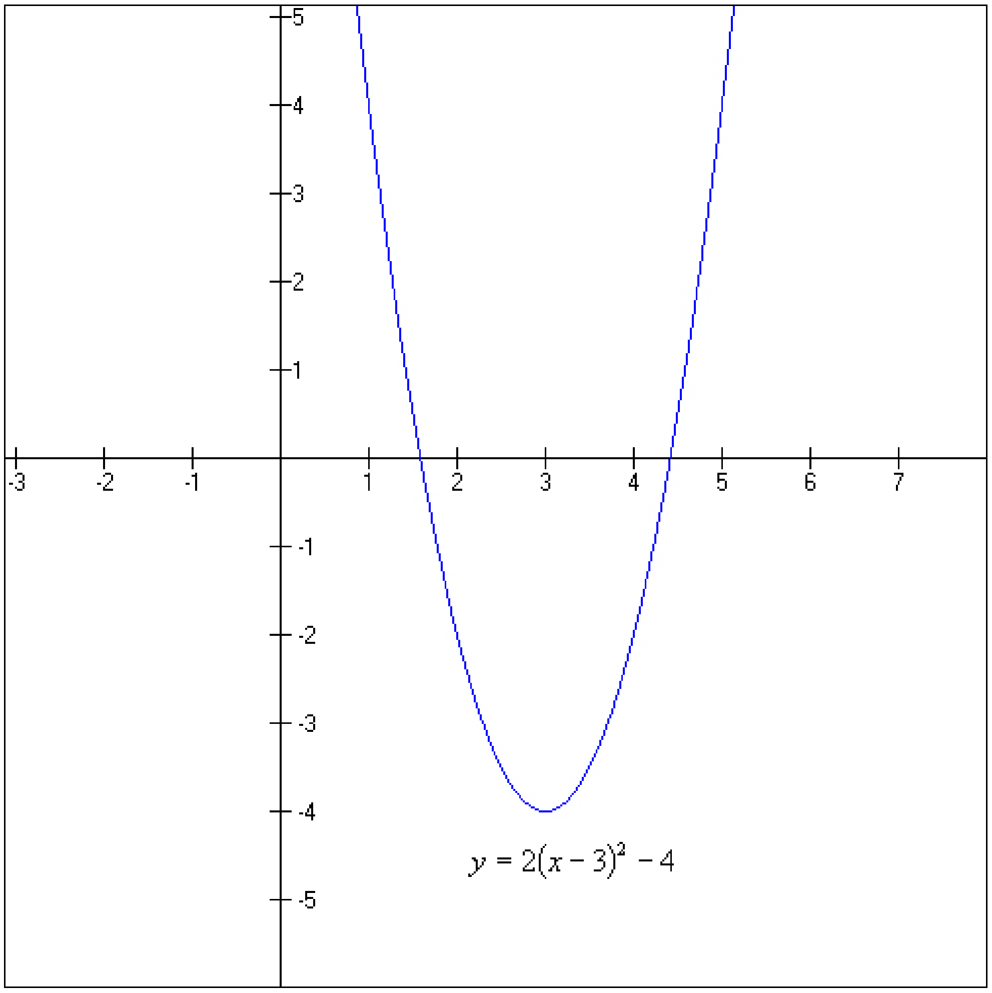


Figure 4.11: Parabola $y = 2(x-3)^2 - 4$ with a 2-unit scale, a horizontal translation of 3 units, and a vertical translation of –4 units. The roots are approximately 1.58 and 4.41.

Conveniently, a mechanical formula exists that allows us to find the roots of a quadratic equation. This formula, called the **quadratic formula**, is as follows:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

To be sure that you understand how this equation works we will do some examples.

Example 1: Find the roots of the following quadratic equation: $x^2 - x - 6$.

Using the quadratic formula we have:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{1 \pm \sqrt{1 - 4(1)(-6)}}{2 \cdot 1} = \frac{1 \pm \sqrt{25}}{2} = \frac{1 \pm 5}{2}$$

So, $x_1 = 3$ and $x_2 = -2$.

Example 2: Find the roots of the following quadratic equation: $x^2 - 2x + 1$.

Using the quadratic formula we have:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{2 \pm \sqrt{4 - 4(1)(1)}}{2 \cdot 1} = \frac{2 \pm 0}{2} = 1$$

So, $x_1 = x_2 = 1$.

Example 3: Find the roots of the following quadratic equation: $x^2 + 2x + 5$.

Using the quadratic formula we have:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-2 \pm \sqrt{4 - 4(1)(5)}}{2 \cdot 1} = \frac{-2 \pm \sqrt{-16}}{2}$$

Recalling that $i^2 = -1$ we obtain:

$$\frac{-2 \pm \sqrt{-16}}{2} = \frac{-2 \pm \sqrt{16i^2}}{2} = \frac{-2 \pm 4i}{2} = -1 \pm 2i$$

So, $x_1 = -1 + 2i$ and $x_2 = -1 - 2i$.

Observe in example three that the roots to the quadratic equation can contain imaginary numbers; a number that includes both a real and an imaginary component is termed a **complex number.**

Exercise

Write a function that inputs (i.e., takes as parameters) the coefficients of a quadratic equation, and outputs the result. The function should return two solutions with two parts: 1) A real part and 2) an imaginary part. Of course, if a solution does not have an imaginary part (or real part, for that matter) then the corresponding component will just be zero (e.g., $0+2i$, $3+0i$). The function should be prototyped as follows:

```
bool QuadraticFormula(float a, float b, float c,
           float& r1, float& i1, float& r2, float& i2);
```

Where a, b, and c are the coefficients of the quadratic equation. And where `r1` denotes the real part of solution 1 and where `i1` denotes the imaginary part of solution 1. Likewise, `r2` denotes the real part of solution 2 and `i2` denotes the imaginary part of solution 2.

Note that the return type is a `bool`. The return value is determined as follows: If the function contains an imaginary part then return `false`, otherwise return `true`. We do this because some applications might not want to work with non-real results (i.e., results that have imaginary parts), and such application can easily test for a non-real result by examining the return value of this function.

Test your function with the coefficients given in the quadratic equations from the above three examples. Your output should be formatted similar to this:

```
Coefficients a=1, b=2, c=5 yield S1 = -1 + 2i, and S2 = -1 – 2i
```

Chapter 5

Classes and Object Oriented Programming

Introduction

Thus far we have been using intrinsic C++ variable types such as `bool`, `char`, `int`, `float`, arrays, etc, and the standard library `std::string` type, to represent simple quantities. These types work well to describe trivial things such as names and numbers. However, many real-world objects which need to be described in code are built as an aggregate set of various intrinsic types (e.g., a game player has many statistical properties such as a name, health, armor, etc). Moreover, many real-world objects can perform actions. A game player can run and fire his/her weapon, for example. The primary theme of this chapter is to learn how to create complex types called classes, which will enable the creation of variables which consist of several components and can perform actions.

Chapter Objectives

- Understand the problems object oriented programming attempts to solve.
- Define a class and instantiate members of that class.
- Learn some basic class design strategies.

5.1 Object Oriented Programming Concepts

Many real-world objects are described by a multitude of data members. For example, a player in a game might have several properties:

```
std::string name;
int hitPoints;
int magicPoints;
float positionX;
float positionY;
std::string weaponName;
int weaponDamage;
int armor;
```

Because there may be many players in a game, we would like to logically group the variables associated with each player via one name. In addition to an object's properties, represented by variables, objects can also perform certain actions. These actions are usually specific to a particular type of object. For example, a player in a role-playing game might be able to perform actions such as `fight`, `talk`, `castSpell`, etc. These verbs can be represented in code with functions; for example, a function called `fight` would execute the code necessary to perform a combat simulation. Moreover, these functions would have access to the player's data properties so that, for example, hit points could be decreased whn the player was injured, magic points increased when a new level is attained, and so on.

Real-world objects consist of data (properties) and perform actions (functions). The key idea of object-oriented programming is to model this observation in code so that new variable types can be defined which can be used to instantiate variables that behave like real-world objects. These new variable types are called **classes.** The motivation for this style of programming is four-fold.

First, it is intuitive. It is much more natural for humans to think in terms of real objects than it is to think in terms of computer instructions and memory. This is especially true in large and complex systems.

Second, object-oriented programming provides a natural organizational system. When creating these classes, all the related code associated with the class is organized in a tight code unit. Instead of having a lot of data variables scattered throughout the code and manipulating that data through functions, we obtain a much more self-contained code unit of related code through classes.

Third, data can be encapsulated with object-oriented programming. The class writer can enforce which parts of the class are visible to outside code, and which parts should be kept purely internal. In this way, the class writer prevents users of the class from accidentally making destructive modifications to internal class data.

Finally, object-oriented programming enables a higher level of code reuse and generalizations through facilities like inheritance, polymorphism, and templates. These topics are covered towards the end of this book.

5.2 Classes

A **class** allows us to define a new variable type. For example, we can define a class called `Wizard`. We can then instantiate `Wizard` **objects** (i.e., **instances**) in the same way we instantiate variables of the intrinsic types. For example, we will be able to write:

```
Wizard wiz0; //Instantiate a variable called wiz0 of type Wizard.
```

Note that the definition of a class is different than an instance of the class. The class definition states to the compiler what properties an object of the class has, but it does not create an object. In the above example, `Wizard` is the class and `wiz0` is the object (i.e., an instance of class `Wizard`).

A conceptual analogy which is often used to explain the difference between a class and an object is a blueprint. A house blueprint, for example, specifies how a house would be made, but it is *not* a house itself. The actual houses built based on that blueprint would be called objects (i.e., instances) of that blueprint. Similarly, a class specifies how a variable would be made, but it is *not* a variable itself. The actual variables built based on that class would be called objects (i.e., instances) of that class.

5.2.1 Syntax

In general, a class is defined with the following syntax:

```
class ClassName
{
	// Methods

	// Data members
};
```

For example, we might define our aforementioned `Wizard` class like so:

```
class Wizard
{
public:
	// Methods
	void fight();
	void talk();
	void castSpell();

	// Data members
	std::string mName;
	int mHitPoints;
	int mMagicPoints;
	int mArmor;
};
```

Methods (also called **member functions**) are the class functions, which specify the actions which objects of this class can perform. **Data members** are the variable components that together form a more complex type, such as a `Wizard`. In the above `Wizard` class, we define a `Wizard` to have a name, hit points, magic points, and armor. Thus we have built a complex `Wizard` type out of simpler components.

> **Note:** We prefix the data members of a class with 'm' (for member) so that we can readily distinguish them from non-class member variables; this is purely for notation and not required.

After our class has been defined, the class methods still need to be implemented. Method implementations (i.e., definitions) occur outside the class definition scope and follow this general syntax:

```
returnType ClassName::MethodName(ParameterList...)
{
	...Body Code
}
```

To illustrate a simple example, we implement the functions of `Wizard`, like so:

```
void Wizard::fight()
{
	cout << "Fighting." << endl;
}

void Wizard::talk()
```

```
{
      cout << "Talking." << endl;
}

void Wizard::castSpell()
{
      cout << "Casting Spell." << endl;
}
```

The only syntactic difference between a method definition and a “regular” function definition is that the method name must be prefixed with the class name followed by the **scope resolution operator** (`::`). The scope resolution operator is used in the same sense in which it was used with namespaces. Recall that the `std::` prefix told the compiler that the standard library code we need belongs to the standard namespace. Similarly with classes, we use the scope resolution operator to specify the class to which the method belongs.

> **Note:** It is possible to define methods inside the class instead of outside. For example, we could implement the `Wizard` methods like so:

```
class Wizard
{
public:
      // Methods
      void fight()
      {
           cout << "Fighting." << endl;
      }
      void talk()
      {
           cout << "Talking." << endl;
      }
      void castSpell()
      {
           cout << "Casting Spell." << endl;
      }

      // Data members
      std::string mName;
      int mHitPoints;
      int mMagicPoints;
      int mArmor;
};
```

> However, for reasons Section 5.2.3 will give, this is usually considered bad form.

5.2.2 Class Access: The Dot Operator

In Section 5.2.1 we defined and implemented a `Wizard` class. This class is now ready to be used. Let us first instantiate an object:

```
Wizard wiz0;
```

Now that we have a `Wizard` object created, we want it to perform actions. So let us call some of its methods. To invoke a class method the **dot operator** (.) is used as follows:

```
wiz0.fight();
wiz0.talk();
```

Note that this is the same dot operator that was used to resize a vector. This is because `std::vector` is actually a class and `resize` is a method of that class, which executes the necessary code to resize the vector.

Since methods are associated with a particular object—that is, we invoke them through an object with the dot operator—we may ask whether a method can access the data members of the calling object. For example, when `wiz0.fight()` is called, and a fight simulation begins, can `fight()` access the armor property `mArmor` or the `mHitPoints` properties of `wiz0`? It would seem reasonable that it could, since `fight()` is as much a part of the object as the data members are. Indeed, a member function can access the data members of the calling object. We will see many examples of how this is done throughout this chapter, and we will see how it actually works in the next chapter.

Suppose now a data member of `wiz0` needs to be read or modified. To access a data member, the dot operator is also used, as this next snippet shows:

```
wiz0.mArmor = 10;

cout << "Player's name = " << wiz0.mName << endl;

// Test to see if player has enough magic points to cast a spell
if( wiz0.mMagicPoints > 4 )
      wiz0.castSpell();
else
      cout << "Not enough magic points!" << endl;
```

Note that each object has its "own" data members. For example, if we have several `Wizard` objects `wiz0`, `wiz1`,…, they each have their own name, number of hit points, and magic points. In other words, the data members of objects are completely independent from each other.

It may be helpful to think of the dot operator as a subscripting symbol. In an array we identify different elements that belong to that array using the bracket operator ([]). With objects, we identify different methods and data members of that object with the dot operator, followed by the method or data member name.

Note: We can create pointers to class types in the same way that we create pointers to intrinsic types. What if we have a pointer to an object? For example, suppose we create a wizard instance like so:

```
Wizard* wiz0 = new Wizard;
```

To invoke a method or access a data member we must first dereference the pointer to get the variable, and then we can use the dot operator as normal:

```
(*wiz0).fight();
(*wiz0).mMagicPoints = 5;
```

We use the parentheses to specify that the dereference operation must come before the dot operation.

However, this syntax is considered cumbersome, so a shorthand syntax was developed. When using a pointer to an object, we can invoke a member function or access a data member using the **indirect membership operator (->):**

```
wiz0->fight();
wiz0->mMagicPoints = 5;
```

5.2.3 Header Files; Class Definitions; Class Implementations

One of the goals of C++ is to separate class definitions from implementation. The motivation behind this is that a programmer should be able to use a class without knowing exactly how it works (i.e., without knowing the implementation details). This occurs when you may be using classes you did not write yourself—for example, when you learn DirectX, you will use classes that Microsoft wrote.

In order to separate class definitions from implementation, two separate files are used: **header files** (.h) and **implementation files** (.cpp). Header files include the class definition, and implementation files contain the class implementation (i.e., method definitions). Eventually, the source code file is compiled to either an object file (.obj) or a library file (.lib). Once that is done, you can distribute the class header file along with the object file or library file to other programmers. With the header file containing the class definition, and with the object or library file containing the compiled class implementation, which is linked into the project with the linker, the programmer has all the necessary components to use the class without ever having to see the implementation file (that is, the .cpp file). For example, the DirectX Software Development Kit includes only the DirectX header files and the DirectX library files—you never see the implementation files (.cpp).

In summary, given the header file and object file or library file, other programmers are able to use your class without ever seeing how it was implemented in the source code file. This does two things: first, others will not be able to modify the implementation, and second, your implementation (intellectual property) is protected.

To illustrate, let us rewrite the `Wizard` class and its implementation, but this time using header files and source code files.

```
// Wiz.h (Wizard header file.)

#ifndef WIZARD_H
#define WIZARD_H
```

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

class Wizard
{
public:
	// Methods
	void fight();
	void talk();
	void castSpell();

	// Data members
	std::string mName;
	int mHitPoints;
	int mMagicPoints;
	int mArmor;
};

#endif // WIZARD_H
```

```
// Wiz.cpp (Wizard implementation file.)

#include "wiz.h"
using namespace std;

void Wizard::fight()
{
	cout << "Fighting." << endl;
}

void Wizard::talk()
{
	cout << "Talking." << endl;
}

void Wizard::castSpell()
{
	cout << "Casting Spell." << endl;
}
```

```
// Main.cpp (The file with main.)

#include <iostream>
#include <string>
#include "wiz.h" //<--Include so we can declare Wizard objects.
using namespace std;

int main()
{
	Wizard wiz0; // Declare a variable called wiz0 of type Wizard.

	wiz0.fight();
	wiz0.talk();

	wiz0.mArmor = 10;
	cout << "Player's name = " << wiz0.mName << endl;

	// Test to see if player has enough magic points to cast a spell
	if( wiz0.mMagicPoints > 4 )
		wiz0.castSpell();
	else
		cout << "Not enough magic points!" << endl;
}
```

Throughout the rest of this course we will separate our class definition from its implementation into two separate files.

> **Note:** A segment of code that uses a class is called a **client** of the class. For example, *Main.cpp* is a client file of `Wizard` because it uses objects of that class.

> **Note:** When including header files in the project which you have written, such as *Wiz.h*, use quotation marks in the include directive instead of the angle brackets (e.g., "Wiz.h")

5.2.2.1 Inclusion Guards

Observe that in *Wiz.h* our class is surrounded with the following:

```
#ifndef WIZARD_H
#define WIZARD_H
…
#endif // WIZARD_H
```

This is called an **inclusion guard.** These statements are called **preprocessor directives** and they direct the compiler on how to compile the code.

The first line, `#ifndef WIZARD_H`, has the compiler ask if a symbol called `WIZARD_H` has *not* been already defined. If it has not been defined then the code between the `#ifndef` and the `#endif` will be compiled. Here the code contained between this "compiler if statement" is the class and another preprocessor directive `#define WIZARD_H`. The directive `#define WIZARD_H` tells the compiler to define a symbol called `WIZARD_H`. Thus, the first time the compiler sees *Wiz.h* in a translation unit, `WIZARD_H` is not defined, so the class will be compiled, and `WIZARD_H` will also be defined. Any other time the compiler sees *Wiz.h* in a translation unit, `WIZARD_H` *will already be defined* and thus the class will not be recompiled again. This prevents a redefinition of the class, which is desirable because it only needs to be compiled once per translation unit.

But why would the same class be included more than once in a translation unit? Consider the `Wizard` class file breakdown *Wiz.h*, *Wiz.cpp*, and *Main.cpp*. We include *iostream* and *string* in *Main.cpp*, but *Main.cpp* also includes *Wiz.h*, which in turn includes *iostream* and *string* as well. Thus *iostream* and *string* would be included twice in the translation unit *Main.cpp*, which would cause a class redefinition error. However, *iostream* and *string* contain the above mentioned inclusion guards, thereby preventing them from being compiled more than once, and thus preventing a redefinition error. This same scenario can occur with our own classes as well, so we too need inclusion guards.

5.2.4 Data Hiding: Private versus Public

C++ class writers can enforce which parts of the class are visible to outside code, and which parts should be kept purely internal. To do this, C++ provides two keywords: `public` and `private`. All code in a class definition following a `public` keyword and up to a `private` keyword is considered public, and all code in a class definition following a `private` keyword up to a `public` keyword is considered private. Note that classes are private by default.

> **Note:** The `public` and `private` keywords are used to enforce which parts of the class are visible to *outside* code. A class's own member functions can access all parts of the class since they are *inside* the class, so to speak.

The public portion of a class is referred to as the **class-interface** because that is the interface exposed to other code units. The private portion of a class is considered to be part of the class implementation, because only the internal implementation has access to it.

The general rule is that data members (class variables) should be kept internal (private) and only methods should be exposed publicly. By making the programmer go through member functions, the class can safely regulate how the internal class data can be accessed and manipulated. Hence, the class can maintain the integrity of its internal data. Why does the integrity need to be maintained? Because the class methods most likely have some expectations about the data, and by going through member functions it can enforce these expectations.

Using this general rule, our Wizard class can be rewritten like so:

```
class Wizard
{
public:
	// Methods
	void fight();
	void talk();
	void castSpell();

private:
	// Data members
	std::string mName;
	int mHitPoints;
	int mMagicPoints;
	int mArmor;
};
```

However, now the data members cannot be accessed and the code fails to compile:

```
wiz0.mArmor = 10;
```

```
	cout << "Player's name = " << wiz0.mName << endl;

	// Test to see if player has enough magic points to cast a spell
	if( wiz0.mMagicPoints > 4 )
		wiz0.castSpell();
	else
		cout << "Not enough magic points!" << endl;
```

The errors generated are:
“error C2248: 'Wizard::mArmor' : cannot access private member declared in class 'Wizard',”
“error C2248: 'Wizard::mName' : cannot access private member declared in class 'Wizard',”
“error C2248: 'Wizard::mMagicPoints' : cannot access private member declared in class 'Wizard'.”

This should not be surprising as these variables were just made private and cannot be accessed directly.

To solve this problem, a similar, but safer, functionality must be provided via the class interface. First, we will add a `Wizard::setArmor` method, which allows a client to set the armor. It is implemented like so:

```
	void Wizard::setArmor(int armor)
	{
		if( armor >= 0 )
			mArmor = armor;
	}
```

Here the member function sets the data member `mArmor` (remember that we said member functions can access the class data). Thus if we write:

```
wiz0.setArmor(10);
```

We are setting the `mArmor` member of `wiz0` to ten.

Why is `Wizard::setArmor` “better” than making `mArmor` public? By forcing the client to go through `Wizard::setArmor`, we can add our own safety checks to maintain data integrity. For example, in our `Wizard::setArmor` implementation, we added a check to make sure `armor` was nonnegative.

Next, a way for the client to get the name of a `Wizard` without giving direct access to the name must be provided. This is done by making a function that returns a *copy* of the name. Thus the client cannot modify the internal name, but only a copy. This function is implemented like so:

```
	std::string Wizard::getName()
	{
		return mName;
	}
```

Finally, the `mMagicPoints` data member must be considered. In actuality, the client should not be getting access to this value. The class itself can test whether or not the wizard has enough magic points internally. We rewrite `Wizard::castSpell` like this:

```
void Wizard::CastSpell()
{
	if( mMagicPoints > 4)
		cout << "Casting Spell." << endl;
	else
		cout << "Not enough magic points!" << endl;
}
```

The client code can now be rewritten with our new interface as follows:

```
int main()
{
	Wizard wiz0;

	wiz0.Fight();
	wiz0.Talk();

	wiz0.setArmor(10);

	cout << "Player's name = " << wiz0.getName() << endl;

	wiz0.CastSpell();
}
```

Note: In addition to defining a class, you can define what is called a **structure**. In C++, a structure is exactly the same as a class except that it is public by default, whereas a class is private by default. For example, we can write a Point2D structure like so:

```
struct Point2D
{
	float x;
	float y;
};
```

We do not need to explicitly designate it as public because structures are public by default. Structures can have member functions as well, but in practice, structures are usually used for types that only have data members. Because structures are essentially the same as classes, we use the two terms interchangeably.

5.2.5 Constructors and Destructors

Every class has a **constructor** and **destructor**, which are special kinds of methods. If you do not explicitly define these methods, the compiler will generate default versions automatically. In short, a constructor is a method that is automatically executed when an object is instantiated and a destructor is a method that is automatically executed when an object is deleted.
A constructor is usually used to initialize data members to some default value or to allocate any dynamic memory the class uses or to execute any initialization code that you want executed as the object is being created. Conversely, the destructor is usually used to free any dynamic memory which the class has allocated because if it does not delete it when the object is being destroyed, it will result in a memory

leak. Note that you never invoke a destructor yourself; rather, the destructor will automatically be called when an object is being deleted from memory.

Constructors and destructors are special methods and they require a specific syntax. In particular, a constructor has no return type and its name is also the name of the class. Likewise, a destructor has no return type, no parameters, and its name is the name of the class but prefixed with the tilde (~). This next snippet shows how the constructor and destructor would be declared in the `Wizard` class definition:

```
class Wizard
{
public:
      // Constructor.
      Wizard();

      // Overloaded constructor.
      Wizard(std::string name, int hp, int mp, int armor);

      // Destructor
      ~Wizard();
...
```

The implementations of these function is done just as any other method, except there is no return type. The following snippet gives a sample implementation:

```
Wizard::Wizard()
{
      // Client called constructor with zero parameters,
      // so construct a "wizard" with default values.
      // We call this a “default” constructor.
      mName        = "DefaultName";
      mHitPoints   = 0;
      mMagicPoints = 0;
      mArmor       = 0;
}

Wizard::Wizard(std::string name, int hp, int mp, int armor)
{
      // Client called constructor with parameters, so
      // construct a "wizard" with the specified values.
      mName        = name;
      mHitPoints   = hp;
      mMagicPoints = mp;
      mArmor       = armor;
}

Wizard::~Wizard()
{
      // No dynamic memory to delete--nothing to cleanup.
}
```

Note: Observe that we have overloaded the constructor function. Recall that the act of defining several different versions—which differ in signature—of a function is called function overloading. We can overload methods in the same way we overload functions.

Constructors are called when an object is created. Thus instead of writing:

```
Wizard wiz0;
```

We now write:

```
Wizard wiz0();// Use “default” constructor.
```

or:

```
Wizard wiz0(“Gandalf”, 20, 100, 5);// Use constructor with
                                   // parameters.
```

Note that the following are actually equivalent; that is, they both use the default constructor:

```
Wizard wiz0;  // Use “default” constructor.
Wizard wiz0();// Use “default” constructor.
```

5.2.6 Copy Constructors and the Assignment Operator

Every class also has a **copy constructor** and an **assignment operator.** If you do not explicitly define these methods, the compiler will generate default ones automatically. A copy constructor is a method that constructs an object via another object of the same type. For example, we should be able to construct a new `Wizard` object from another `Wizard` object—somewhat like a copy:

```
Wizard wiz0;
...
Wizard wiz1(wiz0);// Construct wiz1 from wiz0.
```

If you use the default copy constructor, the object will be constructed by copying the parameter’s bytes, byte-by-byte, into the object being constructed, thereby performing a basic copy. However, there are times when this default behavior is undesirable and you need to implement your own copy constructor code.

Similarly, an assignment operator is a method that specifies how an object can be assigned to another object. For example, how should a `Wizard` object be assigned to another `Wizard` object?

```
Wizard wiz0;
...
Wizard wiz1 = wiz0;// Assign wiz0 to wiz1.
```

If you use the default assignment operator, a simple byte-by-byte copy from the right hand operand’s memory into the left hand operand’s memory will take place, thereby performing a basic copy. However, there are times when this default behavior is undesirable and you need to override it with your own assignment code.

We discuss the details of cases where you would need to implement your own copy constructor and assignment operator in Chapter 7. For now, just be aware that these methods exist.

5.3 RPG Game: Class Examples

To help reinforce the concepts of classes, we will create several classes over the following subsections. We will then use these classes to make a small text-based role-playing game (RPG).

When utilizing the object oriented programming paradigm, the first thing we ask when designing a new program is: "What objects does the program attempt to model?" The answer to this question depends on the program. For example, a paint program might utilize objects such as `Brushes`, `Canvases`, `Pens`, `Lines`, `Circles`, `Curves`, and so on. In the case of our RPG, we require various types of weapon objects, monster objects, player objects, and map objects.

After we have decided on what objects our program will use, we need to design corresponding classes which define the properties of these kinds of objects and the actions they perform. For example, what aggregate set of data members represents a `Player` in the game? What kind of actions can a `Player` perform in the game? In addition to the data and methods of a class, the class design will also need to consider the relationships between the objects of one class and the objects of other classes—for example, how they will interact with each other. The following subsections provide examples for how these class design questions can be answered.

5.3.1 The Range Structure

Our game will rely on "dice rolls" as is common in many role-playing games. We implement random dice rolls with a random number generator. To facilitate random number generation, let us define a range structure, which can be used to define a range in between which we compute random numbers:

```
// Range.h

#ifndef RANGE_H
#define RANGE_H

// Defines a range [mLow, mHigh].
struct Range
{
      int mLow;
      int mHigh;
};

#endif //RANGE_H
```

This class is simple, and it contains zero methods. The data members are the interface to this class. Therefore, there is no reason to make the data private and so we leave them public. (Note as well that we actually used the struct type rather than the class type as discussed in section 5.2.4).

5.3.2 Random Functions

Our game will require a couple of utility functions as well. These functions do not belong to an object, per se, so we will implement them using "regular" functions.

```
// Random.h

#ifndef RANDOM_H
#define RANDOM_H

#include "Range.h"

int Random(Range r);

int Random(int a, int b);

#endif // RANDOM_H
```

```
// Random.cpp

#include "Random.h"
#include <cstdlib>

// Returns a random number in r.
int Random(Range r)
{
	return r.mLow + rand() % ((r.mHigh + 1) - r.mLow);
}

// Returns a random number in [low, high].
int Random(int low, int high)
{
	return low + rand() % ((high + 1) - low);
}
```

- `Random`: This function returns a random number in the specified range. We overload this function to work with the `Range` structure, and also to work with two integer parameters that specify a range. Section 3.4.1 describes how this calculation works.

5.3.3 Weapon Class

Typically in an RPG game there will be many different kinds of weapons players can utilize. Therefore, it makes sense to define a `Weapon` class, from which different kinds of weapon objects can be instantiated. In our RPG game the `Weapon` class is defined like so:

```
// Weapon.h

#ifndef WEAPON_H
#define WEAPON_H

#include "Range.h"
#include <string>

struct Weapon
{
      std::string mName;
      Range       mDamageRange;
};

#endif //WEAPON_H
```

An object of class `Weapon` has a name (i.e., the name of the weapon), and a damage range, which specifies the range of damage the weapon inflicts against an enemy. To describe the damage range, we use our `Range` class. Again, note that this class has no methods because it performs no actions. You might argue that a weapon *attacks* things, but instead of this approach, we decide that game characters (players and monsters) *attack* things and weapons do not; this is simply a design decision. Because there are no methods, there is no reason to protect the data integrity. In fact, the data members are the class interface.

The following code snippet gives some examples of how we might use this class to instantiate different kinds of weapons:

```
      Weapon dagger;
      dagger.mName = "Dagger";
      dagger.mDamageRange.mLow  = 1;
      dagger.mDamageRange.mHigh = 4;

      Weapon sword;
      sword.mName = "Sword";
      sword.mDamageRange.mLow  = 2;
      sword.mDamageRange.mHigh = 6;
```

5.3.4 Monster Class

In addition to weapons, an RPG game will have many different types of monsters. Thus, it makes sense to define a `Monster` class from which different kinds of monster objects can be instantiated. In our RPG game the `Monster` class is defined like so:

```
// Monster.h

#ifndef MONSTER_H
#define MONSTER_H

#include "Weapon.h"
#include <string>

class Player;

class Monster
{
public:
	Monster(const std::string& name, int hp, int acc,
		int xpReward, int armor, const std::string& weaponName,
		int lowDamage, int highDamage);

	bool isDead();

	int         getXPReward();
	std::string getName();
	int         getArmor();

	void attack(Player& player);
	void takeDamage(int damage);
	void displayHitPoints();

private:
	std::string mName;
	int         mHitPoints;
	int         mAccuracy;
	int         mExpReward;
	int         mArmor;
	Weapon      mWeapon;
};

#endif //MONSTER_H
```

The first thing of interest is the first line after the include directives; specifically, the statement `class Player;`. What does this do? This is called a **forward class declaration,** and it is needed in order to use the `Player` class without having yet defined it. The idea is similar to function declarations, where a function is declared first, in order that it can be used, and then defined later.

Monster Class Data:

- `mName`: The name of the monster. For example, we would name an Orc monster "Orc."

- `mHitPoints`: An integer that describes the number of hit points the monster has.

- `mAccuracy`: An integer value used to determine the probability of a monster hitting or missing a game player.

- `mExpReward`: An integer value that describes how many experience points the player receives upon defeating this monster.

- `mArmor`: An integer value that describes the armor strength of the monster.

- `mWeapon`: The monster's weapon. A `Weapon` value describes the name of a weapon and its range of damage.

Note how objects of this class will contain a `Weapon` object, which in turn contains a `Range` object. We can observe this propagation of complexity as we build classes on top of other classes.

Monster Class Methods:

`Monster`:

The constructor simply takes a parameter list, which is used to initialize the data members of a `Monster` object at the time of construction. It is implemented like so:

```
Monster::Monster(const std::string& name, int hp, int acc,
            int xpReward, int armor, const std::string& weaponName,
            int lowDamage, int highDamage)
{
      mName      = name;
      mHitPoints = hp;
      mAccuracy  = acc;
      mExpReward = xpReward;
      mArmor     = armor;
      mWeapon.mName = weaponName;
      mWeapon.mDamageRange.mLow  = lowDamage;
      mWeapon.mDamageRange.mHigh = highDamage;
}
```

As you can see, this function copies the parameters to the data members, thereby initializing the data members. In this way, the property values of a monster object can be specified during construction.

`isDead`:

This simple method returns true if a monster is dead, otherwise it returns false. A monster is defined to be dead if its hit points are less than or equal to zero.

```
bool Monster::isDead()
{
      return mHitPoints <= 0;
}
```

This method is important because during combat, we will need to be able to test whether a monster has been killed.

getXPReward:

This method is a simple accessor method, which returns a copy of the mExpReward data member:

```
int Monster::getXPReward()
{
	return mExpReward;
}
```

getName:

This is another accessor method, which returns a copy of the mName data member:

```
std::string Monster::getName()
{
	return mName;
}
```

getArmor:

This is another accessor method; this one returns a copy of the mArmor data member:

```
int Monster::getArmor()
{
	return mArmor;
}
```

attack:

This method is the only nontrivial method of Monster. This method executes the code which has a monster attack a game Player (a class we will soon define). Because a monster attacks a Player, we pass a reference to a Player into the function. We pass by reference for efficiency; that is, just as we do not want to copy an entire array into a parameter, we do not want to copy an entire Player object. By passing a reference, we merely copy a reference variable (a 32-bit address).

The attack method is responsible for determining if the monster's attack hits or misses the player. We use the following criteria to determine whether a monster hits a player: If the monster's accuracy is greater than a random number in the range [0, 20] then the monster hits the player, else the monster misses the player:

```
	if( Random(0, 20) < mAccuracy )
```

If the monster hits the player, then the next step is to compute the damage the monster inflicts on the player. We start by computing a random number in the range of damage determined by the monster's weapon—mWeapon.mDamageRange:

```
	int damage = Random(mWeapon.mDamageRange);
```

However, armor must be brought into the equation. In particular, we say that armor absorbs some of the damage. Mathematically we describe this by subtracting the player's armor value from the random damage value:

```
	int totalDamage = damage - player.getArmor();
```

It is possible that `damage` is a low value in which case `totalDamage` might be less than or equal to zero. In this case, no damage is actually inflicted—we say that the attack failed to penetrate the armor. Conversely, if `totalDamage` is greater than zero then the player loses hit points. Here is the `attack` function in its entirety:

```
void Monster::attack(Player& player)
{
	cout << "A " << mName << " attacks you "
		<< "with a " << mWeapon.mName << endl;

	if( Random(0, 20) < mAccuracy )
	{
		int damage = Random(mWeapon.mDamageRange);

		int totalDamage = damage - player.getArmor();

		if( totalDamage <= 0 )
		{
			cout << "The monster's attack failed to "
				<< "penetrate your armor." << endl;
		}
		else
		{
			cout << "You are hit for " << totalDamage
				<< " damage!" << endl;

			player.takeDamage(totalDamage);
		}
	}
	else
	{
		cout << "The " << mName << " missed!" << endl;
	}
	cout << endl;
}
```

<u>`takeDamage`</u>:

This method is called when a player hits a monster. The parameter specifies the amount of damage for which the monster was hit, which indicates how many hit points should be subtracted from the monster:

```
void Monster::takeDamage(int damage)
{
	mHitPoints -= damage;
}
```

displayHitPoints:

This method outputs the monster's hit points to the console window. This is used in the game during battles so that the player can see how many hit points the monster has remaining.

```
void Monster::displayHitPoints()
{
      cout << mName << "'s hitpoints = " << mHitPoints << endl;
}
```

5.3.5 Player Class

The `Player` class describes a game character. In our game there is only one player object (single player), however you could extend the game to support multiple players, or let the user control a party of several characters. The `Player` class is defined as follows:

```
// Player.h

#ifndef PLAYER_H
#define PLAYER_H

#include "Weapon.h"
#include "Monster.h"
#include <string>

class Player
{
public:
      // Constructor.
      Player();

      // Methods
      bool isDead();

      std::string getName();
      int         getArmor();

      void takeDamage(int damage);

      void createClass();
      bool attack(Monster& monster);
      void levelUp();
      void rest();
      void viewStats();
      void victory(int xp);
      void gameover();
      void displayHitPoints();

private:
      // Data members.
      std::string mName;
      std::string mClassName;
      int         mAccuracy;
      int         mHitPoints;
      int         mMaxHitPoints;
      int         mExpPoints;
      int         mNextLevelExp;
      int         mLevel;
      int         mArmor;
      Weapon      mWeapon;
};

#endif //PLAYER_H
```

The `Player` class is similar in many ways to the `Monster` class, as can be seen by the similar data members and functions. However, there are some additional data and methods the `Player` class contains which the `Monster` class does not.

Player Class Data:

- `mName`: The name of the character the player controls. You name the character during character creation.
- `mClassName`: A string that denotes the player class type. For example, if you play as a wizard then your class name would be "Wizard."
- `mAccuracy`: An integer value used to determine the probability of a player hitting or missing a monster.
- `mHitPoints`: An integer that describes the current number of hit points the player has.
- `mMaxHitPoints`: An integer that describes the maximum number of hit points the player can currently have.
- `mExpPoints`: An integer that describes the number of experience points the player has currently earned.
- `mNextLevelExp`: An integer that describes the number of experience points the player needs to reach the next level. We define the amount of experience needed to reach the next level in terms of the player's current level; that is, `mNextLevelExp = mLevel * mLevel * 1000;`
- `mLevel`: An integer that describes the current level of the player.
- `mArmor`: An integer value that describes the armor strength of the player.
- `mWeapon`: The player's weapon. A `Weapon` value describes the name of a weapon and its range of damage.

Player Class Methods:

`Player`:

The constructor of our Player class is a default one—it simply initializes the data members to default values. This is not problematic because these values will be changed during character creation.

```
Player::Player()
{
      mName         = "Default";
      mClassName    = "Default";
      mAccuracy     = 0;
      mHitPoints    = 0;
      mMaxHitPoints = 0;
      mExpPoints    = 0;
      mNextLevelExp = 0;
      mLevel        = 0;
      mArmor        = 0;
      mWeapon.mName = "Default Weapon Name";
      mWeapon.mDamageRange.mLow  = 0;
      mWeapon.mDamageRange.mHigh = 0;
}
```

`isDead`:

This simple method returns true if a player is dead, otherwise it returns false. A player is defined to be dead if its hit points are less than or equal to zero.

```
bool Player::isDead()
{
      return mHitPoints <= 0;
}
```

This method is important because during combat, we will need to be able to test whether a player has been killed.

`getArmor`:

An accessor method; this one returns a copy of the `mArmor` data member:

```
int Player::getArmor()
{
      return mArmor;
}
```

`takeDamage`:

This method is called when a monster hits a player. The parameter specifies the amount of damage for which the player was hit, which indicates how many hit points should be subtracted from the player:

```
void Player::takeDamage(int damage)
{
      mHitPoints -= damage;
}
```

`createClass`:

This method is used to execute the code that performs the character generation process. First, it asks the user to enter in the name of the player. Next, it asks the user to select a character class. Then, based on the character class chosen, the properties of the Player object are filled out accordingly. For example, a "fighter" is given more hit points than a "wizard." Similarly, different classes start the game with different weapons.

```
void Player::createClass()
{
      cout << "CHARACTER CLASS GENERATION" << endl;
      cout << "==========================" << endl;

      // Input character's name.
      cout << "Enter your character's name: ";
      getline(cin, mName);

      // Character selection.
      cout << "Please select a character class number..."<< endl;
      cout << "1)Fighter 2)Wizard 3)Cleric 4)Thief : ";

      int characterNum = 1;
      cin >> characterNum;

      switch( characterNum )
      {
      case 1:  // Fighter
            mClassName    = "Fighter";
            mAccuracy     = 10;
            mHitPoints    = 20;
            mMaxHitPoints = 20;
            mExpPoints    = 0;
            mNextLevelExp = 1000;
            mLevel        = 1;
            mArmor        = 4;
            mWeapon.mName = "Long Sword";
            mWeapon.mDamageRange.mLow  = 1;
            mWeapon.mDamageRange.mHigh = 8;
            break;
      case 2:  // Wizard
            mClassName    = "Wizard";
            mAccuracy     = 5;
            mHitPoints    = 10;
            mMaxHitPoints = 10;
            mExpPoints    = 0;
            mNextLevelExp = 1000;
            mLevel        = 1;
            mArmor        = 1;
            mWeapon.mName = "Staff";
            mWeapon.mDamageRange.mLow  = 1;
            mWeapon.mDamageRange.mHigh = 4;
            break;
      case 3:  // Cleric
            mClassName    = "Cleric";
            mAccuracy     = 8;
            mHitPoints    = 15;
            mMaxHitPoints = 15;
            mExpPoints    = 0;
            mNextLevelExp = 1000;
            mLevel        = 1;
            mArmor        = 3;
            mWeapon.mName = "Flail";
            mWeapon.mDamageRange.mLow  = 1;
            mWeapon.mDamageRange.mHigh = 6;
            break;
      default: // Thief
            mClassName    = "Thief";
            mAccuracy     = 7;
            mHitPoints    = 12;
            mMaxHitPoints = 12;
            mExpPoints    = 0;
            mNextLevelExp = 1000;
            mLevel        = 1;
            mArmor        = 2;
            mWeapon.mName = "Short Sword";
            mWeapon.mDamageRange.mLow  = 1;
            mWeapon.mDamageRange.mHigh = 6;
            break;
      }
}
```

attack:

The attack method is essentially the same as `Monster::attack`. However, one important difference is that we give the player an option of what to do on his attack turn. For example, in our game, the player can choose to fight or run:

```
      int selection = 1;
      cout << "1) Attack, 2) Run: ";
      cin >> selection;
      cout << endl;
```

This can be extended to give the player more options such as using an item or casting a spell.

If the player chooses to attack, then the execute code is very similar to `Monster::attack`, except that the roles are reversed; that is, here a player attacks a monster, whereas in `Monster::attack`, a monster attacks a player.

```
switch( selection )
{
```

```
case 1:
      cout << "You attack an " << monster.getName()
            << " with a " << mWeapon.mName << endl;

      if( Random(0, 20) < mAccuracy )
      {
            int damage = Random(mWeapon.mDamageRange);

            int totalDamage = damage - monster.getArmor();

            if( totalDamage <= 0 )
            {
                  cout << "Your attack failed to penetrate "
                        << "the armor." << endl;
            }
            else
            {
                  cout << "You attack for " << totalDamage
                        << " damage!" << endl;

                  // Subtract from monster's hitpoints.
                  monster.takeDamage(totalDamage);
            }
      }
      else
      {
            cout << "You miss!" << endl;
      }
      cout << endl;
      break;
```

On the other hand, if the player chooses to run, then the code computes a random number, where there is a 25% chance that the player can escape.

```
case 2:
      // 25 % chance of being able to run.
      int roll = Random(1, 4);

      if( roll == 1 )
      {
            cout << "You run away!" << endl;
            return true;//<--Return out of the function.
      }
      else
      {
            cout << "You could not escape!" << endl;
            break;
      }
```

Observe that this function returns `true` if the player runs away, otherwise it returns `false`.

<u>`levelUp`</u>:

This method tests whether or not the player has acquired enough experience points to level up. It is called after every battle. If the player does have enough experience points then some of the player's statistics such as hit points and accuracy are randomly increased.

```
void Player::levelUp()
{
      if( mExpPoints >= mNextLevelExp )
      {
            cout << "You gained a level!" << endl;

            // Increment level.
            mLevel++;

            // Set experience points required for next level.
            mNextLevelExp = mLevel * mLevel * 1000;

            // Increase stats randomly.
            mAccuracy     += Random(1, 3);
            mMaxHitPoints += Random(2, 6);
            mArmor        += Random(1, 2);

            // Give player full hitpoints when they level up.
            mLevel = mMaxHitPoints;
      }
}
```

<u>`rest`</u>:

This method is called when the player chooses to rest. Currently, resting simply increases the player's hit points to the maximum. Later you may wish to add the possibility of random enemy encounters during resting or other events.

```
void Player::rest()
{
      cout << "Resting..." << endl;

      mHitPoints = mMaxHitPoints;
}
```

<u>`viewStats`</u>:

Often, a player in an RPG likes to view his player's statistics, so that he knows what items are in his inventory, how many hit points he has, or how many experience points are required to reach the next level. We output this type of information with the `viewStats` method:

```
void Player::viewStats()
{
      cout << "PLAYER STATS" << endl;
      cout << "============" << endl;
      cout << endl;

      cout << "Name           = " << mName         << endl;
      cout << "Class          = " << mClassName    << endl;
```

```
	cout << "Accuracy        = " << mAccuracy     << endl;
	cout << "Hitpoints       = " << mHitPoints    << endl;
	cout << "MaxHitpoints    = " << mMaxHitPoints << endl;
	cout << "XP              = " << mExpPoints    << endl;
	cout << "XP for Next Lvl = " << mNextLevelExp << endl;
	cout << "Level           = " << mLevel        << endl;
	cout << "Armor           = " << mArmor        << endl;
	cout << "Weapon Name     = " << mWeapon.mName << endl;
	cout << "Weapon Damage   = " << mWeapon.mDamageRange.mLow
		<< "-" << mWeapon.mDamageRange.mHigh << endl;

	cout << endl;
	cout << "END PLAYER STATS" << endl;
	cout << "================" << endl;
	cout << endl;
}
```

victory:

This method is called after a player is victorious in battle. It displays a victory message and gives the player an experience point award.

```
void Player::victory(int xp)
{
	cout << "You won the battle!" << endl;
	cout << "You win " << xp
		<< " experience points!" << endl << endl;

	mExpPoints += xp;
}
```

gameover:

This method is called if the player dies in battle. It displays a "game over" string and asks the user to press 'q' to quit:

```
void Player::gameover()
{
	cout << "You died in battle..." << endl;
	cout << endl;
	cout << "================================" << endl;
	cout << "GAME OVER!" << endl;
	cout << "================================" << endl;
	cout << "Press 'q' to quit: ";
	char q = 'q';
	cin >> q;
	cout << endl;
}
```

displayHitPoints:

This method simply outputs the player's hit points to the console window. This is used in the game during battles so that the player can see how many hit points he has left.

```
void Monster::displayHitPoints()
{
	cout << mName << "'s hitpoints = " << mHitPoints << endl;
}
```

5.3.6 Map Class

The final class we implement is called `Map`. An object of this class is used to represent the game board of either the game world, part of the game world, or a dungeon in the game world. In our small game we use a single `Map` object for our limited 2D game world. One responsibility of a `Map` object is to keep track of the player's world position; that is, its coordinates. In doing so, we make the `Map` class responsible for inputting the user's movement input. Additionally, since a `Map` should know where objects are on the map, it should know where the monsters are. Therefore, we also make the `Map` class responsible for handling enemy encounters.

A further extension to the `Map` class would be to define "landmarks" on it, or key areas where you want something special to occur. For example, perhaps at coordinates (2, 3) you want to place a dungeon, so that when the player moves to coordinates (2, 3), the game will describe the exterior of the dungeon and ask if the player wants to enter. A town would be another example.

Let us now look at the header file that contains the `Map` class:

```
// Map.h

#ifndef MAP_H
#define MAP_H

#include "Weapon.h"
#include "Monster.h"
#include <string>

class Map
{
public:

	// Constructor.
	Map();

	// Methods
	int  getPlayerXPos();
	int  getPlayerYPos();
	void movePlayer();
	Monster* checkRandomEncounter();
	void printPlayerPos();
private:
	// Data members.
	int mPlayerXPos;
	int mPlayerYPos;
};

#endif //MAP_H
```

Map Class Data:

- `mPlayerXPos`: The x-coordinate position of the player.
- `mPlayerYPos`: The y-coordinate position of the player.

Map Class Methods:

`Map`:

The constructor initializes the player's position coordinates to the origin; that is, the player starts off at the origin:

```
Map::Map()
{
	// Player starts at origin (0, 0)
	mPlayerXPos = 0;
	mPlayerYPos = 0;
}
```

`getPlayerXPos`:

This method is an accessor function that returns the current x-coordinate of the player.

```
int  Map::getPlayerXPos()
{
	return mPlayerXPos;
}
```

`getPlayerYPos`:

This method is an accessor function that returns the current y-coordinate of the player.

```
int  Map::getPlayerYPos()
{
	return mPlayerYPos;
}
```

`movePlayer`:

As stated, we make it a `Map`'s responsibility to keep track of the player's position. This function is called when the player wants to move. It prompts the user to enter in a direction of movement and then updates the player's coordinates accordingly:

```
void Map::movePlayer()
{
	int selection = 1;
	cout << "1) North, 2) East, 3) South, 4) West: ";
	cin >> selection;

	// Update coordinates based on selection.
	switch( selection )
	{
	case 1: // North
		mPlayerYPos++;
		break;
	case 2: // East
		mPlayerXPos++;
		break;
	case 3: // South
		mPlayerYPos--;
		break;
	default: // West
		mPlayerXPos--;
		break;
	}
	cout << endl;
}
```

<u>`checkRandomEncounter`</u>:

This function is the key function of the `Map` class. It generates a random number in the range [0, 20]. Depending upon which sub-range in which the generated number falls, a different encounter takes place:

- Range [0, 5] – The player encounters no enemy.
- Range [6, 10] – The player encounters an Orc.
- Range [11, 15] – The player encounters a Goblin.
- Range [15, 19] – The player encounters an Ogre.
- Range [20] – The player encounters an Orc Lord.

The bulk of this method code consists of testing in which range the random number falls and then creating the appropriate kind of monster.

```
Monster* Map::checkRandomEncounter()
{
	int roll = Random(0, 20);

	Monster* monster = 0;


	if( roll <= 5 )
	{
```

```
            // No encounter, return a null pointer.
            return 0;
      }
      else if(roll >= 6 && roll <= 10)
      {
            monster = new Monster("Orc", 10, 8, 200, 1,
                  "Short Sword", 2, 7);

            cout << "You encountered an Orc!" << endl;
            cout << "Prepare for battle!" << endl;
            cout << endl;
      }
      else if(roll >= 11 && roll <= 15)
      {
            monster = new Monster("Goblin", 6, 6, 100, 0,
                  "Dagger", 1, 5);

            cout << "You encountered a Goblin!" << endl;
            cout << "Prepare for battle!" << endl;
            cout << endl;
      }
      else if(roll >= 16 && roll <= 19)
      {
            monster = new Monster("Ogre", 20, 12, 500, 2,
                  "Club", 3, 8);

            cout << "You encountered an Ogre!" << endl;
            cout << "Prepare for battle!" << endl;
            cout << endl;
      }
      else if(roll == 20)
      {
            monster = new Monster("Orc Lord", 25, 15, 2000, 5,
                  "Two Handed Sword", 5, 20);

            cout << "You encountered an Orc Lord!!!" << endl;
            cout << "Prepare for battle!" << endl;
            cout << endl;
      }

      return monster;
}
```

Observe that the function returns a pointer to the encountered monster. We chose to use a pointer because with pointers we can return a null pointer. A null pointer is useful in the case in which the player encounters no enemy.

Also note that when we do create a Monster, dynamic memory must be used so that the system does not automatically destroy the memory when the function returns—remember, once we use dynamic memory it is our responsibility to destroy it. We make it the responsibility of the function caller, which receives the pointer, to delete it.

`printPlayerPos`:

When the player is at the main menu, we would like to display the player's current coordinate position on the map. This is what this function is used for.

```
void Map::printPlayerPos()
{
      cout << "Player Position = (" << mPlayerXPos << ", "
            << mPlayerYPos << ")" << endl << endl;
}
```

5.4 The Game

The classes that the objects of our game are members of have now been defined, and we are ready to instantiate these objects and put them to use. But, before looking at the game code, let us look at a sample output of the game so that we have an idea of the game flow.

> **Note:** You can download the executable and source code for this program from the *www.gameinstitute.com* website. You may want to run the program yourself a few times, in order to get familiar with its functionality before examining the code.

```
CHARACTER CLASS GENERATION
==========================
Enter your character's name: Frank
Please select a character class number...
1)Fighter 2)Wizard 3)Cleric 4)Thief : 1
Player Position = (0, 0)

1) Move, 2) Rest, 3) View Stats, 4) Quit: 1
1) North, 2) East, 3) South, 4) West: 1

Player Position = (0, 1)

1) Move, 2) Rest, 3) View Stats, 4) Quit: 1
1) North, 2) East, 3) South, 4) West: 2

You encountered an Ogre!
Prepare for battle!

Frank's hitpoints = 20
Ogre's hitpoints = 20

1) Attack, 2) Run: 1

You attack an Ogre with a Long Sword
You attack for 3 damage!

A Ogre attacks you with a Club
You are hit for 4 damage!

Frank's hitpoints = 16
Ogre's hitpoints = 17
```

```
1) Attack, 2) Run: 1

You attack an Ogre with a Long Sword
You attack for 6 damage!

A Ogre attacks you with a Club
The Ogre missed!

Frank's hitpoints = 16
Ogre's hitpoints = 11

1) Attack, 2) Run: 1

You attack an Ogre with a Long Sword
You miss!

A Ogre attacks you with a Club
You are hit for 3 damage!

Frank's hitpoints = 13
Ogre's hitpoints = 11

1) Attack, 2) Run: 1

You attack an Ogre with a Long Sword
You attack for 5 damage!

A Ogre attacks you with a Club
You are hit for 4 damage!

Frank's hitpoints = 9
Ogre's hitpoints = 6

1) Attack, 2) Run: 1

You attack an Ogre with a Long Sword
You miss!

A Ogre attacks you with a Club
You are hit for 3 damage!

Frank's hitpoints = 6
Ogre's hitpoints = 6

1) Attack, 2) Run: 1

You attack an Ogre with a Long Sword
You attack for 6 damage!

You won the battle!
You win 500 experience points!

Player Position = (1, 1)

1) Move, 2) Rest, 3) View Stats, 4) Quit: 3
```

```
PLAYER STATS
============

Name            = Frank
Class           = Fighter
Accuracy        = 10
Hitpoints       = 6
MaxHitpoints    = 20
XP              = 500
XP for Next Lvl = 1000
Level           = 1
Armor           = 4
Weapon Name     = Long Sword
Weapon Damage   = 1-8

END PLAYER STATS
================

Player Position = (1, 1)

1) Move, 2) Rest, 3) View Stats, 4) Quit: 2
Resting...
Player Position = (1, 1)

1) Move, 2) Rest, 3) View Stats, 4) Quit: 4
```

As the game output shows, the game first proceeds to create a game character. The core game then begins which allows the user to move about the map, fight random monsters, rest, view the player stats, or exit. This is one big loop which continues as long as the player does not die, or the player does not quit.

To create the main game logic, we add a **client file** called *game.cpp* to our project. A "client file" is a file which uses the classes we created in the previous section (e.g., `Weapon`, `Monster`, `Player`). Let us now look at the client file one segment at a time.

5.4.1 Segment 1

```
// game.cpp

#include "Map.h"
#include "Player.h"
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

int main()
{
      srand( time(0) );
      Map gameMap;
```

```
	Player mainPlayer;

	mainPlayer.createClass();
```

Code segment 1 is quite trivial. We include the necessary header files and begin the `main` function. The first thing we do is seed the random number generator (Section 3.4). Next, we instantiate a `Map` object called `gameMap` and we instantiate a `Player` object called `mainPlayer`. Finally, we execute the character class creation code by calling `createClass` for `mainPlayer`.

5.4.2 Segment 2

```
// Begin adventure.
bool done = false;
while( !done )
{
	// Each loop cycly we output the player position and
	// a selection menu.

	gameMap.printPlayerPos();

	int selection = 1;
	cout << "1) Move, 2) Rest, 3) View Stats, 4) Quit: ";
	cin >> selection;

	Monster* monster = 0;
	switch( selection )
	{
```

In segment 2, we begin the main "game loop." This is the loop which will continue to execute until either the player dies or the player quits the game. The key tasks the game loop performs are to display the player's position every loop cycle and to prompt the user to make a menu selection. We then execute different code paths depending on the chosen selection via a switch statement.

5.4.3 Segment 3

```
case 1:
	// Move the player.
	gameMap.movePlayer();

	// Check for a random encounter.  This function
	// returns a null pointer if no monsters are
	// encountered.
	monster = gameMap.checkRandomEncounter();

	// 'monster' not null, run combat simulation.

	if( monster != 0 )
	{
```

```
            // Loop until a 'break' statement.
            while( true )
            {
                  // Display hitpoints.
                  mainPlayer.displayHitPoints();
                  monster->displayHitPoints();
                  cout << endl;

                  // Player's turn to attack first.
                  bool runAway = mainPlayer.attack(*monster);

                  if( runAway )
                        break;

                  if( monster->isDead() )
                  {
                        mainPlayer.victory(monster->getXPReward());
                        mainPlayer.levelUp();
                        break;
                  }

                  monster->attack(mainPlayer);

                  if( mainPlayer.isDead() )
                  {
                        mainPlayer.gameover();
                        done = true;
                        break;
                  }
            }

            // The pointer to a monster returned from
            // checkRandomEncounter was allocated with
            // 'new', so we must delete it to avoid
            // memory leaks.
            delete monster;
            monster = 0;
      }

break;
```

The case where the player moves is the largest code unit. First we call `Map::movePlayer`, which prompts the user to enter the direction of movement. Next we check if the player encountered an enemy with the `Map::checkRandomEncounter` method. Recall that this function returns null if no monster was encountered. This fact allows us to use a simple if statement to determine whether or not a monster was encountered. If the pointer is not null, then a monster was encountered and we proceed to enter a "combat loop;" that is, a loop that continues until the player dies or runs away, or the monster dies.

Inside the combat loop we output the player's hit points and the monster's hit points for every loop cycle so that the game player can observe the progress of the battle. Afterwards, we have the player attack the monster:

```
      bool runAway = mainPlayer.attack(*monster);
```

Before doing anything else, we check to see whether the player ran away, and if so, we break out of the combat loop:

```
if( runAway )
      break;
```

If the player did not run, but rather attacked and killed the monster then we execute our victory message and test to see if the player leveled up:

```
      if( monster->isDead() )
      {
            mainPlayer.victory(monster->getXPReward());
            mainPlayer.levelUp();
            break;
      }
```

If the monster did not die from the previous attack then it is the monster's turn to attack the player:

```
      monster->attack(mainPlayer);
```

If the attack kills the player, we execute the "game over" message and exit the game loop:

```
      if( mainPlayer.isDead() )
      {
            mainPlayer.gameover();
            done = true;
            break;
      }
```

This cycle of turn-based attacking continues until the combat loop is terminated.

Finally, after the battle is over we must delete the pointer to dynamic memory which the `Map::checkRandomEncounter` method returned:

```
      delete monster;
      monster = 0;
```

5.4.4 Segment 4

```
      case 2:
         mainPlayer.rest();
         break;
      case 3:
         mainPlayer.viewStats();
         break;
      case 4:
         done = true;
         break;
      }// End Switch Statement
    }// End While Statement
 }// End main function.
```

The last several segments are trivial. If the player chose to rest then we call the `Player::rest` method. If the player chose to view the character statistics then we call the `Player::viewStats` method. If the player chose to exit then we assign `true` to `done`, which will terminate the game loop.

> **Note:** Hopefully, this program walkthrough has provided you with an idea of how to go about breaking up a program into objects. Try your best to understand how the entire program works together, because the exercises of this chapter will ask you to make some modifications to the program.

5.5 Summary

1. In the real world, objects typically have a multitude of properties that define them (e.g., a fighter jet has a quantity describing its ammo count, its fuel, its altitude, and so on). Furthermore, many kinds of objects can perform actions (e.g., a jet can fly, fire a missile, land, and so on). Classes allow us to model real-world objects in code; that is, with classes we can define new variable types which consist of several properties (data members), and which can perform actions (member functions). Consequently, with classes, we can instantiate variables that behave like real-world objects.

2. The `public` and `private` keywords are used to enforce which parts of the class are visible to *outside* code. A class' own member functions can access all parts of the class since they are *inside* the class, so to speak. The general rule is that data members (class variables) should be kept internal (private) and only methods should be exposed publicly. By making the programmer go through member functions, the class can safely regulate how the internal class data can be manipulated. Hence, the class can maintain the integrity of its internal data.

3. In addition to defining a class, you can define what is called a structure. In C++, a structure is exactly the same as a class except that it is public by default, whereas a class is private by default.

4. One of the goals of C++ is to separate class definitions from implementation. The motivation behind this is that a programmer should be able to use a class without knowing exactly how it works. This occurs when you realize that you may be using classes you did not write yourself—for example, if you learn DirectX, you will use classes that Microsoft wrote. In order to separate class definitions from implementation, two separate files are used: header files (.h) and implementation files (.cpp). Header files include the class definition, and implementation files contain the class implementation (i.e., method definitions). Eventually, the source code file is compiled to either an object file (.obj) or a library file (.lib). Once that is done you can distribute the class header file along with the object file or library file to other programmers. With the

header file containing the class definition, and with the object or library file containing the compiled class implementation, which is linked into the project with the linker, the programmer has all the necessary components to use the class without having seen the implementation file (that is, the .cpp file). For example, the DirectX Software Development Kit includes only the DirectX header files and the DirectX library files—you never see the implementation files (.cpp). This does two things: first, others cannot modify the implementation, and second your implementation (intellectual property) is protected.

5. Every class has a constructor and destructor, which are special kinds of methods. If you do not explicitly define these methods, the compiler will generate default ones automatically. A constructor is a method that is automatically executed when an object is instantiated and a destructor is a method that is automatically executed when an object is deleted. A constructor is usually used to initialize data members to some default value or to allocate any dynamic memory the class uses or to execute any initialization code that you want executed as the object is being created. Conversely, the destructor is usually used to free any dynamic memory which the class has allocated. If it does not delete it when the object is being destroyed, it will result in a memory leak. Note that you never invoke a destructor yourself; rather, the destructor will automatically be called when an object is being deleted from memory.

6. When utilizing the object oriented programming paradigm, the first thing we ask when designing a new program is: "What objects does the program attempt to model?" The answer to this question depends on the program. After we have decided what objects our program will use, we need to design corresponding classes, which define the properties of these kinds of objects and the actions they perform. In addition to the data and methods of a class, the class design will also need to consider the relationships between the objects of one class and the objects of other classes—for example, how they will they interact with each other.

5.6 Exercises

The exercises for this chapter are suggestions for modifications to the role-playing game discussed in Sections 5.3 and 5.4. They are not very specific on the exact details about what must happen—they are kept open-ended on purpose. You are free to implement these solutions any way you choose —add new data and methods to the existing classes, or create new classes.

5.6.1 Gold Modification

Add an integer data member to the `Player` class that keeps track of the player's amount of gold. After each battle, generate a random gold reward that the player receives, in addition to the experience point award. It would make sense for harder enemies (ogres, Orc lords) to provide a larger gold reward then orcs and goblins. Be sure to also modify the `Player::viewStats()` method to display the current amount of gold the player owns.

5.6.2 Character Races

Modify the character creation process so that the user can choose a character race (e.g., dwarf, human, elf, halfling). Additionally, give statistical pros and cons to each race. For example, an elf may start the game with a higher accuracy rating than a dwarf, but lower hit points. Conversely, a dwarf may start out with more hit points than an elf, but less accuracy.

5.6.3 Leveling Up

The leveling up process we implemented in Section 5.3 does not take the player's class into consideration. Modify the leveling up process to reflect the class of the character. For example, a "fighter" should gain more hit points than a "wizard."

5.6.4 Magic Points

Add an integer data member to the Player class that describes the number of magic points the player currently has. Additionally, add a "max magic points" data member that describes the maximum number of magic points the player can have at his/her current level. Be sure to also modify the `Player::viewStats()` method to display the current amount of magic points the player has. Also, be sure to increase the amount of magic points when the character levels up—and a "wizard" should gain more hit points than a "fighter."

After you have added magic points to the system, create a `Spell` structure and then instantiate a few types of spell objects, such as “magic missile,” “fireball,” and “shield.” You may want to implement the Spell structure like so:

```
struct Spell
{
	std::string mName;
	Range       mDamageRange;
	int         mMagicPointsRequired;
};
```

- `mName`: The name of the spell (e.g., “Fireball”).
- `mDamageRange`: The range of damage the spell inflicts.
- `mMagicPointsRequired`: The number of magic points required to cast the spell.

Finally, add a “cast spell” option to the combat menu, which allows the player to select and cast a spell from a list of spells in his/her spell book. Be sure to verify that the player has enough magic points to cast the spell. And also be sure to deduct the magic points required to cast the spell from the player’s magic point count, after a spell is cast.

5.6.5 Random Encounters During Rest

In the current implementation, a player can rest to full hit points after every battle with no risk. Since there is no such thing as a free lunch, add random encounters when resting– maybe a 25% chance of an attack during rest.

5.6.6 A Store

Add a store to the map at some location. That is, in the main game loop do something like:

```
if( gameMap.getPlayerXPos() == 2 &&
    gameMap.getPlayerYPos() == 3 )
{
	Store store;
	store.enter(gamePlayer);
}
```

Thus, if the player lands on the point (2, 4), the player enters the store. And `Store` is a class you define which contains methods for displaying the store’s inventory, and for buying/selling weapons, armor, and items. Note that you may want to write a small armor structure so that you can name armor:

```
struct Armor
```

```
{
	std::string mName;
	int armorValue;
};
```

5.6.7 Items

Allow the player to buy and carry healing potions, which can be used to heal the player during combat. Moreover, allow the user to carry magic fireball potions, which can be used against enemies in battle. You can add an `Item` array (you define the `Item` class) to the `Player` class for storing these items, or use a `std::vector` to store them.

So that the player can use an item during combat, add a "use item" option to the combat menu.

5.6.8 Multiple Enemies

Modify the game so that the player can encounter several enemies at once. For example, `Map::checkRandomEncounter()` can return a pointer to an array of enemies. Then update `Player::attack(Monster& monster)` to instead take an array of monsters: `Player::attack(Monster monsters[])`.

Chapter 6

Strings and Other Topics

Introduction

Recall that a **string** is an ordered set of characters (e.g., ‘h’, ‘e’, ‘l’, ‘l’, ‘o’), which is commonly used to form words and sentences (e.g., “hello”). So far, we have been using the standard library type `std::string` to represent strings, but we may ask: how does `std::string` work internally? The first theme of this chapter is to address that question and to discover how strings can be described using only intrinsic C++ data types. The second theme of this chapter is to survey the additional functionality `std::string` provides via its class method interface. Finally, this chapter closes by discussing some miscellaneous C++ keywords and constructs.

Chapter Objectives

- Understand how C++ natively describes strings.
- Learn some important standard library string functions.
- Review `std::string` and become familiar with some of its methods.
- Become familiar with the `this` pointer.
- Learn about the `friend` and `static` keywords.
- Discover how to create your own namespaces.
- Understand what enumerated types are, how they are defined in C++, and when they would be used.

6.1 `char` Strings

We know that the `char` keyword can represent a character, and we know a string is a set of characters. Thus, it follows that a string can be represented with an array of `char`s -- which is how C++ natively supports strings. A string represented as an array of `char`s is called a **c-string.** For instance, the string “Hello” can be represented as:

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

The character ‘\0’ is a special character called the **null character,** and it is used to mark the end of a c-string. A c-string that ends with a null character is called a **null-terminating string.** The end of the c-string is marked because it then can be determined when the c-string ends. The ability to figure out when a string ends is important when navigating the elements of the string. For example, if we assume the input c-string is a null-terminating string then we can write a function that returns the number of characters in the c-string (excluding the terminating null character) like so:

```
// Assume str is a pointer to a null-terminating string.
int StringLength(char* str)
{
      int cnt = 0;

      // Loop through the array until we reach the end.
      while( str[cnt] != '\0' )
            ++cnt; // Count character.

      // Return the number of characters.
      return cnt;
}
```

Without the null character telling us when the c-string ends, we would not know when to exit the loop and how many characters to count.

Using `StringLength` we can write a function to print out a c-string, element-by-element as shown here:

```
int main()
{
      char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

      cout << "str = ";
      for(int i = 0; i < StringLength(str); ++i)
            cout << str[i];

      cout << endl;
}
```

Again, we need to know how many characters are in the c-string in order to know how many times to loop. Incidentally, you will never write code to print a c-string like this because `cout` is overloaded to print a c-string:

```
int main()
{
      char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

      cout << "str = ";
      cout << str;
      cout << endl;
}
```

Note that even `cout` needs `str` to be a null-terminating string so that it too can figure out how many characters are in the c-string.

You may observe that we could keep track of the number of elements in a c-string in order to do away with `StringLength` and the null character completely, and we would know exactly how many characters are in the c-string. This would work, but it is not convenient to have to carry around an extra "size" variable per c-string. By using a null-terminating c-string, the string size can be deduced from the c-string itself, which is much more compact and convenient.

6.1 String Literals

Recall that a literal such as 3.14f is considered to be of type `float`, but what type is a string literal such as “hello world”? C++ will treat “hello world” as a `const char[12]`. The `const` keyword indicates that the string is a literal and a literal cannot be changed. Because all string literals are specified in the program (i.e., they are written in the source code), C++ can know about every literal string the program uses at compile time. Consequently, all string literals are allocated in a special global segment of memory when the program starts. The important property about the memory for string literals is that it exists for the life of the program.

Because string literals are stored in `char` arrays, pointers to the first element can be acquired like so:

```
char* pStr = "hello world";
```

However, it is important not to modify the elements of the string literal via `pStr`, because “hello world” is constant. For example, as Stroustrup points out, the following would be undefined:

```
char* pStr = "hello world";
pStr[5] = '-'; // undefined
```

Now consider the following:

```
char* LiteralMsg()
{
      char* msg = "hello world";

      return msg;
}

int main()
{
      cout << "msg = " << LiteralMsg() << endl;
}
```

In the function `LiteralMsg` we obtain a pointer to the literal “hello world.” We then return a copy of this pointer back to the caller, which in this case is `main`. You might suspect that this code is flawed because it returns a pointer to a “local” string literal, which would be destroyed after the function returns. However, this does not happen because memory for string literals is not allocated locally, but is allocated in a global memory pool at the start of the program. Thus, the above code is correct.

6.2 Escape Characters

In addition to characters you are already familiar with, there exist some special characters, called **escape characters.** An escape character is symbolized with a backslash \ followed by a regular character(s). For instance, the new-line character is symbolized as ‘\n’.

The following table shows commonly used escape characters:

Symbol	Description
\n	New-line character: Represents a new line.
\t	Tab character: Represents a tab space.
\a	Alert character: Represents an alert.
\\	Backslash: Represents a backslash character.
\'	Single quote mark:
\"	Double quote mark

Interestingly, because the backslash \ is used to denote an escape character, you may wonder how you would actually express the character ‘\’ in a string. C++ solves this by making the backslash character an escape character itself; that is, a backslash followed by a backslash. Similarly, because the single and double quotation marks are used to denote a character literal and string literal, respectively, you may wonder how you would actually express the characters ‘'’ and ‘"’ in a string. C++ solves this by making the quotation mark characters escape characters: ‘\' ’, and ‘\"’.

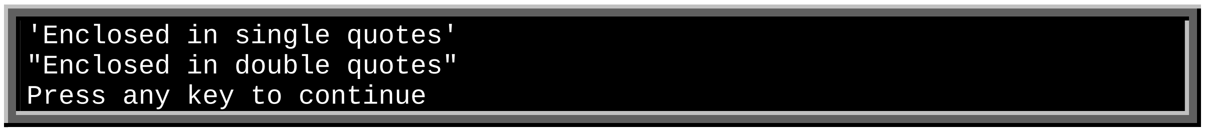
Program 6.1 demonstrates how the escape characters can be used.

Program 6.1: Escape Characters.

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

int main()
{
      cout << "\tAfter Tab" << endl;
      cout << "\nAfter newline" << endl;
      cout << "\aAfter alert" << endl;
      cout << "\\Encloses in backslashes\\" << endl;
      cout << "\'Enclosed in single quotes\'" << endl;
      cout << "\"Enclosed in double quotes\"" << endl;
}
```

Program 6.1 Output

```
        After Tab

After newline
After alert
\Encloses in backslashes\
```



```
'Enclosed in single quotes'
"Enclosed in double quotes"
Press any key to continue
```

The "alert" causes the computer to make a beeping sound. Observe how we can print a new line by outputting a new-line character. Consequently, '\n' can be used as a substitute for `std::endl`. It is worth emphasizing that escape characters are *characters;* they fit in a `char` variable and can be put in strings as such. Even the new-line character, which seems like it occupies a whole line of characters, is just one character.

> **Note:** '\n' and `std::endl` are not exactly equivalent. `std::endl` flushes the output stream each time it is encountered, whereas '\n' does not. By "flushing" the output stream, we mean output is kept buffered up so that many characters can be sent (flushed) to the hardware device at once. This is done purely for efficiency—it is more efficient to send lots of data to the device (e.g., console window output) at one time than it is to send many small batches of data. Thus, you may not want to use `std::endl` frequently since that would mean you are flushing small batches of data frequently, instead of one large batch of data infrequently.

6.2 C-String Functions

The previous section showed how we could represent strings as arrays of `chars` (c-strings). We now look at some standard library functions that operate on c-strings. To include these functions in your code, you will need to include the <cstring> header file. Note that this header file is different than the <string> header file, which is used for `std::string`.

6.2.1 Length

We already talked about length and we even wrote our own function to compute the length of a null-terminating string. Not surprisingly, the standard library already provides this function for us. The function is called `strlen` and the function is prototyped as follows:

```
size_t strlen(const char *string);
```

The type `size_t` is usually defined as a 32-bit unsigned integer. The parameter `string` is a pointer to a null-terminating c-string, and the function returns the number of characters in this input string.

<u>Example</u>:

```
int length = strlen("Hello, world!"); // length = 13
```

6.2.2 Equality

One of the first things we can ask about two strings is whether or not they are they equal. To answer this question the standard library provides the function `strcmp` (string compare):

```
int strcmp(const char *string1, const char *string2);
```

The parameters, `string1` and `string2`, are pointers to null-terminating c-strings, which are to be compared. This function returns three possible types of numbers:

- Zero: If the return value is zero then it means the strings, `string1` and `string2`, are equal.

- Negative: If the return value is negative then it means `string1` is less than `string2`. What does "less than" mean in the context of strings? A string A is less than a string B if the difference between the first two unequal characters `A[k] – B[k]` is less than zero, where *k* is the array index of the first two unequal characters.

  For example, let A = "hella" and B = "hello"; the first two unequal characters are found in element [4]—'a' does not equal 'o'. Since the integer representation of 'a' (97) is less than the integer representation of 'o' (111), A is less than B.

  Consider another example: let A = "abc" and B = "abcd"; the first unequal character is found in element [3] (remember the terminating null!). That is, '\0' does not equal 'd'. Because the integer representation of '\0' (zero) is less than the integer representation of 'd' (100), A is less than B.

- Positive: If the return value is positive then it means `string1` is greater than `string2`. What does "greater than" mean in the context of strings? A string A is greater than a string B if the difference between the first two unequal characters `A[k] – B[k]` is greater than zero, where *k* is the array index of the first two unequal characters.

  For example, let A = "sun" and B = "son"; the first two unequal characters are found in element [1]—'u' does not equal 'o'. Since the integer representation of 'u' (117) is greater than the integer representation of 'o' (111), A is greater than B.

  Consider another example: let A = "xyzw" and B = "xyz"; the first unequal character is found in element [3] (remember the terminating null!). That is, 'w' does not equal '\0'. Because the integer representation of 'w' (119) is greater than the integer representation of '\0' (zero), A is greater than B.

Example:

```
int ret = strcmp("Hello", "Hello");
// ret = 0 (equal)

ret = strcmp("abc", "abcd");
```

```
// ret < 0 ("abc" < "abcd")

ret = strcmp("hello", "hella");
// ret > 0 ("hello" > "hella")
```

6.2.3 Copying

Another commonly needed function is one that copies one (source) string to another (destination) string:

```
char *strcpy(char *strDestination, const char *strSource);
```

The second parameter, `strSource`, is a pointer to a null terminating c-string, which is to be copied to the destination parameter `strDestination`, which is a pointer to an array of `char`s. The c-string to which `strSource` points is made constant to indicate that `strcpy` does not modify it. On the other hand, `strDestination` is not made constant because the function does modify the array to which it points. This function returns a pointer to `strDestination`, which is redundant because we already have a pointer to `strDestination`, but it does this so that the function could be passed as an argument to another function (e.g., `strlen(strcpy(dest, source));`). Also, it is important to realize that `strDestination` must point to a `char` array that is large enough to store `strSource`.

Example:

```
char dest[256];

char* source = "Hello, world!";

strcpy(dest, source);

// dest = "Hello, world!"
```

Note that `strcpy` does not resize the array `dest`; rather, `dest` stores "Hello, world!" at the beginning of the array, and the rest of the characters in the 256 element array are simply unused.

6.2.4 Addition

It would be convenient to be able to add two strings together. For example:

"hello " + "world" = "hello world"

This is called string concatenation (i.e., joining). The standard library provides such a function to do this, called `strcat`:

```
char *strcat(char *strDestination, const char *strSource);
```

The two parameters, `strDestination` and `strSource,` are both pointers to null-terminating c-strings. The c-string to which `strSource` points is made constant to indicate that `strcat` does not modify it. On the other hand, `strDestination` is not made constant because the function does modify the c-string to which it points.

This function appends `strSource` onto the back of `strDestination`, thereby joining them. For example, if the source string S = "world" and the destination string D = "hello", then after the call `strcat(D, S)`, D = "hello world". The function returns a pointer to `strDestination`, which is redundant because we already have a pointer to `strDestination`. It does this so that the function can be passed as an argument to another function (e.g., `strlen(strcat(dest, source));`).

It is important to realize that `strDestination` must be large enough to store the concatenated string (`strDestination` is not a string literal, so the compiler will not be automatically allocating memory for it). Returning to the preceding example, the c-string to which D pointed must have had allocated space to store the concatenated string "hello world". To ensure that the destination string can store the concatenated string, it is common to make D an array with "max size":

```
const int MAX_STRING SIZE = 256;
char dest[MAX_STRING_SIZE];
```

This means that some memory might be wasted. However, we can be more precise by using dynamic memory to obtain an array size that is exactly large enough to store the concatenated string.

<u>Example</u>:

```
char dest[256];
char* source = "Hello, world!";
strcpy(dest, source);
// dest = "Hello, world!"

strcat(dest, " And hello, C++");
// dest = "Hello, world! And hello, C++"
```

6.2.7 Formatting

Sometimes we will need to put variable values, such as integers and floating-point numbers, into strings. That is, we must format a numeric value so that it becomes a string (e.g., 3.14 becomes "3.14"). We can do this with the `sprintf` function:

```
int sprintf(
   char *buffer,
   const char *format,
   [argument] ...
);
```

This function returns the number of characters in the array (excluding the terminating null) to which buffer points after it has received the formatted output.

- `buffer`: A pointer to a `char` array, which will receive the formatted output.

- `format`: A pointer to a null-terminating c-string, which contains a string with some special formatting symbols within. These formatting symbols will be replaced with the arguments specified in the next parameter.

- `argument`: This is an interesting parameter. The ellipses syntax (…) indicates a variable amount of arguments. It is here where the variables are specified whose values are to replace the formatting symbols in `format`. Why a variable number of arguments? Because the format string can contain any number of formatting symbols and we will need values to replace each of those symbols. Since the number of formatting symbols is unknown to the function, the function must take a variable amount of arguments. The following examples illustrate the point.

Suppose that you want to ask the user to enter a number and then put the result into a string. Again, because the number is variable (we do not know what the user will input), we cannot literally specify the number directly into the string—we must use the `sprintf` function:

Program 6.2: The sprintf Function.

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

using namespace std;

int main()
{
	char buffer[256];

	int num0 = 0;
	cout << "Enter a number: ";
	cin >> num0;

	sprintf(buffer, "You entered %d", num0);
	cout << buffer << endl;
}
```

The `format` string contains one formatting symbol called '%d'; this symbol will be replaced by the value stored in the argument `num0`. For example, if we execute this code we get the following output:

Program 6.2 Output

```
Enter a number: 11
You entered 11
Press any key to continue
```

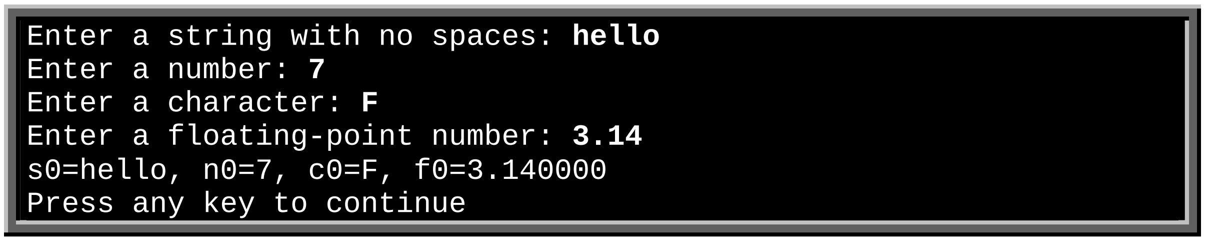
As you can see, the number entered (11) replaced the %d part of the `format` string.

Now suppose that you want the user to enter a string, a character, an integer, and a floating-point number. Because these values are variable (we do not know what the user will input), we cannot literally specify the values directly into the string—we must use the `sprintf` function:

Program 6.3: Another example of the sprintf function.

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

using namespace std;

int main()
{
	char buffer[256];

	char s0[256];
	cout << "Enter a string with no spaces: ";
	cin >> s0;

	int n0 = 0;
	cout << "Enter a number: ";
	cin >> n0;

	char c0 = '\0';
	cout << "Enter a character: ";
	cin >> c0;

	float f0 = 0.0f;
	cout << "Enter a floating-point number: ";
	cin >> f0;

	sprintf(buffer, "s0=%s, n0=%d, c0=%c, f0=%f",s0,n0,c0,f0);

	cout << buffer << endl;
}
```

Program 6.3 Output



```
Enter a string with no spaces: hello
Enter a number: 7
Enter a character: F
Enter a floating-point number: 3.14
s0=hello, n0=7, c0=F, f0=3.140000
Press any key to continue
```

This time the `format` string contains four formatting symbols called %s, ‘%d’, %c, and %f. These symbols are replaced by the values stored in `s0`, `n0`, `c0`, and `f0`, respectively—Figure 6.1 illustrates.

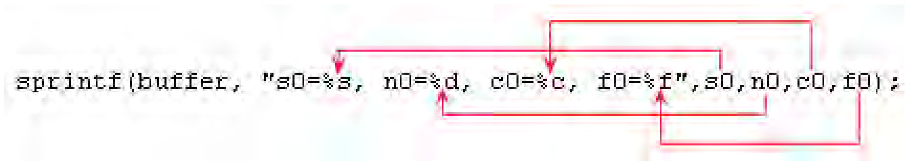

```
sprintf(buffer, "s0=%s, n0=%d, c0=%c, f0=%f",s0,n0,c0,f0);
```

Figure 6.1: Argument list replace formatting symbols.

The following table summarizes the different kinds of format symbols:

%s	Formats a string to a string.
%c	Formats a character to a string.
%n	Formats an integer to a string.
%f	Formats a floating-point number to a string.

Now you can see why the `argument` parameter of `sprintf` requires a variable number of arguments—one argument is needed for each formatting symbol. In our first example, we used one formatting symbol and thus, had one argument. In our second example, we used four formatting symbols and thus, had four arguments.

6.3 std::string

We now know that at the lowest level, we can represent a string using an array of `char`s. So how does `std::string` work? `std::string` is actually a class that uses `char` arrays internally. It hides any dynamic memory that might need to be allocated behind the scenes and it provides many methods which turn out to be clearer and more convenient to use than the standard library c-string functions. In this section we survey some of the more commonly used methods.

6.3.1 Length

As with c-strings, we will often want to know how many characters are in a `std::string` object. To obtain the length of a `std::string` object we can use either the length or the size method (they are different in name only):

```
string s = "Hello, world!";
int length = s.length();
int size   = s.size();
// length = 13 = size
```

6.3.2 Relational Operators

One of the useful things about `std::string` is that it defines relational operators which provide a much more natural syntax than using `strcmp`.

- Equal: We can test if two strings are equal by using the equality operator (==). If two strings A and B are equal, the expression (A == B) evaluates to `true`, otherwise it evaluates to `false`.

- Not Equal: We can test if two strings are not equal by using the not equal operator (`!=`). If two strings A and B are not equal, the expression (A `!=` B) evaluates to `true`, otherwise it evaluates to `false`.

- Less Than: We can test if a string A is less than a string B by using the less than operator (<). If A is less than B then the expression (A < B) evaluates to `true`, otherwise it evaluates to `false`.

- Greater Than: We can test if a string A is greater than a string B by using the greater than operator (>). If A is greater than B then the expression (A > B) evaluates to `true`, otherwise it evaluates to `false`.

- Less Than or Equal To: We can test if a string A is less than or equal to a string B by using the less than or equal to operator (<=). If A is less than or equal to B then the expression (A <= B) evaluates to `true`, otherwise it evaluates to `false`.

- Greater Than or Equal To: We can test if a string A is greater than or equal to a string B by using the greater than or equal to operator (>=). If A is greater than or equal to B then the expression (A >= B) evaluates to `true`, otherwise it evaluates to `false`.

See Section 6.2.2 for a description of how “less than” and “greater than” are defined for strings.

Examples:

```
string s0 = "Hello";
string s1 = "Hello";
string s2 = "abc";
string s3 = "abcd";

(s0 == s1); // true
(s0 != s1); // false
(s1 != s2); // true
(s2 < s3);  // true
(s3 > s2);  // true
```

6.3.3 Addition

We can add two strings together to make a third “sum” string using the addition operator (+):

```
string A = "Hello, ";
string B = "world!";

string C = A + B; // sum = "Hello, world!"

cout << C << endl;
```

This outputs: `Hello, world!`

Furthermore, rather than adding two strings, A and B, to make a third string C, we can directly append a string to another string using the compound addition operator (+=):

```
string A = "Hello, ";
string B = "world!";

A += B; // A = "Hello, world!"

cout << A << endl;
```

The compound addition operator is essentially the `std::string` equivalent to the c-string `strcat` function.

6.3.4 Empty Strings

Sometimes we would like to know if a `std::string` object is an “empty” string (i.e., contains no characters). For example, the string “” is an empty string. To test whether a `std::string` object is empty we use the `empty()` method which returns true if the object is empty and false otherwise. Consider the following short program:

Program 6.4: Empty Strings.

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

int main()
{
    string emptyString = "";
    string notEmptyStr = "abcdef";

    if( emptyString.empty() == true )
        cout << "emptyString is empty." << endl;
    else
        cout << "emptyString is actually not empty." << endl;
    if( notEmptyStr.empty() == true )
        cout << "notEmptyStr is empty." << endl;
    else
        cout << "notEmptyStr is actually not empty." << endl;
}
```

Program 6.4 Output

```
emptyString is empty.
notEmptyStr is actually not empty.
Press any key to continue
```

As the program output verifies, `emptyString` is indeed empty, and so the condition `emptyString.empty() == true` evaluates to `true`, thereby executing the corresponding *if* statement:

```
      cout << "emptyString is empty." << endl;
```

On the other hand, `notEmptyStr` is *not* empty, so the condition `emptyString.empty() == true` evaluates to `false`, therefore the corresponding *else* statement is executed:

```
      cout << "notEmptyStr is actually not empty." << endl;
```

6.3.5 Substrings

Every so often we will want to extract a smaller string contained within a larger string—we call the smaller string a **substring** of the larger string. To do this, we use the `substr` method. Consider the following example:

Program 6.5: Substrings.

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

int main()
{
     string str = "The quick brown fox jumped over the lazy dog.";
     string sub = str.substr(10, 9);
     cout << "str = " << str << endl;
     cout << "str.substr(10, 9) = " << sub << endl;
}
```

Program 6.5 Output

```
str = The quick brown fox jumped over the lazy dog.
str.substr(10, 9) = brown fox
Press any key to continue
```

We pass two arguments to the `substr` method; the first specifies the starting character index of the substring to extract, and the second argument specifies the length of the substring—Figure 6.2 illustrates.

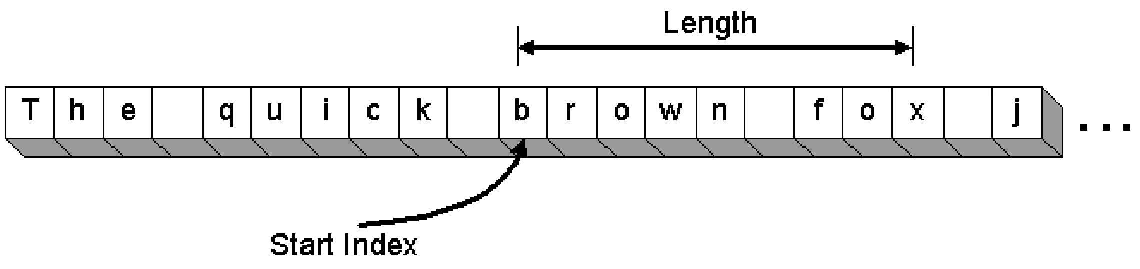


Figure 6.2: Starting index and length.

As the output verifies, the string which starts at index 10 and has a length of 9 is "brown fox".

6.3.6 Insert

At times we may wish to insert a string somewhere into another string—it could be at the beginning, middle, or end. We can do this with the `insert` method, as this next example illustrates:

Program 6.6: String insertion.

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

int main()
{
     string str = "The  fox jumped over the lazy dog.";
     cout << "Before insert: " << str << endl;

     string strToInsert = "quick brown";
     str.insert(4, strToInsert);
     cout << "After insert: " << str << endl;
}
```

Program 6.6 Output

```
Before insert: The  fox jumped over the lazy dog.
After insert: The quick brown fox jumped over the lazy dog.
Press any key to continue
```

We pass two arguments to the insert method; the first is the starting character index specifying where we wish to insert the string; the second argument specifies the string we are inserting. In our example, we specify character [4] as the position to insert the string, which is the character space just before the word "fox." And as the output verifies, the string "brown fox" is inserted there correctly.

6.3.7 Find

Another string operation that we may need is one that looks for a substring in another string. We can do this with the `find` method, which returns the index to the first character of the substring if it is found. Consider the following example:

Program 6.7: String Finding.

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

int main()
{
     string str = "The quick brown fox jumped over the lazy dog.";

     // Get the index into the string where "jumped" starts.
     int index = str.find("jumped");
     cout << "\"jumped\" starts at index: " << index << endl;
}
```

Program 6.7 Output

```
"jumped" starts at index: 20
Press any key to continue
```

Here we are searching for "jumped" within the string, "The quick brown fox jumped over the lazy dog." If we count characters from left-to-right, we note that the substring "jumped" starts at character [20], which is what `find` returned.

6.3.8 Replace

Sometimes we want to replace a substring in a string with a different substring. We can do this with the `replace` method. The following example illustrates:

Program 6.8: String replacing.

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

int main()
{
     string str = "The quick brown fox jumped over the lazy dog.";
     cout << "Before replace: " << str << endl;
     // Replace "quick brown" with "slow blue"
     str.replace(4, 11, "slow blue");
     cout << "After replace: " << str << endl;
}
```

Program 6.8 Output

```
Before replace: The quick brown fox jumped over the lazy dog.
After replace: The slow blue fox jumped over the lazy dog.
Press any key to continue
```

Based on the program output, it is not difficult to see how `replace` works; specifically, it replaced the substring, identified by the character range [4, 11], with "slow blue." If we count the characters in `str` before the `replace` operation occurred, we find that the range [4, 11] identifies the substring "quick brown." Based on the output, this is exactly the substring that was replaced with "slow blue."

6.3.9 Bracket Operator

Sometimes we want to access a specific character in a `std::string` object. We can do this with the bracket operator ([]):

```
string s = "Hello, world!";

char c0 = s[0];  // = 'H'
char c1 = s[1];  // = 'e'
char c2 = s[4];  // = 'o'
char c3 = s[7];  // = 'w'
char c4 = s[12]; // = '!'
```

6.3.10 C-String Equivalent

Some existing C++ libraries do not use `std::string`, but instead work with c-strings. Thus if we want to work with `std::string`, and still be able to use a C++ library that does not (for example, DirectX uses c-strings) then we need a way to convert between the two.

We can create a `std::string` object from a c-string since `std::string` provides a constructor and assignment operator, which both take a c-string parameter. From this c-string representation, an equivalent `std::string` object can be built, which describes the same string:

```
string s = "Assignment";
string t("Constructor");
```

So going from a c-string to a `std::string` object is taken care of.

To go in the other direction, that is, to get a c-string representation from a `std::string`, we use the `c_str` method:

```
string s = "Assignment";
const char* cstring = s.c_str(); // cstring = "Assignment"
```

6.3.11 `getline`

Consider the following small program:

Program 6.9: Attempting to read a line if input with cin.

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

int main()
{
     string s = "";
     cout << "Enter a multiple word string: ";
     cin >> s;

     // Echo the string the user entered back to the console window:
     cout << "You entered: " << s << endl;
}
```

Program 6.9 Output

```
Enter a multiple word string: Hello, world!
You entered: Hello,
Press any key to continue
```

What happened? We enter in “Hello, world!” but the program only echoed the string up to a space (“Hello,”). The problem is that `cin` only reads up to the first whitespace character. To get around this problem we use the `getline` function, which can read up to a line of input. Program 6.10 rewrites the preceding program using the `getline` function.

Program 6.10: The getline Function.

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

int main()
{
     string s = "";
     cout << "Enter a multiple word string: ";
     getline(cin, s);

    // Echo the string the user entered back to the console window:
    cout << "You entered: " << s << endl;
}
```

Program 6.10 Output

```
Enter a multiple word string: Hello, world!
You entered: Hello, world!
Press any key to continue
```

The program now behaves as expected.

The `getline` function takes two parameters. First, it takes a reference to a `std::istream` object, where `istream` is a class that provides an interface for inputting data from various sources. We note that `cin` is actually a global instance of this class that is setup to obtain input from the keyboard. The second parameter is a reference to a `std::string` object through which the function returns the resulting line of string.

In reality, there is a third default parameter where we can specify a *delimiter* (a character indicating where to end the flow of input). By default, `getline` uses the '\n' character as the delimiter. Thus, it reads up to the first new-line character. Alternatively, we could use another delimiter such as 'a', which would instruct `getline` to read data up to the first 'a' character encountered:

```
getline(cin, s, 'a');
```

We will now turn our attention away from strings, and devote the rest of this chapter to some miscellaneous C++ topics.

6.4 The `this` Pointer

Consider the following simple class definition and its implementation:

```
// Class definition
class Person
{
public:
      Person(string name, int age);

      string getName();
      int    getAge();
      void   talk(Person& p);

private:
      string mName;
      int    mAge;
};

// Class implementation
Person::Person(string name, int age)
{
      mName = name;
      age   = age;
}

string Person::getName()
{
      return mName;
}

int Person::getAge()
{
      return mAge;
}

void Person::talk(Person& p)
{
      cout << mName << " is talking to ";
      cout << p.mName << endl;
}
```

We tend to think of a class as defining the properties (data) and actions (methods) that instances of this class contain. For example, if we instantiate the following objects:

```
Person mike("Mike", 32);
Person tim("Tim", 29);
Person vanessa("Vanessa", 20);
```

We say each `Person` object instance has its own name and age, and each `Person` object has its own constructor, `getName()`, `getAge()`, and `talk()` methods, which can access the corresponding data members.

In general, the data properties of each `Person` object are unique; that is, no two persons are exactly alike (Mike has his own name and age, Tim has his own name and age, and Vanessa has her own name and age). Therefore, each object must really have its *own* individual data members—so far, so good. However, what about the member functions? The implementation of a method is the same across all objects. The only difference is with the data members which that method might access. If we call `mike.getName()`, we expect `getName()` to return `mike`'s data member `mike.mName`, if we call `tim.getName()`, we expect `getName()` to return `tim`'s data member `tim.mName`, and if we call `vanessa.getName()`, we expect `getName()` to return `vanessa`'s data member `vanessa.mName`. The same is true for the other member functions as well.

It does not seem practical to have a separate function for each object, all of which do the same thing, but with different data members. So what C++ actually does with member functions is to implement them like normal functions, but it creates the function with a "hidden" parameter which the compiler passes as a pointer to the member function's calling object. For example, the methods that Person defines would really be written like so:

```
Person::Person(Person* this, string name, int age)
{
      this->mName = name;
      this->age   = age;
}

string Person::getName(Person* this)
{
      return this->mName;
}

int Person::getAge(Person* this)
{
      return this->mAge;
}

void Person::talk(Person& p)
{
      cout << mName << " is talking to ";
      cout << p.mName << endl;
}
```

When we invoke methods like so:

```
Person mike("Mike", 32);
mike.getName();
tim.getAge();
vanessa.talk(tim);
```

This really evaluates to the following:

```
Person::Person(&mike, "Mike", 32);
Person::getName(&mike);
Person::getAge(&tim);
Person::talk(&vanessa, tim);
```

Since these are member functions, they are able to directly access the data members of the passed-in object pointer, whether the data is private or public. A class can always access its entirety within itself. The `public` and `private` keywords only indicate how external code can access the class.

By passing a hidden pointer into the member function, which points to the object which called the function, the member function is able to access the data members of the calling object.

Although this is all hidden from the programmer, it is not completely hidden. C++ allows you to access the hidden pointer parameter inside your method definition with the **`this` pointer** keyword. The name `this` is just the name of the hidden pointer parameter passed into the member function (which you do not actually see), so that the member function can access the data members of the object which called it. For example, writing the following:

```
Person::Person(string name, int age)
{
      mName = name;
      age   = age;
}

string Person::getName()
{
      return mName;
}

int Person::getAge()
{
      return mAge;
}

void Person::talk(Person& p)
{
      cout << mName << " is talking to ";
      cout << p.mName << endl;
}
```

This is equivalent to writing:

```
Person::Person(string name, int age)
{
      this->mName = name;
      this->age   = age;
}

string Person::getName()
{
      return this->mName;
}

int Person::getAge()
{
      return this->mAge;
}
```

```
void Person::talk(Person& p)
{
	cout << this->mName << " is talking to ";
	cout << p.mName << endl;
}
```

In the latter case, the `this` pointer is used explicitly, and in the former case it is used implicitly.

6.5 Friends

6.5.1 Friend Functions

Sometimes we will have a non-member function that is closely related to a class. It is so closely related that we would like to give that function the ability to access the class' private data and methods. To do this, the class must declare the function a **friend.** A friend of the class can then access the class' private members directly—it does *not* need to go through the class' public interface. Consider the following example:

```
class Point
{
	friend void PrintPoint(Point& p);
public:
	Point(float x, float y, float z);

private:
	float mX;
	float mY;
	float mZ;
};

Point::Point(float x, float y, float z)
{
	mX = x;
	mY = y;
	mZ = z;
}

void PrintPoint(Point& p)
{
	cout << "(" << p.mX << ", ";
	cout << p.mY << ", ";
	cout << p.mZ << ")" << endl;
}

int main()
{
	Point p(1.0f, 2.0f, 3.0f);

	PrintPoint(p);
}
```

Even though `PrintPoint` is *not* a member function of class `Point`, it is still able to access the private data members of a `Point` object. This is because we made the function `PrintPoint` a friend of class `Point`. To make a function a friend you use the `friend` keyword, followed by the function prototype in the class definition. The friend statement can be written anywhere in the class definition.

Note that this example is for illustrative purposes only; in reality, the print function should be a member function of `Point`.

6.5.2 Friend Classes

Friend classes extend the idea of friend functions. Instead of making a function a friend, you make *all* the methods of another class a friend. The syntax to make all the methods of a class friends with our class is similar to functions. The `friend` keyword followed by the class prototype in the class definition is used:

```
class A{...};

class B
{
      friend class A;
...
};
```

All the methods of class A would now be able to access the private components of an object of class B.

6.6 The `static` Keyword

Variable declarations can be prefixed with the `static` keyword. For example:

```
static int staticVar;
```

The meaning of `static` depends on the context of the variable.

6.6.1 Static Variables in Functions

A static variable declared inside a function has an interesting property. It is created and initialized once and *not* destroyed when the function terminates; that is, its value persists across function calls. A simple application of a static variable inside a function is used when you want the function to execute some special code the very first time it is called:

```
void Func()
{
	// Created and initialized once at program start.
	static bool firstTime = true;

	if( firstTime )
	{
		// Do work the first time the function is called.
		// ...

		// Set firstTime to false so that this first-time
		// work is not executed in subsequent calls.
		firstTime = false;
	}
}
```

The static variable `firstTime` will be initialized once, at the start of the program, to `true`. Thus, the first time the function is called it will be true and the if-statement body will execute. However, the if-statement body then sets `firstTime` to `false`. This value will persist even across all calls to this function. Therefore, `firstTime` will be `false` for all subsequent calls to this function, and the if-statement body will not execute. Thus, by using a static variable in a function, we were able to control the execution of some code the first time the function was called.

6.6.2 Static Data Members

Sometimes you will want to have a universal class variable that is not part of the objects of that class, but rather is associated with the class itself. For example, we may want an object counter, which keeps track of how many objects of some class have been instantiated. This is called **reference counting.** This might be useful if you want to know how many "enemy" opponents are left in a game level, for example. Clearly, each object does not need to "own" a copy of this counter since the count will be the same across all objects. So the variable should not be part of the objects; rather, because we are counting objects of a particular class, and we only need one counter, it makes sense that the counter should be "owned" by the class itself. We can express this kind of class variable by prefixing its declaration with the `static` keyword:

```
class Enemy
{
public:
	Enemy();
	~Enemy();

	// ...

private:
	static int NUM_ENEMY_OBJECTS;
};
```

```
// Special syntax to initialize a class variable.
// We have the variable type, followed by the class
// name, followed by the identifier name, followed by
// assignment.
int Enemy::NUM_ENEMY_OBJECTS = 0;
```

Because this is a class variable, all object instances of that class can still access this universal variable (but there is only one; that is, each object does *not* have its "own"). To continue with our reference counting example, every time an object is created we want to increment `NUM_ENEMY_OBJECTS`, and every time an object is destroyed we want to decrement `NUM_ENEMY_OBJECTS`. In this way, `NUM_ENEMY_OBJECTS` will store how many objects are "alive." To do this, the following lines are added to the class constructor and destructor:

```
Enemy::Enemy()
{
      // The constructor was called which implies an object
      // is being created, so increment the count.
      ++NUM_ENEMY_OBJECTS;
}

Enemy::~Enemy()
{
      // The destructor was called which implies an object
      // is being destroyed, so decrement the count.
      --NUM_ENEMY_OBJECTS;
}
```

Recall that the constructor function is called automatically when an object is created and the destructor is called automatically when an object is destroyed. Thus, we have implemented the required functionality we seek—the reference count is incremented when objects are created and it is decremented when objects are destroyed.

6.6.3 Static Methods

We now have a static class variable `NUM_ENEMY_OBJECTS`, which stores the number of objects of class `Enemy` which currently exist. However, that variable is private and cannot be accessed by anything except instances of that class. To remedy this, a public static accessor method is created called `GetEnemyObjectCount`.

```
class Enemy
{
public:
      Enemy();
      ~Enemy();

      // ...

      static int GetEnemyObjectCount();
private:
	static int NUM_ENEMY_OBJECTS;
};

int Enemy::GetEnemyObjectCount()
{
	return NUM_ENEMY_OBJECTS;
}
```

To access a public static class member from outside the class, the class name is used, followed by the scope resolution operator, followed by the static identifier. Continuing with the `Enemy` class example, we would get a copy of `NUM_ENEMY_OBJECTS` like so:

```
int cnt = Enemy::GetEnemyObjectCount();

cout << "Num enemies = " << cnt << endl;
```

Observe how we can access the static method without an object—we access it directly through the class.

> **Note:** Because a static method is not associated with any particular object instance, but rather the class itself, it does not have the hidden `this` pointer parameter. Because there is no `this` pointer, there are no data members which can be accessed. Therefore, static methods can only access static class variables.

6.7 Namespaces

In the first chapter of this text we made the analogy that as folders are used to organize groups of related files and prevent file name clashes, namespaces are used to organize groups of related code and prevent code name clashes. At this point in your programming career, the possibility of name clashes may not be apparent. For instance, you might say that you will ensure that you do not give two things the same name, thereby avoiding name clashes altogether. This ideology is reasonable on a small development scale. However, once you begin large-scale developments where you are working on a team with dozens of programmers and where you are using several third party C++ libraries, the possibility of name clashes increases dramatically. For example, when developing a 3D game (using polygons for the graphics) you may need to define a class called `Polygon`, but the Win32 API (a C/C++ library used for developing Windows applications) already defines a function called `Polygon`. Thus the name `Polygon` becomes ambiguous. Does it refer to the function name or the class name? This is a name clash.

The solution to this problem is a namespace. Since our `Polygon` concerns 3D graphics we might put it in a `gfx3D` namespace. The following code snippet shows the syntax for creating a namespace and adding a class to it:

```
namespace gfx3D
{
	class Polygon
	{
		// ...
	};
}
```

To add something to a namespace, we must define it (for classes) or declare it (variables and functions) inside the namespace block. Note that we can add members to a particular namespace across source code files. For example, the following shows the addition of a class, function, and variable to the `gfx3D` namespace from three separate files:

```
// Polygon.h
namespace gfx3D
{
	class Polygon
	{
		// ...
	};
}

// Area.h
#include "Polygon.h"

namespace gfx3D
{
	float Area(Polygon& p);
}

// Variable.h

namespace gfx3D
{
	const float PI = 3.14f;
}
```

To use the classes, functions, or variables, which exist in a namespace, the namespace name must be specified, followed by the scope resolution operator, followed by the identifier. Examples:

```
cout << gfx3D::PI << endl;

gfx3D::Polygon poly;

float f = gfx3D::Area(poly);
```

The class `Polygon` would no longer clash with the Win32 API function called `Polygon`, because the class is actually referred to as `gfx3D::Polygon`, which is a separate name. Note that this is the same idea as when we had to prefix things in the standard library with `std::`. Just as we used the `using` clause for the `std` namespace, we can use it with our namespaces as well, thereby bypassing the need to specify the namespace prefix:

```
using namespace gfx3D;

...

cout << PI << endl;
Polygon poly;
float f = Area(poly);
```

However, doing this circumvents the whole point of the namespace to begin with. Once we add `using namespace gfx3D`, the `Polygon` class becomes ambiguous with the Win32 API function `Polygon`, because we no longer specify the `gfx3D` part which enabled the compiler to distinguish between the two. Therefore, even though we specify "`using namespace std`" in the example programs for convenience, in general practice, it is not a good idea to use the `using` clause unless you are confident that it will not cause trouble (e.g. you are sure that you will only be using the one namespace).

6.7.1 Variations of the "using" Clause

Sometimes we only want to use some objects in a namespace. For example, instead of "using" the entire `std` namespace, we can specify only certain objects in it. Consider the following trivial program:

```
#include <iostream>

using std::cout; // using cout
using std::endl; // using end

int main()
{
	cout << "Hello, world!" << endl;
}
```

The only two standard objects we use are `std::cout` and `std::endl`. Thus we do not need to "use" the entire `std` namespace. Consequently, we specify with the `using` clause that we will only be using these two objects:

```
using std::cout; // using cout
using std::endl; // using end
```

6.8 Enumerated Types

Sometimes we will want to create a variable that can only be assigned a few select values. For example, a variable of type DAY should only be assigned Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, or Saturday—any other value would not make sense for a DAY type. Another example would be a MONTH type. Only a select pool of values can be assigned to a month. To facilitate types such as these naturally in the language, C++ supports enumerated types. An enumerated type is created with the enum keyword. Here is an example using DAY:

```
enum DAY
{
      Sunday,
      Monday,
      Tuesday,
      Wednesday,
      Thursday,
      Friday,
      Saturday
};
```

To instantiate a DAY variable, we write:

```
DAY day1;
```

We can only assign to any DAY variable a member specified in the DAY enumerated listing; namely, Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday.

```
day1 = Monday;

DAY day2 = Friday;

DAY day3 = 15; // Error: cannot convert from 'int' to 'DAY'
```

As an implementation detail, an enumerated type is actually represented as an integer internally, and so each enumerated value can be associated with an integer:

```
enum DAY
{
      Sunday    = 0,
      Monday    = 1,
      Tuesday   = 2,
      Wednesday = 3,
      Thursday  = 4,
      Friday    = 5,
      Saturday  = 6
};
```

Consequently, you can cast an enumerated type to an integer and vice versa. However, you must be careful that the cast is valid. For example, there is no 8th day, so casting 8 to a `DAY` would be invalid. In the following code snippet, we initialize an array of seven `DAY`s to the days of the week:

```
int main()
{
	DAY days[7];

	for(int i = 0; i < 7; ++i)
	{
		days[i] = (DAY)i;
	}
}
```

6.9 Summary

1. A `char` variable can represent a character, so we can represent a string with an array of `chars`. A string represented as an array of `chars` is called a **c-string.** A c-string that ends with a null character ('\0') is called a **null-terminating string.** The end is marked as such because the ability to figure out when a string ends is important when navigating the elements of the string.

2. `std::string` is a class that uses `char` arrays internally. It hides any dynamic memory that might need to be allocated behind the scenes and it provides many methods, which turn out to be clearer and more convenient to use than the standard library c-string functions.

3. Member functions are similar to non-member functions. The difference is that a hidden pointer to the object which invoked the method is passed into a hidden parameter of the function. Thus, the hidden pointer parameter, which is given the name `this`, points to the object which invoked the method. In this way, the member function can access the data members of the object that called it via the `this` pointer.

4. The `static` keyword has several meanings, which depend upon context. A `static` function variable is initialized once at the program start, persists across function calls, and is not destroyed until the program ends. A `static` variable declared inside class is a universal class variable that is not part of the objects of that class, but rather is associated with the class name itself. A method can be made `static` as well. A `static` method is not associated with any particular object instance, and therefore does not have a hidden `this` pointer parameter; rather, a `static` method is associated with the class name. We refer to class (`static`) variables and method using the scope resolution operator: *className::identifier*.

5. Namespaces are used to organize groups of related code and prevent code name clashes. You can create your own namespaces using the `namespace` keyword. Any class definitions, function declarations or variable declarations made inside the namespace scope become contained in that namespace.

6. Enumerated types are used when we want to create a variable that can only be assigned a few select values. For example, a variable of type `DAY` should only be assigned Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, or Saturday—any other value would not make sense for a `DAY` type. An enumerated type can be defined with the `enum` keyword.

6.10 Exercises

For the following exercises you can use c-strings or std::string or a combination of both.

6.10.1 String Reverse

Write a program that does the following:

1. Ask the user to enter up to a line of text and store it in a string *s*.
2. Reverse the string *s*.
3. Output the reversed string to the console window.

Your program output should look like this:

```
Enter a string: Hello, World!
Reversed string = !dlroW ,olleH
Press any key to continue
```

6.10.2 To-Upper

Write a program that does the following:

1. Ask the user to enter up to a line of text and store it in a string *s*.
2. Transform each alphabetical character in *s* into its uppercase form. If a character is not an alphabetical character—do not modify it.
3. Output the uppercase string to the console window.

Your program output should look like this:

```
Enter a string: Hello, World!
Uppercase string = HELLO, WORLD!
Press any key to continue
```

6.10.3 To-Lower

Write a program that does the following:

1. Ask the user to enter up to a line of text and store it in a string *s*.
2. Transform each alphabetical character in *s* into its lowercase form. If a character is not an alphabetical character—do not modify it.
3. Output the lowercase string to the console window.

Your program output should look like this:

```
Enter a string: Hello, World!
Lowercase string = hello, world!
Press any key to continue
```

6.10.4 Palindrome

Dictionary.com defines a palindrome as follows: "A word, phrase, verse, or sentence that reads the same backward or forward. For example: *A man, a plan, a canal, Panama!*" For our purposes, we will generalize and say that a palindrome can be any string that reads the same backwards or forwards and does not have to form a real word or sentence. Thus, some simpler examples may be:

"abcdedcba"
"C++C"
"ProgrammingnimmargorP"

Write a program that does the following:

1. Asks the user to enter up to a line of text and store it in a string *s*.
2. Tests if the string *s* is a palindrome.
3. If *s* is a palindrome then output "*s* is a palindrome" else output "*s* is not a palindrome."

Your program output should look like this:

```
Enter a string: Hello, World!
Hello, World! is not a palindrome
Press any key to continue
```

Another sample output:

```
Enter a string: abcdedcba
abcdedcba is a palindrome
Press any key to continue
```

Chapter 7

Operator Overloading

Introduction

In the previous chapter, we saw that we could access a character in a `std::string` object using the bracket operator ([]). Moreover, we also saw that we could add two `std::strings` together using the addition operator (+) and that we could use the relational operators (==, `!=`, <, etc) with `std::string` objects as well. So it appears that we can use (some) C++ operators with `std::string`. We may assume that perhaps these operators are defined for every class. However, a quick test verifies that this is not the case:

```
class Fraction
{
public:
      Fraction();
      Fraction(float num, float den);
      float mNumerator;
      float mDenominator;
};

Fraction::Fraction()
{
      mNumerator   = 0.0f;
      mDenominator = 1.0f;
}

Fraction::Fraction(float num, float den)
{
      mNumerator   = num;
      mDenominator = den;
}

int main()
{
      Fraction f(1.0f, 2.0f);
      Fraction g(3.0f, 4.0f);

      Fraction p = f * g;
      bool b = f > g;
}
```

The above code yields the errors:

C2676: binary '*' : 'Fraction' does not define this operator or a conversion to a type acceptable to the predefined operator
C2676: binary '>' : 'Fraction' does not define this operator or a conversion to a type acceptable to the predefined operator.

Why can `std::string` use the C++ operators but we cannot? We actually can, but the functionality is not available by default. We have to define (or *overload*) these C++ operators in our class definitions. These overloaded operators are defined similarly to regular class methods, and they specify what the operator does in the context of the particular class in which it is being overloaded. For example, what does it mean to multiply two `Fractions`? We know from basic math that to multiply two fractions we multiply the numerators and the denominators. Thus, we would overload the multiplication operator for `Fraction` and give an implementation like so:

```
Fraction Fraction::operator *(const Fraction& rhs)
{
      Fraction P;
      P.mNumerator   = mNumerator * rhs.mNumerator;
      P.mDenominator = mDenominator * rhs.mDenominator;
      return P; // return the fraction product.
}
```

With operator overloading we can make our user-defined types behave very similarly to the C++ built-in types (e.g., `float`, `int`). Indeed, one of the primary design goals of C++ was for user-defined types to behave similarly to built-in C++ types.

The rest of this chapter describes operator overloading in detail by looking at two different class examples. The first class which we build will model a mathematical vector, which is an essential tool for 3D computer graphics and 3D game programming.

However, before proceeding, note that operator overloading is not recommended for every class; you should only use operator overloading if it makes the class easier and more natural to work with. Do not overload operators and implement them with non-intuitive and confusing behavior. To illustrate an extreme case: You should not overload the '+' operator such that it performs a subtraction operation, as this kind of behavior would be very confusing.

Chapter Objectives

- Learn how to overload the arithmetic operators.
- Discover how to overload the relational operators.
- Overload the conversion operators.
- Understand the difference between deep copies and shallow copies.
- Find out how to overload the assignment operator and copy constructor to perform deep copies.

7.1 Vector Mathematics

In this section we discuss the mathematics of vectors, in order to understand what they are (the data) and what we can do with them (the methods). Clearly we need to know this information if we are to model a vector in C++ with a class.

In 3D computer graphics programming (and many other fields for that matter), you will use a *vector* to model quantities that consist of a magnitude and a direction. Examples of such quantities are physical forces (forces are applied in a certain direction and have a strength or magnitude associated with them), and velocities (speed and direction).

Geometrically, we represent a vector as a directed line segment—Figure 7.1. The direction of the line segment describes the vector direction and the length of the line segment describes the magnitude of the vector.

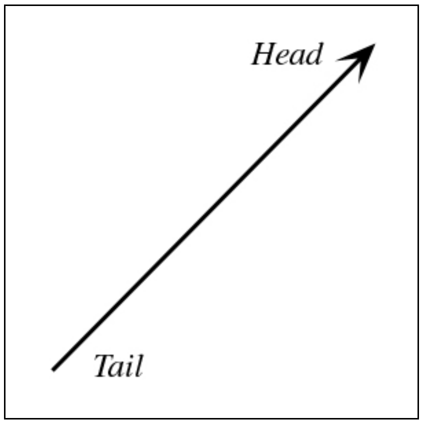


Figure7.1: Geometric interpretation of a vector.

Note that vectors describe a direction and magnitude, but they say nothing about location. Therefore, we are free to choose a convenient location from which they originate. In particular, for solving problems, it is convenient to define all vectors such that their "tails" originate from the origin of the working coordinate system, as seen in Figure 7.2.

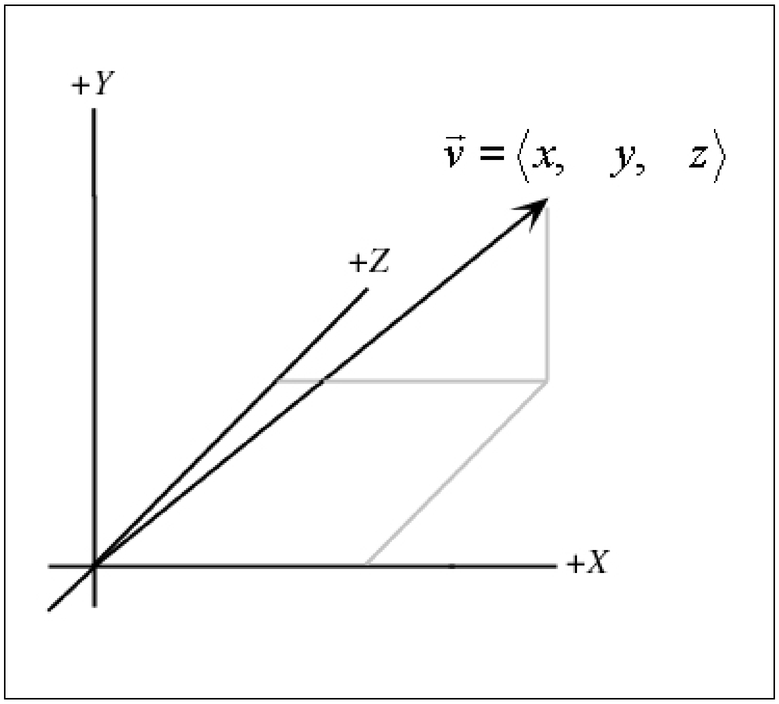


Figure 7.2: A vector with its tail fixed at the origin. Observe that by specifying the coordinates of the vector's tail we can control its magnitude and direction.

Thus we can describe a vector analytically by merely specifying the coordinates of its "head." This motivates the following structural representation of a 3D vector:

```
class Vector3
{
    // Methods...

    // Data
    float mX;
    float mY;
    float mZ;
};
```

At first glance, it is easy to confuse a vector with a point, as they both are specified via three coordinate components. However, recall that vector coordinates are interpreted differently than point coordinates; specifically, vector coordinates indicate the end point to which a directed line segment (originating from the origin) connects. Conversely, point coordinates specify a location in space and say nothing about directions and/or magnitudes.

What follows is a description of important vector operations. For now, we do not need to worry about the detailed understanding of this math or why it is this way; rather, our goal is simply to understand the vector operation descriptions well enough to implement C++ methods which perform these operations.

> **Note:** The *Game Mathematics* and *Graphics Programming with DirectX 9 Part I* courses at Game Institute explain vectors in detail.

Throughout this discussion we restrict ourselves to 3D vectors. Let $\vec{u} = \langle u_x,\ u_y,\ u_z \rangle$ *and* $\vec{v} = \langle v_x,\ v_y,\ v_z \rangle$ *be any vectors in 3-space, and let* $\vec{p} = \langle 1,\ 2,\ 3 \rangle$ *and* $\vec{q} = \langle 5,\ -3,\ 1 \rangle$.

<u>Vector Equality</u>:

Two vectors are equal if and only if their corresponding components are equal. That is, $\vec{u} = \vec{v}$ if and only if $u_x = v_x, u_y = v_y$, and $u_z = v_z$.

<u>Vector Addition</u>:

The sum of two vectors is found by adding corresponding components:

$$\vec{u} + \vec{v} = \langle u_x,\ u_y,\ u_z \rangle + \langle v_x,\ v_y,\ v_z \rangle = \langle u_x + v_x,\ u_y + v_y,\ u_z + v_z \rangle .$$

Example:

$$\vec{p} + \vec{q} = \langle 1,\ 2,\ 3 \rangle + \langle 5,\ -3,\ 1 \rangle = \langle 1+5,\ 2-3,\ 3+1 \rangle = \langle 6,\ -1,\ 4 \rangle .$$

Geometrically, we add two vectors $\vec{u}+\vec{v}$ by parallel translating $\vec{v}$ so that its tail is coincident with the head of $\vec{u}$ and then the sum $\vec{u}+\vec{v}$ is the vector that originates at the tail of $\vec{u}$ and terminates at the head of $\vec{v}$. Figure 7.3 illustrates.

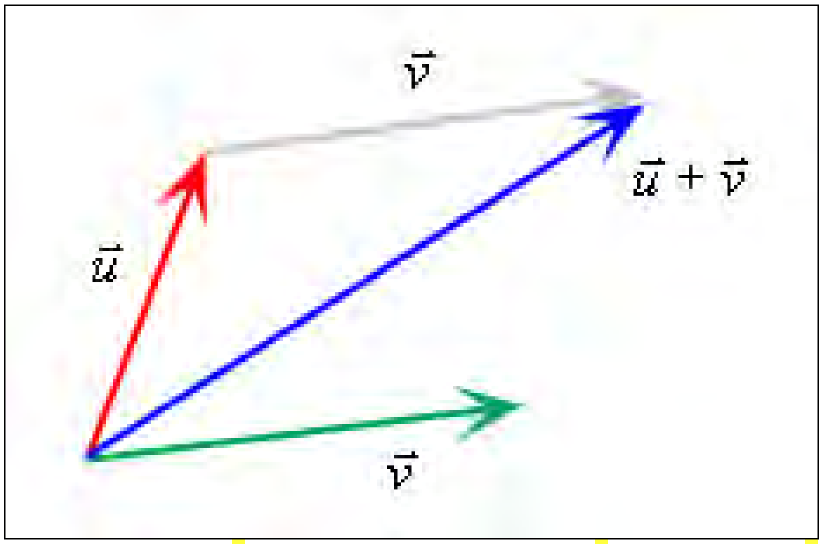


Figure 7.3: Vector addition. We translate $\vec{v}$ so that its tail coincides with $\vec{u}$. Then the sum $\vec{u}+\vec{v}$ is the vector from the tail of $\vec{u}$ to the head of translated $\vec{v}$.

A key observation to note regarding parallel translation of a vector is that the length and direction of the vector is preserved; that is, translating a vector parallel to itself does not change its properties and therefore it is legal to parallel transport them around for visualization.

<u>Vector Subtraction</u>:

The difference between two vectors is found by subtracting corresponding components:

$$\vec{u}-\vec{v}=\langle u_x,\quad u_y,\quad u_z\rangle-\langle v_x,\quad v_y,\quad v_z\rangle=\langle u_x-v_x,\quad u_y-v_y,\quad u_z-v_z\rangle.$$

Example:

$$\vec{p}-\vec{q}=\langle 1,\quad 2,\quad 3\rangle-\langle 5,\quad -3,\quad 1\rangle=\langle 1-5,\quad 2-(-3),\quad 3-1\rangle=\langle -4,\quad 5,\quad 2\rangle.$$

Geometrically, we can view the difference $\vec{u}-\vec{v}$ as the vector that originates from the head of $\vec{v}$ and terminates at the head of $\vec{u}$. Figure 7.4 illustrates.

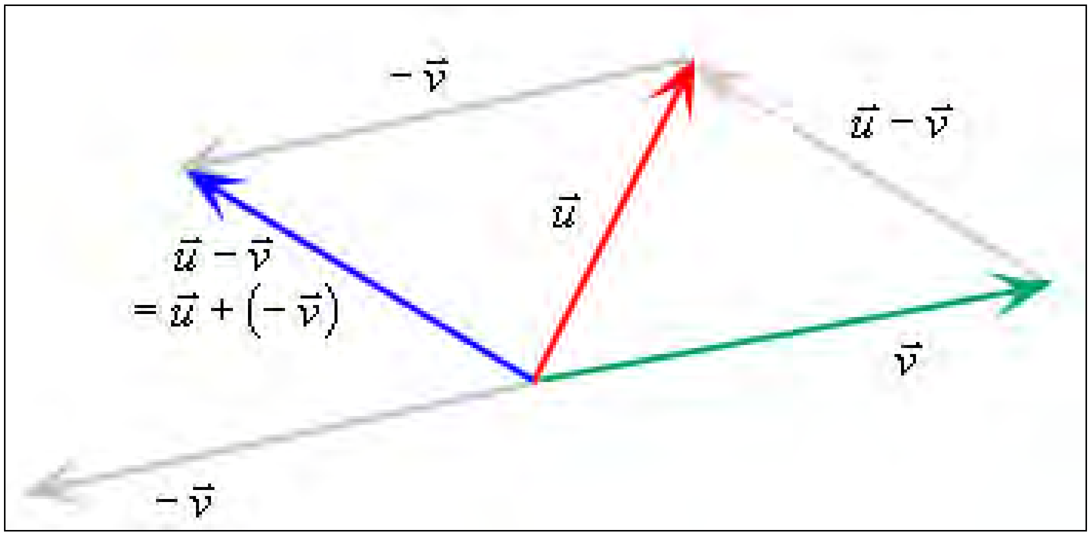


Figure 7.4: Vector subtraction. We view vector subtraction as the sum $\vec{u}-\vec{v}=\vec{u}+(-\vec{v})$. So first we negate $\vec{v}$ and then translate $-\vec{v}$ so that its tail coincides with $\vec{u}$. Then $\vec{u}+(-\vec{v})$ is the vector originating from the tail of $\vec{u}$ and terminating at the head of $-\vec{v}$.

Observe that subtraction can be viewed as an addition; that is, $\vec{u}-\vec{v}=\vec{u}+(-\vec{v})$, where negating a vector flips its direction.

<u>Scalar Multiplication</u>:

A vector can be multiplied by a scalar, which modifies the magnitude of the vector but not its direction. To multiply a vector by a scalar we multiply each vector component by the scalar:

$$k\vec{v}=k\langle v_1,\quad v_2,\quad v_3\rangle=\langle kv_1,\quad kv_2,\quad kv_3\rangle$$

Example:

$$3\vec{p}=3\langle 1,\quad 2,\quad 3\rangle=\langle 3(1),\quad 3(2),\quad 3(3)\rangle=\langle 3,\quad 6,\quad 9\rangle$$

As the name implies, scalar multiplication scales the length of a vector. Figure 7.5 shows some examples.

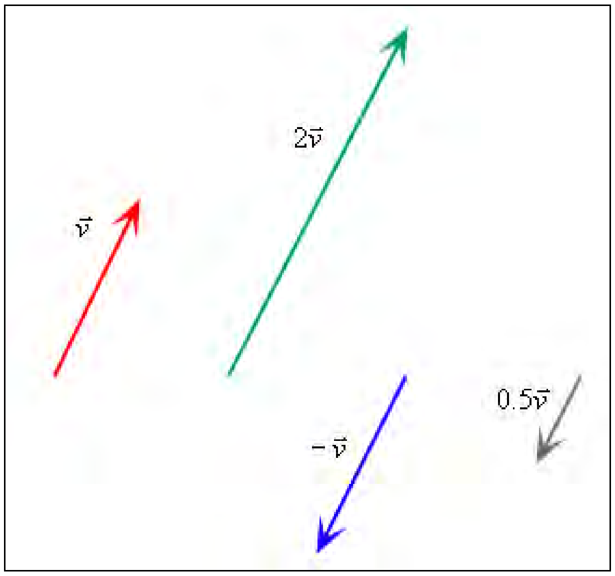


Figure 7.5: Scalar multiplication. Multiplying a vector by a scalar changes the magnitude of the vector. A negative scalar flips the direction.

<u>Vector Magnitude</u>:

We use a double vertical bar notation to denote the magnitude of a vector; for example, the magnitude of the vector $\vec{v}$ is denoted as $\|\vec{v}\|$. The magnitude of a vector is found by computing the distance from the tail of a vector to its head:

$$\|\vec{v}\| = \sqrt{{v_1}^2 + {v_2}^2 + {v_3}^2}$$

Example:

$$\|\vec{p}\| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{14}$$

Geometrically, the magnitude of a vector is its length—see Figure 7.6.

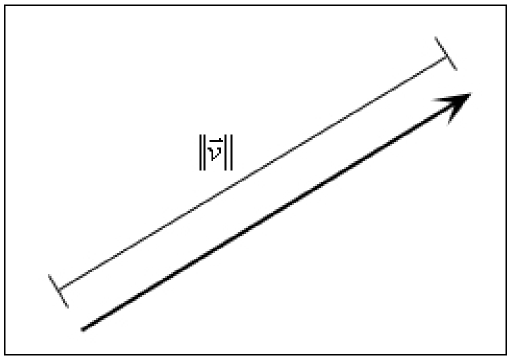


Figure 7.6: Vector magnitude. The magnitude of a vector is its length.

<u>Normalizing a Vector</u>:

Normalizing a vector makes its length equal to 1.0. We call this a *unit vector* and denote it by putting a "hat" on it (e.g., $\hat{\vec{v}}$). We normalize a vector by scalar multiplying the vector by the reciprocal of its magnitude:

$$\hat{\vec{v}} = \vec{v} / \|\vec{v}\|$$

Example:

$$\hat{\vec{p}} = \vec{p} / \|\vec{p}\| = 1/\sqrt{14}\langle 1, \quad 2, \quad 3\rangle = \left\langle 1/\sqrt{14}, \quad 2/\sqrt{14} \quad 3/\sqrt{14}\right\rangle.$$

<u>The Dot Product</u>:

The dot product of two vectors is the sum of the products of corresponding components:

$$\vec{u} \cdot \vec{v} = \left\langle u_x, \quad u_y, \quad u_z\right\rangle \cdot \left\langle v_x, \quad v_y, \quad v_z\right\rangle = u_x v_x + u_y v_y + u_z v_z$$

It can be proved that $\vec{u} \cdot \vec{v} = \|u\| \cdot \|v\| \cos\theta$, where θ is the angle between $\vec{u}$ and $\vec{v}$. Consequently, a dot product can be useful for finding the angle between two vectors.

Example:

$$\vec{p} \cdot \vec{q} = \langle 1, \quad 2, \quad 3\rangle \cdot \langle 5, \quad -3, \quad 1\rangle = 1(5) + 2(-3) + 3(1) = 5 - 6 + 3 = 2$$

Observe that this vector product returns a scalar—not a vector.

The dot product of a vector $\vec{v}$ with a unit vector $\hat{\vec{n}}$ evaluates to the *magnitude* of the projection of $\vec{v}$ onto $\hat{\vec{n}}$, as Figure 7.7 shows.

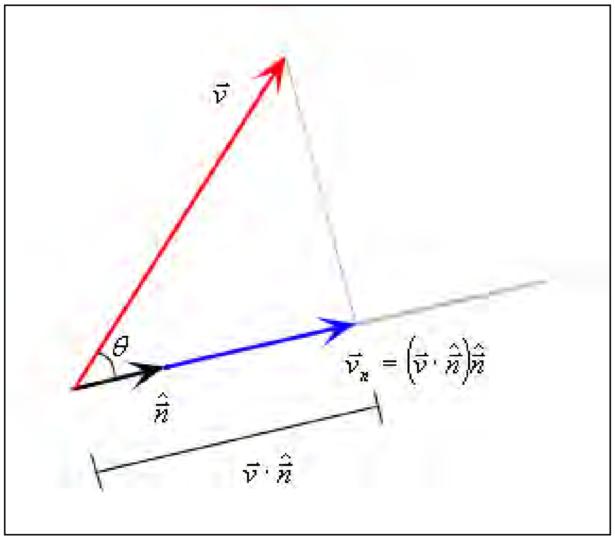

$\vec{v}$
θ
$\hat{\vec{n}}$
$\vec{v}_n = (\vec{v}\cdot\hat{\vec{n}})\hat{\vec{n}}$
$\vec{v}\cdot\hat{\vec{n}}$

Figure 7.7: The dot product of a vector $\vec{v}$ with a unit vector $\hat{\vec{n}}$ evaluates to the *magnitude* of the projection of $\vec{v}$ onto $\hat{\vec{n}}$. We can get the actual projected vector $\vec{v}_n$ by scaling $\hat{\vec{n}}$ by that magnitude.

Given the magnitude of the projection of $\vec{v}$ onto $\hat{\vec{n}}$, the actual projected vector is: $\vec{v}_n = (\vec{v}\cdot\hat{\vec{n}})\hat{\vec{n}}$, which makes sense: $\vec{v}\cdot\hat{\vec{n}}$ returns the magnitude of the projection, and $\hat{\vec{n}}$ is the unit vector along which the projection lies. Therefore, to get the projected vector we simply scale the unit vector $\hat{\vec{n}}$ by the magnitude of the projection.

Sometimes we will want the vector $\vec{v}_\perp$ perpendicular to the projected vector $\vec{v}_n$ (Figure 7.8). Using our geometric interpretation of vector subtraction we see that $\vec{v}_\perp = \vec{v} - \vec{v}_n$. But this then implies $\vec{v} = \vec{v}_n + \vec{v}_\perp$, which means a vector can be written as a sum of its perpendicular components.

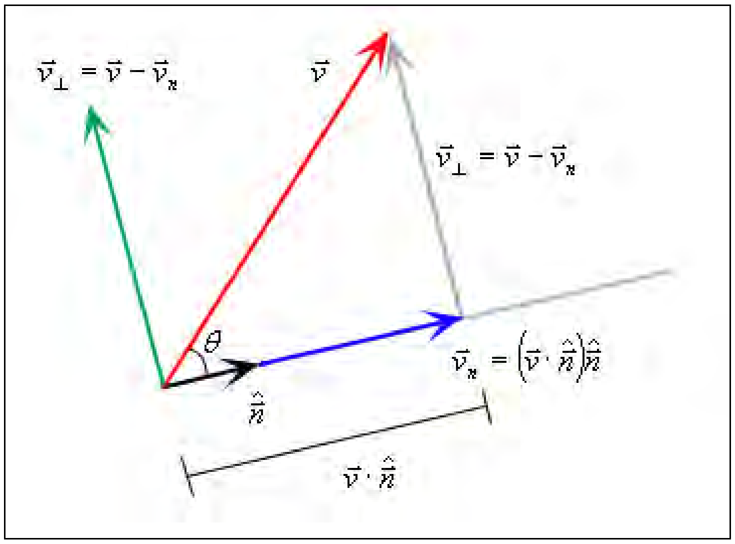

$\vec{v}_\perp = \vec{v} - \vec{v}_n$
$\vec{v}$
$\vec{v}_\perp = \vec{v} - \vec{v}_n$
θ
$\hat{\vec{n}}$
$\vec{v}_n = (\vec{v}\cdot\hat{\vec{n}})\hat{\vec{n}}$
$\vec{v}\cdot\hat{\vec{n}}$

Figure 7.8: Finding the vector perpendicular to the vector projected on $\hat{\vec{n}}$.

7.2 A Vector Class

Our goal now is to design a class that represents a 3D vector. We know how to describe a 3D vector (with an ordered triplet of coordinates) and we also know what operators are defined for vectors (from the preceding section); that is, what things we can do with vectors (methods). Based on this, we define the following class:

```
// Vector3.h

#ifndef VECTOR3_H
#define VECTOR3_H

#include <iostream>

class Vector3
{
public:

      Vector3();
      Vector3(float coords[3]);
      Vector3(float x, float y, float z);


      bool equals(const Vector3& rhs);
      Vector3 add(const Vector3& rhs);
      Vector3 sub(const Vector3& rhs);
      Vector3 mul(float scalar);
      float length();
      void normalize();
      float dot(const Vector3& rhs);

      float* toFloatArray();

      void print();

      void input();

      float mX;
      float mY;
      float mZ;
};
#endif // VECTOR3_H
```

Note: Why do we keep the data public even though the general rule is that data should always be private? There are two reasons. The first is practical: typically, vector components need to be accessed quite frequently by outside code, so it would be cumbersome to have to go through accessor functions. Second, there is no real data to protect; that is, a vector can take on any value for its components, so there is nothing we would want to restrict.

The data members are obvious—a coordinate value for each axis, which thereby describes a 3D vector. The methods are equally obvious—most come straight from our discussion of vector operations from the previous section. But how should we implement these methods? Let us now take a look at the implementation of these methods one-by-one.

7.2.1 Constructors

We provide three constructors. The first one, with no parameters, creates a default vector, which we define to be the **null vector.** The null vector is defined to have zero for all of its components. The second constructor constructs a vector based on a three-element input array. Element [0] will contain the x-component; element [1] will contain the y-component; and element [2] will contain the z-component. The last constructor directly constructs a vector out of the three passed-in components. The implementations for these constructors are trivial:

```
Vector3::Vector3()
{
      mX = 0.0f;
      mY = 0.0f;
      mZ = 0.0f;
}

Vector3::Vector3(float coords[3])
{
      mX = coords[0];
      mY = coords[1];
      mZ = coords[2];
}

Vector3::Vector3(float x, float y, float z)
{
      mX = x;
      mY = y;
      mZ = z;
}
```

7.2.2 Equality

The next method we specified was the `equals` method. The equals method returns `true` if the method which calls vector (`this` vector) is equal to the vector passed into the parameter; otherwise, it returns `false`. Recall that two vectors are equal if and only if their corresponding components are equal.

```
bool Vector3::equals(const Vector3& rhs)
{
      // Return true if the corresponding components are equal.
      return
            mX == rhs.mX &&
            mY == rhs.mY &&
            mZ == rhs.mZ;
}
```

7.2.3 Addition and Subtraction

We next implement two methods to perform vector addition and subtraction. Recall that we add two vectors by adding corresponding components, and that we subtract two vectors by subtracting corresponding components. The following implementations do exactly that, and return the sum or difference.

```
Vector3 Vector3::add(const Vector3& rhs)
{
	Vector3 sum;
	sum.mX = mX + rhs.mX;
	sum.mY = mY + rhs.mY;
	sum.mZ = mZ + rhs.mZ;

	return sum;
}

Vector3 Vector3::sub(const Vector3& rhs)
{
	Vector3 dif;
	dif.mX = mX - rhs.mX;
	dif.mY = mY - rhs.mY;
	dif.mZ = mZ - rhs.mZ;

	return dif;
}
```

7.2.4 Scalar Multiplication

After the subtraction method we define the `mul` method, which multiplies a scalar by `this` vector (the vector object that invoked the method), and returns the resulting vector. Again, it is straightforward to translate the mathematical computations described in Section 7.1 into code. To refresh your memory, to multiply a vector with a scalar we simply multiply each vector component with the scalar:

```
 Vector3 Vector3::mul(float scalar)
{
	Vector3 p;
	p.mX = mX * scalar;
	p.mY = mY * scalar;
	p.mZ = mZ * scalar;

	return p;
}
```

7.2.5 Length

The `length` method is responsible for returning the length (or magnitude) of the calling vector. Translating the mathematic formula $\|\vec{v}\| = \sqrt{v_1{}^2 + v_2{}^2 + v_3{}^2}$ into code we have:

```
float Vector3::length()
{
	return sqrtf(mX*mX + mY*mY + mZ*mZ);
}
```

7.2.6 Normalization

Writing a method to normalize the calling vector is as equally easy. Recall that normalizing a vector makes its length equal to 1.0, and that we normalize a vector by scalar multiplying the vector by the reciprocal of its magnitude:

$$\hat{\vec{v}} = \vec{v} / \|\vec{v}\|$$

Translating this math into code yields:

```
void Vector3::normalize()
{
	// Get 'this' vector's length.
	float len = length();

	// Divide each component by the length.
	mX /= len;
	mY /= len;
	mZ /= len;
}
```

7.2.7 The Dot Product

The last method, which is mathematical in nature, implements the dot product. Translating the following mathematical formula results in the code seen below:

$$\vec{u} \cdot \vec{v} = \langle u_x, \quad u_y, \quad u_z \rangle \cdot \langle v_x, \quad v_y, \quad v_z \rangle = u_x v_x + u_y v_y + u_z v_z$$

```
float Vector3::dot(const Vector3& rhs)
{
	float dotP = mX*rhs.mX + mY*rhs.mY + mZ*rhs.mZ;
	return dotP;
}
```

7.2.8 Conversion to `float` Array

The conversion method `toFloatArray` does not correspond to a mathematical vector operation. Rather, it returns a *pointer* to the three-element `float` representation of the calling object (`this`). Why would we ever want to convert our `Vector3` representation to a three-element `float` array representation? A good example might be if we were using the OpenGL 3D rendering library. This library has no idea about `Vector3`, and instead expects vector parameters to be passed in using a three-element `float` array representation. Providing a method to convert our `Vector3` object to a three-element `float` array would allow us to use the `Vector3` class seamlessly with OpenGL.

The implementation of this function might not be obvious.

```
float* Vector3::toFloatArray()
{
      return &mX;
}
```

This code returns the address of the first component of `this` vector. However, we must remember that the memory of class objects is contiguous, just like we saw with arrays.

```
float mX;
float mY;
float mZ;
```

The memory of `mY` comes directly after `mX`, and the memory for `mZ` comes directly after `mY`. The above memory layout is equivalent to:

```
float v[3];
```

Thus, by getting a pointer to `mX`, we are implicitly getting a pointer to the first element in a three-element array, which represents the vector components. We can now access the x-, y-, and z-components using the array bracket operator:

```
Vector3 w(-5.0f, 2.0f, 0.0f);
float* wArray = w.toFloatArray();

// wArray[0] == w.x == -5.0f
// wArray[1] == w.y ==  2.0f
// wArray[2] == w.z ==  0.0f
```

7.2.9 Printing

The print method is responsible for displaying the calling vector to the console window:

```
void Vector3::print()
{
	cout << "<" << mX << ", " << mY << ", " << mZ << "> \n";
}
```

7.2.10 Inputting

Finally, the last method is used to initialize a vector based on user input from the keyboard. In other words, it prompts the user to enter in the components of a vector one-by-one.

```
void Vector3::input()
{
	cout << "Enter x: ";
	cin >> mX;
	cout << "Enter y: ";
	cin >> mY;
	cout << "Enter z: ";
	cin >> mZ;
}
```

7.2.11 Example: `Vector3` in Action

Let us now look at a driver program, which uses our `Vector3` class.

Program 7.1: Using the `Vector3` class.

```
// main.cpp

#include "Vector3.h"
#include <iostream>
using namespace std;

int main()
{
	// Part 1: Construct three vectors.
	float coords[3] = {1.0f, 2.0f, 3.0f};
	Vector3 u;
	Vector3 v(coords);
	Vector3 w(-5.0f, 2.0f, 0.0f);

	// Part 2: Print the three vectors.
	cout << "u = ";
	u.print();
```

```
      cout << "v = ";
      v.print();
      cout << "w = ";
      w.print();
      cout << endl;

      // Part3: u = v + w
      u = v.add(w);
      cout << "v.add(w) = ";
      u.print();
      cout << endl;

      // Part 4: v = v / ||v||
      v.normalize();
      cout << "unit v = ";
      v.print();
      cout << "v.length() = " << v.length() << endl;
      cout << endl;

      // Part 5: dotP = u * w
      float dotP = u.dot(w);
      cout << "u.dot(w) = " << dotP;

      // Part 6: Convert to array representation.
      float* vArray = v.toFloatArray();

      // Print out each element and verify it matches the
      // components of v.
      cout <<
            "[0] = " << vArray[0] << ", "
            "[1] = " << vArray[1] << ", "
            "[2] = " << vArray[2] << endl;
      cout << endl;

      // Part 7: Create a new vector and have user specify its
      // components, then print the vector.
      cout << "Input vector..." << endl;
      Vector3 m;
      m.input();
      cout << "m = ";
      m.print();
}
```

Program 7.1 Output

```
u = <0, 0, 0>
v = <1, 2, 3>
w = <-5, 2, 0>

v.add(w) = <-4, 4, 3>

unit v = <0.267261, 0.534522, 0.801784>
v.length() = 1

u.dot(w) = 28[0] = 0.267261, [1] = 0.534522, [2] = 0.801784
```

```
Input vector...
Enter x: 9
Enter y: 8
Enter z: 7
m = <9, 8, 7>
Press any key to continue
```

The code in Program 7.1 is pretty straightforward, so we will only briefly summarize it. In Part 1, we have:

```
float coords[3] = {1.0f, 2.0f, 3.0f};
Vector3 u;
Vector3 v(coords);
Vector3 w(-5.0f, 2.0f, 0.0f);
```

Here, we construct three different vectors using the different constructors we have defined. The vector `u` takes no parameters and is constructed with the default constructor. The second vector `v` uses the array constructor; that is, we pass a three-element array where element [0] specifies the x-component, element [1] specifies the y-component, and element [2] specifies the z-component. Lastly, the vector `w` is constructed using the constructor in which we can directly specify the x-, y-, and z-components.

Part 2 of the code simply prints each vector to the console window. In this way, we can check the program output to verify that the vectors were indeed constructed with the values we specified.

Part 3 performs an addition operation, and then prints the sum.

```
u = v.add(w);
cout << "v.add(w) = ";
u.print();
cout << endl;
```

In particular, Part 3 computes u = v + w. Observe how the method caller `v` is the "left hand side" of the addition operation, and the argument `w` is the "right hand side" of the addition operation. Also, notice that the `add` method returns the sum, which we store in `u`.

The first key statement in Part 4 is the following:

```
v.normalize();
```

This statement simply makes the vector `v` a unit vector (length equal to one). The second key statement in Part 4 is:

```
cout << "v.length() = " << v.length() << endl;
```

Here we call the `length` function for `v`, which will return the length of `v`. Because `v` was just normalized, the length should be equal to 1. A quick check at the resulting output confirms that the length is indeed one.

Part 5 performs a dot product:

```
float dotP = u.dot(w);
```

In words, this code reads: Take the dot product of `u` and `w` and return the result in `dotP`, or in mathematical symbols, $dotP = \vec{u} \cdot \vec{w}$. As with addition, observe how the method caller `u` is the "left hand side" of the dot product operation, and the argument `w` is the "right hand side" of the dot product operation.

In Part 6 we obtain a `float` pointer to the first element (component) of the vector `v`. With this pointer, which is essentially a pointer to a three-element array, we can access all three components of `v` using the subscript operator.

```
float* vArray = v.toFloatArray();

      // Print out each element and verify it matches the
      // components of v.
      cout <<
            "[0] = " << vArray[0] << ", "
            "[1] = " << vArray[1] << ", "
            "[2] = " << vArray[2] << endl;
      cout << endl;
```

Finally, Part 7 creates a new vector and prompts the user to specify the components via the `input` method. Based on the program's output, we can see the program echoed the values we input, thereby confirming that the `input` method initialized the components correctly.

```
cout << "Input vector..." << endl;
Vector3 m;
m.input();
cout << "m = ";
m.print();
```

7.3 Overloading Arithmetic Operators

In the previous section we defined and implemented a `Vector3` class. However, the class interface is unnatural. Since a `Vector3` object is mathematical in nature and supports mathematical operators, it would be useful if it were possible to add, subtract and multiply vectors using the natural syntax:

```
u = v + w;          // Addition
v = w - u;          // Subtraction
w = v * 10.0f;      // Scalar multiplication
float dotP = u * w; // Dot product
```

This would be used instead of the currently exposed syntax:

```
u = v.add(w);
v = w.sub(u);
w = v.mul(10.0f);
float dotP = u.dot(w);
```

This is where operator overloading comes in. Instead of defining method names for `Vector3`, we will overload C++ operators for `Vector3` objects to perform the desired function. For example, using the + operator will be functionally equivalent to using the `add` method. In this way, we will be able to perform all of our vector operations using a natural mathematical syntax.

7.3.1 Operator Overloading Syntax

Overloading an operator is simple. It is just like defining a method, except that instead of using a method name, the `operator` keyword is used followed by the operator symbol which we are overloading as the method name. For example, we can overload the + operator like so:

```
Vector3 operator+(const Vector3& rhs);
```

We treat the name "operator +" as the method name. When we implement the overloaded operator, we just implement it as we would a regular method with name "operator +":

```
Vector3 Vector3::operator+(const Vector3& rhs)
{
      Vector3 sum;
      sum.mX = mX + rhs.mX;
      sum.mY = mY + rhs.mY;
      sum.mZ = mZ + rhs.mZ;

      return sum;
}
```

This is quite convenient because we want our overloaded + operator to be functionally equivalent to the `add` function. We can simply replace `add` with `operator+`. Now that we overloaded the + operator between two vectors we can now perform vector addition with this syntax:

```
u = v + w;
```

7.3.2 Overloading the Other Arithmetic Operators

Overloading the other mathematical operators is equally easy, especially since we already have the desired functionality already written. We already wrote the code which subtracts two vectors, multiplies a scalar and a vector, and takes a dot product. Thus, all we have to do is replace the method "word" names `sub`, `mul`, and `dot`, with the operator symbol equivalent: - and *:

```
Vector3 operator-(const Vector3& rhs);
Vector3 operator*(float scalar);
float   operator*(const Vector3& rhs);
```

Note: Just as we can overload function names several times using a different function signature, we can overload operators several times using different signatures. Thus we can overload the * operator twice, since we use a different signature. The first, which takes a scalar argument, performs a scalar multiplication. The second version, which takes a `Vector3` argument, performs a dot product.

The implementations to these operator methods are exactly the same as we had before:

```
Vector3 Vector3::operator-(const Vector3& rhs)
{
      Vector3 dif;
      dif.mX = mX - rhs.mX;
      dif.mY = mY - rhs.mY;
      dif.mZ = mZ - rhs.mZ;

      return dif;
}

Vector3 Vector3::operator*(float scalar)
{
      Vector3 p;
      p.mX = mX * scalar;
      p.mY = mY * scalar;
      p.mZ = mZ * scalar;

      return p;
}

float Vector3::operator*(const Vector3& rhs)
{
      float dotP = mX*rhs.mX + mY*rhs.mY + mZ*rhs.mZ;

      return dotP;
}
```

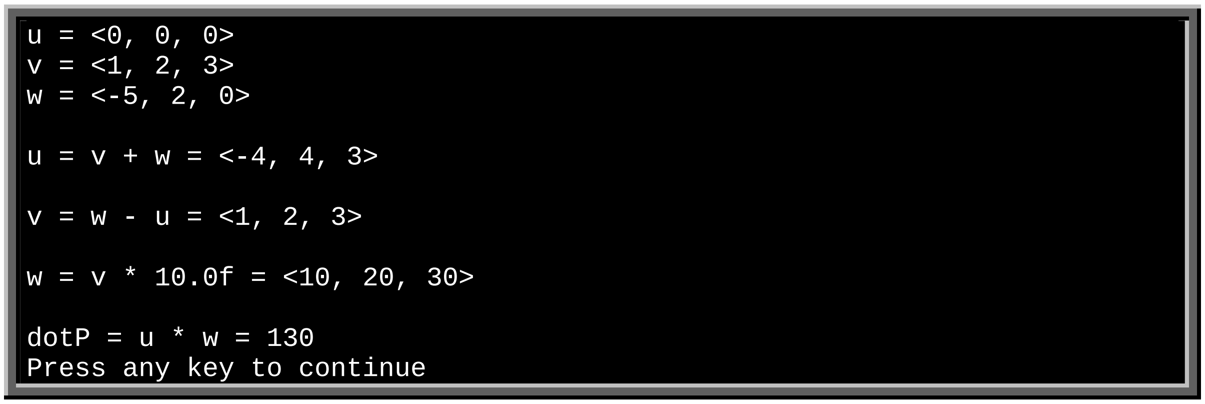
7.3.3 Example using our Overloaded Operators

To show off our new overloaded operators, let us rewrite parts of Program 7.1:

Program 7.2: Overloaded Arithmetic Operators.

```
// main.cpp

#include "Vector3.h"
#include <iostream>
using namespace std;

int main()
{
      // Part 1: Construct three vectors and print.
      float coords[3] = {1.0f, 2.0f, 3.0f};
      Vector3 u;
      Vector3 v(coords);
      Vector3 w(-5.0f, 2.0f, 0.0f);

      cout << "u = ";    u.print();
      cout << "v = ";  v.print();
      cout << "w = ";    w.print();
      cout << endl;

      // Part 2: u = v + w
      u = v + w;
      cout << "u = v + w = ";
      u.print();
      cout << endl;

      // Part 3: subtraction
      v = w - u;
      cout << "v = w - u = ";
      v.print();
      cout << endl;

      // Part 4: w = v * 10.0f
      w = v * 10.0f;
      cout << "w = v * 10.0f = ";
      w.print();
      cout << endl;

      // Part 5: dotP = u * w
      float dotP = u * w;
      cout << "dotP = u * w = " << dotP;
      cout << endl;
}
```

Program 7.2 Output



```
u = <0, 0, 0>
v = <1, 2, 3>
w = <-5, 2, 0>

u = v + w = <-4, 4, 3>

v = w - u = <1, 2, 3>

w = v * 10.0f = <10, 20, 30>

dotP = u * w = 130
Press any key to continue
```

There is not much to discuss here. We have omitted some parts of Program 7.1 and added some new parts. The key point of our new version is how we perform arithmetic operations using the C++ arithmetic symbols +, -, and *, instead of the word name `add`, `sub`, `mul`, and `dot`.

> **Note 1:** We did not overload the division operator (/) in our `Vector3` class because we did not need it. You could overload it and define a meaning to it in your classes. The general syntax would be:
>
> ```
> ReturnType ClassName::operator/(type rhs)
> ```
>
> **Note 2:** We have overloaded the * operator such that we can write `v * 10.0f`, for example. However, equally correct would be `10.0f * v`, but we cannot write this as is because our member function version is setup such that the `Vector3` object is the left hand operand and the `float` scalar is the right hand operand:
>
> ```
> Vector3 Vector3::operator*(float scalar)
> ```
>
> The trick is to make a global `operator*` where the `float` scalar is the first parameter and the `Vector3` object the second parameter:
>
> ```
>  Vector3 operator*(float scalar, Vector3& v);
> ```
>
> In this way, you can maintain an elegant symmetry, and write, `10.0f * v` , as well.

7.4 Overloading Relational Operators

Thus far, for our `Vector3` class, we have overloaded the arithmetic operators and have achieved pleasant results. However, what about relational type operators? At present, to determine if two vectors are equal we write:

```
if( u.equals(v) )
      // equal
else
      // not equal
```

We do this using the `Vector3::equals` method, which, for reference, is implemented like so:

```
bool Vector3::equals(const Vector3& rhs)
{
      // Return true if the corresponding components are equal.
      return
            mX == rhs.mX &&
            mY == rhs.mY &&
            mZ == rhs.mZ;
}
```

However, our goal is to make our user-defined types behave more like the built-in C++ types where it makes sense to do so. Therefore, we ask if it is possible to also overload the relational operators (==, `!=`, <, >, <=, >=). To be sure, we would *not* be spending time on a section called "overloading relation operators" if it were *not* possible!. So indeed we can, and in particular, for `Vector3`, we are interested in overloading the equality operator (==) and the not equals operator (`!=`). Overloading the relational operators is essentially the same as overloading the arithmetic operators; in both cases, we define the method as usual, but replace the method name with the `operator` keyword followed by the operator's symbol:

```
bool Vector3::operator==(const Vector3& rhs)
{
      // Return true if the corresponding components are equal.
      return
            mX == rhs.mX &&
            mY == rhs.mY &&
            mZ == rhs.mZ;
}

bool Vector3::operator!=(const Vector3& rhs)
{
      // Return true if any one corresponding components
      // are _not_ equal.
      return
            mX != rhs.mX ||
            mY != rhs.mY ||
            mZ != rhs.mZ;
}
```

Recall from Chapter 2 that the relational operators always return `bool` values. The relational operators you overload should do the same to preserve consistency and intuition. With our overloaded equals and not equals operators we can write code with `Vector3` objects that looks like this:

```
if( u == v )
      // equal
else
      // not equal

if( u != v )
      // not equal
else
      // equal
```

What about the other relational operators such as the less than (<) and greater than (>) operators? We could overload these operators (in the same way we overload the other relational operators) and say a vector `u` is less than a vector `v` if `u`'s length is less than `v`'s length. Likewise, a vector `u` is greater than a vector `v` if `u`'s length is greater than `v`'s length. However, this is not standard convention.

7.5 Overloading Conversion Operators

Recall the casting operator, which would allow us to, for example, convert a `float` to an `int`:

```
float pi = 3.14f;
int three = (int)pi; // three = 3 due to truncation.
```

The decimal part of the `float` is lost because an integer cannot represent it. But can we define casting operators that convert between our user-defined types and other types? Indeed we can, by overloading the casting operators for our class.

In our `Vector3` example, we can essentially view the method

```
float* toFloatArray();
```

as a conversion of a `Vector3` object to a three-element `float` array object (we return a pointer to the first element in the array). Our goal now is to overload an operator so that instead of calling `toFloatArray` to obtain the array representation, we can simply cast our `Vector3` object to a `float*`. That is:

```
// Convert to array representation.
Vector3 v(1.0f, 2.0f, 3.0f);
float* vArray = (float*)v;

// Print out each element and verify it matches the
// components of v.
cout <<
      "[0] = " << vArray[0] << ", "  // 1.0f
      "[1] = " << vArray[1] << ", "  // 2.0f
      "[2] = " << vArray[2] << endl; // 3.0f
```

Overloading a conversion operator takes on the following syntax:

```
operator type();
```

Where `type` is the type of object to which we wish to define a conversion. In our `Vector3` example, we wish to convert to type `float*`. So we replace `toFloatArray` with our overloaded conversion operator:

```
operator float*();
```

The implementation of our conversion operator is exactly the same as `toFloatArray` because they perform the same function.

```
Vector3::operator float*()
{
	return &mX;
}
```

To summarize, we have overloaded the conversion operator which converts our `Vector3` to type `float*` so that instead of calling `toFloatArray` to obtain the array representation, we can simply cast our `Vector3` object to a `float*` like so:
`float* vArray = (float*)v;`.

> **Note:** You can define as many conversion operators as you want for a class—just make sure it makes sense for your class to convert to those other types. For example, we could define a conversion from a `Vector3` object to a `double` three-element array or an `int` three-element array:
>
> ```
> operator double*();
> operator int*();
> ```

7.6 Overloading the Extraction and Insertion Operators

We have come a long way with our `Vector3` class in making it behave very much like the built-in C++ types, but we can go even further. Recall that we can output and input the built-in types to and from the console window with `cout <<`, and `cin >>`, respectively. We might wonder whether we could make it so that we could use `cout <<` and `cin >>` with our user-defined types. Predictably, we can.

In particular, we need to overload the **insertion operator (<<)** and the **extraction operator (>>)**, and specify how our user-defined types will be output and input with these operators. Note that these operators are also sometimes called the **output** and **input** operators, respectively. However, one key difference between the insertion and extraction operators that differs from the other operators we have discussed is where we overload these operators. Previously, we always made the operators member functions, but if we look at the syntax:

```
cout << variable;
```

We note that this is translated to `cout.operator<<(variable)`. In other words, `cout` is the object invoking the operator <<.

> **Note:** For reference, `cout` and `cin` are defined in the standard namespace like so:

```
extern ostream cout;
extern istream cin;
```

That is, they are object instances of class `ostream` and `istream`.

Obviously, the writers of `ostream` (the class of which `cout` is an instance) and of `istream` (the class of which `cin` is an instance) did not know about our user-defined class `Vector3`; that is, there is no method:

```
std::ostream& std::ostream::operator<<(const Vector3& v);
```

nor

```
std::istream& std::istream::operator<<(Vector3& v);
```

Therefore, `cout` and `cin` do not work with our objects (by default). Moreover, because we did not write the `istream` or `ostream` class, we cannot add such member functions to them. Therefore, the only way to overload << and >> so that they work with `cout` and `cin` as we would expect, is to make them non-member functions:

```
std::istream& operator>>(std::istream& is, Vector3& v);
std::ostream& operator<<(std::ostream& os, const Vector3& v);
```

The implementations simply do the same thing we had `input` and `print` do:

```
std::istream& operator>>(std::istream& is, Vector3& v)
{
      cout << "Enter x: ";
      cin >> v.mX;
      cout << "Enter y: ";
      cin >> v.mY;
      cout << "Enter z: ";
      cin >> v.mZ;

      return is;
}

std::ostream& operator<<(std::ostream& os, const Vector3& v)
{
      cout << "<" << v.mX << ", " << v.mY << ", " << v.mZ << "> \n";

      return os;
}
```

Note: Both of these overloaded operators (<< & >>) access the data of a `Vector3` object. This is legal since the data members of `Vector3` are public in this case. However, this will not always be the case. Therefore, typically you will find the overloaded << and >> operators declared as *friends* (Section 6.5) of the class they operate on. For example, if `Vector3`'s data were private, we would write:

```
class Vector3
{
   friend std::istream& operator>>(std::istream& is, Vector3& v);
   friend std::ostream& operator<<(std::ostream& os, const Vector3& v);
...
```

In this way, the overloaded `<<` and `>>` operators could access the private data of `Vector3` without any problems.

With these overloaded operators, we can now output and input `Vector3` objects to and from the console using `cout` and `cin`, respectively:

Program 7.3: Using the overloaded insertion and extraction operators.

```
// main.cpp

#include "Vector3.h"
#include <iostream>
using namespace std;

int main()
{
      // Part 1: Construct three vectors and print.
      float coords[3] = {1.0f, 2.0f, 3.0f};
      Vector3 u;
      Vector3 v(coords);
      Vector3 w(-5.0f, 2.0f, 0.0f);

      cout << "u = ";    cout << u;
      cout << "v = ";    cout << v;
      cout << "w = ";    cout << w;
      cout << endl;

      cin >> u;

      cout << u;
}
```

Program 7.3 Output

```
u = <0, 0, 0>
v = <1, 2, 3>
w = <-5, 2, 0>

Enter x: 9
Enter y: 8
Enter z: 7
<9, 8, 7>
Press any key to continue
```

7.7 A String Class; Overloading the Assignment Operator, Copy Constructor, and Bracket Operator

We now shift focus from the `Vector3` class to a custom made `String` class. Before we begin, understand that this `String` class will be incomplete and we will use it only to further illustrate some additional operator overloading concepts—it is not meant to replace `std::string`.

Internally, we represent the string data with a c-string. However, because the size of the string is unknown, we use dynamic memory so we can resize the string accordingly:

```
class String
{
public:
      String();
      String(const char* rhs);
      String(const String& rhs);

      String& operator=(const String& rhs);

private:
      char* mData;
};
```

7.7.1 Construction and Destruction

We would like to be able to construct a `String` object from a c-string so we added the constructor:

```
String(const char* rhs);
```

(Note: RHS stands for right-hand-side.) At first, you may suspect that all we need to do is this:

```
String::String(const char* rhs)
{
      mData = rhs;
}
```

That is, make our internal `char` pointer point to the c-string. However, our `String` objects are not merely pointers to other c-strings; rather, they are completely independent string objects, which use a c-string as a source to *copy* characters into their *own* c-string memory destination. Think about it. Suppose we did construct a `String` object `dest` using a c-string `C`, and we let `dest.mData` point to `C`. What happens if `C` changes? If `C` changes then `dest` changes as well, since internally, `dest`'s data representation is a pointer to `C`. What happens if `C` is destroyed? If `C` is destroyed then `dest` becomes invalid automatically, since `dest`'s internal data representation is a pointer to `C` (which was destroyed). This kind of behavior might be surprising to someone using your `String` class, who would expect

`String` objects to be self-contained and not dependent upon external data. Therefore, when given a c-string in the constructor, we allocate enough memory to store a copy of this c-string, and then copy the c-string into the `String`'s own internal data:

```
String::String(const char* rhs)
{
	// Get the length of the rhs c-string.
	int len = strlen(rhs);

	// Allocate enough memory to store a copy of the c-string plus
	// one more for null-character.
	mData = new char[len+1];

	// Copy characters over to our own data.
	for(int i = 0; i < len; ++i)
		mData[i] = rhs[i];

	// Set null-character in the last spot.
	mData[len] = '\0';
}
```

Because we have used dynamic memory, we must free the memory when a `String` object is destroyed. Recall that the destructor is a function that is called when an object gets destroyed. Hence, the destructor is the logical place to put this memory release code:

```
String::~String()
{
	delete[] mData;
	mData = 0;
}
```

7.7.2 Assignment Operator

Recall that every class automatically gets a constructor, destructor, copy constructor, and assignment operator, even if it does not manually define one (the compiler will do this for you). What does the default assignment operator do? The default assignment operator will perform a simple byte-by-byte copy from the right hand operand's memory into the left hand operand's memory; this is called a **shallow copy.**

Consider the following:

```
String dest("Hello");
String source("C++");

dest = source;
```

A shallow copy implies a direct memory copy, which means the pointer address will by copied from `source` to `dest`; that is, `dest.mData = source.mData;`

This raises two problems:

1. The memory to which `dest.mData` pointed was never deleted before `source.mData` was assigned to it. This results in a memory leak.

2. The second problem with a shallow copy in the case of our `String` class is the same problem we mentioned in the previous section with the constructor: a shallow copy will simply copy pointers over. That is, `dest.mData == source.mData`. Consequently, `dest` is no longer independent (it depends on `source` now—`source.mData`, specifically) and we get all the problems we mentioned in the preceding section; namely, if `source` changes then `dest` changes, and if `source` is destroyed then `dest` contains a pointer to deleted memory.

To fix these problems, we must overload the assignment operator and perform a **deep copy.** In a deep copy, we do what we did in the constructor: we allocate enough memory to store a copy of the `rhs.mData` c-string, and then we copy the c-string into the `String`'s own internal data:

```
String& String::operator =(const String& rhs)
{
      // Prevent self assignment.  We say two Strings
      // are equal if their memory addresses are equal.
      if( this == &rhs )
            return *this;

      // Get the length of the rhs c-string.
      int len = strlen(rhs.mData);

      // Free existing memory.
      delete [] mData;

      // Allocate enough memory to store a copy of the c-string plus
      // one more for null-character.
      mData = new char[len+1];

      // Copy characters over to our own data.
      for(int i = 0; i < len; ++i)
            mData[i] = rhs.mData[i];

      // Set null-character.
      mData[len] = '\0';

      // Return a reference to *this object.
      return *this;
}
```

Note: As a rule, if your classes contain pointers, you should always override the copy constructor and assignment operator and perform a deep copy—otherwise you simply copy pointers over and get two objects that point to one thing, thereby making the objects dependent on each other, instead of self-contained, independent objects.

There are a few other things happening in this assignment operator, which you should understand.

- We first check for self assignment; for example:

  ```
  String dest("Hello");
  dest = dest;
  ```

  For our purposes, we define two `String` objects to be equal if their addresses are equal. You may prefer to say two `String` objects are equal if they represent equivalent strings (e.g., “hello” == “hello”). Which option you choose depends on the needs of the application.

- Second, we return a reference to the left-hand-side operand; that is, `*this`. We do this because it allows us to chain assignments like so:

  ```
  String a;
  String b;
  String c;
  String d("Hello");

  a = b = c = d;
  ```

  If we returned, say, void instead of a reference to `*this`, we would not be able to chain assignments like this, because recall that these operator symbols really look like the following, as far as the compiler is concerned:

  ```
  a.operator=(b.operator=(c.operator=(d)));
  ```

  You have to return something from the “operator function” if you want to be able to pass the function as an argument into another function.

7.7.3 Copy Constructor

A copy constructor is a method that constructs an object via another object of the same type. If you use the default copy constructor, the object will be constructed by copying the parameter’s bytes, byte-by-byte, into the object being constructed, thereby performing a shallow copy. From the previous section, we know this default behavior is unacceptable for our `String` class (or virtually any class that uses dynamic memory).

Because the copy constructor constructs an object via another object of the same type, which is essentially what our `String` assignment operator does, we can actually implement the copy constructor in terms of the assignment operator like so:

```
String::String(const String& rhs)
{
      mData = 0; // Does not point to anything, yet.
      *this = rhs;
}
```

Our copy constructor now behaves correctly and does a deep copy (because we overloaded the assignment operator to do a deep copy).

7.7.4 Overloading the Bracket Operator

Recall that with a `std::string`, we could access the individual characters of that string with the bracket operator [] like so:

```
string s = "Hello, world!";

char c0 = s[0];  // = 'H'
char c1 = s[1];  // = 'e'
char c2 = s[4];  // = 'o'
char c3 = s[7];  // = 'w'
char c4 = s[12]; // = '!'
```

We can add this exact functionality to our String class by overloading the bracket operator:

```
char& String::operator[](int i)
{
      return mData[i];
}
```

As you can see, the bracket operator takes one parameter, which is the index into some container (a `char` array in our case). We use this index to fetch the appropriate character out of our internal `char` array.

7.8 Summary

1. One of the primary design goals of C++ was for user-defined types to behave similarly to intrinsic C++ types (e.g., `float`, `int`). To facilitate this goal, operator overloading was added to C++. Operator overloading enables programmers to overload the C++ operators (e.g., '+', '*', '<', '!=", '<<', '>>', etc) and define new behaviors for them. Consequently, we can overload C++ operators with our user-defined types, which results in behavior similar to the intrinsic C++ types.

2. Operator overloading is not for every class. Only use operator overloading if it makes the class easier and more natural to work with. Do not overload operators and implement them with non-intuitive and confusing behavior. To illustrate an extreme case: You would not overload the '+' operator such that it performs a subtraction operation, as this kind of behavior would be very confusing.

3. A vector is a mathematical object used to represent magnitudes and directions. Examples of such quantities are physical forces (forces are applied in a certain direction and have a strength and magnitude associated with them), and velocities (speed and direction). Geometrically, we represent a vector as a directed line segment. The direction of the line segment describes the vector direction, and the length of the line segment describes the magnitude of the vector.

4. A shallow copy refers to copying memory byte-by-byte from one segment of memory to another. A deep copy refers to copying dynamic memory values by allocating enough memory in the destination to store all the values contained in the source, and then copying the source values into the destination memory.

7.9 Exercises

7.9.1 `Fraction` Class

Define and implement a `Fraction` class with the following specifications:

- Contains a floating-point data member representing the numerator and contains a floating-point data member representing the denominator.

- Contains a default constructor that takes no parameters and which initializes the fraction to $0/1$.

- Contains a non-default constructor which takes a floating-point parameter that specifies the numerator, and takes another floating-point point parameter that specifies the denominator.

- Overloads the arithmetic operators (+, -, *, /) to perform fraction addition, fraction subtraction, fraction multiplication, and fraction division. (You do not need to reduce the fraction, but you can if you are motivated to do so.)

- Overloads the relational operators (==, !=, <, >, <=, >=) so that, for any two `Fractions` `A` and `B`, we can determine if `A == B`, `A != B`, `A < B`, `A > B`, `A <= B`, or `A >= B`.

- Overloads the `float` and `double` conversion operator so that you can convert a `Fraction` object to a decimal of type `double` or `float`; for example, so you can write:

  ```
  Fraction frac(22, 7); // (22 / 7)
  float decimal = (float)frac; // convert to decimal;
                               // decimal ≈ 3.142857.
  ```

  Note that the decimal representation of a fraction is computed simply as:

  decimal = numerator / denominator.

- Overloads the insertion (<<) and extraction (>>) operators so that you can output `Fractions` with `cout` and input `Fractions` with `cin`.

Be sure to test every operator of your `Fraction` class thoroughly to verify you have implemented the operators correctly. Also, be sure to watch out for divisions by zero.

7.9.2 Simple `float` Array Class

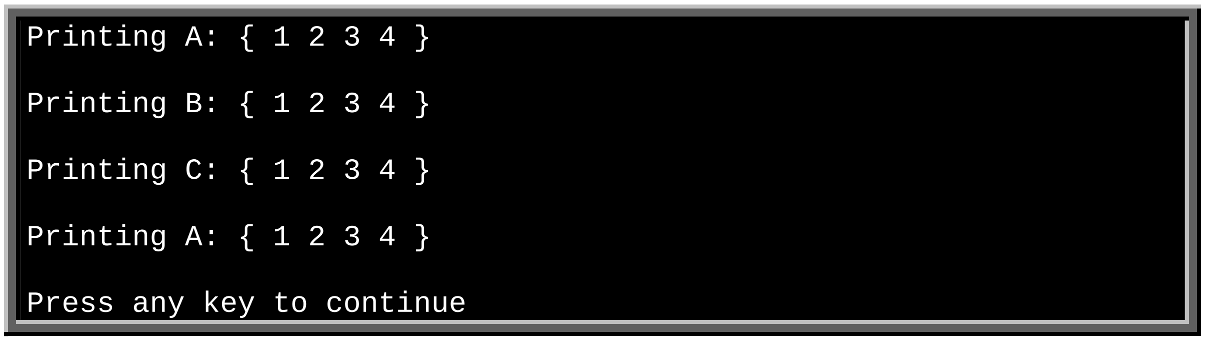
Consider the following class definition:

```
// FloatArray.h

#ifndef FLOAT_ARRAY_H
#define FLOAT_ARRAY_H

class FloatArray
{
public:
      // Create a FloatArray with zero elements.
      FloatArray();

      // Create a FloatArray with 'size' elements.
      FloatArray(int size);

      // Create a FloatArray from another FloatArray--
      // be sure to prevent memory leaks!
      FloatArray(const FloatArray& rhs);

      // Free dynamic memory.
      ~FloatArray();

      // Define how a FloatArray shall be assigned to
      // another FloatArray--be sure to prevent memory
      // leaks!
      FloatArray& operator=(const FloatArray& rhs);

      // Resize the FloatArray to a new size.
      void resize(int newSize);

      // Return the number of elements in the array.
      int size();

      // Overload bracket operator so client can index
      // into FloatArray objects and access the elements.
      float& operator[](int i);

private:
      float* mData; // Pointer to array of floats (dynamic memory).
      int    mSize; // The number of elements in the array.
};

#endif // FLOAT_ARRAY_H
```

Your task for this exercise is to provide the implementation for `FloatArray`. Read the comments above each method for instructions on what the method should do (i.e., how you should implement the method). (*Hint:* Reread Program 4.9 from Chapter 4 Section 4.5.4 for help on how to implement the `resize` method.) After you are finished with the implementation, write the following driver program to test it:

```cpp
// FloatArrayDriver.cpp

#include "FloatArray.h"
#include <iostream>
using namespace std;

void PrintFloatArray(FloatArray& fa)
{
      cout << "{ ";
      for(int i = 0; i < fa.size(); ++i)
            cout << fa[i] << " ";

      cout << "}" << endl << endl;
}

int main()
{
      FloatArray A;

      A.resize(4);
      A[0] = 1.0f;
      A[1] = 2.0f;
      A[2] = 3.0f;
      A[3] = 4.0f;

      cout << "Printing A: ";
      PrintFloatArray(A);

      FloatArray B(A);

      cout << "Printing B: ";
      PrintFloatArray(B);

      FloatArray C = B = A;

      cout << "Printing C: ";
      PrintFloatArray(C);

      A = A = A = A;

      cout << "Printing A: ";
      PrintFloatArray(A);
}
```

If you implemented the `FloatArray` class correctly, this driver should compile and display the following expected output without errors:



```
Printing A: { 1 2 3 4 }

Printing B: { 1 2 3 4 }

Printing C: { 1 2 3 4 }

Printing A: { 1 2 3 4 }

Press any key to continue
```

Does your program match the expected output and execute without errors? If not, reconsider your implementation of `FloatArray`.

Chapter 8

File Input and Output

Introduction

Many games do not expect the player to complete the game in one sitting, and we can further assume that most gamers do not wish to start a game from the beginning each time they play. Therefore, it is necessary that a game be able to be saved at certain points of progress, and then resumed from those points at a later time. In order to satisfy this requirement, we will need to be able to save/load game information to/from a place where it can persist after the program has terminated, and after the computer has been turned off. The obvious place for such storage is the hard drive. Thus, the primary theme of this chapter is saving files from our program to disk (file output) and loading files from disk into our program (file input).

Chapter Objectives

- Learn how to load and save text files to and from your program.
- Learn how to load and save binary files to and from your program.

8.1 Streams

Recall that `cout` and `cin` are instances of the class `ostream` and `istream`, respectively:

```
extern ostream cout;
extern istream cin;
```

What are `ostream` and `istream`? For starters, the ‘o’ in `ostream` stands for “output,” and thus `ostream` means “output stream.” Likewise, the ‘i’ in `istream` stands for “input,” and thus `istream` means “input stream.” A **stream** is a flow of data from a source to a destination. It is used analogously to a water stream. As water flows down a stream so data flows as well. In the context of `cout`, the stream flows data from our program to the console window for display. In the context of `cin`, the stream flows data from the keyboard into our program.

We discuss `cout` and `cin` because file I/O works similarly. Indeed we will use streams for file I/O as well. In particular, we instantiate objects of type `ofstream` to create an “output file stream” and objects of type `ifstream` to create an “input file stream.” An `ofstream` object flows data from our program to a file, thereby “writing (saving) data to the file.” An `ifstream` object flows data from a file into our program, thereby “reading (loading) data from the file.”

8.2 Text File I/O

In this section we concern ourselves with saving and loading text files. Text files contain data written in a format readable by humans, as opposed to binary files (which we will examine later) which simply contain pure numeric data. We will use two standard classes to facilitate file I/O:

- `ofstream`: An instance of this class contains methods that are used to write (save) data to a file.

- `ifstream`: An instance of this class contains methods that are used to read (load) data from a file.

  Note: In order to use `ofstream` and `ifstream`, you must include the standard library header file <fstream> (file stream) into your source code file. Also realize that these objects exist in the standard namespace.

The overall process of file I/O can be broken down into three simple steps:

1. Open the file.
2. Write data to the file or read data from the file.
3. Close the file.

8.2.1 Saving Data

To open a file which we will *write* to, we have two options:

1) We can create an `ofstream` object and pass a string specifying the filename we wish to write to (if this file does not exist then it will be created by the object)

2) We can create an `ofstream` object using the default constructor and then call the `open` method.

Both styles are illustrated next, and one is not necessarily preferable over the other.

```
ofstream outFile("data.txt");
```

Or:

```
ofstream outFile;
outFile.open("data.txt");
```

Interestingly, `ofstream` overloads the conversion operator to type `bool`. This conversion returns `true` if the stream is valid and `false` otherwise. For example, to verify that `outFile` was constructed (or `open`ed) correctly we can write:

```
if( outFile )
      // outFile valid--construction/open OK.
else
      // construction/open failed.
```

Once we have an open file, data can be "dumped" from our program into the stream. The data will flow down the stream and into the file. To do this, the insertion operator (<<) is used, just as with `cout`:

```
outFile << "Hello, world!" << endl;
float pi = 3.14f;
outFile << "pi = " << pi << endl;
```

This would write the following output to the file:

```
Hello, world!
pi = 3.14
```

The symmetry between `cout` and `ofstream` becomes more apparent now. Whereas `cout` sends data from our program to the console window to be displayed, `ofstream` sends data from our program to be written to a file for storage purposes.

Finally, to close the file, the `close` method is called:

```
outFile.close();
```

8.2.2 Loading Data

To open a file, which we will *read* from, we have two options:

1) We can create an `ifstream` object and pass a string specifying the filename from which we wish to read.

2) We can create an `ifstream` object using the default constructor and then call the `open` method.

Both styles are illustrated next, and one is not necessarily preferable over the other.

```
ifstream inFile("data.txt");
```

Or:

```
ifstream inFile;
inFile.open("data.txt");
```

`ifstream` also overloads the conversion operator to type `bool`. This conversion returns `true` if the stream is valid and `false` otherwise. For example, to verify that `inFile` was constructed (or `opened`) correctly we can write:

```
if( inFile )
      // inFile valid--construction/open OK.
else
      // construction/open failed.
```

Once we have an open file, data can be read from the input file stream into our program. The data will flow down the stream from the file into our program. To do this, the extraction operator (>>) is used, as with `cin`:

```
string data;
inFile >> data; // Read a string from the file.
float f;
inFile >> f; // Read a float from the file.
```

The symmetry between `cin` and `ifstream` is more apparent now. Whereas `cin` reads data from the console window, `ifstream` reads data from a file.

Finally, to close the file, the `close` method is called:

```
inFile.close();
```

8.2.3 File I/O Example

Now that you are familiar with the concepts of file I/O and the types of objects and methods we will be working with, let us look at an example program. Recall the `Wizard` class from Chapter 5, which we present now in a modified form:

```
// Wiz.h

#ifndef WIZARD_H
#define WIZARD_H

#include <fstream>
#include <string>

class Wizard
{
public:
      Wizard();
      Wizard(std::string name, int hp, int mp, int armor);

      // [...] other methods snipped

      void print();

      void save(std::ofstream& outFile);
      void load(std::ifstream& inFile);

private:
      std::string mName;
      int mHitPoints;
      int mMagicPoints;
      int mArmor;
};
#endif // WIZARD_H
```

```
// Wiz.cpp

#include "Wiz.h"
#include <iostream>
using namespace std;

Wizard::Wizard()
{
      mName        = "Default";
      mHitPoints   = 0;
      mMagicPoints = 0;
      mArmor       = 0;
}

Wizard::Wizard(string name, int hp, int mp, int armor)
{
      mName        = name;
      mHitPoints   = hp;
      mMagicPoints = mp;
      mArmor       = armor;
}

void Wizard::print()
{
      cout << "Name= "  << mName        << endl;
      cout << "HP= "    << mHitPoints   << endl;
      cout << "MP= "    << mMagicPoints << endl;
      cout << "Armor= " << mArmor       << endl;
      cout << endl;
}

// [...] ‘save’ and ‘load’ implementations follow shortly.
```

Specifically, we have removed methods which are of no concern to us in this chapter. Additionally, we added two methods, `save` and `load`, which do what their names imply. The `save` method writes a `Wizard` object to file, and the `load` method reads a `Wizard` object from file. Let us look at the implementation of these two methods one at a time:

```
void Wizard::save(ofstream& outFile)
{
      outFile << "Name= "  << mName        << endl;
      outFile << "HP= "    << mHitPoints   << endl;
      outFile << "MP= "    << mMagicPoints << endl;
      outFile << "Armor= " << mArmor       << endl;
      outFile << endl;
}
```

The `save` method has a reference parameter to an `ofstream` object called `outFile`. `outFile` is the output file stream through which our data will be sent. Inside the `save` method, our data is "dumped" into the output file stream using the insertion operator (<<) just as we would with `cout`.

To apply our `save` method, consider the following driver program:

Program 8.1: Saving text data to file.

```
// main.cpp

#include "Wiz.h"
using namespace std;

int main()
{
      // Create wizards with specific data.
      Wizard wiz0("Gandalf", 25, 100, 10);
      Wizard wiz1("Loki", 50, 150, 12);
      Wizard wiz2("Magius", 10, 75, 6);

      // Create a stream which will transfer the data from
      // our program to the specified file "wizdata.tex".
      ofstream outFile("wizdata.txt");

      // If the file opened correctly then call save methods.
      if( outFile )
      {
            // Dump data into the stream.
            wiz0.save(outFile);
            wiz1.save(outFile);
            wiz2.save(outFile);

            // Done with stream--close it.
            outFile.close();
      }
}
```

This program does not output anything. Rather, it creates a text file called "wizdata.txt" in the project's working directory[4]. If we open that file, we find the following data was saved to it:

"wizdata.txt"

```
Name= Gandalf
HP= 25
MP= 100
Armor= 10

Name= Loki
HP= 50
MP= 150
Armor= 12
```

[4] When you specify the string to the `ofstream` constructor or the open method, you can specify a path as well. For example, you can specify "C:\wizdata.txt" to write the file "wizdata.txt" to the root of the C-drive.

```
Name= Magius
HP= 10
MP= 75
Armor= 6
```

From the file output, it is concluded that the program did indeed save `wiz0`, `wiz1`, and `wiz2` correctly.

Now let us examine the `load` method:

```
void Wizard::load(ifstream& inFile)
{
	string garbage;
	inFile >> garbage >> mName;       // read name
	inFile >> garbage >> mHitPoints;  // read hit points
	inFile >> garbage >> mMagicPoints;// read magic points
	inFile >> garbage >> mArmor;      // read armor
}
```

This method is symmetrically similar to the `save` method. The `load` method has a reference parameter to an `ifstream` object called `inFile`. `inFile` is the input file stream from which we will extract the file data and into our program. Inside the `load` method we extract the data out of the stream using the extraction operator (>>), just like we would with `cin`.

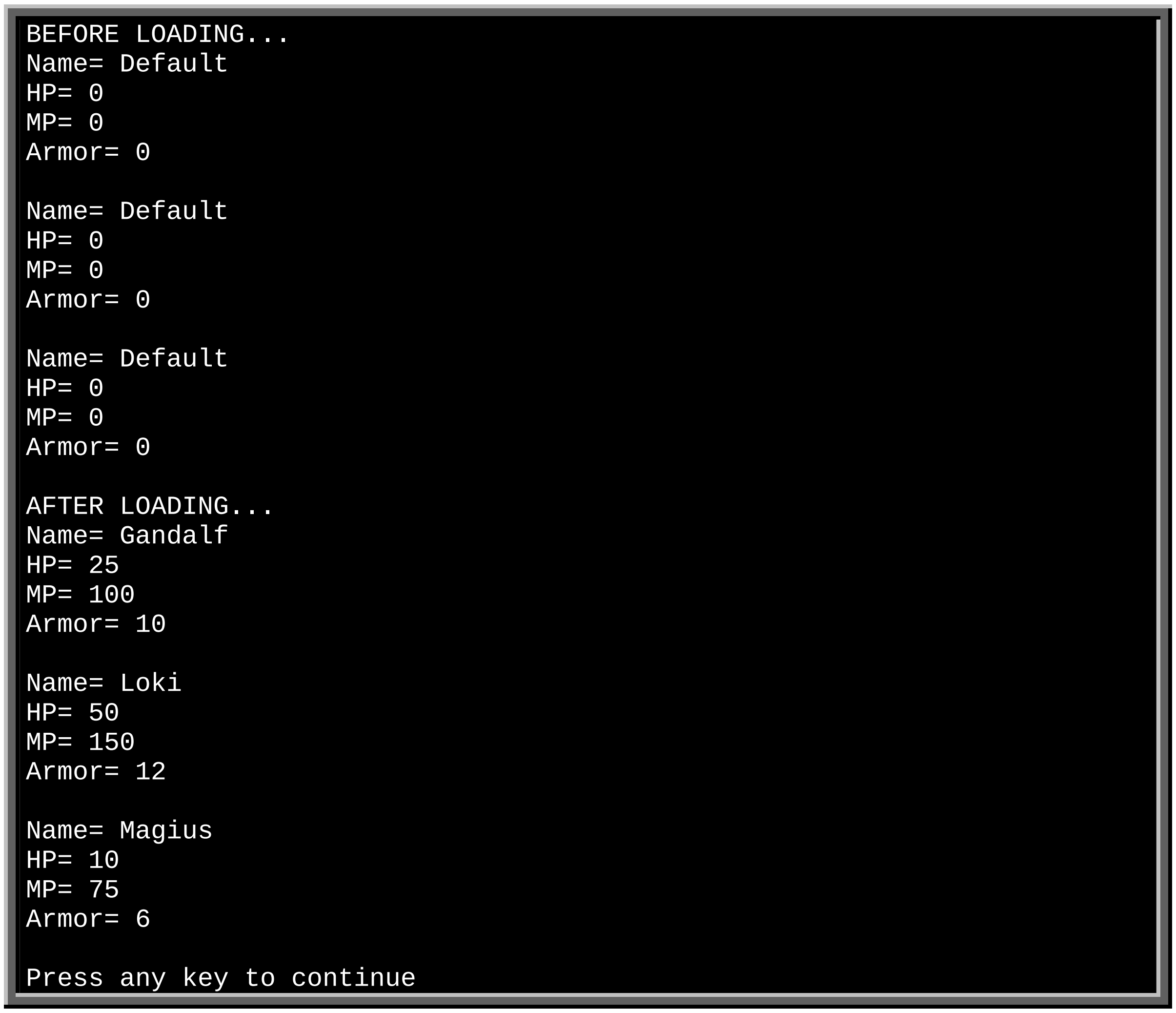
The `garbage` extraction may seem odd at first (`inFile >> garbage`). However, note that when we saved the wizard data, we wrote out a string describing the data (see "wizdata.txt"). For example, before we wrote `mName` to file in `save`, we first wrote "`Name   =`". Before we can extract the actual wizard name from the file, we must first extract "`Name   =`". To do that, we feed it into a "garbage" variable because it is not used.

To apply our `load` method, consider the following driver program:

Program 8.2: Loading text data from file.

```
// main.cpp

#include "Wiz.h"
#include <iostream>
using namespace std;

int main()
{
	// Create some 'blank' wizards, which we will load
	// data from file into.
	Wizard wiz0;
	Wizard wiz1;
	Wizard wiz2;

	// Output the wizards before they are loaded.
	cout << "BEFORE LOADING..." << endl;
	wiz0.print();
```

```
    wiz1.print();
    wiz2.print();

    // Create a stream which will transfer the data from
    // the specified file "wizdata.txt" to our program.
    ifstream inFile("wizdata.txt");

    // If the file opened correctly then call load methods.
    if( inFile )
    {
        wiz0.load(inFile);
        wiz1.load(inFile);
        wiz2.load(inFile);
    }

    // Output the wizards to show the data was loaded correctly.
    cout << "AFTER LOADING..." << endl;
    wiz0.print();
    wiz1.print();
    wiz2.print();
}
```



```
BEFORE LOADING...
Name= Default
HP= 0
MP= 0
Armor= 0

Name= Default
HP= 0
MP= 0
Armor= 0

Name= Default
HP= 0
MP= 0
Armor= 0

AFTER LOADING...
Name= Gandalf
HP= 25
MP= 100
Armor= 10

Name= Loki
HP= 50
MP= 150
Armor= 12

Name= Magius
HP= 10
MP= 75
Armor= 6

Press any key to continue
```

As the output shows, the data was successfully extracted from “wizdata.txt.”

> **Note:** When extracting data with `ifstream`, we run into the same problem we did with `cin`; that is, `cin` reads up to a space character. To get around this problem we use the same solution we used with cin—the `getline` function, which can read up to a line of input. Recall that `getline`’s first parameter is a reference to an `istream` object and not an `ifstream` object; however, we can still use `ifstream` with `getline`, since `ifstream` is a kind of `istream`.

8.3 Binary File I/O

When working with text files, there is some overhead that occurs when converting between numeric and text types. Additionally, a text-based representation tends to consume more memory. Thus, we have two motivations for using binary-based files:

1. Binary files tend to consume less memory than equivalent text files.
2. Binary files store data in the computer’s native binary representation so no conversion needs to be done when saving or loading the data.

However, text files are convenient because a human can read them, and this makes the files easier to edit manually, and I/O bugs easier to fix.

Creating file streams that work in binary instead of text is quite straightforward. An extra flag modifier, which specifies binary usage, must be passed to the file stream’s constructor or open method:

```
ofstream outFile("pointdata.txt", ios_base::binary);
ifstream inFile("pointdata.txt", ios_base::binary);
```

Or:

```
outFile.open("pointdata.txt", ios_base::binary);
inFile.open("pointdata.txt", ios_base::binary);
```

Where `ios_base` is a member of the standard namespace; that is, `std::ios_base`.

8.3.1 Saving Data

Because there is no necessary conversion required in binary mode, we do not need to worry about writing specific types. All that is required for transferring data in binary form is to stream raw bytes. When writing data, we specify a pointer to the first byte of the data-chunk, and the number of bytes it contains. All the bytes of the data-chunk will then be streamed directly to the file in their byte (binary) form. Consequently, a large amount of bytes can be streamed if they are contiguous, like an array or class object, with one method call.

To write data to a binary stream the `write` method is used, as the following code snippet illustrates:

```
struct Point
{
	int x;
	int y;
};

float fArray[4] = {1, 2, 3, 4};
Point p = {0, 0};
int x = 10;

outFile.write((char*)fArray, sizeof(float)*4);
outFile.write((char*)&p, sizeof(Point));
outFile.write((char*)&x, sizeof(int));
```

The first parameter is a `char` pointer. Recall that a `char` is one byte. By casting our data-chunk (be it a built-in type, a class, or array of any type) to a `char` pointer, we are returning the address of the first byte of the data-chunk. The second parameter is the number of bytes we are going to stream in this call, starting from the first byte pointed to by the first parameter. Typically, we use the `sizeof` operator to get the number of bytes of the entire data-chunk so that the whole data-chunk is streamed to the file.

8.3.2 Loading Data

Loading binary data is similar to writing it. Once we have a binary input file stream setup, we simply specify the number of bytes we wish to stream in from the file into our program. As with writing bytes, we can stream in a large amount of contiguous bytes with one function call, labeled `read`.

```
float fArray[4];
Point p;
int x;

inFile.read((char*)fArray, sizeof(float)*4);
inFile.read((char*)&p, sizeof(Point));
inFile.read((char*)&x, sizeof(int));
```

The `read` method is the inverse of the `write` method. The first parameter is a pointer to the first byte of the data-chunk into which we wish to read the bytes. The second parameter is the number of bytes to stream into the data-chunk specified by the first parameter.

8.3.3 Examples

Now that we are familiar with the basics of binary file writing and reading, let us look at a full example. Figure 8.1 shows the vertices of a unit cube.

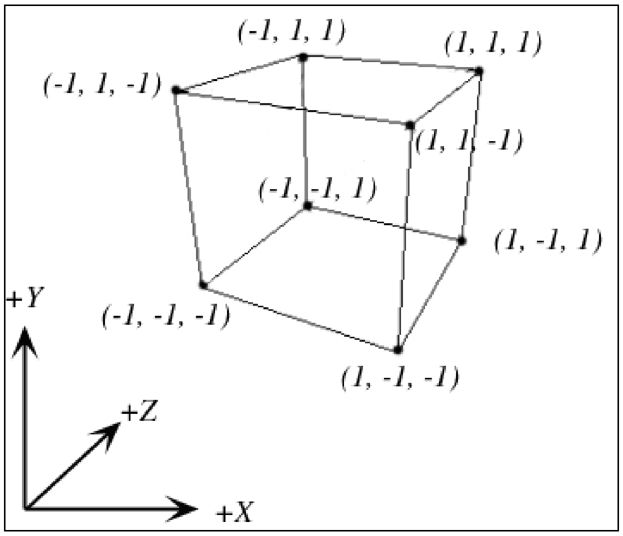


Figure 8.1: Unit cube with vertices specified.

In the first program, we will create the vertices of a unit cube and stream the data to a binary file called "pointdata.txt." In the second program, we will do the inverse operation and stream the point data contained in "pointdata.txt" into our program.

First, we create a basic data structure to represent a point in 3D space:

```
struct Point3
{
      Point3();
      Point3(float x, float y, float z);
      float mX;
      float mY;
      float mZ;
};

// Implementation
Point3::Point3()
{
      mX = mY = mZ = 0.0f;
}

Point3::Point3(float x, float y, float z)
{
      mX = x;
      mY = y;
      mZ = z;
}
```

The first program is written as follows:

Program 8.3: Saving binary data to file.

```
#include <fstream>
#include <iostream>
#include “Point.h”
using namespace std;

int main()
{
      // Create 8 points to define a unit cube.
      Point3 cube[8];

      cube[0] = Point3(-1.0f, -1.0f, -1.0f);
      cube[1] = Point3(-1.0f,  1.0f, -1.0f);
      cube[2] = Point3( 1.0f,  1.0f, -1.0f);
      cube[3] = Point3( 1.0f, -1.0f, -1.0f);
      cube[4] = Point3(-1.0f, -1.0f,  1.0f);
      cube[5] = Point3(-1.0f,  1.0f,  1.0f);
      cube[6] = Point3( 1.0f,  1.0f,  1.0f);
      cube[7] = Point3( 1.0f, -1.0f,  1.0f);

      // Create a stream which will transfer the data from
      // our program to the specified file "pointdata.tex".
      // Observe how we add the binary flag modifier
      // ios_base::binary.
      ofstream outFile("pointdata.txt", ios_base::binary);

      // If the file opened correctly then save the data.
      if( outFile )
      {
            // Dump data into the stream in binary format.
            // That is, stream the bytes of the entire array.
            outFile.write((char*)cube, sizeof(Point3)*8);

            // Done with stream--close it.
            outFile.close();
      }
}
```

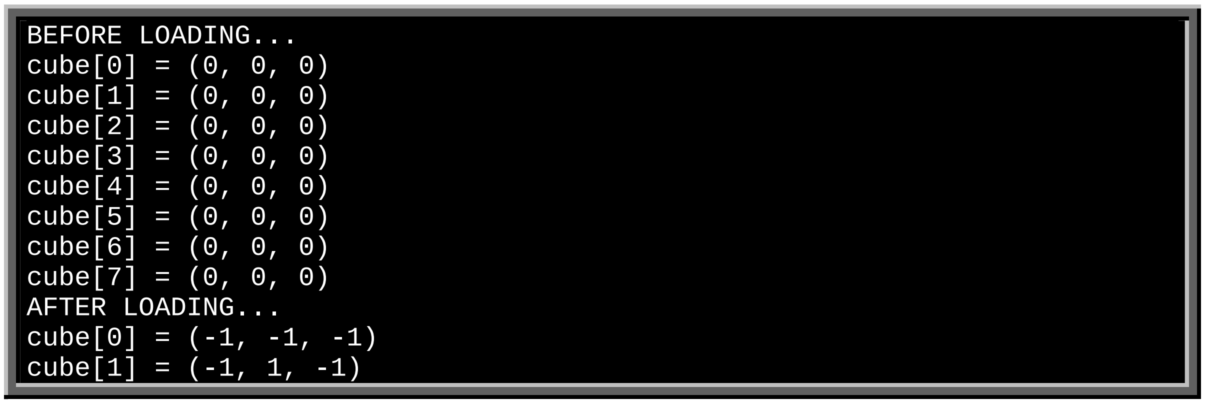
This program produces no console output. However, it does create the file “pointdata.txt” and writes the eight points of the cube to that file. If you open “pointdata.txt” in a text editor, you will see what appears to be nonsense, because the data is written in binary.

We now proceed to write the inverse program.

Program 8.4: Loading binary data from file.

```
#include <fstream>
#include <iostream>
#include “Point.h”
using namespace std;
```

```
int main()
{
        Point3 cube[8];

        cout << "BEFORE LOADING..." << endl;
        for(int i = 0; i < 8; ++i)
        {
                cout << "cube[" << i << "] = ";
                cout << "(";
                cout << cube[i].mX << ", ";
                cout << cube[i].mY << ", ";
                cout << cube[i].mZ << ")" << endl;
        }

        // Create a stream which will transfer the data from
        // the specified file "pointdata.txt" to our program.
        // Observe how we add the binary flag modifier
        // ios_base::binary.
        ifstream inFile("pointdata.txt", ios_base::binary);

        // If the file opened correctly then call load methods.
        if( inFile )
        {
                // Stream the bytes in from the file into our
                // program array.
                inFile.read((char*)cube, sizeof(Point3)*8);

                // Done with stream--close it.
                inFile.close();
        }

        // Output the points to show the data was loaded correctly.
        cout << "AFTER LOADING..." << endl;
        for(int i = 0; i < 8; ++i)
        {
                cout << "cube[" << i << "] = ";
                cout << "(";
                cout << cube[i].mX << ", ";
                cout << cube[i].mY << ", ";
                cout << cube[i].mZ << ")" << endl;
        }
}
```



```
BEFORE LOADING...
cube[0] = (0, 0, 0)
cube[1] = (0, 0, 0)
cube[2] = (0, 0, 0)
cube[3] = (0, 0, 0)
cube[4] = (0, 0, 0)
cube[5] = (0, 0, 0)
cube[6] = (0, 0, 0)
cube[7] = (0, 0, 0)
AFTER LOADING...
cube[0] = (-1, -1, -1)
cube[1] = (-1, 1, -1)
```

```
cube[2] = (1, 1, -1)
cube[3] = (1, -1, -1)
cube[4] = (-1, -1, 1)
cube[5] = (-1, 1, 1)
cube[6] = (1, 1, 1)
cube[7] = (1, -1, 1)
Press any key to continue
```

As the output shows, the data was successfully extracted from “pointdata.txt.”

8.4 Summary

1. Use file I/O to save data files from your programs to the hard drive and to load previously saved data files from the hard drive into your programs.

2. We generally use two standard library classes for file I/O:

   a. `ofstream`: an instance of this class contains methods that are used to write (save) data to a file
   b. `ifstream`: An instance of this class contains methods that are used to read (load) data from a file.

   To use these objects you must include <fstream> (file stream).

3. A stream is essentially the flow of data from a source to a destination. It is used analogously to a water stream. In the context of `cout`, the stream flows data from our program to the console window for display. In the context of `cin`, the stream flows data from the keyboard into our program. Similarly, an `ofstream` object flows data from our program to a file, thereby “writing (saving) data to the file,” and an `ifstream` object flows data from a file into our program, thereby “reading (loading) data from the file.”

4. There are two different kinds of files we work with: text files and binary files. Text files are convenient because they are readable by humans, thereby making the files easier to edit and making file I/O bugs easier to fix. Binary files are convenient because they tend to consume less memory than equivalent text files and they are streamed more efficiently since binary data is not converted to a text format; rather, the raw bytes of the data are directly saved. When constructing a binary file, remember to specify the `ios_base::binary` flag to the second parameter of the constructor or to the `open` method.

5. When writing and reading to and from text files you use the insertion (<<) and extraction (>>) operators, just as you would with `cout` and `cin`, respectively. When writing and reading to and from binary files you use the `ofstream::write` and `ifstream::read` methods, respectively.

8.5 Exercises

8.5.1 Line Count

Write a program that prompts the user to enter in a string path directory to a text file. For example:

C:/Data/file.txt

Your program must then open this text file and count how many lines the text file contains. Before terminating the program, you should output to the console window how many lines the file contains. Example output:

```
Enter a text file: C:/Data/file.txt
C:/Data/file.txt contained 478 lines.
Press any key to continue
```

Because, in general, you do not know how many lines a text file has, you will need a way to determine when the end of the file is reached. You can do that with the `ifstream::eof` (eof = end of file) method, which returns `true` if the end of the file has been reached, and `false` otherwise. So, your algorithm for this exercise will look something like:

```
while( not end of file )
      read line
      increment line counter
```

8.5.2 Rewrite

1. Rewrite Programs 8.1 and 8.2 of this chapter to use binary files instead of text files.

2. Rewrite Programs 8.3 and 8.4 of this chapter to use text files instead of binary files.

Chapter 9

Inheritance and Polymorphism

Introduction

Any modern programming language must provide mechanisms for code reuse whenever possible. The main benefits of code reuse are increased productivity and easier maintenance. Part of code reuse is functions and classes, but generalizing and abstracting code is also equally important.

In programming, we generalize things for the same reason the mathematician does. The mathematician generalizes mathematical objects so that he/she does not solve the same problem more than once in different forms. By solving a problem with the most general form, the solution applies to all of the specific forms as well. Similarly, in C++, via the **inheritance** mechanism, we can define a generalized class, and give all the data properties and functionality of that generalized class to more specific classes. In this way, we only have to write the general "shared" code once, and we can reuse it with several specific classes. So, whereas the mathematician saves work by applying a general solution to a variety of specific problems, the programmer saves work by applying general class code to a variety of specific classes. The concept of code reuse via inheritance is the first theme of this chapter.

In addition to basic generalization, we would like to work with a set of specific class objects at a general level. For example, suppose we are writing an art program and we need various specific class shapes such as `Lines`, `Rectangles`, `Circles`, `Curves`, and so on. Since these are all shapes, we would likely use inheritance and give them properties and functions from a general class `Shape`. By combining all the `Line` objects, `Rectangle` objects, `Circle` objects, and `Curve` objects into one `Shape` list (e.g., array), we give ourselves the ability to work with *all* shapes at a higher level. For instance, we can iterate over all the shapes and have them draw themselves, without regard to the specific shape. Moreover, it is possible to generalize the specific shape objects up to `Shape`, and for them to still "know" how to draw themselves (that is, the specific shape) correctly in this general form, with the use of **polymorphism.**

Chapter Objectives

- Understand what inheritance means in C++ and why it is a useful code construct.
- Understand the syntax of polymorphism, how it works, and why it is useful.
- Learn how to create general abstract types and interfaces.

9.1 Inheritance Basics

Inheritance allows a **derived class** (also called a **child class or subclass**) to inherit the data and methods of a **base class** (also called a **parent class or superclass**). For example, suppose we are working on a futuristic spaceship simulator game, where earthlings must fight off an enemy alien race from another galaxy. We start off designing our class as generally as possible, with the hopes of reusing its general properties and methods for more specific classes, and thereby avoiding code duplication. First we will have a class `Spaceship`, which is quite general, as there may be many different kinds of models of `Spaceships` (such as cargo ships, mother ships, fighter ships, bomber ships, and so on). At the very least, we can say a `Spaceship` has a model name, a position in space, a velocity specifying its speed and direction, a fuel level, and a variable to keep track of the ship damage. As far as methods go—that is, what actions a `Spaceship` can do—we will say *all* spaceships can fly and print their statistics, but we do not say anything else about them at this general level. It is not hard to imagine some additional properties and possible methods that would fit at this general level, but this is good enough for our purposes in this example. Our general `Spaceship` class (and implementation) now looks like this:

```
// From Spaceship.h
class Spaceship
{
public:
      Spaceship();
      Spaceship(
            const string& name,
            const Vector3& pos,
            const Vector3& vel,
            int fuel,
            int damage);

      void fly();
      void printStats();

protected:
      string  mName;
      Vector3 mPosition;
      Vector3 mVelocity;
      int     mFuelLevel;
      int     mDamage;
};

//=========================================================================
// From Spaceship.cpp
Spaceship::Spaceship()
{
      mName      = "DefaultName";
      mPosition  = Vector3(0.0f, 0.0f, 0.0f);
      mVelocity  = Vector3(0.0f, 0.0f, 0.0f);
      mFuelLevel = 100;
      mDamage    = 0;
}
```

```
Spaceship::Spaceship(const string& name,
                         const Vector3& pos,
                         const Vector3& vel,
                         int fuel,
                         int damage)
{
      mName      = name;
      mPosition  = pos;
      mVelocity  = vel;
      mFuelLevel = fuel;
      mDamage    = damage;
}



void Spaceship::fly()
{
      cout << "Spaceship flying" << endl;
}

void Spaceship::printStats()
{
      // Print out ship statistics.

      cout << "==========================" << endl;
      cout << "Name      = " << mName      << endl;
      cout << "Position  = " << mPosition  << endl;
      cout << "Velocity  = " << mVelocity  << endl;
      cout << "FuelLevel = " << mFuelLevel << endl;
      cout << "Damage    = " << mDamage    << endl;
}
```

Note that we do not include any specific attributes, such as weapon properties nor specific methods such as attack, because such properties and methods are specific to particular kinds of spaceships—and are not *general* attributes for *all* spaceships. Remember, we are starting off generally first.

> **Note:** Observe the new keyword `protected`, which we have used in place of `private`. Recall that only the class itself and `friends` can access data members in the `private` area. This would prevent a derived class from accessing the data members. We do not want such a restriction with a class that is designed for the purposes of inheritance and the derivation of child classes. After all, what good is inheriting properties and methods you cannot access?. In order to achieve the same effect as `private`, but allow derived classes to access the data members, C++ provides the `protected` keyword. The result is that derived classes get access to such members, but outsiders are still restricted.

With `Spaceship` defining the general properties and methods of spaceships, we can define some particular kinds of spaceships which inherit the properties and methods of `Spaceships`—after all, these specific spaceships *are* kinds of spaceships:

```
// From Spaceship.h
class FighterShip : public Spaceship
{
public:
      FighterShip(
            const string& name,
            const Vector3& pos,
            const Vector3& vel,
            int fuel,
            int damage,
            int numMissiles);

      void fireLaserGun();
      void fireMissile();

private:
      int mNumMissiles;
};



class BomberShip : public Spaceship
{
public:
      BomberShip(
            const string& name,
            const Vector3& pos,
            const Vector3& vel,
            int fuel,
            int damage,
            int numBombs);

      void dropBomb();

private:
      int mNumBombs;
};


//========================================================================
// From Spaceship.cpp
FighterShip::FighterShip(const string& name,
                             const Vector3& pos,
                             const Vector3& vel,
                             int fuel,
                             int damage,
                             int numMissiles)
      // Call spaceship constructor to initialize "Spaceship" part.
      : Spaceship(name, pos, vel, fuel, damage)
{
      // Initialize "FighterShip" part.
      mNumMissiles = numMissiles;
}
```

```
void FighterShip::fireLaserGun()
{
      cout << "Firing laser gun." << endl;
}

void FighterShip::fireMissile()
{
      // Check if we have missiles left.
      if( mNumMissiles > 0 )
      {
            // Yes, so fire the missile.
            cout << "Firing missile." << endl;

            // Decrement our missile count.
            mNumMissiles--;
      }
      else // Nope, no missiles left.
            cout << "Out of missiles." << endl;
}


BomberShip::BomberShip(const string& name,
                           const Vector3& pos,
                           const Vector3& vel,
                           int fuel,
                           int damage,
                           int numBombs)
    // Call spaceship constructor to initialize "Spaceship" part.
      : Spaceship(name, pos, vel, fuel, damage)
{
      // Initialize "BomberShip" part.
      mNumBombs = numBombs;
}

void BomberShip::dropBomb()
{
      // Check if we have bombs left.
      if( mNumBombs > 0 )
      {
            // Yes, so drop the bomb.
            cout << "Dropping bomb." << endl;

            // Decrement our bomb count.
            mNumBombs--;
      }
      else // Nope, no bombs left.
            cout << "Out of bombs." << endl;
}
```

We only show two specific spaceships here, but you could easily define and implement a cargo ship and a mother ship, as appropriate. Note how we added specific data; that is, bombs are specific to a `BomberShip`, and missiles are specific to a `FighterShip`. In a real game we would probably need to add more data and methods, but this will suffice for illustration.

Observe the colon syntax that follows the class name. Specifically:

```
: public Spaceship
```

This is the inheritance syntax, and reads "inherits publicly from `Spaceship`." So the line:

```
class FighterShip : public Spaceship
```

says that the class `FighterShip` inherits publicly from `Spaceship`. Also, the line:

```
class BomberShip : public Spaceship
```

says that the class `BomberShip` inherits publicly from `Spaceship`. We discuss what public inheritance means and how it differs from, say, private inheritance in Section 9.2.3.

Another important piece of syntax worth emphasizing is where we call the parent constructor:

```
// Call spaceship constructor to initialize "Spaceship" part.
: Spaceship(name, pos, vel, fuel, damage)
```

As we shall discuss in more detail later on, we can view a derived class as being made up of two parts: the parent class part, and the part specific to the derived class. Consequently, we can invoke the parent's constructor to construct the parent part of the class. What is interesting here is *where* we call the parent constructor—we do it after the derived constructor's parameter list and following a colon, but before the derived constructor's body. This is called a **member initialization list**:

```
ClassName::ClassName(parameter-list…)
:   // Member initialization list
{
}
```

When an object is instantiated, the memory of its members is first constructed (or initialized) to something before the class constructor code is executed. We can explicitly specify how a member variable should be constructed in the member initialization list. And in particular, if we want to invoke the parent constructor to construct the parent part of the class, then we *must* invoke it in the member initialization list—where the parent part is being constructed. Note that we are not limited to calling parent constructors in the member initialization list. We can also specify how other variables are initialized. For example, we could rewrite the `FighterShip` constructor like so:

```
FighterShip::FighterShip(const string& name,
                             const Vector3& pos,
                             const Vector3& vel,
                             int fuel,
                             int damage,
                             int numMissiles)
      // Call spaceship constructor to initialize "Spaceship" part.
      : Spaceship(name, pos, vel, fuel, damage),
        mNumMissiles(numMissiles) // Initialize "FighterShip" part.
{}
```

Here we directly construct the integer `mNumMissiles` with the value `numMissiles`, rather than make an assignment to it in the constructor body after it has been constructed. That is:

```
mNumMissiles = numMissiles;
```

This has performance implications, as Scott Meyers points out in *Effective C++*. Specifically, by using a member initialization list, we only do one operation—construction. If we do *not* use a member initialization list we end up doing two operations: 1) construction to a default value, and 2) an assignment in the constructor body. So by using a member initialization list, we can reduce two operations down to one. Such a reduction can become significant with large classes and with large arrays of classes.

> **Note:** Inheritance relationships are often depicted graphically. For example, our spaceship inheritance hierarchy would be drawn as follows:

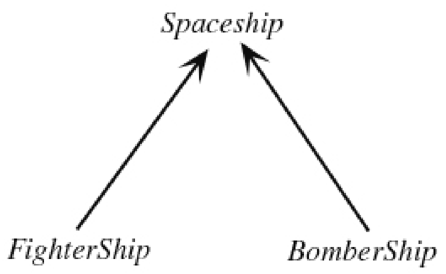


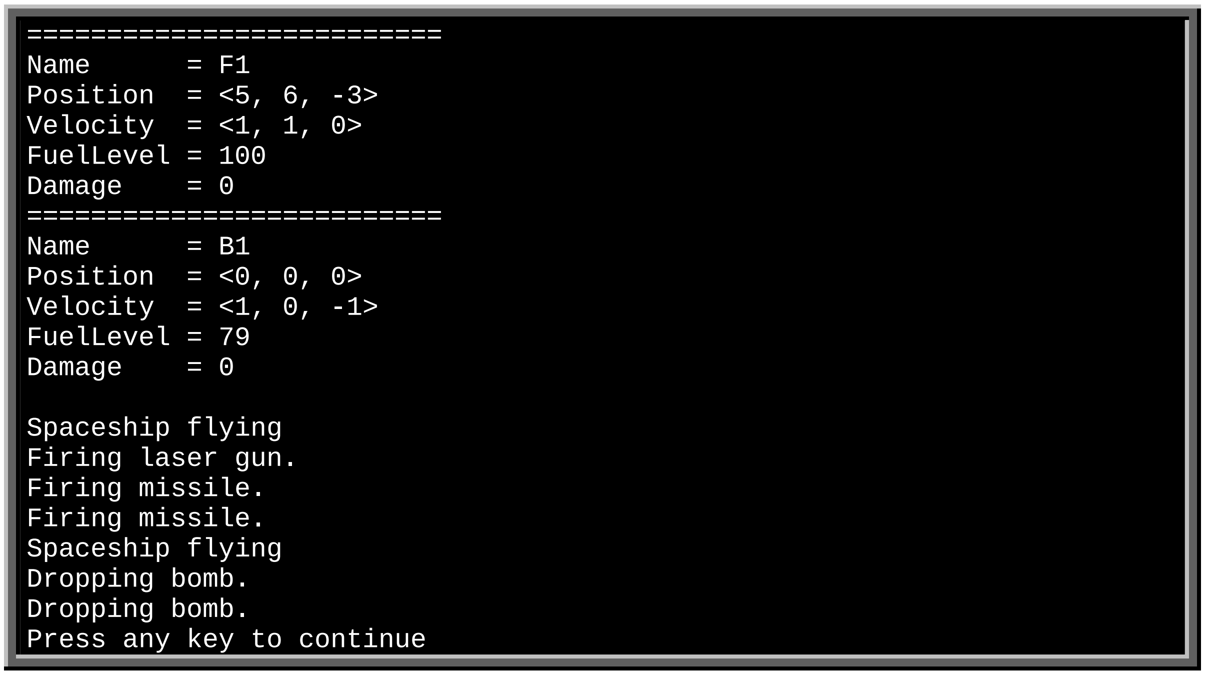
Figure 9.1: A simple graphical inheritance relationship.

Now that we have some specific spaceships, let us put them to use in a small sample program.

Program 9.1: Using derived classes.

```
// main.cpp

#include <iostream>
#include "Spaceship.h"
using namespace std;

int main()
{
      FighterShip fighter("F1", Vector3(5.0f, 6.0f, -3.0f),
            Vector3(1.0f, 1.0f, 0.0f), 100, 0, 10);

      BomberShip bomber("B1", Vector3(0.0f, 0.0f, 0.0f),
            Vector3(1.0f, 0.0f, -1.0f), 79, 0, 5);

      fighter.printStats();
      bomber.printStats();
      cout << endl;
      fighter.fly();
      fighter.fireLaserGun();
      fighter.fireMissile();
      fighter.fireMissile();

      bomber.fly();
      bomber.dropBomb();
      bomber.dropBomb();
}
```

Program 9.1 Output



```
==========================
Name      = F1
Position  = <5, 6, -3>
Velocity  = <1, 1, 0>
FuelLevel = 100
Damage    = 0
==========================
Name      = B1
Position  = <0, 0, 0>
Velocity  = <1, 0, -1>
FuelLevel = 79
Damage    = 0

Spaceship flying
Firing laser gun.
Firing missile.
Firing missile.
Spaceship flying
Dropping bomb.
Dropping bomb.
Press any key to continue
```

This is a simple program where we just instantiate a few objects and call their methods. What is of interest to us is how `fighter` and `bomber` can call the methods of `Spaceship`; in particular, they both call the `fly` and `printStats` methods. Also notice that `printStats` prints the stats of `fighter` and `bomber`, thereby showing that they inherited the data members of `Spaceship`. Thus we can see that they have their own copies of these data members. Again, this is because `FighterShip` and `BomberShip` inherit from `Spaceship`.

Do you see the benefit of inheritance? If we had not used inheritance then we would have had to duplicate all the data and methods (and their implementations) contained in `Spaceship` for both `FighterShip` and `BomberShip`, and any other new kind of spaceship we wanted to add. However, with inheritance, all of that information and functionality is inherited by the derived classes automatically, and we do not have to duplicate it. Hopefully this gives you a more intuitive notion of inheritance and its benefits.

9.2 Inheritance Details

9.2.1 Repeated Inheritance

In the previous section we used inheritance for a single generation; that is, parent and child. Naturally, one might wonder whether we can create more complex relationships, such as grandparent, parent, and child. In fact, we can create inheritance hierarchies as large and as deep as we like—there is no limit imposed. Figure 9.2 shows a more complex spaceship inheritance hierarchy.

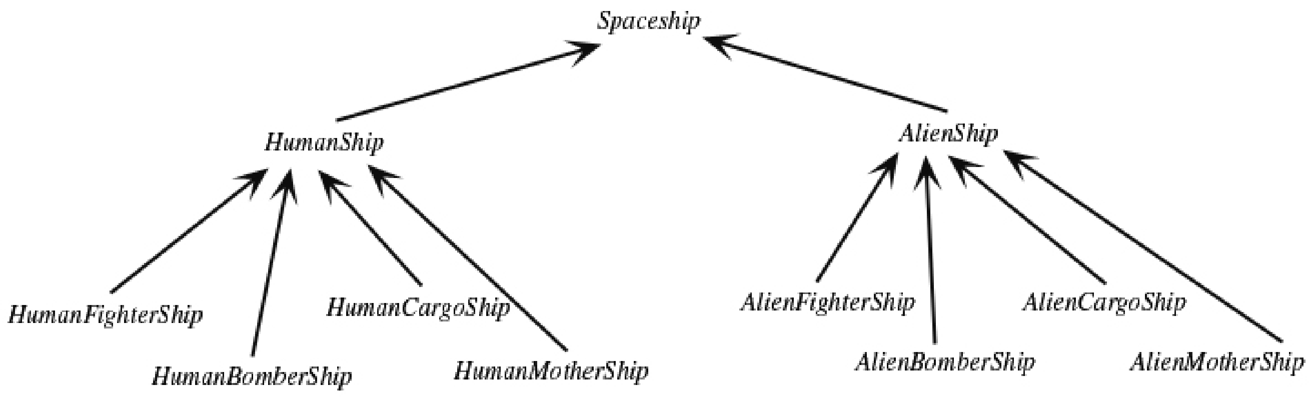


Figure 9.2: An inheritance hierarchy.

To create this hierarchy in code we write the following (with class details omitted for brevity):

```
class Spaceship { [...] };

class AlienShip        : public Spaceship { [...] };
class AlienFighterShip : public AlienShip { [...] };
class AlienBomberShip  : public AlienShip { [...] };
class AlienCargoShip   : public AlienShip { [...] };
class AlienMotherShip  : public AlienShip { [...] };

class HumanShip        : public Spaceship { [...] };
class HumanFighterShip : public HumanShip { [...] };
class HumanBomberShip  : public HumanShip { [...] };
class HumanCargoShip   : public HumanShip { [...] };
class HumanMotherShip  : public HumanShip { [...] };
```

9.2.2 *isa* versus *hasa*

When a class contains a variable of some type *T* as a member variable, we say the class "**has a**" *T*. For example, the data members of `Spaceship` were:

```
string  mName;
```

```
Vector3 mPosition;
Vector3 mVelocity;
int     mFuelLevel;
int     mDamage;
```

We say a `Spaceship` *has a* `string`, two `Vector3`s, and two `int`s. Incidentally, when we compose a class out of other types, object oriented programmers use the term **composition** to denote this. That is, the class is 'composed of' those other types.

When a class *A* inherits publicly from a class *B*, object oriented programmers say that we are modeling an "**is a**" relationship; that is, *A is a B*, but not conversely. Essentially, this is what public inheritance means—*is a*. For example, in our previous spaceship examples, our specific spaceships `FighterShip` and `BomberShip` inherited publicly from `Spaceship`. This is conceptually correct because `FighterShip` *is a* kind of `Spaceship` and `Bombership` *is a* kind of `Spaceship`. However, the reverse is not true. That is, a `Spaceship` is not necessarily a `FighterShip` and a `Spaceship` is not necessarily a `Bombership`. This is important terminology. C++ Guru Scott Meyers says this about the terminology in his book *Effective C++*: "[…] the single most important rule in object-oriented programming with C++ is this: public inheritance means "isa." Commit this rule to memory."

9.2.3 Moving Between the Base Class and Derived Class

Why is an *is a* relationship important? As we said, when a class *A* inherits publicly from a class *B*, we specify the relationship that *A is a B*. Consequently, with this relationship defined, C++ allows us to convert an *A* object into a *B* object. After all, an *A* object *is a* kind of *B* object. To better illustrate, let us take a moment to review.

Recall that inheritance extends a class. For example, consider a class called `Base`:

```
class Base
{
public:
      void f();
      void g();
protected:
      int mBaseData1;
      float mBaseData2;
      std::string mBaseData3;
};
```

Suppose we need to create a new distinct class, called `Derived`, which contains all the methods and data of `Base`, but adds some additional data and methods specific to `Derived`. In other words, `Derived` extends `Base`. Instead of recopying the data and functionality from `Base` into `Derived`, we can take advantage of the C++ inheritance mechanism to do this for us:

```
class Derived : public Base
```

```
{
public:
      void h();
protected:
      char  mDerivedData1[4];
      std::vector<int> mDerivedData2;
};
```

We note that since `Derived` inherits publicly from `Base`, it contains all the methods[5] and data of `Base`, plus the additional methods and data specific to `Derived`. Moreover, because of the public inheritance we specify that `Derived` *is a* `Base`.

Because `Derived` inherits the data of `Base`, the data layout of a `Derived` object consists of a `Base` part. Figure 9.3 illustrates:

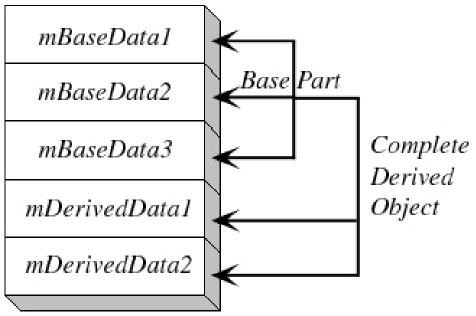


Figure9.3: A derived object consists of a Base part.

Furthermore, because `Derived` *is a* `Base` (public inheritance), we can switch back and forth between the `Derived` object and its `Base` part via pointer casting:

```
Derived* derived = new Derived();
Base*    base    = (Base*)derived; // upcast
Derived* d2      = (Derived*)base; // downcast
```

The upcast is considered safe and can be done implicitly as shown here:

```
Base*    base    = new Derived();  // upcast done implicitly
Derived* d2      = (Derived*)base; // downcast
```

We use the term **upcast** when casting up the inheritance chain; that is, from a derived object to its base part. Likewise, we use the term **downcast** when casting down the inheritance chain; that is, from the base part to the derived object. Note that downcasting is *not* always safe. Remember, a `Derived` object *is a* specific kind of `Base` object, but not conversely. In other words, a `Derived` object will always

[5] Excepting constructors, destructors, and the assignment operator. Obviously the constructor and destructor are not inherited since they say nothing about creating or destroying the derived object. The assignment operator is masked since every class has its own assignment operator, by default; if you do not implement one, the compiler implements a default one for you.

have a base part, but a Base object is not necessarily part of a Derived object. For example, we can write this:

```
Base* pureBase = new Base();
```

Here pureBase is purely a Base object—it is not part of a Derived object and therefore it is illegal to downcast. If you think about it for a moment you will realize why – the memory for the Derived object data was never allocated since a Derived object was never constructed. Only the Base data exists after the new call. Using a downcasted pointer to access derived data members that were never created would result in a nasty problem indeed. The following, on the other hand, is different:

```
Derived* derived = new Derived();
Base*    base    = (Base*)derived; // upcast
Derived* d2      = (Derived*)base; // downcast
```

In this case we can downcast from base to d2. The difference is that base is part of a Derived object (i.e., we first upcasted from a Derived object to base).

Before we conclude this section, let us examine some useful vocabulary. Consider the following:

```
Base* base = new Derived();
```

We say that the variable base has the **static type** Base and the **dynamic type** Derived. Note that this assignment is completely legal since the memory for the Derived class object has been fully allocated in the new call. The base pointer would be able to access all of the Base class components and could be safely downcast to a Derived type pointer as needed. This is legal since any downcasted Derived pointer would have an actual Derived object (with all of its memory fully intact) to work with.

9.2.4 Public versus Private Inheritance

We have discussed public inheritance as modeling an *is a* relationship, but it seems that if we must explicitly specify public inheritance then there must be another type of inheritance. Indeed there is, and it is called **private inheritance.** To specify private inheritance you just replace public with private in the inheritance syntax:

```
class Derived : private Base
```

Private inheritance does *not* mean *is a*, and consequently, we should not be able to upcast and downcast the inheritance hierarchy. With private inheritance, inherited members and methods automatically become private no matter their previously declared access level.

Private inheritance is useful when you want to reuse code via inheritance, but do not want to make the *is a* relationship claim. That is, you want to prevent upcasts and downcasts, because the *is a* relationship does not make sense for what you are modeling. Private inheritance is a way to express that in the language. To quote Scott Meyers's *Effective C++*: “If you make a class D privately inherit from a class

B, you do so because you are interested in taking advantage of some of the code that has already been written for class B, not because there is any conceptual relationship between objects of type B and objects of type D."

9.2.5 Method Overriding

Recall that in Program 9.1 we wrote the line:

```
fighter.printStats();
```

which produced the output:

```
==========================
Name      = F1
Position  = <5, 6, -3>
Velocity  = <1, 1, 0>
FuelLevel = 100
Damage    = 0
```

Also recall that `printStats` was a method inherited from `Spaceship`, and was implemented like so:

```
void Spaceship::printStats()
{
      // Print out ship statistics.

      cout << "==========================" << endl;
      cout << "Name      = " << mName      << endl;
      cout << "Position  = " << mPosition  << endl;
      cout << "Velocity  = " << mVelocity  << endl;
      cout << "FuelLevel = " << mFuelLevel << endl;
      cout << "Damage    = " << mDamage    << endl;
}
```

The problem here is that `FighterShip` has properties which are specific to it, but `Spaceship` being the more general class, does not know about them and cannot print them. In particular, `FighterShip` has the member `mNumMissiles`, and most likely we would want to print the number of missiles a `FighterShip` has remaining along with its other statistics. Thus, what we must do is **override** `printStats` for `FighterShip`s:

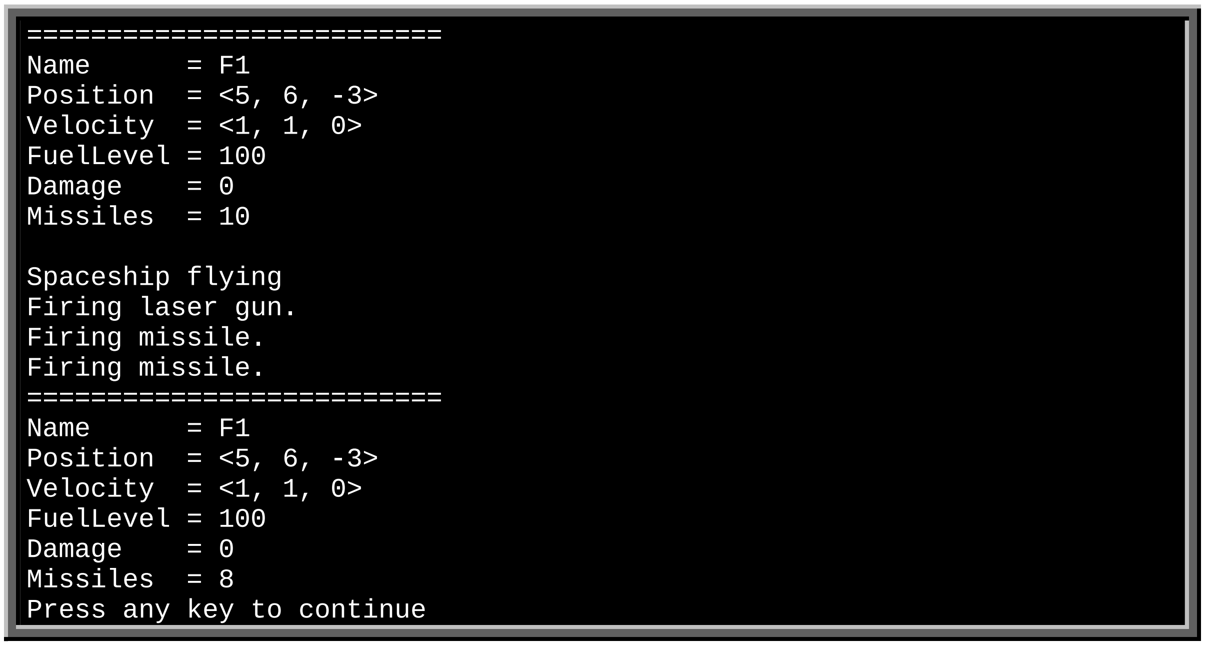
```
void FighterShip::printStats()
{
      Spaceship::printStats();
      cout << "Missiles  = " << mNumMissiles << endl;
}
```

Luckily, we can still reuse the code from `Spaceship`. As you can see, we call the parent version `Spaceship::printStats` to print the `Spaceship` part of `FighterShip`, and then we add only the new code specific to `FighterShip`—so we are still reusing code. We could also do something similar for `BomberShip`. Program 9.2 shows our new overridden method in action:

Program 9.2: Overridden method.

```
#include <iostream>
#include "Spaceship.h"
using namespace std;

int main()
{
      FighterShip fighter("F1", Vector3(5.0f, 6.0f, -3.0f),
            Vector3(1.0f, 1.0f, 0.0f), 100, 0, 10);

      fighter.printStats();
      cout << endl;

      fighter.fly();
      fighter.fireLaserGun();
      fighter.fireMissile();
      fighter.fireMissile();

      fighter.printStats();
}
```

Program 9.2 Output



```
==========================
Name      = F1
Position  = <5, 6, -3>
Velocity  = <1, 1, 0>
FuelLevel = 100
Damage    = 0
Missiles  = 10

Spaceship flying
Firing laser gun.
Firing missile.
Firing missile.
==========================
Name      = F1
Position  = <5, 6, -3>
Velocity  = <1, 1, 0>
FuelLevel = 100
Damage    = 0
Missiles  = 8
Press any key to continue
```

As you can see, when we call `printStats`, it prints the number of missiles the ship has remaining.

Now having said all of this, in general you should *not* override methods in this fashion—instead use polymorphism, which we will discuss later. The reason is that if you override the method at different levels of the inheritance hierarchy, and you cast up and down the hierarchy as Section 9.2.3 describes, you will change what version of the method gets called depending where you are in the inheritance hierarchy. This is usually not desired.

9.3 Constructors and Destructors with Inheritance

Consider the following simple program:

Program 9.3: Derived object's construction and destruction order.

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

class Spaceship
{
public:
      Spaceship();
      ~Spaceship();

};

class AlienShip : public Spaceship
{
public:
      AlienShip();
      ~AlienShip();

};

class AlienBomberShip : public AlienShip
{
public:
      AlienBomberShip();
      ~AlienBomberShip();

};

Spaceship::Spaceship()
{
      cout << "Spaceship() Constructor called." << endl;
}

Spaceship::~Spaceship()
{
      cout << "~Spaceship() Destructor called." << endl;
}

AlienShip::AlienShip()
{
      cout << "AlienShip() Constructor called." << endl;
}
```

```
AlienShip::~AlienShip()
{
      cout << "~AlienShip() Destructor called." << endl;
}


AlienBomberShip::AlienBomberShip()
{
      cout << "AlienBomberShip() Constructor called." << endl;
}

AlienBomberShip::~AlienBomberShip()
{
      cout << "~AlienBomberShip() Destructor called." << endl;
}

int main()
{
      // Construct an AlienBomberShip
      AlienBomberShip alienShip;

      // 'alienShip' will be destroyed when 'main' exits.
}
```

Program 9.3 Output

```
Spaceship() Constructor called.
AlienShip() Constructor called.
AlienBomberShip() Constructor called.
~AlienBomberShip() Destructor called.
~AlienShip() Destructor called.
~Spaceship() Destructor called.
Press any key to continue
```

The output of Program 9.3 illustrates that when an object of a child class is constructed, the parent constructors are invoked in a descending order starting from the top of the hierarchy. For example, in Program 9.3, first the `Spaceship` part of `AlienBomberShip` is constructed, then the `AlienShip` part of `AlienBomberShip` is constructed, and finally the `AlienBomberShip` constructor is invoked. Destruction occurs in the reverse order; that is, the destructors are invoked in an ascending order starting at the bottom of the inheritance hierarchy. Particularly in Program 9.3, first the `AlienBomberShip` destructor is called, then the `AlienShip` destructor is called, and finally the `Spaceship` destructor is called.

9.4 Multiple Inheritance

One might wonder if a class can inherit from more than one class. After all, classes in the real world can be found that share properties of more than one class. For example, humans inherit traits from both of their parents. C++, unlike some other programming languages, *does* support multiple inheritance and the syntax is rather trivial; we simply specify an additional class to inherit from using a separating comma like so:

```
class AlienShip : public Spaceship, public DrawableObject
{
	...
}
```

Here we have said an `AlienShip` *is a* `Spaceship`, but it *is* also *a* `DrawableObject` (drawable in the sense that our 3D engine is capable of drawing an `AlienShip`.)

Although multiple inheritance can solve some problems quite elegantly, it is not without its pitfalls. However, we will not discuss them in this text because, although multiple inheritance is useful in some situations, it is not encountered very frequently. There is no need to spend a lot of time on a feature that is rarely used and indeed, multiple inheritance is not used at all in this course. Multiple inheritance is a feature that you need only be aware of in case you come across a situation where it might be useful. At that time, you can investigate the potential problems with using it. Scott Meyers' *Effective C++* spends a solid fourteen pages on the problems of multiple inheritance and how to use multiple inheritance effectively if you are interested in a deeper look.

9.5 Polymorphism

In order to introduce polymorphism, we will first present a problem. Our first attempt to solve this problem will result in failure. We will then show a correct solution to this problem using polymorphism, but we will not explain why and how polymorphism works until the following section.

Problem: Create a `Shape` class and derive two other classes, `Circle` and `Rectangle`, from it. Each shape should know its type (a string describing what kind of shape it is; e.g., "circle" and "rectangle"), and be able to calculate its area and perimeter. Using these classes, instantiate five different circle objects and five different rectangle objects, and upcast them to `Shape` so that you can store all the circles and rectangles in a common `Shape` pointer container (i.e., array/`std::vector`). Finally, iterate through each element in the `Shape` container and output the element's shape type (is it a circle or rectangle?), its area, and its perimeter.

9.5.1 First Attempt (Incorrect Solution)

The restrictions which the problem imposes are what make the problem difficult. In particular, a problem will occur when we upcast our `Circle` and `Rectangle` objects to `Shapes`. Let us attempt to do what the problem requires and see where it takes us.

We create a general `Shape` class like so:

```
Class Shape
{
public:
      string type();
      float  area();
      float  perimeter();

};
```

We are already running into problems when we try to implement this class. Specifically, what is the type of a shape? What is the area of a shape? What is the perimeter of a shape? We only know types, areas and perimeters of specific shapes like circles and rectangles, but not of a general shape. In fact, the entire concept of a shape is abstract. In order to continue, let us have the `Shape` implementations of these methods return "undefined" for the shape type, and zero for area and perimeter. Then we will override the `type`, `area` and `perimeter` methods in the derived classes with a proper implementation. The following code shows the agreed implementation of `Shape`:

```
string Shape::type()
{
      return "undefined";
}

float Shape::area()
{
      return 0.0f;
}

float Shape::perimeter()
{
      return 0.0f;
}
```

Moving on to the `Circle` and `Rectangle` class, we have the following class definitions:

```
class Circle : public Shape
{
public:
      Circle(float rad);

      string type();
      float  area();
      float  perimeter();
protected:
	float mRadius;
};

//==============================================================

class Rectangle : public Shape
{
public:
	Rectangle(float w, float l);

	string type();
	float  area();
	float  perimeter();

protected:
	float mWidth;
	float mLength;
};
```

We have the implementations:

```
Circle::Circle(float rad)
: mRadius(rad)
{
}

string Circle::type()
{
	return "Circle";
}

const float PI = 3.14f;

float Circle::area()
{
	return PI*mRadius*mRadius;
}

float Circle::perimeter()
{
	return 2.0f*PI*mRadius;
}

//==============================================================

Rectangle::Rectangle(float w, float l)
: mWidth(w), mLength(l)
{
}

string Rectangle::type()
{
	return "Rectangle";
}
```

```
float Rectangle::area()
{
      return mWidth*mLength;
}

float Rectangle::perimeter()
{
      return 2.0f*mWidth + 2.0f*mLength;
}
```

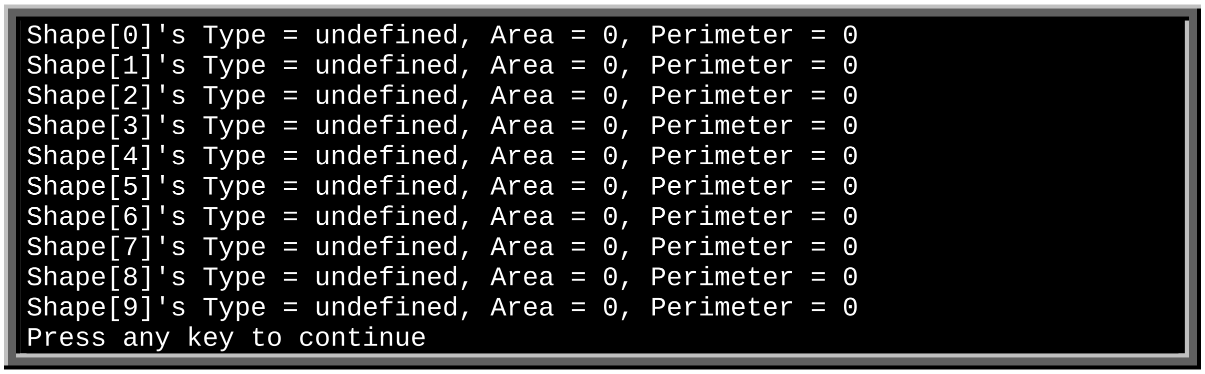
No problems are encountered at this specific level. Now that we have our classes defined, the problem instructs us to create five different `Circle`s and five different `Rectangle`s, and to upcast them into a container of `Shape` pointers. Doing so yields the following code:

```
int main()
{
      Shape* shapes[10];

      shapes[0] = new Circle(1.0f);
      shapes[1] = new Circle(2.0f);
      shapes[2] = new Circle(3.0f);
      shapes[3] = new Circle(4.0f);
      shapes[4] = new Circle(5.0f);

      shapes[5] = new Rectangle(1.0f, 2.0f);
      shapes[6] = new Rectangle(2.0f, 4.0f);
      shapes[7] = new Rectangle(3.0f, 1.0f);
      shapes[8] = new Rectangle(4.0f, 6.0f);
      shapes[9] = new Rectangle(5.0f, 2.0f);
```

Our final task is to iterate through each element in `shapes` and output the element's shape type (a circle or rectangle), its area, and its circumference. Doing so yields the following code:

```
      for(int i = 0; i < 10; ++i)
      {
            string type = shapes[i]->type();
            float area  = shapes[i]->area();
            float peri  = shapes[i]->perimeter();
            cout << "Shape[" << i << "]'s ";
            cout << "Type = " << type << ", ";
            cout << "Area = " << area << ", ";
            cout << "Perimeter = " << peri << endl;
      }

      // Delete the memory.
      for(int i = 0; i < 10; ++i)
      {
          delete shapes[i];
      }
}// end main
```

Now if we execute this program we get the following output:



```
Shape[0]'s Type = undefined, Area = 0, Perimeter = 0
Shape[1]'s Type = undefined, Area = 0, Perimeter = 0
Shape[2]'s Type = undefined, Area = 0, Perimeter = 0
Shape[3]'s Type = undefined, Area = 0, Perimeter = 0
Shape[4]'s Type = undefined, Area = 0, Perimeter = 0
Shape[5]'s Type = undefined, Area = 0, Perimeter = 0
Shape[6]'s Type = undefined, Area = 0, Perimeter = 0
Shape[7]'s Type = undefined, Area = 0, Perimeter = 0
Shape[8]'s Type = undefined, Area = 0, Perimeter = 0
Shape[9]'s Type = undefined, Area = 0, Perimeter = 0
Press any key to continue
```

This is not correct at all. What happened? The `Shape` versions of `type`, `area`, and `circumference` were called. This is not surprising as we upcast our objects to `Shape`. We hinted at this problem at the end of Section 9.2.5 and it might seem that we are at a dead end. However, it is possible to generalize the specific shape objects up to `Shape`, and for them to still "know" how to draw themselves (that is, the specific shape) correctly in this general form, by using **polymorphism**.

This brings us to our second attempt at solving the problem. Before we start, remember the following statement:

```
Base* base = new Derived();
```

We say that the variable `base` has the **static type** `Base` and the **dynamic type** `Derived`. What we want is to be able to upcast a derived type to the base type, but for the dynamic type of the object to be "remembered" so that the dynamic type methods can be invoked, and *not* the base (static type) methods.

9.5.2 Second Attempt (Correct Solution)

We begin as we did in the first attempt, by defining and implementing the `Shape` class. We quickly run into the same previous problem; namely, what is the type of a shape? What is the area of a shape? What is the circumference of a shape? We again choose to ignore this problem and return "dummy" values. So far it seems we are taking the same road we took in the previous attempt. However, we diverge by modifying the methods `type`, `area`, and `circumference` with the `virtual` keyword:

```
class Shape
{
public:
	virtual string type();
	virtual float  area();
	virtual float  perimeter();

};

string Shape::type()
{
	return "undefined";
}
```

```
float Shape::area()
{
	return 0.0f;
}

float Shape::perimeter()
{
	return 0.0f;
}
```

Methods prefixed with the `virtual` keyword are called **virtual functions** or **virtual methods** (usually the former).

We now proceed to finish the program exactly as we did in the first attempt. Moving on to the `Circle` and `Rectangle` class, we again have the following class definitions (same as in our first attempt):

```
class Circle : public Shape
{
public:
	Circle(float rad);

	string type();
	float  area();
	float  perimeter();

protected:
	float mRadius;
};

//================================================================

class Rectangle : public Shape
{
public:
	Rectangle(float w, float l);

	string type();
	float  area();
	float  perimeter();

protected:
	float mWidth;
	float mLength;
};
```

Implementations:

```
Circle::Circle(float rad)
: mRadius(rad)
{
}
```

```
string Circle::type()
{
	return "Circle";
}

const float PI = 3.14f;

float Circle::area()
{
	return PI*mRadius*mRadius;
}

float Circle::perimeter()
{
	return 2.0f*PI*mRadius;
}

//==============================================================

Rectangle::Rectangle(float w, float l)
: mWidth(w), mLength(l)
{
}

string Rectangle::type()
{
	return "Rectangle";
}

float Rectangle::area()
{
	return mWidth*mLength;
}

float Rectangle::perimeter()
{
	return 2.0f*mWidth + 2.0f*mLength;
}
```

Now that we have our classes defined, the problem instructs us to create five different `Circles` and five different `Rectangles`, and to upcast them into a container of `Shape` pointers. Doing so yields the following code:

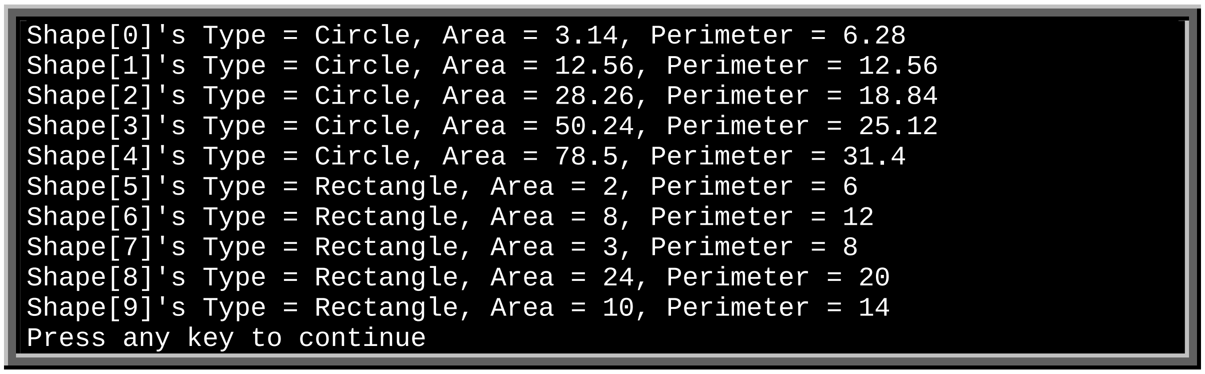
```
int main()
{
	Shape* shapes[10];

	shapes[0] = new Circle(1.0f);
	shapes[1] = new Circle(2.0f);
	shapes[2] = new Circle(3.0f);
	shapes[3] = new Circle(4.0f);
	shapes[4] = new Circle(5.0f);

	shapes[5] = new Rectangle(1.0f, 2.0f);
	shapes[6] = new Rectangle(2.0f, 4.0f);
```

```
      shapes[7] = new Rectangle(3.0f, 1.0f);
      shapes[8] = new Rectangle(4.0f, 6.0f);
      shapes[9] = new Rectangle(5.0f, 2.0f);
```

Our final task is to iterate through each element in `shapes` and output the element's shape type (a circle or rectangle), its area, and its circumference. Doing so yields the following code:

```
      for(int i = 0; i < 10; ++i)
      {
            string type = shapes[i]->type();
            float area  = shapes[i]->area();
            float peri  = shapes[i]->perimeter();
            cout << "Shape[" << i << "]'s ";
            cout << "Type = " << type << ", ";
            cout << "Area = " << area << ", ";
            cout << "Perimeter = " << peri << endl;
      }

    // Delete the memory.
    for(int i = 0; i < 10; ++i)
    {
        delete shapes[i];
    }

}// end main
```

Now if we execute this program we get the following output:



```
Shape[0]'s Type = Circle, Area = 3.14, Perimeter = 6.28
Shape[1]'s Type = Circle, Area = 12.56, Perimeter = 12.56
Shape[2]'s Type = Circle, Area = 28.26, Perimeter = 18.84
Shape[3]'s Type = Circle, Area = 50.24, Perimeter = 25.12
Shape[4]'s Type = Circle, Area = 78.5, Perimeter = 31.4
Shape[5]'s Type = Rectangle, Area = 2, Perimeter = 6
Shape[6]'s Type = Rectangle, Area = 8, Perimeter = 12
Shape[7]'s Type = Rectangle, Area = 3, Perimeter = 8
Shape[8]'s Type = Rectangle, Area = 24, Perimeter = 20
Shape[9]'s Type = Rectangle, Area = 10, Perimeter = 14
Press any key to continue
```

We note that this time, the output is correct. We call this behavior **polymorphism;** that is, we have upcasted our objects to a more general `Shape` type, which behave like their original type. The `Shapes` have "many forms" (polymorphism). Classes with virtual functions are termed **polymorphic.**

We emphasize that the only change we made in this second attempt was the `virtual` keyword. So it seems that virtual functions are the solution to our problems. By using virtual functions, the program "magically remembers" the dynamic type of the object and can therefore invoke the specific methods even after the objects are upcasted to a more general type up the inheritance ladder. The next section sets out to show that this is not magic, and describes how and why it works. In other words, we will learn what the `virtual` keyword is instructing the compiler to do behind the scenes.

Before we move on though, let us address the following question: Why use polymorphism? If it were not for the problem description, we would not have had to upcast our `Circles` and `Rectangles` to `Shapes`, and we could have just kept separate containers of `Circles` and `Rectangles` to output the same information. It is difficult to answer this question at this point. Polymorphism is best appreciated through experience. However, we can say that it is convenient to have a centralized general container of objects without having to realize their specific type. In particular, by having such a container we can work with and apply operations to the objects in the most general form, without regard to the specific form. This is convenient because we do not need to have separate branches of code for each specific case—the program "knows" to invoke the method of the dynamic type. We also explore another utility of polymorphism in Section 9.9.

9.6 How Virtual Functions Work

We demonstrated polymorphism with virtual functions but now we will discuss how they work. Typically[6], a virtual table implementation is used. It works like this: When a class contains a virtual function it is given a corresponding **virtual table** (vtable) by the compiler, which is an array of pointers to all of its virtual functions. Note that a vtable contains only virtual functions, not regular functions. Also note that *classes* are given vtables, not *object instances*. Consider the following class hierarchy:

```
class Base
{
public:
      virtual ~Base();
      virtual void f();
      virtual void g();
      virtual void h();
      virtual void r();
      void s(); // non virtual
      void t(); // non virtual
};

class Derived : public Base
{
public:
      virtual ~Derived();
      virtual void f(); // override
      virtual void g(); // override
      void x(); // new method, non virtual
      void y(); // new method, non virtual
};
```

The corresponding virtual tables which the compiler will set aside would look like this:

[6] We say typically because the actual implementation is compiler dependent.

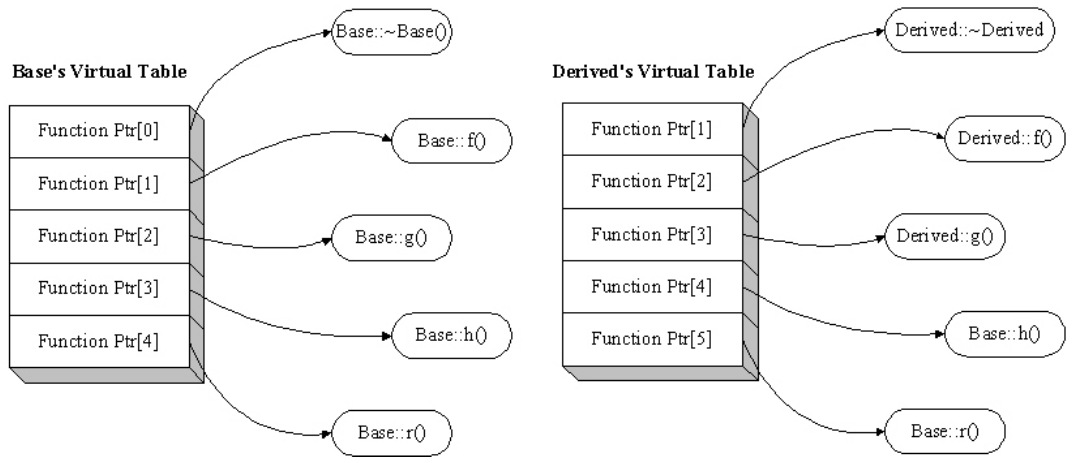


Figure9.4: A class' virtual table contains function pointers to the corresponding class methods. Note that only virtual functions are in the virtual table. Also note that if a derived class does not override a virtual function, then the corresponding derived class' virtual table entries point to the base class' implementations (see h() and r()).

In addition to the vtable, *each* object instance whose class contains virtual functions is given a **virtual table pointer**[7] (vptr), which points to the vtable that corresponds with the object's dynamic type. We note that during the construction of an object we know its dynamic type, and that is the time the vptr can be initialized. For example,

```
Base* obj1 = new Base();
Base* obj2 = new Derived();
```

Here, `obj1`'s vptr points to the vtable of class `Base`. Likewise, `obj2`'s vptr points to the vtable of class `Derived`. Figure 9.5 shows a conceptual diagram to illustrate.

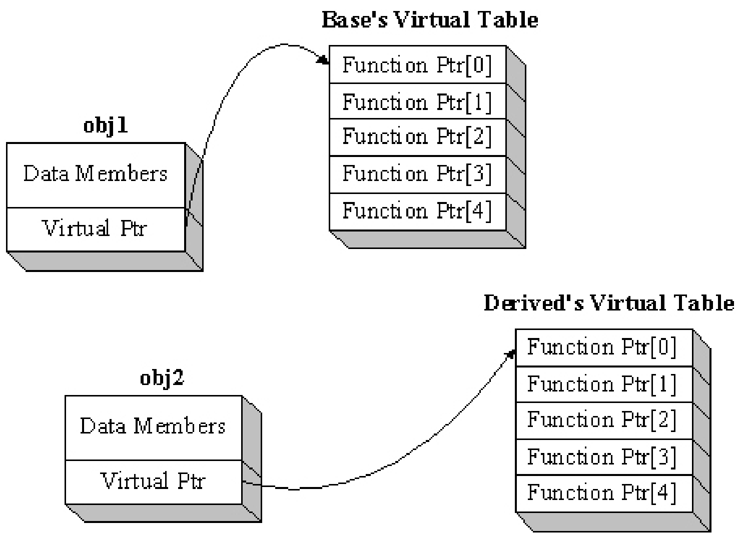


Figure 9.5: An object's virtual pointer points to the virtual table of the object's dynamic type.

[7] The virtual table pointer is stored in some part of the object that only the compiler knows about. It is "hidden" from the programmer.

We now see how virtual functions work—the vptr points to the vtable that stores the "right" methods that correspond to the given object's dynamic type. For example:

```
obj2->h();
```

In this call, the compiler fetches `obj2`'s vptr and follows it to the vtable to which it points, which in this case is `Derived`'s vtable. In this case, once at the vtable it can quickly offset to the right function, `Derived::h()` since each virtual function is given a unique index within the vtable. Finally, `Derived::h()` can be invoked, thus performing the desired result.

> **Note:** If you have virtual functions in your class, then the rule of thumb is to also have a virtual destructor. For example, suppose that `Base` and `Derived` do not have virtual destructors, *which means the destructor is not in the vtable*. What happens when you write:
>
> ```
> Base* base = new Derived();
> ...
> delete base;
> ```
>
> Technically, your program is undefined by the C++ standard. However, what typically happens is that only `Base::~Base` will be called, which means `Derived::~Derived` is not called, which in turn means that you may get memory leaks and errors if you are expecting `Derived::~Derived` to be called. The solution to this problem is to make the destructor virtual so that it gets put into the vtable. Thus "`delete base;`" will call the destructor of the dynamic type (`Derived`) and destroy the object correctly. (Recall that when a derived object is deleted, its local destructor is called, followed by its base class destructor.)

9.7 The Cost of Virtual Functions

1. Increased memory taken up by storing a vtable for each class that contains virtual functions.

2. Increased memory taken up by storing a vptr in each object instance whose class contains virtual functions.

3. Some extra work associated with calling a virtual function; that is, accessing the vptr, dereferencing the vptr to get to the vtable, offsetting into the vtable to get to the correct function pointer, and dereferencing the function pointer to invoke the actual function.

4. Virtual functions are usually not inlined since the compiler cannot determine which function to call at compile time—remember that polymorphism is done at runtime, since dynamic types can be set at runtime.

Finally, to show some supporting evidence of the virtual function implementation just described (vtable/vptr), consider the following program:

Program 9.4: Providing evidence for the vptr.

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

class NonVirtual
{
public:
	void f(){}
protected:
	int number;
	float x;
};

class Virtual
{
public:
	virtual void greeting(){}
protected:
	int number;
	float x;
};

int main()
{
	NonVirtual n;
	Virtual v;
	cout << "Size of NonVirtual = " << sizeof(n) << endl;
	cout << "Size of Virtual    = " << sizeof(v) << endl;
}
```

Program 9.4 Output

```
Size of NonVirtual = 8
Size of Virtual    = 12
Press any key to continue
```

We see that the size of `Virtual` is *four bytes more* than that of `NonVirtual`, even though they have the same data. Where did these extra four bytes come from? They came from the vptr that is added to objects whose class contains virtual functions.

9.8 Abstract Classes

Despite solving the problem posed in Section 9.5 with polymorphism (i.e., virtual functions), we noted that it did not make sense to instantiate a `Shape` object. More specifically, we did not know how to implement the `type`, `area`, and `perimeter` functions of a `Shape` because a `Shape` is an abstract concept and we need concrete details to implement these functions. Consequently, we implemented `Shape`'s methods with "dummy" implementations like so:

```
string Shape::type()
{
	return "undefined";
}

float Shape::area()
{
	return 0.0f;
}

float Shape::perimeter()
{
	return 0.0f;
}
```

Another oddity is the fact that, at present, we can instantiate `Shape` objects:

```
Shape shape; // no error.
```

But such objects are completely meaningless. All the methods of `Shape` return worthless values. It only makes sense to use `Shapes` when using polymorphism where we can upcast from concrete `Shapes` (e.g., `Circles`, `Rectangles`), because then the methods of the dynamic type will be invoked.

All of this is quite inelegant. There are two things we would like to do in this situation.

1. Provide the general `Shape` method prototypes in the `Shape` class (`type`, `area`, `perimeter`), but not an implementation since it does not make sense for abstract `Shapes`. So we want to force derived classes to override and implement these prototyped methods where it does make sense (at the concrete level).

2. Prevent a `Shape` class from being instantiated altogether; that is, to make the restriction that we can only instantiate concrete types and upcast to `Shape`.

C++ accomplishes these two things with **abstract classes.** An abstract class is a class that contains **pure virtual functions.** A pure virtual function is a virtual function that is declared in the base class (the abstract class) but is not implemented in the base class. It is the responsibility of derived classes to override the pure virtual functions and provide an implementation for them.

> **Note:** Derived classes *must* override and implement *every* pure virtual function an abstract class declares. Also realize that not every method in an abstract class must be "pure virtual." However, it only takes one to make the class abstract.

A virtual function can be modified to "pure virtual" by appending the function signature with the syntax '= 0'. Let us now rewrite `Shape` as an abstract class:

```
class Shape
{
public:
	// Pure virtual functions.
	virtual string type()      = 0;
	virtual float  area()      = 0;
	virtual float  perimeter() = 0;
};
```

Note that we also delete the old "dummy" implementations—they are not needed with an abstract class since derived classes are guaranteed to override these pure virtual functions and provide an implementation.

Now that we have made `Shape` abstract we observe that the following produces an error:

```
Shape shape; // error, Shape abstract.
```

In particular, the error: "C2259: 'Shape' : cannot instantiate abstract class."

9.9 Interfaces

We mentioned that polymorphism is useful because it enables us to have a centralized general container of objects without having to realize their specific concrete type. In particular, by having such a container we can work with and apply operations to the objects in the most general form, without regard to the specific form. This is convenient because we do not need to have separate branches of code for each specific case—the program "knows" to invoke the method of the dynamic type.

Interfaces provide yet another utility of polymorphism. Interfaces are closely related to abstract classes. In fact, an interface is usually defined as a class that consists of only pure virtual functions. To illustrate, let us suppose we have a class, called `GraphicsEngine`, which handles drawing 3D objects to the screen using the video card. In particular, we want to draw various 3D objects. One approach would be to have a method that draws each kind of shape the program supports:

```
class GraphicsEngine
{
public:
	void drawFighterShip();
	void drawMothership();
	void drawBomberShip();
	void drawBomberShip();
	void drawPlanet();
	void drawAsteroid();
	// ... etc
};
```

This approach is cumbersome and inelegant. Every time we introduce a new kind of 3D object into the program, we must define a new method. Moreover, if `GraphicsEngine` was part of a third party library, we would not be able to modify it. This would be very restrictive! An approach that is much more scalable and robust (from a software engineering perspective), is to use *interfaces*.

We will define an `Object3D` interface with a pure virtual `draw` method.

```
class Object3D
{
public:
      virtual void draw() = 0;
};
```

All of our concrete 3D object classes will inherit from this interface and implement the `draw` method, thereby specifying how to draw themselves. Then `GraphicsEngine` will have its own `draw` method implemented like so:

```
void GraphicsEngine::draw(Object3D* obj)
{
      // Prepare hardware for drawing.

      obj->draw(); // draw the object

      // Do post drawing work.
}
```

`GraphicsEngine::draw` takes a pointer to an `Object3D` and calls `Object3D::draw` at the appropriate time. `GraphicsEngine` does not know how any specific *concrete* `Object3D` draws itself—and it does not care. All it cares about is that the method `draw` exists and has been overridden in the concrete class, so that it can invoke the `draw` method, and the correct method corresponding to the object's dynamic type will be invoked. This condition is guaranteed since `draw` is a pure virtual function in `Object3D`—therefore, it *must* be overridden and implemented by derived classes.

Whenever we need to add a new 3D object to the program, we simply create a new class representing the 3D object, have it inherit from `Object3D`, implement the `draw` method, and then pass it off to `GraphicsEngine::draw`. Then, due to polymorphism, the method corresponding to the object's dynamic type will be invoked. Incidentally, when a class inherits from an interface and implements the pure virtual functions, we say the class **implements the interface.**

We now come to the analogy of an interface being viewed as a contract. Just as a contract guarantees some agreement, an interface guarantees that a method or set of methods exist and are implemented in any derived class.

9.10 Summary

1. Inheritance allows a derived class (also called a child class or subclass) to inherit the data and methods of a base class (also called a parent class or superclass). In this way, we only have to write the general "shared" code once, and we can pass it along to the specific classes via inheritance, thereby saving work and reusing code. Furthermore, from an object oriented programming standpoint, C++ inheritance enables us to model real world inheritance relationships in code, allowing us to more closely model real world objects with software objects.

2. When a class contains a variable of some type *T* as a member variable, we say the class "has a" *T*. When we compose a class out of other types, object oriented programmers use the term "composition" to denote this; that is, the class is composed of those other types. When a class *A* inherits publicly from a class *B*, object oriented programmers say that we are modeling an "is a" relationship; that is, *A is a B*, but not conversely. Essentially, this is what public inheritance means—*is a*.

3. If `Derived` *is a* `Base` then we can switch back and fourth between the `Derived` object and its `Base` part via pointer casting. We use the term "upcast" when casting up the inheritance chain; that is, from a derived object to its base part. Likewise, we use the term "downcast" when casting down the inheritance chain; that is, from the base part to the derived object. Note that downcasting is *not* always safe. Remember, a `Derived` object *is a* specific kind of `Base` object, but not conversely. In other words, a `Derived` object will always have a base part, but a `Base` object is not necessarily part of a `Derived` object

4. In `Base* base = new Derived();` we say that the variable `base` has the "static type" `Base` and the "dynamic type" `Derived`.

5. Polymorphism allows us to upcast concrete types to a more general type, such that the program still knows to invoke the methods that correspond to the object's dynamic types. Virtual functions are what make a type polymorphic.

6. An abstract class is a class that contains pure virtual functions. A pure virtual function is a virtual function that is declared in the base class (the abstract class) but is not implemented in the base class. It is the responsibility of derived classes to override the pure virtual functions and provide an implementation for them. Derived classes *must* override and implement *every* pure virtual function an abstract class declares. Also realize that not every method in an abstract class must be pure virtual. However, it only takes one to make the class abstract. A virtual function can be modified to "pure virtual" by appending the function signature with the syntax '= 0'.

7. Interfaces provide yet another utility of polymorphism. Interfaces are closely related to abstract classes. In fact, an interface is usually defined as a class that consists of only pure virtual functions. A class that inherits from an interface and implements the pure virtual functions is said to "implement the interface." Interfaces enforce contracts; that is, they say: "all classes that inherit from me must implement my interface." In this way, other classes can be guaranteed that

all subclasses of an interface implement the pure virtual functions of that interface (i.e., they implement the interface). This allows software to call the methods of other objects without even knowing how they are implemented. Consequently, this leads to more general, expandable, and elegant software.

9.11 Exercises

9.11 Employee Database

Background Information

Thus far we have been working with `std::vector` as a resizable array; that is, we call the `resize` method and then access elements using the bracket operator []. However, another, perhaps more convenient way, to work with `std::vector` is to view it as a container where we can add and remove items to and from the container. Instead of calling `resize` and assigning values directly to elements, we simply call an "add" method, which will add a specified item to the next "free" element in the `std::vector`. We can also remove items from the container with a "remove" method, which will erase an item at a specified index. Moreover, the `std::vector` will resize itself automatically as needed to grow and shrink as you add/remove items.

The "add" and "remove" methods of `std::vector` are called `push_back` and `erase`, respectively, and they are summarized as follows:

- `push_back(item)`: Add a copy of the item specified by the parameter to the next "free" element in the `std::vector`.

- `erase(iterator)`: Removes the element specified by the iterator from the `std::vector`. We discuss iterators when we discuss the STL in Chapter 13 in the next module. For now, just think of it as a special object that identifies an element in a `std::vector`. We can get iterators to the vector elements using an offset and the `begin` method. The begin method returns an iterator to the element at index [0]. We can then add an offset value to this iterator to get iterators to the other elements. E.g., `vec.begin() + 1` evaluates to an iterator to element [1], `vec.begin() + 2` evaluates to an iterator to element [2], and so on.

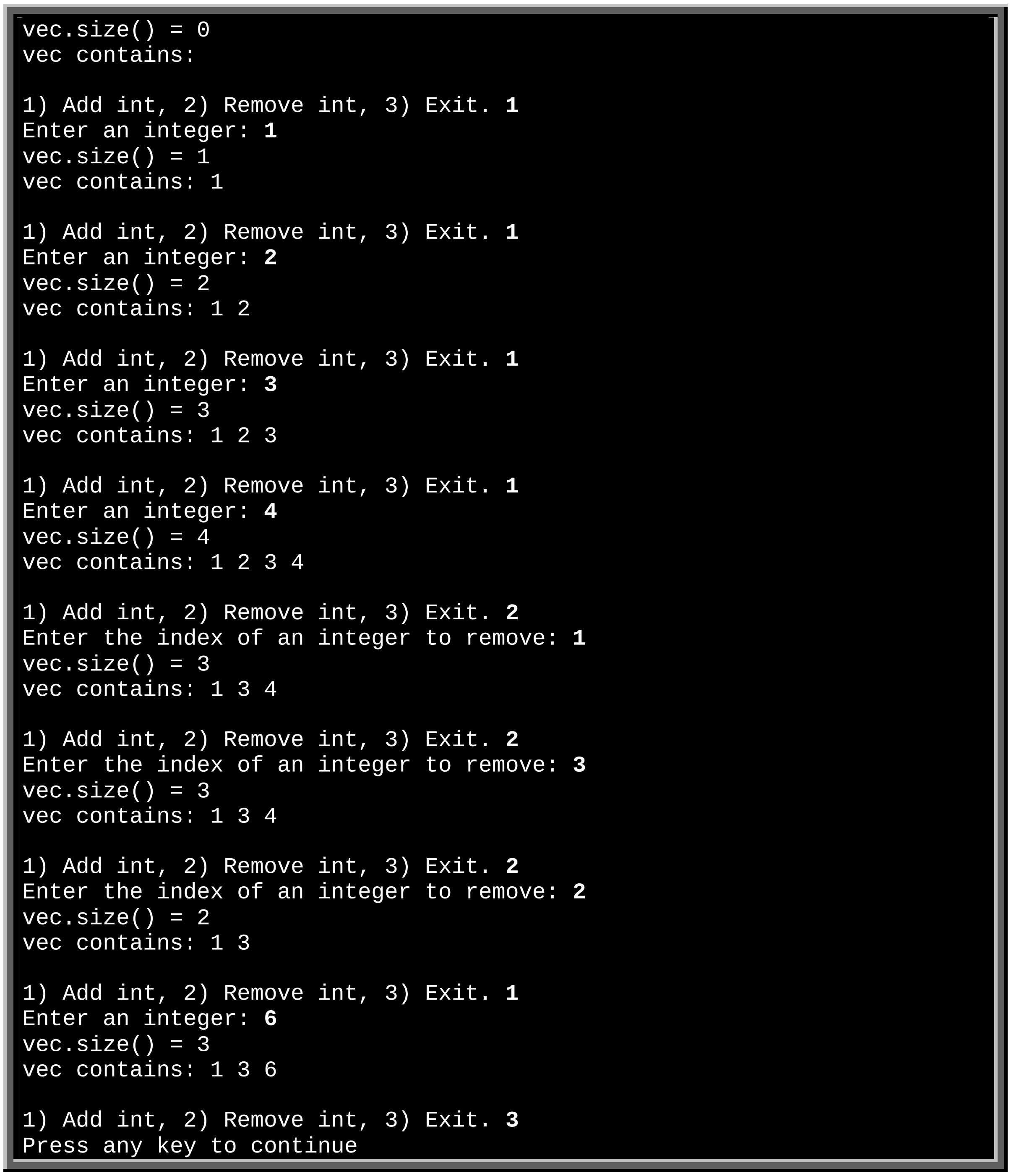
As you know, we can always get the current size of the `std::vector` with the size method.

The following program displays the contents of a vector and its size for each loop cycle. Furthermore, for each loop cycle, the user can add or remove a new item to the vector. In this way, you can see how items are added and removed to the vector container in real-time.

Program 9.5: Using the std::vector methods push_back and erase.

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

int main()
{
      vector<int> vec;

      bool quit = false;
      while( !quit )
      {
            // Output size.
            cout << "vec.size() = " << vec.size() << endl;

            // Output vector contents.
            cout << "vec contains: ";
            for(int i = 0; i < vec.size(); ++i)
                  cout << vec[i] << " ";
            cout << endl << endl;

            // Display option menu.
            cout << "1) Add int, 2) Remove int, 3) Exit. ";

            // Get menu input.
            int input = 1;
            cin >> input;

            // Do operation based on item chosen.
            switch( input )
            {
            case 1:
                  // Add an inputted integer to the vector.
                  cout << "Enter an integer: ";
                  cin >> input;
                  vec.push_back( input );
                  break;
            case 2:
                  // Remove the element at the inputted index.
                  cout << "Enter the index of an integer to remove: ";
                  cin >> input;

                  // Make sure index is inbounds.
                  if( input > 0 && input < vec.size() )
                        vec.erase( vec.begin() + input );

                  break;
            case 3:
                  // Exit.
                  quit = true;
                  break;
            }
      }
}
```

Program 9.5 Output



```
vec.size() = 0
vec contains:

1) Add int, 2) Remove int, 3) Exit. 1
Enter an integer: 1
vec.size() = 1
vec contains: 1

1) Add int, 2) Remove int, 3) Exit. 1
Enter an integer: 2
vec.size() = 2
vec contains: 1 2

1) Add int, 2) Remove int, 3) Exit. 1
Enter an integer: 3
vec.size() = 3
vec contains: 1 2 3

1) Add int, 2) Remove int, 3) Exit. 1
Enter an integer: 4
vec.size() = 4
vec contains: 1 2 3 4

1) Add int, 2) Remove int, 3) Exit. 2
Enter the index of an integer to remove: 1
vec.size() = 3
vec contains: 1 3 4

1) Add int, 2) Remove int, 3) Exit. 2
Enter the index of an integer to remove: 3
vec.size() = 3
vec contains: 1 3 4

1) Add int, 2) Remove int, 3) Exit. 2
Enter the index of an integer to remove: 2
vec.size() = 2
vec contains: 1 3

1) Add int, 2) Remove int, 3) Exit. 1
Enter an integer: 6
vec.size() = 3
vec contains: 1 3 6

1) Add int, 2) Remove int, 3) Exit. 3
Press any key to continue
```

Note that you can accomplish the same thing with `resize`, and doing some of your own "bookkeeping." However, `push_back` and `erase` provide a much simpler interface—it does the "bookkeeping" for you.

Exercise

Suppose a company has the following types of employees: 1) Manager, 2) Engineer, and 3) Researcher. The following box summarizes the data properties each kind of employee has.

All employees have the following:

1. First name.
2. Last name.
3. Salary.

In addition to what all employees have, managers have the following:

1. Number of meetings per week.
2. Number of vacation days per year.

In addition to what all employees have, engineers have the following:

1. A value specifying whether or not they know C++.
2. Number of years of experience.
3. A string denoting the type of engineer they are (e.g., “mechanical,” “electric,” “software.”

In addition to what all employees have, researchers have the following:

1. A string specifying the school they received their PhD from.
2. A string specifying the topic of their PhD thesis.

Your task is to write the following program. Create a console application that allows the user to add employees to and delete employees from a database. Use exactly one `std::vector` of `Employee`s as the database. Furthermore, the program should allow the user to save the database to a file (use a text file); in particular, implement a `save` method, which is responsible for writing the data of one employee to the file; so to save all the employees you iterate over the database and call the `save` method for each employee.

You should start this program by deciding on what the inheritance hierarchy should look like, what methods you need (i.e., what should the constructors look like, etc), which functions should be made virtual/pure virtual, which methods should be overridden in derived classes, and which data types to use to represent the properties of the various employees.

The menu displayed to the user should look something like this:

1) Add an Employee, 2) Delete an Employee 3) Save Database, 4) Exit.

For example, when the user presses the “1” key, then a new menu “Add an Employee” should be displayed. Similarly for “Delete an Employee” and “Save Database.”

1. **Add an Employee.** This should display a submenu: a) Add a Manager, b) Add an Engineer, c) Add a Researcher.

    a) *Add a Manager.* This should ask the user to input the necessary information to construct a `Manager`. Using this information, construct a new `Manager` and add it to the database (i.e., the `std::vector`).

    b) *Add an Engineer.* This should ask the user to input the necessary information to construct an `Engineer`. Using this information, construct a new `Engineer` and add it to the database (i.e., the `std::vector`).

    c) *Add a Researcher.* This should ask the user to input the necessary information to construct a `Researcher`. Using this information, construct a new `Researcher` and add it to the database (i.e., the `std::vector`).

2. **Delete an Employee.** This should ask for the last name of the employee (your program does not need to handle duplicate last names). You then need to write code to *search* the database for the given employee and delete him/her from the database (i.e., the `std::vector`). If the user enters in a name that does not exist in the database, report that information to the user and return to the main menu.

3. **Save Database.** This should traverse the entire database (i.e., the array/`std::vector`) and call the `save` method of each `Employee`, thereby saving the data of each employee to file.

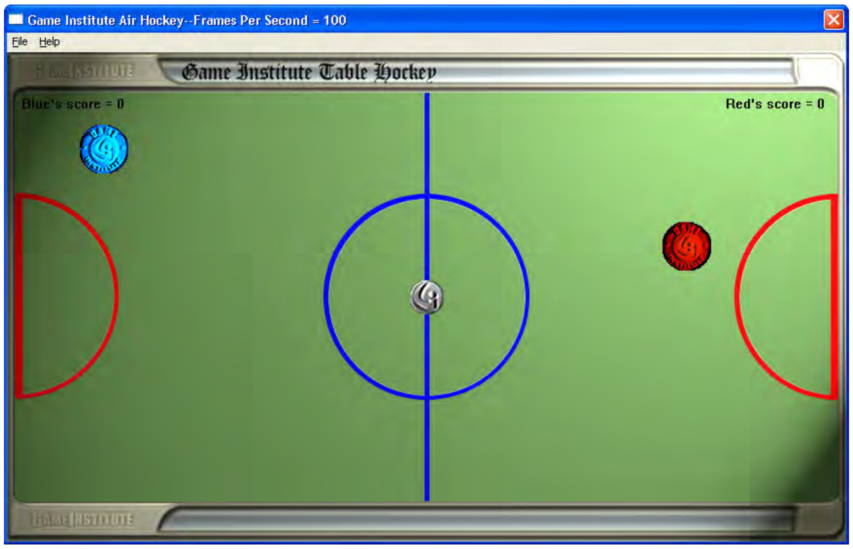
After you have implemented this program, run the program and manually enter in three distinct managers, three distinct engineers, and three distinct researchers. Then delete one manager, one engineer, and one researcher from the database. Third, save the database to a file. Finally, inspect the file to verify that the information saved was the information you entered, and that the three deleted employees were not saved. If the file contains the correct information, then you are done with this exercise. If you are feeling ambitious, add a new option that reads the database from file back into RAM.

C++ Module I Conclusion

Congratulations on completing a first course on the C++ programming language. It is hoped by now that you are proficient with core C++ topics such as variables, console input and output, functions, loops, programming logic, pointers, strings, and classes; in addition, you should have a basic understanding of more complex C++ subject matter, such as operator overloading, file input and output, inheritance, and polymorphism. Mastery in said advance C++ topics will come with time and experience, as your C++ programming matures.

In the second C++ course, offered at Game Institute, we will begin to move away from the text-based console applications we have been building, and begin to examine Windows programming. By making the move to Windows, we enter a whole new world of programming; a world in which we will be exposed to a set of functions and data structures, collectively called the *Win32 API*, which is used to develop Windows programs. With the Win32 API, we will be able to write programs the way you are, no doubt, familiar to seeing; ones with resizable windows, mouse input, graphics, menus, toolbars, scroll bars, dialog boxes, and controls.

Of particular interest to us as game programmers is the ability to do graphics with the Win32 API, something which is not possible with pure C++[8]. We will learn about fundamental graphic concepts such as double buffering, sprites, animation and timing, and masking. By the end of the course, we will have developed a fully functional 2D Air Hockey game (see Figure), complete with graphics, physics, artificial intelligence, and input via the mouse.


Game Institute Air Hockey--Frames Per Second = 100
File
Help
Game Institute Table Hockey
Blue's score = 0
Red's score = 0

However, before we make the move to Windows, we need to make three stops, and examine some last minute C++ techniques; in particular, template programming, exception handling, alternative number systems and bit operations, and a primer of the STL (standard template library).

[8] That is, C++ has nothing to say about graphic functionality—graphic routines must be exposed by the particular platform you are working on.

After completing the next C++ module, you will be adequately prepared for your first course in 3D graphics programming. One of the benefits from that point forward is that you will be writing all sorts of interesting 3D, AI, physics, and other game related applications in C++, which gives you the opportunity to continue to mature your C++ programming abilities. By the time you graduate from the full program, not only will you be a well-trained game developer, but you will also be a highly skilled C++ programmer. This will open up a lot of career opportunities that you might not have even considered.

Be sure to study hard for the final exam, and we hope to see you back here in short order so that you can begin Module II and start making games!