1

A Crash Course in C++

The goal of this chapter is to cover briefly the most important parts of C++ so that you have a base of knowledge before embarking on the rest of the book. This chapter is not a comprehensive lesson in the C++ programming language. The very basic points (like what a program is and the difference between = and ==) are not covered. The very esoteric points (remember what a `union` is? how about the `volatile` keyword?) are also omitted. Certain parts of the C language that are less relevant in C++ are also left out, as are parts of C++ that get in-depth coverage in later chapters.

This chapter aims to cover the parts of C++ that programmers encounter on a daily basis. If you've been away from C++ for a while and you've forgotten the syntax for a `for` loop, you'll find that in this chapter. If you're fairly new to C++ and you don't understand what a reference variable is, you'll learn that here as well.

If you already have significant experience with C++, skim this chapter to make sure that there aren't any fundamental parts of the language on which you need to brush up. If you're new to C++, take the time to read this chapter carefully and make sure that you understand the examples. If you need additional introductory information, consult the titles listed in Appendix B.

The Basics of C++

The C++ language is often viewed as a "better C" or a "superset of C." Many of the annoyances or rough edges of the C language were addressed when C++ was designed. Because C++ is based on C, much of the syntax you'll see in this section will look familiar to you if are an experienced C programmer. The two languages certainly have their differences, though. As evidence, *The C++ Programming Language* by C++ creator Bjarne Stroustrup weighs in at 911 pages, while Kernighan and Ritchie's *The C Programming Language* is a scant 274 pages. So if you're a C programmer, be on the lookout for new or unfamiliar syntax!

The Obligatory Hello, World

In all its glory, the following code is the simplest C++ program you're likely to encounter.

```
// helloworld.cpp

#include <iostream>

int main(int argc, char** argv)
{
    std::cout << "Hello, World!" << std::endl;

    return 0;
}
```

This code, as you might expect, prints the message Hello, World! on the screen. It is a simple program and unlikely to win any awards, but it does exhibit several important concepts about the format of a C++ program.

Comments

The first line of the program is a *comment*, a message that exists for the programmer only and is ignored by the compiler. In C++, there are two ways to delineate a comment. In the preceding example, two slashes indicate that whatever follows on that line is a comment.

```
// helloworld.cpp
```

The same behavior (this is to say, none) would be achieved by using a *C-style comment*, which is also valid in C++. C-style comments start with `/*` and end with `*/`. In this fashion, C-style comments are capable of spanning multiple lines. The code below shows a C-style comment in action (or, more appropriately, inaction).

```
/* this is a multiline
 * C-style comment. The
 * compiler will ignore
 * it.
 */
```

Comments are covered in detail in Chapter 7.

Preprocessor Directives

Building a C++ program is a three-step process. First, the code is run through a *preprocessor*, which recognizes metainformation about the code. Next, the code is *compiled*, or translated into machine-readable object files. Finally, the individual object files are *linked* together into a single application. Directives that are aimed at the preprocessor start with the `#` character, as in the line `#include <iostream>` in the previous example. In this case, an include directive tells the preprocessor to take everything from the iostream header file and make it available to the current file. The most common use of header files is to declare functions that will be defined elsewhere. Remember, a *declaration* tells the compiler how a function is called. A *definition* contains the actual code for the function. The iostream header declares the input and output mechanisms provided by C++. If the program did not include it, it would be unable to perform its only task of outputting text.

In C, included files usually end in .h, such as `<stdio.h>`. In C++, the suffix is omitted for standard library headers, such as `<iostream>`. Your favorite standard headers from C still exist in C++, but with new names. For example, you can access the functionality from `<stdio.h>` by including `<cstdio>`.

The table below shows some of the most common preprocessor directives.

Preprocessor Directive	Functionality	Common Uses
`#include [file]`	The specified file is inserted into the code at the location of the directive.	Almost always used to include header files so that code can make use of functionality that is defined elsewhere.
`#define [key] [value]`	Every occurrence of the specified key is replaced with the specified value.	Often used in C to define a constant value or a macro. C++ provides a better mechanism for constants. Macros are often dangerous so `#define` is rarely used in C++. See Chapter 12 for details.

`#ifdef [key]`
`#ifndef [key]`
`#endif`
	Code within the `ifdef` ("if defined") or `ifndef` ("if not defined") blocks are conditionally included or omitted based on whether the specified value has been defined with `#define`.	Used most frequently to protect against circular includes. Each included file defines a value initially and surrounds the rest of its code with a `#ifndef` and `#endif` so that it won't be included multiple times.

`#pragma`	Varies from compiler to compiler. Often allows the programmer to display a warning or error if the directive is reached during preprocessing.	Because usage of `#pragma` is not standard across compilers, we advocate not using it.

The main function

`main()` is, of course, where the program starts. An `int` is returned from `main()`, indicating the result status of the program. `main()` takes two parameters: `argc` gives the number of arguments passed to the program, and `argv` contains those arguments. Note that the first argument is always the name of the program itself.

I/O Streams

If you're new to C++ and coming from a C background, you're probably wondering what `std::cout` is and what has been done with trusty old `printf()`. While `printf()` can still be used in C++, a much better input/output facility is provided by the streams library.

I/O streams are covered in depth in Chapter 14, but the basics of output are very simple. Think of an output stream as a laundry chute for data. Anything you toss into it will be output appropriately. `std::cout` is the chute corresponding to the user console, or *standard out*. There are other chutes, including `std::cerr`, which outputs to the error console. The `<<` operator tosses data down the chute. In the preceding example, a quoted string of text is sent to standard out. Output streams allow multiple data of varying types to be sent down the stream sequentially on a single line of code. The following code outputs text, followed by a number, followed by more text.

```
std::cout << "There are " << 219 << " ways I love you." << std::endl;
```

`std::endl` represents an end of line character. When the output stream encounters `std::endl`, it will output everything that has been sent down the chute so far and move to the next line. An alternate way of representing the end of a line is by using the '`\n`' character. The `\n` character is an *escape character*, which refers to a new-line character. Escape characters can be used within any quoted string of text. The list below shows the most common escape characters.

- **`\n`** new line
- **`\r`** carriage return
- **`\t`** tab
- **`\\`** the backslash character
- **`\"`** quotation mark

Streams can also be used to accept input from the user. The simplest way to do this is to use the `>>` operator with an input stream. The `std::cin` input stream accepts keyboard input from the user. User input can be tricky because you can never know what kind of data the user will enter. See Chapter 14 for a full explanation of how to use input streams.

Namespaces

Namespaces address the problem of naming conflicts between different pieces of code. For example, you might be writing some code that has a function called `foo()`. One day, you decide to start using a third-party library, which also has a `foo()` function. The compiler has no way of knowing which version of `foo()` you are referring to within your code. You can't change the library's function name, and it would be a big pain to change your own.

Namespaces come to the rescue in such scenarios because you can define the context in which names are defined. To place code in a namespace, simply enclose it within a namespace block:

```
 // namespaces.h

namespace mycode {
    void foo();
}
```

The implementation of a method or function can also be handled in a namespace:

```
// namespaces.cpp

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

namespace mycode {

    void foo() {
        std::cout << "foo() called in the mycode namespace" << std::endl;
    }
}
```

By placing your version of `foo()` in the namespace "mycode," it is isolated from the `foo()` function provided by the third-party library. To call the namespace-enabled version of `foo()`, prepend the namespace onto the function name as follows.

```
mycode::foo();    // Calls the "foo" function in the "mycode" namespace
```

Any code that falls within a "mycode" namespace block can call other code within the same namespace without explicitly prepending the namespace. This implicit namespace is useful in making the code more precise and readable. You can also avoid prepending of namespaces with the `using` directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the code that follows:

```
// usingnamespaces.cpp

#include "namespaces.h"

using namespace mycode;

int main(int argc, char** argv)
{
    foo();  // Implies mycode::foo();
}
```

A single source file can contain multiple `using` directives, but beware of overusing this shortcut. In the extreme case, if you declare that you're using every namespace known to humanity, you're effectively eliminating namespaces entirely! Name conflicts will again result if you are using two namespaces that contain the same names. It is also important to know in which namespace your code is operating so that you don't end up accidentally calling the wrong version of a function.

You've seen the namespace syntax before — we used it in the Hello, World program. `cout` and `endl` are actually names defined in the `std` namespace. We could have rewritten Hello, World with the `using` directive as shown here:

```
// helloworld.cpp

#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    cout << "Hello, World!" << endl;

    return 0;
}
```

The `using` directive can also be used to refer to a particular item within a namespace. For example, if the only part of the `std` namespace that you intend to use is `cout`, you can refer to it as follows:

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

Subsequent code can refer to `cout` without prepending the namespace, but other items in the `std` namespace will still need to be explicit:

```
using std::cout;

cout << "Hello, World!" << std::endl;
```

Variables

In C++, *variables* can be declared just about anywhere in your code and can be used anywhere in the current block below the line where they are declared. In practice, your engineering group should decide whether variables will be declared at the start of each function or on an as-needed basis. Variables can be declared without being given a value. These undeclared variables generally end up with a semirandom value based on whatever is in memory at the time and are the source of countless bugs. Variables in C++ can alternatively be assigned an initial value when they are declared. The code that follows shows both flavors of variable declaration, both using `int`s, which represent integer values.

```
// hellovariables.cpp

#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
  int uninitializedInt;
  int initializedInt = 7;

  cout << uninitializedInt << " is a random value" << endl;
  cout << initializedInt << " was assigned an initial value" << endl;

  return (0);
}
```

When run, this code will output a random value from memory for the first line and the number 7 for the second. This code also shows how variables can be used with output streams.

The table that follows shows the most common variable types used in C++.

Type	Description	Usage
`int`	Positive and negative integers (range depends on compiler settings)	`int i = 7;`
`short`	Short integer (usually 2 bytes)	`short s = 13;`
`long`	Long integer (usually 4 bytes)	`long l = -7;`

`unsigned int`
`unsigned short`
`unsigned long`
	Limits the preceding types to values >= 0	`unsigned int i =2;`
		`unsigned short s = 23;`
		`unsigned long l = 5400;`

`float` `double`	Floating-point and double precision numbers	`float f = 7.2;` `double d = 7.2`
`char`	A single character	`char ch = 'm';`
`bool`	`true` or `false` (same as non-0 or 0)	`bool b = true;`

C++ does not provide a basic string type. However, a standard implementation of a string is provided as part of the standard library as described later in this chapter and in Chapter 13.

Variables can be converted to other types by *casting* them. For example, an `int` can be cast to a `bool`. C++ provides three ways of explicitly changing the type of a variable. The first method is a holdover from C, but is still the most commonly used. The second method seems more natural at first but is rarely seen. The third method is the most verbose, but often considered the cleanest.

```
bool someBool = (bool)someInt;       // method 1

bool someBool = bool(someInt);       // method 2

bool someBool = static_cast<bool>(someInt);    // method 3
```

The result will be `false` if the integer was 0 and `true` otherwise. In some contexts, variables can be automatically cast, or *coerced*. For example, a `short` can be automatically converted into a `long` because a `long` represents the same type of data with additional precision.

```
long someLong = someShort;          // no explicit cast needed
```

When automatically casting variables, you need to be aware of the potential loss of data. For example, casting a `float` to an `int` throws away information (the fractional part of the number). Many compilers

will issue a warning if you assign a `float` to an `int` without an explicit cast. If you are certain that the left-hand-side type is fully compatible with the right-hand side type, it's okay to cast implicitly.

Operators

What good is a variable if you don't have a way to change it? The table below shows the most common *operators* used in C++ and sample code that makes use of them. Note that operators in C++ can be *binary* (operate on two variables), *unary* (operate on a single variable), or even *ternary* (operate on three variables). There is only one ternary operator in C++ and it is covered in the next section, "Conditionals."

Operator	Description	Usage

`=`	Binary operator to assign the value on the right to the variable on the left.	`int ;`
		`i = 3;`
		`int j;`
		`j = i;`

`!`	Unary operator to negate the true/false (non-0/0) status of a variable.	`bool b = !true;` `bool b2 = !b;`

`+`	Binary operator for addition.	`int i = 3 + 2;`
		`int j = i + 5;`
		`int k = i + j;`

`-`
`*`
`/`
	Binary operators for subtraction, multiplication, and division.	`int i = 5-1;`
		`int j = 5*2;`
		`int k = j / i;`

`%`	Binary operator for remainder of a division operation. Also referred to as the *mod* operator.	`int remainder = 5 % 2;`
`++`	Unary operator to increment a variable by 1. If the operator occurs before the variable, the result of the expression is the unincremented value. If the operator occurs after the variable, the result of the expression is the new value.	`i++;` `++i;`
`--`	Unary operator to decrement a variable by 1.	`i--;` `--i;`
`+=`	Shorthand syntax for `i = i + j`	`i += j;`

`-=`
`*=`
`/=`
`%=`
	Shorthand syntax for
	`i = i - j;`
	`i = i * j;`
	`i = i / j;`
	`i = i % j;`
		`i -= j;`
		`i *= j;`
		`i /= j;`
		`i %= j;`

`&` `&=`	Takes the raw bits of one variable and performs a bitwise "and" with the other variable.	`i = j & k;` `j &= k;`
`|`	Takes the raw bits of one variable and performs a bitwise "or" with the other variable.	`i = j | k;` `j |= k;`

Operator	Description	Usage

`<<`
`>>`
`<<=`
`>>=`
	Takes the raw bits of a variable and "shifts" each bit left (<<) or right (>>) the specified number of places.	`i = i << 1;`
		`i = i >> 4;`
		`i <<= 1;`
		`i >>= 4;`

`^` `^=`	Performs a bitwise "exclusive or" operation on the two arguments.	`i = i ^ j;` `i ^= j;`

The following program shows the most common variable types and operators in action. If you're unsure about how variables and operators work, try to figure out what the output of this program will be, and then run it to confirm your answer.

```
// typetest.cpp

#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    int someInteger = 256;
    short someShort;
    long someLong;
    float someFloat;
    double someDouble;

    someInteger++;
    someInteger *= 2;
    someShort = (short)someInteger;
    someLong = someShort * 10000;
    someFloat = someLong + 0.785;
    someDouble = (double)someFloat / 100000;

    cout << someDouble << endl;
}
```

The C++ compiler has a recipe for the order in which expressions are evaluated. If you have a complicated line of code with many operators, the order of execution may not be obvious. For that reason, it's probably better to break up a complicated statement into several smaller statements or explicitly group expressions using parentheses. For example, the following line of code is confusing unless you happen to know the C++ operator precedence table by heart:

```
int i = 34 + 8 * 2 + 21 / 7 % 2;
```

Adding parentheses makes it clear which operations are happening first:

```
int i = 34 + (8 * 2) + ( (21 / 7) % 2 );
```

Breaking up the statement into separate lines makes it even clearer:

```
int i = 8 * 2;
int j = 21 / 7;
j %= 2;
i = 34 + i + j;
```

For those of you playing along at home, all three approaches are equivalent and end up with `i` equal to 51. If you assumed that C++ evaluated expressions from left to right, your answer would have been 1. In fact, C++ evaluates `/`, `*`, and `%` first (in left to right order), followed by addition and subtraction, then bitwise operators. Parenthesis let you explicitly tell the compiler that a certain operation should be evaluated separately.

Types

In C++, you can use the basic types (`int`, `bool`, etc.) to build more complex types of your own design. Once you are an experienced C++ programmer, you will rarely use the following techniques, which are features brought in from C, because classes are far more powerful. Still, it is important to know about the two most common ways of building types so that you will recognize the syntax.

Enumerated Types

An integer really represents a value within a sequence — the sequence of numbers. *Enumerated types* let you define your own sequences so that you can declare variables with values in that sequence. For example, in a chess program, you *could* represent each piece as an `int`, with constants for the piece types, as shown in the following code. The integers representing the types are marked `const` to indicate that they can never change.

```
const int kPieceTypeKing = 0;
const int kPieceTypeQueen = 1;
const int kPieceTypeRook = 2;
const int kPieceTypePawn = 3;
//etc.

int myPiece = kPieceTypeKing;
```

This representation is fine, but it can become dangerous. Since the piece is just an `int`, what would happen if another programmer added code to increment the value of the piece? By adding one, a king becomes a queen, which really makes no sense. Worse still, someone could come in and give a piece a value of -1, which has no corresponding constant.

Enumerated types resolve these problems by tightly defining the range of values for a variable. The following code declares a new type, `PieceT`, that has four possible values, representing four of the chess pieces.

```
typedef enum { kPieceTypeKing, kPieceTypeQueen, kPieceTypeRook,
               kPieceTypePawn
             } PieceT;
```

Behind the scenes, an enumerated type is just an integer value. The real value of `kPieceTypeKing` is zero. However, by defining the possible values for variables of type `PieceT`, your compiler can give you a warning or error if you attempt to perform arithmetic on `PieceT` variables or treat them as integers. The following code, which declares a `PieceT` variable then attempts to use it as an integer, results in a warning on most compilers.

```
PieceT myPiece;

myPiece = 0;
```

Structs

Structs let you encapsulate one or more existing types into a new type. The classic example of a struct is a database record. If you are building a personnel system to keep track of employee information, you will need to store the first initial, last initial, middle initial, employee number, and salary for each employee. A struct that contains all of this information is shown in the header file that follows.

```
// employeestruct.h

typedef struct {
    char    firstInitial;
    char    middleInitial;
    char    lastInitial;
    int     employeeNumber;
    int     salary;
} EmployeeT;
```

A variable declared with type `EmployeeT` will have all of these *fields* built-in. The individual fields of a struct can be accessed by using the "." character. The example that follows creates and then outputs the record for an employee.

```
// structtest.cpp

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

using namespace std;

int main(int argc, char** argv)
{
    // Create and populate an employee.
    EmployeeT anEmployee;

    anEmployee.firstInitial = 'M';
    anEmployee.middleInitial = 'R';
    anEmployee.lastInitial = 'G';
    anEmployee.employeeNumber = 42;
    anEmployee.salary = 80000;

    // Output the values of an employee.
```

```
    cout << "Employee: " << anEmployee.firstInitial <<
                           anEmployee.middleInitial <<
                           anEmployee.lastInitial << endl;
    cout << "Number: " << anEmployee.employeeNumber << endl;
    cout << "Salary: $" << anEmployee.salary << endl;

    return 0;
}
```

Conditionals

Conditionals let you execute code based on whether or not something is true. There are three main types of conditionals in C++.

If/Else Statements

The most common conditional is the `if` statement, which can be accompanied by `else`. If the condition given inside the `if` statement is true, the line or block of code is executed. If not, execution continues to the `else` case if present, or to the code following the conditional. The following pseudocode shows a *cascading if statement*, a fancy way of saying that the `if` statement has an `else` statement that in turn has another `if` statement, and so on.

```
if (i > 4) {
    // Do something.
} else if (i > 2) {
    // Do something else.
} else {
    // Do something else.
}
```

The expression between the parentheses of an `if` statement must be a Boolean value or evaluate to a Boolean value. Conditional operators, described below, provide ways of evaluating expressions to result in a `true` or `false` Boolean value.

Switch Statements

The `switch` statement is an alternate syntax for performing actions based on the value of a variable. In `switch` statements, the variable must be compared to a constant, so the greater-than `if` statements above could not be converted to switch statements. Each constant value represents a "case". If the variable matches the case, the subsequent lines of code are executed until the `break` statement is reached. You can also provide a `default` case, which is matched if none of the other cases match.

`switch` statements are generally used when you want to do something based on the specific value of a variable, as opposed to some test on the variable. The following pseudocode shows a common use of the `switch` statement.

```
switch (menuItem) {
    case kOpenMenuItem:
        // Code to open a file
        break;
```

```
        case kSaveMenuItem:
            // Code to save a file
            break;
        default:
            // Code to give an error message
            break;
    }
```

If you omit the `break` statement, the code for the subsequent case will be executed whether or not it matches. This is sometimes useful, but more frequently a source of bugs.

The Ternary Operator

C++ has one operator that takes three arguments, known as the *ternary operator*. It is used as a shorthand conditional expression of the form "if [*something*] then [*perform action*], otherwise [*perform some other action*]". The ternary operator is represented by a `?` and a `:`. The following code will output "yes" if the variable `i` is greater than 2, and "no" otherwise.

```
std::cout << ((i > 2) ? "yes" : "no");
```

The advantage of the ternary operator is that it can occur within almost any context. In the preceding example, the ternary operator is used within code that performs output. A convenient way to remember how the syntax is used is to treat the question mark as though the statement that comes before it really is a question. For example, "Is `i` greater than 2? If so, the result is 'yes': if not, the result is 'no.'"

Unlike an `if` statement or a `switch` statement, the ternary operator doesn't execute code based on the result. Instead, it is used *within* code, as shown in the preceding example. In this way, it really is an operator (like `+` and `-`) as opposed to a true conditional, such as `if` and `switch`.

Conditional Operators

You have already seen a *conditional operator* without a formal definition. The `>` operator compares two values. The result is "true" if the value on the left is greater than the value on the right. All conditional operators follow this pattern — they all result in a `true` or `false`.

The table below shows other common conditional operators.

Operator	Description	Usage

`<`
`<=`
`>`
`>=`
	Determines if the left-hand side is less than, less than or equal to, greater than,or greater than or equal to the right-hand side.	`if (i <= 0) {`
		`  std::cout << "i is negative";`
		`}`

`==`	Determines if the left-hand side equals the right-hand side. Don't confuse this with the `=` (assignment) operator!	`if (i == 3) {`
		`  std::cout << "i is 3";`
		`}`

Table continued on following page

Operator	Description	Usage

`!=`	Not equals. The result of the statement is true if the left-hand side does *not* equal the right-hand side.	`if (i != 3) {`
		`  std::cout << "i is not 3";`
		`}`

`!`	Logical not. Negates the true/false status of a Boolean expression. This is a unary operator.	`if (!someBoolean) {`
		`  std::cout << "someBoolean is false";`
		`}`

`&&`	Logical and. The result is true if both parts of the expression are true.	`if (someBoolean && someOtherBoolean) {`
		`  std::cout << "both are true";`
		`}`

`||`	Logical or. The result is true if either part of the expression is true.	`if (someBoolean || someOtherBoolean) {`
		`  std::cout << "at least one is true";`
		`}`

C++ uses *short-circuit logic* when evaluating an expression. That means that once the final result is certain, the rest of the expression won't be evaluated. For example, if you are doing a logical or of several Boolean expressions as shown below, the result is known to be true as soon as one of them is found to be true. The rest won't even be checked.

```
bool result = bool1 || bool2 || (i > 7) || (27 / 13 % i + 1) < 2;
```

In the example above, if `bool1` is found to be `true`, the entire expression must be true, so the other parts aren't evaluated. In this way, the language saves your code from doing unnecessary work. It can, however, be a source of hard-to-find bugs if the later expressions in some way influence the state of the program (for example, by calling a separate function). The following code shows a statement using `&&` that will short-circuit after the second term because 1 always evaluates to `true`.

```
bool result = bool1 && 1 && (i > 7) && !done;
```

Loops

Computers are great for doing the same thing over and over. C++ provides three types of looping structures.

The While Loop

`while` loops let you perform a block of code repeatedly as long as an expression evaluates to `true`. For example, the following completely silly code will output "This is silly." five times.

```
int i = 0;
while (i < 5) {
    std::cout << "This is silly." << std::endl;
    i++;
}
```

The keyword `break` can be used within a loop to immediately get out of the loop and continue execution of the program. The keyword `continue` can be used to return to the top of the loop and reevaluate the `while` expression. Both are often considered poor style because they cause the execution of a program to jump around somewhat haphazardly.

The Do/While Loop

C++ also has a variation on the `while` loop called `do/while`. It works similarly to the `while` loop, except that the code to be executed comes first, and the conditional check for whether or not to continue happens at the end. In this way, you can use a loop when you want a block of code to always be executed at least once and possibly additional times based on some condition. The example that follows will output "This is silly." once even though the condition will end up being false.

```
int i = 100;
do {
    std::cout << "This is silly." << std::endl;
    i++;
} while (i < 5);
```

The For Loop

The *`for` loop* provides another syntax for looping. Any `for` loop can be converted to a `while` loop and vice versa. However, the `for` loop syntax is often more convenient because it looks at a loop in terms of a starting expression, an ending condition, and a statement to execute at the end of every iteration. In the following code, `i` is initialized to 0, the loop will continue as long as `i` is less than 5, and at the end of every iteration, `i` is incremented by 1. This code does the same thing as the `while` loop example, but to some programmers, it is easier to read because the starting value, ending condition, and per-iteration statement are all visible on one line.

```
for (int i = 0; i < 5; i++) {
    std::cout << "This is silly." << std::endl;
}
```

Arrays

Arrays hold a series of values, all of the same type, each of which can be accessed by its position in the array. In C++, you must provide the size of the array when the array is declared. You cannot give a variable as the size — it must be a constant value. The code that follows shows the declaration of an array of 10 integers followed by a `for` loop that initializes each integer to zero.

```
int myArray[10];

for (int i = 0; i < 10; i++) {
    myArray[i] = 0;
}
```

The preceding example shows a one-dimensional array, which you can think of as a line of integers, each with its own numbered compartment. C++ allows multidimensional arrays. You might think of a two-dimensional array as a checkerboard, where each location has a position along the x-axis and a position along the y-axis. Three-dimensional and higher arrays are harder to picture and are rarely used. The code below shows the syntax for allocating a two-dimensional array of characters for a Tic-Tac-Toe board and then putting an "o" in the center square.

```
char ticTacToeBoard[3][3];

ticTacToeBoard[1][1] = 'o';
```

Figure 1-1 shows a visual representation of this board with the position of each square.

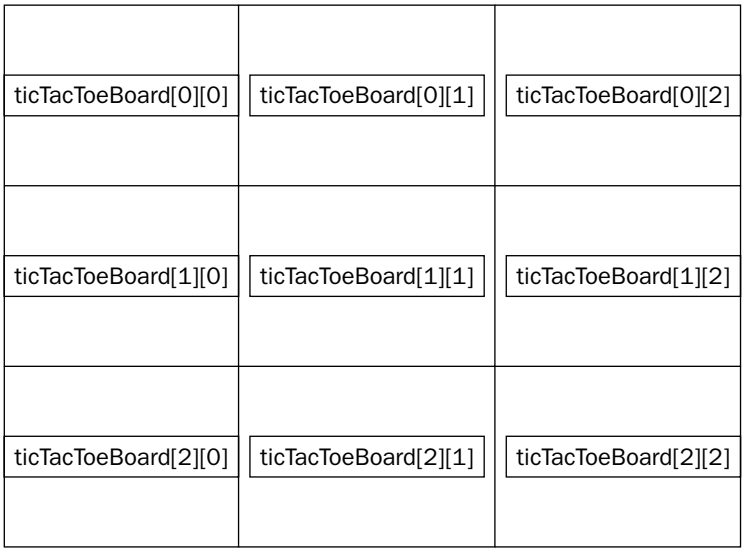

Figure 1-1

In C++, the first element of an array is always at position 0, not position 1! The last position of the array is always the size of the array minus 1!

Functions

For programs of any significant size, placing all the code inside of `main()` is unmanageable. To make programs easy to understand, you need to break up, or *decompose*, code into concise functions.

In C++, you first declare a function to make it available for other code to use. If the function is used inside a particular file of code, you generally declare and define the function in the source file. If the function is for use by other modules or files, you generally put the declaration in a header file and the definition in a source file.

Function declarations are often called "function prototypes" or "signatures" to emphasize that they represent how the function can be accessed, but not the code behind it.

A function declaration is shown below. This example has a return type of `void`, indicating that the function does not provide a result to the caller. The caller must provide two arguments for the function to work with — an integer and a character.

```
void myFunction(int i, char c);
```

Without an actual definition to match this function declaration, the link stage of the compilation process will fail because code that makes use of the function `myFunction()` will be calling nonexistent code. The following definition simply prints the values of the two parameters.

```
void myFunction(int i, char c)
{
   std::cout << "the value of i is " << i << std::endl;
   std::cout << "the value of c is " << c << std::endl;
}
```

Elsewhere in the program, you can make calls to `myFunction()` and pass in constants or variables for the two parameters. Some sample function calls are shown here:

```
myFunction(8, 'a');
myFunction(someInt, 'b');
myFunction(5, someChar);
```

In C++, unlike C, a function that takes no parameters just has an empty parameter list. It is not necessary to use "void" to indicate that no parameters are taken. However, you should still use "void" to indicate when no value is returned.

C++ functions can also *return* a value to the caller. The following function declaration and definition is for a function that adds two numbers and returns the result.

```
int addNumbers(int number1, int number2);

int addNumbers(int number1, int number2)
{
    int result = number1 + number2;
    return (result);
}
```

Those Are the Basics

At this point, you have reviewed the basic essentials of C++ programming. If this section was a breeze, skim the next section to make sure that you're up to speed on the more advanced material. If you

struggled with this section, you may want to obtain one of the fine introductory C++ books mentioned in Appendix D before continuing.

Diving Deeper into C++

Loops, variables, and conditionals are terrific building blocks, but there is much more to learn. The topics covered next include many features that are designed to help C++ programmers with their code as well as a few features that are often more confusing than helpful. If you are a C programmer with little C++ experience, you should read this section carefully.

Pointers and Dynamic Memory

Dynamic memory allows you to build programs with data that is not of fixed size at compile time. Most nontrivial programs make use of dynamic memory in some form.

The Stack and the Heap

Memory in your C++ application is divided into two parts — the *stack* and the *heap*. One way to visualize the stack is as a deck of cards. The current top card represents the current scope of the program, usually the function that is currently being executed. All variables declared inside the current function will take up memory in the top stack frame, the top card of the deck. If the current function, which we'll call `foo()` calls another function `bar()`, a new card is put on the deck so that `bar()` has its own *stack frame* to work with. Any parameters passed from `foo()` to `bar()` are copied from the `foo()` stack frame into the `bar()` stack frame. The mechanics of parameter passing and stack frames are covered in Chapter 13. Figure 1-2 shows what the stack might look like during the execution of a hypothetical function `foo()` that has declared two integer values.

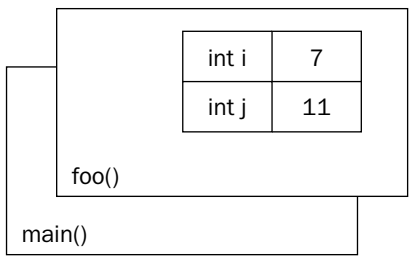

Figure 1-2

Stack frames are nice because they provide an isolated memory workspace for each function. If a variable is declared inside the `foo()` stack frame, calling the `bar()` function won't change it unless you specifically tell it to. Also, when the `foo()` function is done running, the stack frame goes away, and all of the variables declared within the function no longer take up memory.

The *heap* is an area of memory that is completely independent of the current function or stack frame. You can put variables on the heap if you want them to exist even when the function in which they were declared has completed. The heap is less structured than the stack. You can think of it as just a pile of bits. Your program can add new bits to the pile at any time or modify bits that are already in the pile.

Dynamically Allocated Arrays

Due to the way that the stack works, the compiler must be able to determine at compile time how big each stack frame will be. Since the stack frame size is predetermined, you cannot declare an array with a variable size. The following code will not compile because the `arraySize` is a variable, not a constant.

```
int arraySize = 8;
int myVariableSizedArray[arraySize];   // This won't compile!
```

Because the entire array must go on the stack, the compiler needs to know exactly what size it will be so variables aren't allowed. However, it is possible to specify the size of an array at run time by using *dynamic memory* and placing the array in the heap instead of the stack.

> **Some C++ compilers actually do support the preceding declaration, but is not currently a part of the C++ specification. Most compilers offer a "strict" mode that will turn off these nonstandard extensions to the language.**

To allocate an array dynamically, you first need to declare a *pointer*:

```
int* myVariableSizedArray;
```

The `*` after the `int` type indicates that the variable you are declaring refers to some integer memory in the heap. Think of the pointer as an arrow that points at the dynamically allocated heap memory. It does not yet point to anything specific because you haven't assigned it to anything; it is an uninitialized variable.

To initialize the pointer to new heap memory, you use the `new` command:

```
myVariableSizedArray = new int[arraySize];
```

This allocates memory for enough integers to satisfy the `arraySize` variable. Figure 1-3 shows what the stack and the heap both look like after this code is executed. As you can see, the pointer variable still resides on the stack, but the array that was dynamically created lives on the heap.

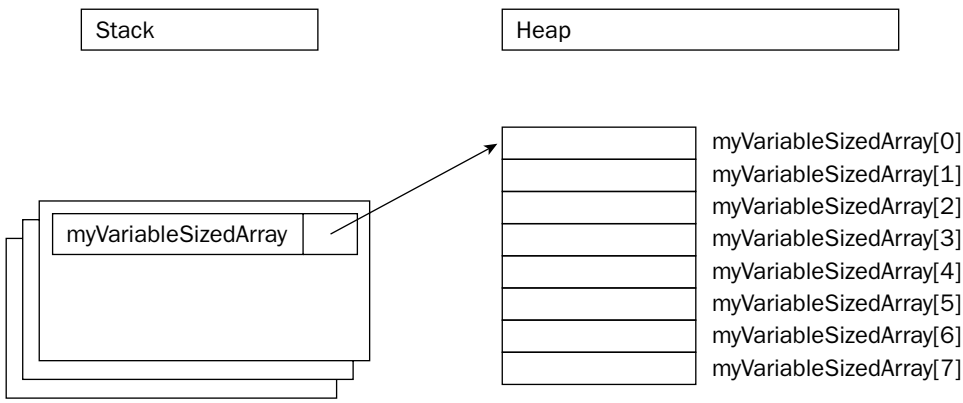

Figure 1-3

Now that the memory has been allocated, you can work with `myVariableSizedArray` as though it were a regular stack-based array:

```
myVariableSizedArray[3] = 2;
```

When your code is done with the array, it should remove it from the heap so that other variables can use the memory. In C++, you use the `delete` command to do this.

```
delete[] myVariableSizedArray;
```

The brackets after `delete` indicate that you are deleting an array.

> **The C++ commands `new` and `delete` are similar to `malloc()` and `free()` from C. The syntax of `new` and `delete` is simpler because you don't need to know how many bytes of memory are required.**

Working with Pointers

There are other reasons to use heap memory besides dynamically allocating arrays. You can put any variable in the heap by using a similar syntax:

```
int* myIntegerPointer = new int;
```

In this case, the pointer points to just a single integer value. To access this value, you need to *dereference* the pointer. Think of dereferencing as following the pointer's arrow to the actual value in the heap. To set the value of the newly allocated heap integer, you would use code like the following:

```
*myIntegerPointer = 8;
```

Notice that this is not the same as setting `myIntegerPointer` to the value 8. You are not changing the pointer, you are changing the memory that it points to. If you were to reassign the pointer value, it would point to the memory address 8, which is probably random garbage that will eventually make your program crash.

Pointers don't always point to heap memory. You can declare a pointer that points to a variable on the stack, even another pointer. To get a pointer to a variable, you use the `&` "address of" operator:

```
int i = 8;
int* myIntegerPointer = &i;  // Points to the variable with the value 8
```

C++ has a special syntax for dealing with pointers to structures. Technically, if you have a pointer to a structure, you can access its fields by first dereferencing it with `*`, then using the normal `.` syntax, as in the code that follows, which assumes the existence of a function called `getEmployee()`.

```
EmployeeT* anEmployee = getEmployee();

cout << (*anEmployee).salary << endl;
```

This syntax is a little messy. The -> (arrow) operator lets you perform both the dereference and the field access in one step. The following code is equivalent to the preceding code, but is easier to read.

```
EmployeeT* anEmployee = getEmployee();

cout << anEmployee->salary << endl;
```

Normally, when you pass a variable into a function, you are *passing by value.* If a function takes an integer parameter, it is really a copy of the integer that you pass in. Pointers to stack variables are often used in C to allow functions to modify variables in other stack frames, essentially *passing by reference*. By dereferencing the pointer, the function can change the memory that represents the variable even though that variable isn't in the current stack frame. This is less common in C++ because C++ has a better mechanism, called *references*, which is covered below.

Strings in C++

There are three ways to work with strings of text in C++. There is the C-style, which represents strings as arrays of characters; the C++ style, which wraps that representation in an easier-to-use string type; and the general class of nonstandard approaches.

C-Style Strings

A string of text like "Hello, World" is internally represented as an array of characters with the character `'\0'` representing the end of the string. As you've seen, arrays and pointers are sometimes related. You could use either one to represent a string, as shown here:

```
char arrayString[20] = "Hello, World";
char* pointerString = "Hello, World";
```

For the `arrayString`, the compiler allocates space for 20 characters on the stack. The first 13 characters in the array are filled in with `'H'`, `'e'`, etc., ending with the character `'\0'`. The characters in positions 13 to 19 contain whatever random values happen to be in memory. The `'\0'` character tells code that uses the string where the content of the string ends. Even though the array has a length of 20, functions that process or output the string should ignore everything after the `'\0'` character.

For the `pointerString`, the compiler allocates enough memory on the stack just to hold the pointer. The pointer points to an area of memory that the compiler has set aside to hold the constant string "Hello, World." In this string, there is also a `'\0'` character after the `'d'` character.

The C language provides a number of standard functions for working with strings, which are described in the <cstring> header file. The details of the standard library are not covered here because C++ provides a much cleaner and simpler way of working with strings.

C++ Strings

C-style strings are important to understand because they are still frequently used by C++ programmers. However, C++ includes a much more flexible string type. The `string` type, described by the <string> header file, acts just like a basic type. Just like I/O streams, the `string` type lives in the "`std`" package. The example that follows shows how strings can be used just like character arrays.

```
// stringtest.cpp

#include <string>
#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    string myString = "Hello, World";

    cout << "The value of myString is " << myString << endl;

    return 0;
}
```

The magic of C++ strings is that you can use standard operators to work with them. Instead of using a function, like `strcat()` in C to concatenate two strings, you can simply use `+`. If you've ever tried to use the `==` operator to compare two C-style strings, you've discovered that it doesn't work. `==` when used on C-style strings is actually comparing the address of the character arrays, not their contents. With C++ strings, `==` actually compares two strings. The example that follows shows some of the standard operators in use with C++ strings.

```
// stringtest2.cpp

#include <string>
#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    string str1 = "Hello";
    string str2 = "World";
    string str3 = str1 + " " + str2;

    cout << "str1 is " << str1 << endl;
    cout << "str2 is " << str2 << endl;
    cout << "str3 is " << str3 << endl;

    if (str3 == "Hello World") {
        cout << "str3 is what it should be." << endl;
    } else {
        cout << "Hmmm . . . str3 isn't what it should be." << endl;
    }

    return (0);
}
```

The preceding examples show just a few of the many features of C++ strings. Chapter 13 goes into further detail.

Nonstandard Strings

There are several reasons why many C++ programmers don't use C++-style strings. Some programmers simply aren't aware of the `string` type because it was not always part of the C++ specification. Others have discovered over the years that the C++ `string` doesn't provide the behavior they need and have developed their own string type. Perhaps the most common reason is that development frameworks and operating systems tend to have their own way of representing strings, such as the `CString` class in Microsoft's MFC. Often, this is for backward compatibility or legacy issues. When starting a project in C++, it is very important to decide ahead of time how your group will represent strings.

References

The pattern for most functions is that they take in zero or more parameters, do some calculations, and return a single result. Sometimes, however, that pattern is broken. You may be tempted to return two values or you may want the function to be able to change the value of one of the variables that were passed in.

In C, the primary way to accomplish such behavior is to pass in a pointer to the variable instead of the variable itself. The only problem with this approach is that it brings the messiness of pointer syntax into what is really a simple task. In C++, there is an explicit mechanism for "pass-by-reference." Attaching `&` to a type indicates that the variable is a reference. It is still used as though it was a normal variable, but behind the scenes, it is really a pointer to the original variable. Below are two implementations of an `addOne()` function. The first will have no effect on the variable that is passed in because it is passed by value. The second uses a reference and thus changes the original variable.

```
void addOne(int i)
{
    i++;  // Has no real effect because this is a copy of the original
}
```

```
void addOne(int& i)
{
    i++;  // Actually changes the original variable
}
```

The syntax for the call to the `addOne()` function with an integer reference is no different than if the function just took an integer.

```
int myInt = 7;
addOne(myInt);
```

Exceptions

C++ is a very flexible language, but not a particularly safe one. The compiler will let you write code that scribbles on random memory addresses or tries to divide by zero (computers don't deal well with infinity). One of the language features that attempts to add a degree of safety back to the language is *exceptions.*

An exception is an unexpected situation. For example, if you are writing a function that retrieves a Web page, several things could go wrong. The Internet host that contains the page might be down, the page might come back blank, or the connection could be lost. In many programming languages, you would handle this situation by returning a special value from the function, such as the `NULL` pointer. Exceptions provide a much better mechanism for dealing with problems.

Exceptions come with some new terminology. When a piece of code detects an exceptional situation, it *throws* an exception. Another piece of code *catches* the exception and takes appropriate action. The following example shows a function, `divideNumbers()`, that throws an exception if the caller passes in a denominator of zero.

```
#include <stdexcept>

double divideNumbers(double inNumerator, double inDenominator)
{
    if (inDenominator == 0) {
        throw std::exception();
    }

    return (inNumerator / inDenominator);
}
```

When the throw line is executed, the function will immediately end without returning a value. If the caller surrounds the function call with a `try-catch` block, as shown in the following code, it will receive the exception and be able to handle it.

```
#include <iostream>
#include <stdexcept>

int main(int argc, char** argv)
{
    try {
        std::cout << divideNumbers(2.5, 0.5) << std::endl;
        std::cout << divideNumbers(2.3, 0) << std::endl;
    } catch (std::exception exception) {
        std::cout << "An exception was caught!" << std::endl;
    }
}
```

The first call to `divideNumbers()` executes successfully, and the result is output to the user. The second call throws an exception. No value is returned, and the only output is the error message that is printed when the exception is caught. The output for the preceding block of code is:

```
5
An exception was caught!
```

Exceptions can get tricky in C++. To use exceptions properly, you need to understand what happens to the stack variables when an exception is thrown, and you have to be careful to properly catch and handle the necessary exceptions. The preceding example used the built-in `std::exception` exception type, but it is preferable to write your own exception types that are more specific to the error being thrown. Unlike the Java language, the C++ compiler doesn't force you to catch every exception that might occur.

If your code never catches any exceptions but an exception is thrown, it will be caught by the program itself, which will be terminated. These trickier aspects of exceptions are covered in much more detail in Chapter 15

The Many Uses of const

The keyword `const` can be used in several different ways in C++. All of its uses are related, but there are subtle differences. One of the authors has discovered that the subtleties of `const` make for excellent interview questions! In Chapter 12, you will learn all of the ways that `const` can be used. The following sections outline the most frequent uses.

Const Constants

If you assumed that the keyword `const` has something to do with constants, you have correctly uncovered one of its uses. In the C language, programmers often use the preprocessor `#define` mechanism to declare symbolic names for values that won't change during the execution of the program, such as the version number. In C++, programmers are encouraged to avoid `#define` in favor of using `const` to define constants. Defining a constant with `const` is just like defining a variable, except that the compiler guarantees that code cannot change the value.

```
const float kVersionNumber = "2.0";
const string kProductName = "Super Hyper Net Modulator";
```

Const to Protect Variables

In C++, you can cast a `non-const` variable to a `const` variable. Why would you want to do this? It offers some degree of protection from other code changing the variable. If you are calling a function that a coworker of yours is writing, and you want to ensure that the function doesn't change the value of a parameter you pass in, you can tell your coworker to have the function take a `const` parameter. If the function attempts to change the value of the parameter, it will not compile.

In the following code, a `char*` is automatically cast to a `const char*` in the call to `mysteryFunction()`. If the author of `mysteryFunction()` attempts to change the values within the character array, the code will not compile. There are actually ways around this restriction, but using them requires conscious effort. C++ only protects against accidentally changing `const` variables.

```
// consttest.cpp
void mysteryFunction(const char* myString);

int main(int argc, char** argv)
{
    char* myString = new char[2];
    myString[0] = 'a';
    myString[1] = '\0';

    mysteryFunction(myString);

    return (0);
}
```

Const References

You will often find code that uses `const` reference parameters. At first, that seems like a contradiction. Reference parameters allow you to change the value of a variable from within another context. `const` seems to prevent such changes.

The main value in `const` reference parameters is efficiency. When you pass a variable into a function, an entire copy is made. When you pass a reference, you are really just passing a pointer to the original so the computer doesn't need to make the copy. By passing a `const` reference, you get the best of both worlds — no copy is made but the original variable cannot be changed.

`const` references become more important when you are dealing with objects because they can be large and making copies of them can have unwanted side effects. Subtle issues like this are covered in Chapter 12.

C++ as an Object-Oriented Language

If you are a C programmer, you may have viewed the features covered so far in this chapter as convenient additions to the C language. As the name C++ implies, in many ways the language is just a "better C." There is one major point that this view overlooks. Unlike C, C++ is an object-oriented language.

Object-oriented programming (OOP) is a very different, arguably more natural, way to write code. If you are used to procedural languages such as C or Pascal, don't worry. Chapter 3 covers all the background information you need to know to shift your mindset to the object-oriented paradigm. If you already know the theory of OOP, the rest of this section will get you up to speed (or refresh your memory) on basic C++ object syntax.

Declaring a Class

A *class* defines the characteristics of an object. It is somewhat analogous to a struct except a class defines behaviors in addition to properties. In C++, classes are usually declared in a header file and fully defined in a corresponding source file.

A basic class definition for an airline ticket class is shown below. The class can calculate the price of the ticket based on the number of miles in the flight and whether or not the customer is a member of the "Elite Super Rewards Program." The definition begins by declaring the class name. Inside a set of curly braces, the *data members* (properties) of the class and its *methods* (behaviors) are declared. Each data member and method is associated with a particular access level: `public`, `protected`, or `private`. These labels can occur in any order and can be repeated.

```
// AirlineTicket.h

#include <string>

class AirlineTicket
{
    public:
        AirlineTicket();
```

```
        ~AirlineTicket();

        int     calculatePriceInDollars();

        std::string  getPassengerName();
        void         setPassengerName(std::string inName);
        int          getNumberOfMiles();
        void         setNumberOfMiles(int inMiles);
        bool         getHasEliteSuperRewardsStatus();
        void         setHasEliteSuperRewardsStatus(bool inStatus);

    private:
        std::string  mPassengerName;
        int          mNumberOfMiles;
        bool         fHasEliteSuperRewardsStatus;
};
```

The method that has the same name of the class with no return type is a *constructor*. It is automatically called when an object of the class is created. The method with a tilde (~) character followed by the class name is a *destructor*. It is automatically called when the object is destroyed.

The sample program that follows makes use of the class declared in the previous example. This example shows the creation of a stack-based `AirlineTicket` object as well as a heap-based object.

```
 // AirlineTicketTest.cpp

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

using namespace std;

int main(int argc, char** argv)
{
    AirlineTicket myTicket;  // Stack-based AirlineTicket

    myTicket.setPassengerName("Sherman T. Socketwrench");
    myTicket.setNumberOfMiles(700);
    int cost = myTicket.calculatePriceInDollars();
    cout << "This ticket will cost $" << cost << endl;

    AirlineTicket* myTicket2; // Heap-based AirlineTicket

    myTicket2 = new AirlineTicket(); // Allocate a new object
    myTicket2->setPassengerName("Laudimore M. Hallidue");
    myTicket2->setNumberOfMiles(2000);
    myTicket2->setHasEliteSuperRewardsStatus(true);
    int cost2 = myTicket2->calculatePriceInDollars();
    cout << "This other ticket will cost $" << cost2 << endl;
    delete myTicket2;

    return 0;
}
```

The definitions of the AirlineTicket class methods are shown below.

```
 // AirlineTicket.cpp

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

using namespace std;

AirlineTicket::AirlineTicket()
{
    // Initialize data members
    fHasEliteSuperRewardsStatus = false;
    mPassengerName = "Unknown Passenger";
    mNumberOfMiles = 0;
}

AirlineTicket::~AirlineTicket()
{
    // Nothing much to do in terms of cleanup
}

int AirlineTicket::calculatePriceInDollars()
{
    if (getHasEliteSuperRewardsStatus()) {
        // Elite Super Rewards customers fly for free!
        return 0;
    }

    // The cost of the ticket is the number of miles times
    // 0.1. Real airlines probably have a more complicated formula!
    return static_cast<int>((getNumberOfMiles() * 0.1));
}

string AirlineTicket::getPassengerName()
{
    return mPassengerName;
}

void AirlineTicket::setPassengerName(string inName)
{
    mPassengerName = inName;
}

int AirlineTicket::getNumberOfMiles()
{
    return mNumberOfMiles;
}

void AirlineTicket::setNumberOfMiles(int inMiles)
{
    mNumberOfMiles = inMiles;
}
```

```
bool AirlineTicket::getHasEliteSuperRewardsStatus()
{
    return (fHasEliteSuperRewardsStatus);
}

void AirlineTicket::setHasEliteSuperRewardsStatus(bool inStatus)
{
    fHasEliteSuperRewardsStatus = inStatus;
}
```

The preceding example exposes you to the general syntax for creating and using classes. Of course, there is much more to learn. Chapters 8 and 9 go into more depth about the specific C++ mechanisms for defining classes.

Your First Useful C++ Program

The following program builds on the employee database example used earlier when discussing structs. This time, you will end up with a fully functional C++ program that uses many of the features discussed in this chapter. This real-world example includes the use of classes, exceptions, streams, arrays, namespaces, references, and other language features.

An Employee Records System

A program to manage a company's employee records needs to be flexible and have useful features. The feature set for this program includes the following.

- The ability to add an employee
- The ability to fire an employee
- The ability to promote an employee
- The ability to view all employees, past and present
- The ability to view all current employees
- The ability to view all former employees

The design for this program divides the code into three parts. The `Employee` class encapsulates the information describing a single employee. The `Database` class manages all the employees of the company. A separate `UserInterface` file provides the interactivity of the program.

The Employee Class

The `Employee` class maintains all the information about an employee. Its methods provide a way to query and change that information. `Employee`s also know how to display themselves on the console. Methods also exist to adjust the employee's salary and employment status.

Employee.h

The `Employee.h` file declares the behavior of the `Employee` class. The sections of this file are described individually in the material that follows.

```
// Employee.h

#include <iostream>

namespace Records {
```

The first few lines of the file include a comment indicating the name of the file and the inclusion of the stream functionality.

This code also declares that the subsequent code, contained within the curly braces, will live in the Records namespace. Records is the namespace that is used throughout this program for application-specific code.

```
    const int kDefaultStartingSalary = 30000;
```

This constant, representing the default starting salary for new employees, lives in the Records namespace. Other code that lives in Records can access this constant simply as `kDefaultStartingSalary`. Elsewhere, it must be referenced as `Records::kDefaultStartingSalary`.

```
    class Employee
    {
        public:

            Employee();

            void     promote(int inRaiseAmount = 1000);
            void     demote(int inDemeritAmount = 1000);
            void     hire();     // Hires or rehires the employee
            void     fire();     // Dismisses the employee
            void     display();  // Outputs employee info to the console

            // Accessors and setters
            void          setFirstName(std::string inFirstName);
            std::string   getFirstName();
            void          setLastName(std::string inLastName);
            std::string   getLastName();
            void          setEmployeeNumber(int inEmployeeNumber);
            int           getEmployeeNumber();
            void          setSalary(int inNewSalary);
            int           getSalary();
            bool          getIsHired();
```

The `Employee` class is declared, along with its public methods. The `promote()` and `demote()` methods both have integer parameters that are specified with a default value. In this way, other code can omit the integer parameters and the default will automatically be used.

A number of accessors provide mechanisms to change the information about an employee or query the current information about an employee:

```
        private:
            std::string   mFirstName;
            std::string   mLastName;
            int           mEmployeeNumber;
            int           mSalary;
            bool          fHired;
    };
}
```

Finally, the data members are declared as `private` so that other parts of the code cannot modify them directly. The accessors provide the only public way of modifying or querying these values.

Employee.cpp

The implementations for the `Employee` class methods are shown here:

```
// Employee.cpp

#include <iostream>

#include "Employee.h"

using namespace std;

namespace Records {

    Employee::Employee()
    {
        mFirstName = "";
        mLastName = "";
        mEmployeeNumber = -1;
        mSalary = kDefaultStartingSalary;
        fHired = false;
    }
```

The `Employee` constructor sets the initial values for the `Employee`'s data members. By default, new employees have no name, an employee number of -1, the default starting salary, and a status of not hired.

```
    void Employee::promote(int inRaiseAmount)
    {
        setSalary(getSalary() + inRaiseAmount);
    }

    void Employee::demote(int inDemeritAmount)
    {
        setSalary(getSalary() - inDemeritAmount);
    }
```

The `promote()` and `demote()` methods simply call the `setSalary()` method with a new value. Note that the default values for the integer parameters do not appear in the source file. They only need to exist in the header.

```
void Employee::hire()
{
    fHired = true;
}

void Employee::fire()
{
    fHired = false;
}
```

The `hire()` and `fire()` methods just set the `fHired` data member appropriately.

```
void Employee::display()
{
    cout << "Employee: " << getLastName() << ", " << getFirstName() << endl;
    cout << "-------------------------" << endl;
    cout << (fHired ? "Current Employee" : "Former Employee") << endl;
    cout << "Employee Number: " << getEmployeeNumber() << endl;
    cout << "Salary: $" << getSalary() << endl;
    cout << endl;
}
```

The `display()` method uses the console output stream to display information about the current employee. Because this code is part of the Employee class, it *could* access data members, such as `mSalary`, directly instead of using the `getSalary()` accessor. However, it is considered good style to make use of accessors when they exist, even within the class.

```
// Accessors and setters

void Employee::setFirstName(string inFirstName)
{
    mFirstName = inFirstName;
}

string Employee::getFirstName()
{
    return mFirstName;
}

void Employee::setLastName(string inLastName)
{
    mLastName = inLastName;
}

string Employee::getLastName()
{
    return mLastName;
}
```

```
    void Employee::setEmployeeNumber(int inEmployeeNumber)
    {
        mEmployeeNumber = inEmployeeNumber;
    }

    int Employee::getEmployeeNumber()
    {
        return mEmployeeNumber;
    }

    void Employee::setSalary(int inSalary)
    {
        mSalary = inSalary;
    }

    int Employee::getSalary()
    {
        return mSalary;
    }

    bool Employee::getIsHired()
    {
        return fHired;
    }

}
```

A number of accessors and setters perform the simple task of getting and setting values. Even though these methods seem trivial, it's better to have trivial accessors and setters than to make your data members `public`. In the future, you may want to perform bounds checking in the `setSalary()` method, for example.

EmployeeTest.cpp

As you write individual classes, it is often useful to test them in isolation. The following code includes a `main()` function that performs some simple operations using the `Employee` class. Once you are confident that the `Employee` class works, you should remove or comment-out this file so that you don't attempt to compile your code with multiple `main()` functions.

```
// EmployeeTest.cpp

#include <iostream>

#include "Employee.h"

using namespace std;
using namespace Records;

int main (int argc, char** argv)
{
    cout << "Testing the Employee class." << endl;
```

```
    Employee emp;

    emp.setFirstName("Marni");
    emp.setLastName("Kleper");
    emp.setEmployeeNumber(71);
    emp.setSalary(50000);
    emp.promote();
    emp.promote(50);
    emp.hire();
    emp.display();

    return 0;
}
```

The Database Class

The `Database` class uses an array to store `Employee` objects. An integer called `mNextSlot` is used as a marker to keep track of the next unused array slot. This method for storing objects is probably not ideal because the array is of a fixed size. In Chapters 4 and 21, you will learn about data structures in the C++ standard library that you can use instead

Database.h

```
// Database.h

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

namespace Records {

    const int kMaxEmployees = 100;
    const int kFirstEmployeeNumber = 1000;
```

Two constants are associated with the database. The maximum number of employees is a constant because the records are kept in a fixed-size array. Because the database will also take care of automatically assigning an employee number to a new employee, a constant defines where the numbering begins.

```
    class Database
    {
        public:
            Database();
            ~Database();

            Employee& addEmployee(std::string inFirstName, std::string inLastName);
            Employee& getEmployee(int inEmployeeNumber);
            Employee& getEmployee(std::string inFirstName, std::string inLastName);
```

The database provides an easy way to add a new employee by providing a first and last name. For convenience, this method will return a reference to the new employee. External code can also get an

employee reference by calling the `getEmployee()` method. Two versions of this method are declared. One allows retrieval by employee number. The other requires a first and last name.

```
            void        displayAll();
            void        displayCurrent();
            void        displayFormer();
```

Because the database is the central repository for all employee records, it has methods that will output all employees, the employees who are currently hired, and the employees who are no longer hired.

```
        protected:
            Employee    mEmployees[kMaxEmployees];
            int         mNextSlot;
            int         mNextEmployeeNumber;
    };
}
```

The `mEmployees` array is a fixed-size array that contains the `Employee` objects. When the database is created, this array will be filled with nameless employees, all with an employee number of -1. When the `addEmployee()` method is called, one of these blank employees will be populated with real data. The `mNextSlot` data member keeps track of which blank employee is next in line to be populated. The `mNextEmployeeNumber` data member keeps track of what employee number will be assigned to the new employee.

Database.cpp

```
// Database.cpp

#include <iostream>
#include <stdexcept>

#include "Database.h"

using namespace std;

namespace Records {

    Database::Database()
    {
        mNextSlot = 0;
        mNextEmployeeNumber = kFirstEmployeeNumber;
    }

    Database::~Database()
    {
    }
```

The Database constructor takes care of initializing the next slot and next employee number members to their starting values. `mNextSlot` is initialized to zero so that when the first employee is added, it will go into slot 0 of the `mEmployees` array.

```
Employee& Database::addEmployee(string inFirstName, string inLastName)
{
    if (mNextSlot >= kMaxEmployees) {
        cerr << "There is no more room to add the new employee!" << endl;
        throw exception();
    }

    Employee& theEmployee = mEmployees[mNextSlot++];
    theEmployee.setFirstName(inFirstName);
    theEmployee.setLastName(inLastName);
    theEmployee.setEmployeeNumber(mNextEmployeeNumber++);
    theEmployee.hire();

    return theEmployee;
}
```

The `addEmployee()` method fills in the next "blank" employee with actual information. An initial check makes sure that the `mEmployees` array is not full and throws an exception if it is. Note that after their use, the `mNextSlot` and `mNextEmployeeNumber` data members are incremented so that the next employee will get a new slot and number.

```
Employee& Database::getEmployee(int inEmployeeNumber)
{
    for (int i = 0; i < mNextSlot; i++) {
        if (mEmployees[i].getEmployeeNumber() == inEmployeeNumber) {
            return mEmployees[i];
        }
    }

    cerr << "No employee with employee number " << inEmployeeNumber << endl;
    throw exception();
}

Employee& Database::getEmployee(string inFirstName, string inLastName)
{
    for (int i = 0; i < mNextSlot; i++) {
        if (mEmployees[i].getFirstName() == inFirstName &&
            mEmployees[i].getLastName() == inLastName) {
            return mEmployees[i];
        }
    }

    cerr << "No match with name " << inFirstName << " " << inLastName << endl;
    throw exception();
}
```

Both versions of `getEmployee()` work in similar ways. The methods loop over all nonblank employees in the `mEmployees` array and check to see if each `Employee` is a match for the information passed to the method. If no match is found, an error is output and an exception is thrown.

```
    void Database::displayAll()
    {
        for (int i = 0; i < mNextSlot; i++) {
            mEmployees[i].display();
        }
    }

    void Database::displayCurrent()
    {
        for (int i = 0; i < mNextSlot; i++) {
            if (mEmployees[i].getIsHired()) {
                mEmployees[i].display();
            }
        }
    }

    void Database::displayFormer()
    {
        for (int i = 0; i < mNextSlot; i++) {
            if (!mEmployees[i].getIsHired()) {
                mEmployees[i].display();
            }
        }
    }
}
```

The display methods all use a similar algorithm. They loop through all nonblank employees and tell each employee to display itself to the console if the criterion for display matches.

DatabaseTest.cpp

A simple test for the basic functionality of the database follows:

```
    // DatabaseTest.cpp

#include <iostream>

#include "Database.h"

using namespace std;
using namespace Records;

int main(int argc, char** argv)
{
    Database myDB;

    Employee& emp1 = myDB.addEmployee("Greg", "Wallis");
    emp1.fire();

    Employee& emp2 = myDB.addEmployee("Scott", "Kleper");
    emp2.setSalary(100000);
```

```
    Employee& emp3 = myDB.addEmployee("Nick", "Solter");
    emp3.setSalary(10000);
    emp3.promote();

    cout << "all employees: " << endl;
    cout << endl;
    myDB.displayAll();

    cout << endl;
    cout << "current employees: " << endl;
    cout << endl;
    myDB.displayCurrent();

    cout << endl;
    cout << "former employees: " << endl;
    cout << endl;
    myDB.displayFormer();
}
```

The User Interface

The final part of the program is a menu-based user interface that makes it easy for users to work with the employee database.

UserInterface.cpp

```
// UserInterface.cpp

#include <i0stream>
#include <stdexcept>

#include "Database.h"

using namespace std;
using namespace Records;

int displayMenu();
void doHire(Database& inDB);
void doFire(Database& inDB);
void doPromote(Database& inDB);
void doDemote(Database& inDB);


int main(int argc, char** argv)
{
  Database employeeDB;
  bool done = false;

  while (!done) {
    int selection = displayMenu();

    switch (selection) {
```

```
      case 1:
        doHire(employeeDB);
        break;
      case 2:
        doFire(employeeDB);
        break;
      case 3:
        doPromote(employeeDB);
        break;
      case 4:
        employeeDB.displayAll();
        break;
      case 5:
        employeeDB.displayCurrent();
        break;
      case 6:
        employeeDB.displayFormer();
        break;
      case 0:
        done = true;
        break;
      default:
        cerr << "Unknown command." << endl;
      }
    }

      return 0;
  }
```

The `main()` function is a loop that displays the menu, performs the selected action, then does it all again. For most actions, separate functions are defined. For simpler actions, like displaying employees, the actual code is put in the appropriate case.

```
  int displayMenu()
  {
      int selection;

      cout << endl;
      cout << "Employee Database" << endl;
      cout << "-----------------" << endl;
      cout << "1) Hire a new employee" << endl;
      cout << "2) Fire an employee" << endl;
      cout << "3) Promote an employee" << endl;
      cout << "4) List all employees" << endl;
      cout << "5) List all current employees" << endl;
      cout << "6) List all previous employees" << endl;
      cout << "0) Quit" << endl;
      cout << endl;
      cout << "---> ";

      cin >> selection;

      return selection;
  }
```

The `displayMenu()` function simply outputs the menu and gets input from the user. One important note is that this code assumes that the user will "play nice" and type a number when a number is requested. When you read about I/O in Chapter 14, you will learn how to protect against bad input

```
void doHire(Database& inDB)
{
    string firstName;
    string lastName;

    cout << "First name? ";
    cin >> firstName;
    cout << "Last name? ";
    cin >> lastName;

    try {
        inDB.addEmployee(firstName, lastName);
    } catch (std::exception ex) {
        cerr << "Unable to add new employee!" << endl;
    }
}
```

The `doHire()` function simply gets the new employee's name from the user and tells the database to add the employee. It handles errors somewhat gracefully by outputting a message and continuing.

```
void doFire(Database& inDB)
{
    int employeeNumber;

    cout << "Employee number? ";
    cin >> employeeNumber;

    try {
        Employee& emp = inDB.getEmployee(employeeNumber);
        emp.fire();
        cout << "Employee " << employeeNumber << " has been terminated." << endl;
    } catch (std::exception ex)
        cerr << "Unable to terminate employee!" << endl;
    }
}

void doPromote(Database& inDB)
{
    int employeeNumber;
    int raiseAmount;

    cout << "Employee number? ";
    cin >> employeeNumber;

    cout << "How much of a raise? ";
    cin >> raiseAmount;
```

```
        try {
            Employee& emp = inDB.getEmployee(employeeNumber);
            emp.promote(raiseAmount);
        } catch (std::exception ex) {
            cerr << "Unable to promote employee!" << endl;
        }
    }
```

`doFire()` and `doPromote()` both ask the database for an employee by their employee number and then use the public methods of the `Employee` object to make changes.

Evaluating the Program

The preceding program covers a number of topics from the very simple to the more obscure. There are a number of ways that you could extend this program. For example, the user interface does not expose all of the functionality of the `Database` or `Employee` classes. You could modify the UI to include those features. You could also change the `Database` class to remove fired employees from the `mEmployees` array, potentially saving space.

If there are parts of this program that don't make sense, consult the preceding sections to review those topics. If something is still unclear, the best way to learn is to play with the code and try things out. For example, if you're not sure how to use the ternary operator, write a short `main()` function that tries it out.

Summary

Now that you know the fundamentals of C++, you are ready to become a professional C++ programmer. The next five chapters will introduce you to several important design concepts. By covering design at a high-level without getting into too much actual code, you will gain an appreciation for good program design without getting bogged down in the syntax.

When you start getting deeper into the C++ language later in the book, refer back to this chapter to brush up on parts of the language that you may need to review. Going back to some of the sample code in this chapter may be all you need to see to bring a forgotten concept back to the forefront of your mind.