PART I
Introduction to Professional C++

1

A Crash Course in C++ and the Standard Library

WHAT'S IN THIS CHAPTER?

- A brief overview of the most important parts and syntax of the C++ language and the Standard Library
- The basics of smart pointers

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of the chapter's code download on this book's website at `www.wrox.com/go/proc++4e` on the Download Code tab.

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 this book. This chapter is not a comprehensive lesson in the C++ programming language or the Standard Library. Certain basic points, such as what a program is and what recursion is, are not covered. Esoteric points, such as the definition of a `union`, or 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 every day. For example, if you've been away from C++ for a while and you've forgotten the syntax of a `for` loop, you'll find that syntax in this chapter. Also, if you're fairly new to C++ and don't understand what a reference variable is, you'll learn about that kind of variable here, as well. You'll also learn the basics on how to use the functionality available in the Standard Library, such as `vector` containers, `string` objects, and smart pointers.

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++, read this chapter carefully and make sure 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." It was mainly designed to be an object-oriented C, commonly called as "C with classes." Later on, many of the annoyances and rough edges of the C language were addressed as well. Because C++ is based on C, much of the syntax you'll see in this section will look familiar to you if you are an experienced C programmer. The two languages certainly have their differences, though. As evidence, *The C++ Programming Language* by C++ creator Bjarne Stroustrup (Fourth Edition; Addison-Wesley Professional, 2013) weighs in at 1,368 pages, while Kernighan and Ritchie's *The C Programming Language* (Second Edition; Prentice Hall, 1988) 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()
{
    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 the following important concepts about the format of a C++ program:

- Comments
- Preprocessor directives
- The `main()` function
- I/O streams

These concepts are briefly explained in the following sections.

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 and following examples, 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 *multiline comment*. Multiline comments start with `/*` and end with `*/`. The following code shows a multiline comment in action (or, more appropriately, inaction).

```
/* This is a multiline comment.
   The compiler will ignore it.
 */
```

Comments are covered in detail in Chapter 3.

Preprocessor Directives

Building a C++ program is a three-step process. First, the code is run through a *preprocessor*, which recognizes meta-information 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 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. A function *declaration* tells the compiler how a function is called, declaring the number and types of parameters, and the function return type. A *definition* contains the actual code for the function. In C++, declarations usually go into *header files*, typically with extension .h, while definitions usually go into *source files*, typically with extension .cpp. A lot of other programming languages, such as C# and Java, do not separate declarations and definitions into separate files.

The `<iostream>` header declares the input and output mechanisms provided by C++. If the program did not include that header, it would be unable to perform its only task of outputting text.

> **NOTE** *In C, the names of the Standard Library header files usually end in .h, such as* `<stdio.h>`*, and namespaces are not used.*
>
> *In C++, the .h suffix is omitted for Standard Library headers, such as* `<iostream>`*, and everything is defined in the* `std` *namespace or a sub-namespace of* `std`*.*
>
> *The Standard Library headers from C still exist in C++ but in two versions:*
>
> - *The new and recommended versions without a .h suffix but with a c prefix. These versions put everything in the* `std` *namespace (for example,* `<cstdio>`*).*
> - *The old versions with the .h suffix. These versions do not use namespaces (for example,* `<stdio.h>`*).*

The following table 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 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 better mechanisms for constants and most types of macros. Macros can be dangerous, so use them cautiously. See Chapter 11 for details.

`#ifdef [key]` `#endif`
 	
`#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 key has been defined with `#define`.	Used most frequently to protect against circular includes. Each header file starts with an `#ifndef` checking the absence of a key, followed by a `#define` directive to define that key. The header file ends with an `#endif`. This prevents the file from being included multiple times; see the example after this table.

`#pragma [xyz]`	`xyz` is compiler dependent. It often allows the programmer to display a warning or error if the directive is reached during preprocessing.	See the example after this table.

One example of using preprocessor directives is to avoid multiple includes, as shown here:

```
#ifndef MYHEADER_H
#define MYHEADER_H
// ... the contents of this header file
#endif
```

If your compiler supports the `#pragma once` directive, and most modern compilers do, then this can be rewritten as follows:

```
#pragma once
// ... the contents of this header file
```

Chapter 11 discusses this in more details.

The main() Function

`main()` is, of course, where the program starts. The return type of `main()` is an `int`, indicating the result status of the program. You can omit any explicit return statements in `main()`, in which

case zero is returned automatically. The `main()` function either takes no parameters, or takes two parameters as follows:

```
int main(int argc, char* argv[])
```

`argc` gives the number of arguments passed to the program, and `argv` contains those arguments. Note that `argv[0]` can be the program name, but it might as well be an empty string, so do not rely on it; instead, use platform-specific functionality to retrieve the program name. The important thing to remember is that the actual parameters start at index 1.

I/O Streams

I/O streams are covered in depth in Chapter 13, but the basics of output and input 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 types of data 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 sequence. 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 sequence*, which refers to a new-line character. Escape sequences can be used within any quoted string of text. The following table shows the most common ones:

`\n`	new line
`\r`	carriage return
`\t`	tab
`\\`	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. Here is an example:

```
int value;
std::cin >> value;
```

User input can be tricky because you can never know what kind of data the user will enter. See Chapter 13 for a full explanation of how to use input streams.

If you're new to C++ and coming from a C background, you're probably wondering what has been done with the trusty old `printf()` and `scanf()` functions. While these functions can still be used in C++, I recommend using the streams library instead, mainly because the `printf()` and `scanf()` family of functions do not provide any type safety.

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, enclose it within a namespace block. For example, the following could be the contents of a file called `namespaces.h`:

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

The implementation of a method or function can also be handled in a namespace. The `foo()` function, for instance, could be implemented in `namespaces.cpp` as follows:

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

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

Or alternatively:

```
#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," you are isolating it 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 by using `::`, also called the *scope resolution operator*, 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 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:

```
#include "namespaces.h"

using namespace mycode;
```

```
int main()
{
    foo();  // Implies mycode::foo();
    return 0;
}
```

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—you used it in the Hello, World program, where `cout` and `endl` are actually names defined in the `std` namespace. You could have written Hello, World with the `using` directive as shown here:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

A `using` declaration can 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;
```

> **WARNING** *Never put a* `using` *directive or* `using` *declaration in a header file; otherwise, you force it on everyone who is including your header file.*

C++17

C++17 makes it easier to work with *nested namespaces*. A nested namespace is a namespace inside another one. Before C++17, you had to use nested namespaces as follows:

```
namespace MyLibraries {
    namespace Networking {
        namespace FTP {
            /* ... */
        }
    }
}
```

This can be simplified a lot with C++17:

```
namespace MyLibraries::Networking::FTP {
    /* ... */
}
```

A *namespace alias* can be used to give a new and possibly shorter name to another namespace. For example: `namespace MyFTP = MyLibraries::Networking::FTP;`

Literals

Literals are used to write numbers or strings in your code. C++ supports a number of standard literals. Numbers can be specified with the following literals (the examples in the list represent the same number, 123):

- Decimal literal, `123`
- Octal literal, `0173`
- Hexadecimal literal, `0x7B`
- Binary literal, `0b1111011`

Other examples of literals in C++ include

- A floating-point value (such as `3.14f`)
- A double floating-point value (such as `3.14`)
- A single character (such as `'a'`)
- A zero-terminated array of characters (such as `"character array"`)

It is also possible to define your own type of literals, which is an advanced feature explained in Chapter 11.

Digits separators can be used in numeric literals. A digits separator is a single quote character. For example,

- `23'456'789`
- `0.123'456f`

C++17

C++17 adds support for hexadecimal floating-point literals—for example, `0x3.ABCp-10`, `0Xb.cp12l`.

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. Variables can be declared without being given a value. These uninitialized variables generally end up with a semi-random value based on whatever is in memory at that time, and are therefore 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.

```
int uninitializedInt;
int initializedInt = 7;
cout << uninitializedInt << " is a random value" << endl;
cout << initializedInt << " was assigned an initial value" << endl;
```

> **NOTE** *Most compilers will issue a warning or an error when code is using uninitialized variables. Some compilers will generate code that will report an error at run time.*

The following table shows the most common types used in C++.

TYPE	DESCRIPTION	USAGE

`(signed) int`
`signed`
	Positive and negative integers; the range depends on the compiler (usually 4 bytes).	`int i = -7;`
		`signed int i = -6;`
		`signed i = -5;`

`(signed) short (int)`	Short integer (usually 2 bytes)	`short s = 13;`
		`short int s = 14;`
		`signed short s = 15;`
		`signed short int s = 16;`

`(signed) long (int)`	Long integer (usually 4 bytes)	`long l = -7L;`
`(signed) long long (int)`	Long long integer; the range depends on the compiler, but is at least the same as for `long` (usually 8 bytes).	`long long ll = 14LL;`

`unsigned (int)`
`unsigned short (int)`
`unsigned long (int)`
`unsigned long long (int)`
	Limits the preceding types to values >= 0	`unsigned int i = 2U;`
		`unsigned j = 5U;`
		`unsigned short s = 23U;`
		`unsigned long l = 5400UL;`
		`unsigned long long ll = 140ULL;`

`float`	Floating-point numbers	`float f = 7.2f;`
`double`	Double precision numbers; precision is at least the same as for `float`.	`double d = 7.2;`
`long double`	Long double precision numbers; precision is at least the same as for `double`.	`long double d = 16.98L;`
`char`	A single character	`char ch = 'm';`
`char16_t`	A single 16-bit character	`char16_t c16 = u'm';`
`char32_t`	A single 32-bit character	`char32_t c32 = U'm';`

continues

(continued)

TYPE	DESCRIPTION	USAGE
`wchar_t`	A single wide character; the size depends on the compiler.	`wchar_t w = L'm';`
`bool`	A Boolean type that can have one of two values: `true` or `false`	`bool b = true;`
C++17 `std::byte`[1]	A single byte. Before C++17, a `char` or `unsigned char` was used to represent a byte, but those types make it look like you are working with characters. `std::byte` on the other hand clearly states your intention, that is, a single byte of memory.	`std::byte b{42};`[2]

[1]Requires an include directive for the `<cstddef>` header file.

[2]Initialization of an `std::byte` requires direct list initialization with a single-element list. See the "Direct List Initialization versus Copy List Initialization" section later in this chapter for the definition of direct list initialization.

> **NOTE** *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 more detail in Chapter 2.*

Variables can be converted to other types by *casting* them. For example, a `float` can be cast to an `int`. C++ provides three ways to *explicitly* change the type of a variable. The first method is a holdover from C; it is not recommended but unfortunately still commonly used. The second method is rarely used. The third method is the most verbose, but is also the cleanest one, and is therefore recommended.

```
float myFloat = 3.14f;
int i1 = (int)myFloat;                  // method 1
int i2 = int(myFloat);                  // method 2
int i3 = static_cast<int>(myFloat);     // method 3
```

The resulting integer will be the value of the floating-point number with the fractional part truncated. Chapter 11 describes the different casting methods in more detail. 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 at least the same 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). Most compilers will issue a warning or even an error 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 following table 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 expressions), *unary* (operate on a single expression), or even *ternary* (operate on three expressions). There is only one ternary operator in C++, and it is explained in the "Conditional Statements" section later in this chapter.

OPERATOR	DESCRIPTION	USAGE

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

`!`	Unary operator to complement the true/false (non-0/0) status of an expression	`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 the remainder of a division operation. This is also referred to as the *mod* or *modulo* operator.	`int remainder = 5 % 2;`
`++`	Unary operator to increment an expression by 1. If the operator occurs after the expression, or *post-increment*, the result of the expression is the unincremented value. If the operator occurs before the expression, or *pre-increment*, the result of the expression is the new value.	`i++;` `++i;`
`--`	Unary operator to decrement an expression 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;`

continues

(continued)

OPERATOR	DESCRIPTION	USAGE
`&` `&=`	Takes the raw bits of one expression and performs a bitwise "AND" with the other expression	`i = j & k;` `j &= k;`
`|` `|=`	Takes the raw bits of one expression and performs a bitwise "OR" with the other expression	`i = j | k;` `j |= k;`

`<<`
`>>`
`<<=`
`>>=`
	Takes the raw bits of an expression 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," also called "XOR" operation, on two expressions	`i = i ^ j;` `i ^= j;`

The following program shows the most common variable types and operators in action. If you are 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.

```
int someInteger = 256;
short someShort;
long someLong;
float someFloat;
double someDouble;

someInteger++;
someInteger *= 2;
someShort = static_cast<short>(someInteger);
someLong = someShort * 10000;
someFloat = someLong + 0.785f;
someDouble = static_cast<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 expression into several smaller expressions, or explicitly group sub-expressions by 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 );
```

For those of you playing along at home, both 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. C++ evaluates `/`, `*`, and `%` first (in left-to-right order), followed by addition and subtraction, then bitwise operators. Parentheses 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`, and so on) 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 following 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 PieceTypeKing = 0;
const int PieceTypeQueen = 1;
const int PieceTypeRook = 2;
const int PieceTypePawn = 3;
//etc.
int myPiece = PieceTypeKing;
```

This representation is fine, but it can become dangerous. Since a piece is just an `int`, what would happen if another programmer added code to increment the value of a piece? By adding 1, 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 solve these problems by tightly defining the range of values for a variable. The following code declares a new type, `PieceType`, which has four possible values, representing four of the chess pieces:

```
enum PieceType { PieceTypeKing, PieceTypeQueen, PieceTypeRook, PieceTypePawn };
```

Behind the scenes, an enumerated type is just an integer value. The real value of `PieceTypeKing` is 0. However, by defining the possible values for variables of type `PieceType`, your compiler can give you a warning or an error if you attempt to perform arithmetic on `PieceType` variables or treat them as integers. The following code, which declares a `PieceType` variable, and then attempts to use it as an integer, results in a warning or an error on most compilers:

```
PieceType myPiece;
myPiece = 0;
```

It's also possible to specify the integer values for members of an enumeration. The syntax is as follows:

```
enum PieceType { PieceTypeKing = 1, PieceTypeQueen, PieceTypeRook = 10, PieceTypePawn };
```

In this example, `PieceTypeKing` has the integer value 1, `PieceTypeQueen` has the value 2 assigned by the compiler, `PieceTypeRook` has the value 10, and `PieceTypePawn` has the value 11 assigned automatically by the compiler.

If you do not assign a value to an enumeration member, the compiler automatically assigns it a value that is the previous enumeration member incremented by 1. If you do not assign a value to the first enumeration member yourself, the compiler assigns it the value 0.

Strongly Typed Enumerations

Enumerations as explained in the previous section are not strongly typed, meaning they are not *type safe*. They are always interpreted as integers, and thus you can compare enumeration values from completely different enumeration types.

The strongly-typed `enum class` enumerations solve this problem. For example, the following defines a type-safe version of the earlier-defined `PieceType` enumeration:

```
enum class PieceType
{
    King = 1,
    Queen,
    Rook = 10,
    Pawn
};
```

For an `enum class`, the enumeration value names are not automatically exported to the enclosing scope, which means that you always have to use the scope resolution operator:

```
PieceType piece = PieceType::King;
```

This also means that you can give shorter names to the enumeration values, for example, `King` instead of `PieceTypeKing`.

Additionally, the enumeration values are not automatically converted to integers, which means the following is illegal:

```
if (PieceType::Queen == 2) {...}
```

By default, the underlying type of an enumeration value is an integer, but this can be changed as follows:

```
enum class PieceType : unsigned long
{
    King = 1,
    Queen,
    Rook = 10,
    Pawn
};
```

> **NOTE** *It is recommended to use the strongly-typed* `enum class` *enumerations instead of the type-unsafe* `enum` *enumerations.*

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 might want to store the first initial, last initial, employee number, and salary for each employee. A struct that contains all of this information is shown in the `employeestruct.h` header file that follows:

```
struct Employee {
    char firstInitial;
    char lastInitial;
    int  employeeNumber;
    int  salary;
};
```

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

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

using namespace std;

int main()
{
    // Create and populate an employee.
    Employee anEmployee;
    anEmployee.firstInitial = 'M';
    anEmployee.lastInitial = 'G';
    anEmployee.employeeNumber = 42;
    anEmployee.salary = 80000;
    // Output the values of an employee.
    cout << "Employee: " << anEmployee.firstInitial <<
                            anEmployee.lastInitial << endl;
    cout << "Number: " << anEmployee.employeeNumber << endl;
    cout << "Salary: $" << anEmployee.salary << endl;
    return 0;
}
```

Conditional Statements

Conditional statements let you execute code based on whether or not something is true. As shown in the following sections, there are three main types of conditional statements in C++: if/else statements, switch statements, and conditional operators.

if/else Statements

The most common conditional statement is the `if` statement, which can be accompanied by an `else`. If the condition given inside the `if` statement is true, the line or block of code is executed. If not, execution continues with the `else` case if present, or with the code following the conditional statement. The following code 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. A value of 0 evaluates to `false`, while any non-zero value evaluates to `true`. For example: `if(0)` is equivalent to `if(false)`. Logical evaluation operators, described later, provide ways of evaluating expressions to result in a `true` or `false` Boolean value.

C++17

Initializers for if Statements

C++17 allows you to include an initializer inside an `if` statement using the following syntax:

```
if (<initializer> ; <conditional_expression>) { <body> }
```

Any variable introduced in the `<initializer>` is only available in the `<conditional_expression>` and in the `<body>`. Such variables are not available outside the `if` statement.

It is too early in this book to give a useful example of this feature, but here is what it looks like:

```
if (Employee employee = GetEmployee() ; employee.salary > 1000) { ... }
```

In this example, the initializer gets an employee and the condition checks whether the salary of the retrieved employee exceeds 1000. Only in that case is the body of the `if` statement executed.

More concrete examples will be given throughout this book.

switch Statements

The `switch` statement is an alternate syntax for performing actions based on the value of an expression. In C++, the expression of a `switch` statement must be of an integral type, a type convertible to an integral type, an enumerated type, or a strongly typed enumeration, and must be compared to constants. Each constant value represents a "case." If the expression matches the case, the subsequent lines of code are executed until a `break` statement is reached. You can also provide a `default` case, which is matched if none of the other cases match. The following pseudocode shows a common use of the `switch` statement:

```
switch (menuItem) {
    case OpenMenuItem:
        // Code to open a file
        break;
    case SaveMenuItem:
        // Code to save a file
        break;
    default:
        // Code to give an error message
        break;
}
```

A `switch` statement can always be converted into `if/else` statements. The previous `switch` statement can be converted as follows:

```
if (menuItem == OpenMenuItem) {
    // Code to open a file
```

```
} else if (menuItem == SaveMenuItem) {
    // Code to save a file
} else {
    // Code to give an error message
}
```

`switch` statements are generally used when you want to do something based on more than 1 specific value of an expression, as opposed to some test on the expression. In such a case, the `switch` statement avoids cascading `if-else` statements. If you only need to inspect 1 value, an `if` or `if-else` statement is fine.

Once a `case` expression matching the `switch` condition is found, all statements that follow it are executed until a `break` statement is reached. This execution continues even if another `case` expression is encountered, which is called *fallthrough*. The following example has a single set of statements that is executed for several different cases:

```
switch (backgroundColor) {
    case Color::DarkBlue:
    case Color::Black:
        // Code to execute for both a dark blue or black background color
        break;
    case Color::Red:
        // Code to execute for a red background color
        break;
}
```

C++17

Fallthrough can be a source of bugs, for example if you accidentally forget a break statement. Because of this, compilers might give a warning if a fallthrough is detected in a `switch` statement, unless the case is empty as in the above example. Starting with C++17, you can tell the compiler that a fallthrough is intentional using the `[[fallthrough]]` attribute as follows:

```
switch (backgroundColor) {
    case Color::DarkBlue:
        doSomethingForDarkBlue();
        [[fallthrough]];
    case Color::Black:
        // Code is executed for both a dark blue or black background color
        doSomethingForBlackOrDarkBlue();
        break;
    case Color::Red:
    case Color::Green:
        // Code to execute for a red or green background color
        break;
}
```

C++17

Initializers for switch Statements

Just as for `if` statements, C++17 adds support for initializers to `switch` statements. The syntax is as follows:

```
switch (<initializer> ; <expression>) { <body> }
```

Any variables introduced in the `<initializer>` are only available in the `<expression>` and in the `<body>`. They are not available outside the `switch` statement.

The Conditional Operator

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

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

The parentheses around `i > 2` are optional, so the following is equivalent:

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

The advantage of the conditional operator is that it can occur within almost any context. In the preceding example, the conditional 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 conditional operator doesn't execute code blocks 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 statement, such as `if` and `switch`.

Logical Evaluation Operators

You have already seen a *logical evaluation 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 logical evaluation operators follow this pattern—they all result in a `true` or `false`.

The following table shows common logical evaluation operators:

OP	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";`
		`}`

`!=`	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";`
		`}`

OP	DESCRIPTION	USAGE

`!`	Logical NOT.
	This complements 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 logical expressions. That means that once the final result is certain, the rest of the expression won't be evaluated. For example, if you are performing a logical OR operation of several Boolean expressions, as shown in the following code, 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 this example, 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 short-circuits after the second term because 0 always evaluates to `false`:

```
bool result = bool1 && 0 && (i > 7) && !done;
```

Short-circuiting can be beneficial for performance. You can put cheaper tests first so that more expensive tests are not even executed when the logic short-circuits. It is also useful in the context of pointers to avoid parts of the expression to be executed when a pointer is not valid. Pointers and short-circuiting with pointers are discussed later in this chapter.

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 only a particular file, 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.

> **NOTE** Function declarations *are often called* function prototypes *or* function headers *to emphasize that they represent how the function can be accessed, but not the code behind it. The term* function signature *is used to denote the combination of the function name and its parameter list, but without the return type.*

A function declaration is shown in the following code. 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 will be calling nonexistent code. The following definition 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 arguments for the two parameters. Some sample function calls are shown here:

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

> **NOTE** *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 must still use* `void` *to indicate when no value is returned.*

C++ functions can also *return* a value to the caller. The following function adds two numbers and returns the result:

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

This function can be called as follows:

```
int sum = addNumbers(5, 3);
```

Function Return Type Deduction

With C++14, you can ask the compiler to figure out the return type of a function automatically. To make use of this functionality, you need to specify `auto` as the return type:

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

The compiler deduces the return type based on the expressions used for the `return` statements. There can be multiple `return` statements in the function, but they should all resolve to the same type. Such a function can even include recursive calls (calls to itself), but the first `return` statement in the function must be a non-recursive call.

Current Function's Name

Every function has a local predefined variable `__func__` containing the name of the current function. One use of this variable would be for logging purposes:

```
int addNumbers(int number1, int number2)
{
    std::cout << "Entering function " << __func__ << std::endl;
    return number1 + number2;
}
```

C-Style 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, or a *constant expression (constexpr)*. Constant expressions are discussed in Chapter 11. The code that follows shows the declaration of an array of three integers followed by three lines to initialize the elements to 0:

```
int myArray[3];
myArray[0] = 0;
myArray[1] = 0;
myArray[2] = 0;
```

> **WARNING** *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!*

The next section discusses loops that you can use to initialize each element. However, instead of using loops, or using the previous initialization mechanism, you can also accomplish the *zero-initialization* with the following one-liner:

```
int myArray[3] = {0};
```

You can even drop the 0 as follows:

```
int myArray[3] = {};
```

An array can also be initialized with an initializer list, in which case the compiler can deduce the size of the array automatically. For example,

```
int myArray[] = {1, 2, 3, 4}; // The compiler creates an array of 4 elements.
```

If you do specify the size of the array, and the initializer list has less elements than the given size, the remaining elements are set to 0. For example, the following code only sets the first element in the array to the value 2, and sets all the other elements to 0:

```
int myArray[3] = {2};
```

To get the size of a stack-based C-style array, you can use the C++17 `std::size()` function (requires `<array>`). For example:

```
unsigned int arraySize = std::size(myArray);
```

If your compiler is not yet C++17 compliant, the old trick to get the size of a stack-based C-style array is to use the `sizeof` operator. The `sizeof` operator returns the size of its argument in bytes. To get the number of elements in a stack-based array, you divide the size in bytes of the array by the size in bytes of the first element. For example:

```
unsigned int arraySize = sizeof(myArray) / sizeof(myArray[0]);
```

The preceding examples show a one-dimensional array, which you can think of as a line of integers, each with its own numbered compartment. C++ allows multi-dimensional 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 following code 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.

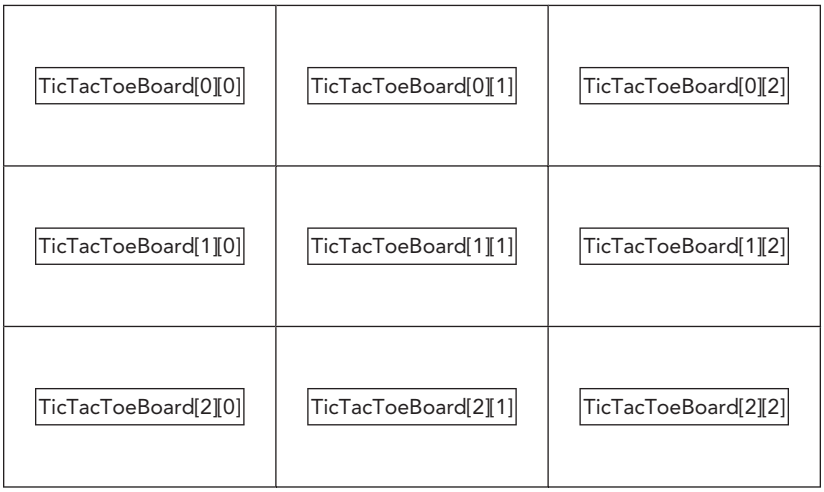

FIGURE 1-1

> **NOTE** *In C++, it's best to avoid C-style arrays as discussed in this section, and instead use Standard Library functionality, such as* `std::array`*, and* `std::vector`*, as discussed in the next two sections.*

std::array

The arrays discussed in the previous section come from C, and still work in C++. However, C++ has a special type of fixed-size container called `std::array`, defined in the `<array>` header file. It's basically a thin wrapper around C-style arrays.

There are a number of advantages to using `std::array`s instead of C-style arrays. They always know their own size, are not automatically cast to a pointer to avoid certain types of bugs, and have iterators to easily loop over the elements. Iterators are discussed in detail in Chapter 17.

The following example demonstrates how to use the `array` container. The use of angle brackets after `array`, as in `array<int, 3>`, will become clear during the discussion of templates in Chapter 12. However, for now, just remember that you have to specify two parameters between the angle brackets. The first parameter represents the type of the elements in the array, and the second one represents the size of the array.

```
array<int, 3> arr = {9, 8, 7};
cout << "Array size = " << arr.size() << endl;
cout << "2nd element = " << arr[1] << endl;
```

> **NOTE** *Both the C-style arrays and the* `std::array`*s have a fixed size, which must be known at compile time. They cannot grow or shrink at run time.*

If you want an array with a dynamic size, it is recommended to use `std::vector`, as explained in the next section. A `vector` automatically increases in size when you add new elements to it.

std::vector

The C++ Standard Library provides a number of different non-fixed-size containers that can be used to store information. `std::vector`, declared in `<vector>`, is an example of such a container. The `vector` replaces the concept of C-style arrays with a much more flexible and safer mechanism. As a user, you need not worry about memory management, as the `vector` automatically allocates enough memory to hold its elements. A `vector` is dynamic, meaning that elements can be added and removed at run time. Chapter 17 goes into more detail regarding containers, but the basic use of a `vector` is straightforward, which is why it's introduced in the beginning of this book so that it can be used in examples. The following code demonstrates the basic functionality of `vector`.

```
// Create a vector of integers
vector<int> myVector = { 11, 22 };

// Add some more integers to the vector using push_back()
myVector.push_back(33);
myVector.push_back(44);

// Access elements
cout << "1st element: " << myVector[0] << endl;
```

myVector is declared as vector<int>. The angle brackets are required to specify the template parameters, just as with std::array. A vector is a generic container. It can contain almost any kind of object; that's why you have to specify the type of object you want in your vector between the angle brackets. Templates are discussed in detail in Chapters 12 and 22.

To add elements to a vector, you can use the push_back() method. Individual elements can be accessed using a similar syntax as for arrays, i.e. operator[].

C++17

Structured Bindings

C++17 introduces the concept of *structured bindings*. Structured bindings allow you to declare multiple variables that are initialized with elements from an array, struct, pair, or tuple.

For example, assume you have the following array:

```
std::array<int, 3> values = { 11, 22, 33 };
```

You can declare three variables, x, y, and z, initialized with the three values from the array as follows. Note that you have to use the auto keyword for structured bindings. You cannot, for example, specify int instead of auto.

```
auto [x, y, z] = values;
```

The number of variables declared with the structured binding has to match the number of values in the expression on the right.

Structured bindings also work with structures if all non-static members are public. For example,

```
struct Point { double mX, mY, mZ; };
Point point;
point.mX = 1.0; point.mY = 2.0; point.mZ = 3.0;
auto [x, y, z] = point;
```

Examples with std::pair and std::tuple are given in chapters 17 and 20 respectively.

Loops

Computers are great for doing the same thing over and over. C++ provides four looping mechanisms: the while loop, do/while loop, for loop, and *range-based* for loop.

The while Loop

The while loop lets 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. However, using `continue` in loops is often considered poor style because it causes the execution of a program to jump around somewhat haphazardly, so use it sparingly.

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 outputs the statement, "This is silly." once, even though the condition ends 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 continues 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 is more readable 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;
}
```

The Range-Based for Loop

The *range-based* `for` loop is the fourth looping mechanism. It allows for easy iteration over elements of a container. This type of loop works for C-style arrays, initializer lists (discussed later in this chapter), and any type that has `begin()` and `end()` methods returning iterators (see Chapter 17), such as `std::array`, `std::vector`, and all other Standard Library containers discussed in Chapter 17.

The following example first defines an array of four integers. The range-based `for` loop then iterates over a *copy* of every element in this array and prints each value. To iterate over the elements themselves *without making copies*, use a reference variable, as I discuss later in this chapter.

```
std::array<int, 4> arr = {1, 2, 3, 4};
for (int i : arr) {
    std::cout << i << std::endl;
}
```

Initializer Lists

Initializer lists are defined in the `<initializer_list>` header file and make it easy to write functions that can accept a variable number of arguments. The `initializer_list` class is a template and so it requires you to specify the type of elements in the list between angle brackets, similar to how you have to specify the type of object stored in a `vector`. The following example shows how to use an initializer list:

```
#include <initializer_list>

using namespace std;

int makeSum(initializer_list<int> lst)
{
    int total = 0;
    for (int value : lst) {
        total += value;
    }
    return total;
}
```

The function `makeSum()` accepts an initializer list of integers as argument. The body of the function uses a range-based `for` loop to accumulate the total sum. This function can be used as follows:

```
int a = makeSum({1,2,3});
int b = makeSum({10,20,30,40,50,60});
```

Initializer lists are type safe and define which type is allowed to be in the list. For the `makeSum()` function shown here, all elements of the initializer list must be integers. Trying to call it with a `double` results in a compiler error or warning, as shown here:

```
int c = makeSum({1,2,3.0});
```

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 are 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 B 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 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.

Strings in C++

There are three ways to work with strings of text in C++: 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. Chapter 2 provides a detailed discussion.

For now, the only thing you need to know is that the C++ `string` type is defined in the `<string>` header file, and that you can use a C++ `string` almost like a basic type. Just like I/O streams, the `string` type lives in the `std` namespace. The following example shows that `string`s can be used just like character arrays:

```
string myString = "Hello, World";
cout << "The value of myString is " << myString << endl;
cout << "The second letter is " << myString[1] << endl;
```

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 I'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. Figure 1-2 shows what the stack might look like during the execution of a hypothetical function `foo()` that has declared two integer values.

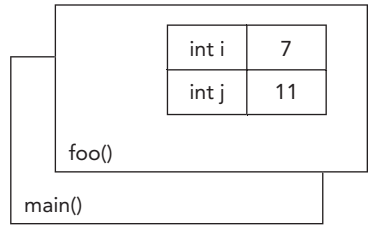

FIGURE 1-2

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. Variables that are stack-allocated do not need to be deallocated (deleted) by the programmer; it happens automatically.

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 created 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. You have to make sure that you deallocate (delete) any memory that you allocated on the heap. This does not happen automatically, unless you use smart pointers, which are discussed in the section "Smart Pointers."

Working with Pointers

You can put anything on the heap by explicitly allocating memory for it. For example, to put an integer on the heap, you need to allocate memory for it, but first you need to declare a *pointer*:

```
int* myIntegerPointer;
```

The `*` after the `int` type indicates that the variable you are declaring refers or points to some integer memory. 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*. Uninitialized variables should be avoided at all times, and especially uninitialized pointers because they point to some random place in memory. Working with such pointers will most likely make your program crash. That's why you should always declare and initialize your pointers at the same time. You can initialize them to a null pointer (`nullptr`—for more information, see the "Null Pointer Constant" section) if you don't want to allocate memory right away:

```
int* myIntegerPointer = nullptr;
```

A null pointer is a special default value that no valid pointer will ever have, and converts to `false` when used in a Boolean expression. For example:

```
if (!myIntegerPointer) { /* myIntegerPointer is a null pointer */ }
```

You use the `new` operator to allocate the memory:

```
myIntegerPointer = new int;
```

In this case, the pointer points to the address of 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 on 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.

After you are finished with your dynamically allocated memory, you need to deallocate the memory using the `delete` operator. To prevent the pointer from being used after having deallocated the memory it points to, it's recommended to set your pointer to `nullptr`:

```
delete myIntegerPointer;
myIntegerPointer = nullptr;
```

> **WARNING** *A pointer must be valid before it is dereferenced. Dereferencing a null pointer or an uninitialized pointer causes undefined behavior. Your program might crash, but it might just as well keep running and start giving strange results.*

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()`.

```
Employee* 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:

```
Employee* anEmployee = getEmployee();
cout << anEmployee->salary << endl;
```

Remember the concept of short-circuiting logic, which was discussed earlier in this chapter? This can be useful in combination with pointers to avoid using an invalid pointer, as in the following example:

```
bool isValidSalary = (anEmployee && anEmployee->salary > 0);
```

Or, a little bit more verbose:

```
bool isValidSalary = (anEmployee != nullptr && anEmployee->salary > 0);
```

`anEmployee` is only dereferenced to get the salary if it is a valid pointer. If it is a null pointer, the logical operation short-circuits, and the `anEmployee` pointer is not dereferenced.

Dynamically Allocated Arrays

The heap can also be used to dynamically allocate arrays. You use the `new[]` operator to allocate memory for an array.

```
int arraySize = 8;
int* 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.

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

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

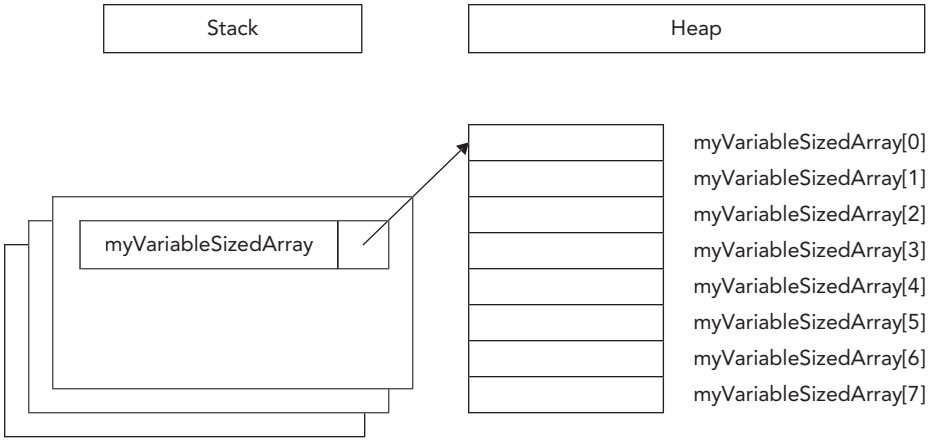

FIGURE 1-3

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

```
delete[] myVariableSizedArray;
myVariableSizedArray = nullptr;
```

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

> **NOTE** *Avoid using* `malloc()` *and* `free()` *from C. Instead, use* `new` *and* `delete`*, or* `new[]` *and* `delete[]`*.*

> **WARNING** *To prevent memory leaks, every call to* `new` *should be paired with a call to* `delete`*, and every call to* `new[]` *should be paired with a call to* `delete[]`*. Not calling* `delete` *or* `delete[]`*, or mismatching calls, results in memory leaks. Memory leaks are discussed in Chapter 7.*

Null Pointer Constant

Before C++11, the constant `NULL` was used for null pointers. `NULL` is simply defined as the constant 0, and this can cause problems. Take the following example:

```
void func(char* str) {cout << "char* version" << endl;}
void func(int i) {cout << "int version" << endl;}

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

The `main()` function is calling `func()` with parameter `NULL`, which is supposed to be a null pointer constant. In other words, you are expecting the `char*` version of `func()` to be called with a null pointer as argument. However, since `NULL` is not a pointer, but identical to the integer 0, the integer version of `func()` is called.

This problem is solved with the introduction of a real null pointer constant, `nullptr`. The following code calls the `char*` version:

```
func(nullptr);
```

Smart Pointers

To avoid common memory problems, you should use *smart pointers* instead of "raw," also called "naked," C-style pointers. Smart pointers automatically deallocate memory when the smart pointer object goes out of scope, for example, when the function has finished executing.

The following are the two most important smart pointer types in C++, both defined in `<memory>` and in the `std` namespace:

- `std::unique_ptr`
- `std::shared_ptr`

`unique_ptr` is analogous to an ordinary pointer, except that it automatically frees the memory or resource when the `unique_ptr` goes out of scope or is deleted. As such, `unique_ptr` has sole ownership of the object pointed to. One advantage of a `unique_ptr` is that memory and resources are always freed, even when return statements are executed, or when exceptions (discussed later in this chapter) are thrown. This, for example, simplifies coding when a function has multiple return statements, because you don't have to remember to free the resources before each return statement.

To create a `unique_ptr`, you should use `std::make_unique<>()`. For example, instead of writing the following,

```
Employee* anEmployee = new Employee;
// ...
delete anEmployee;
```

you should write this:

```
auto anEmployee = make_unique<Employee>();
```

Note that you do not call `delete` anymore; it happens automatically for you. The `auto` keyword is discussed in more detail in the "Type Inference" section later in this chapter. For now, it suffices to know that the `auto` keyword tells the compiler to automatically deduce the type of a variable, so that you don't have to manually specify the full type.

`unique_ptr` is a generic smart pointer that can point to any kind of memory. That's why it is a template. Templates require the angle brackets, `< >`, to specify the template parameters. Between the brackets, you have to specify the type of memory you want your `unique_ptr` to point to. Templates are discussed in detail in Chapters 12 and 22, but the smart pointers are introduced in Chapter 1 so that they can be used throughout the book—and as you will see, they are easy to use.

`make_unique()` has been available since C++14. If your compiler is not yet C++14 compliant, you can make your `unique_ptr` as follows (note that you now have to specify the type, `Employee`, twice):

```
unique_ptr<Employee> anEmployee(new Employee);
```

You can use the `anEmployee` smart pointer in the same way as a normal pointer, for example:

```
if (anEmployee) {
    cout << "Salary: " << anEmployee->salary << endl;
}
```

A `unique_ptr` can also be used to store a C-style array. The following example creates an array of ten `Employee` instances, stores it in a `unique_ptr`, and shows how to access an element from the array:

```
auto employees = make_unique<Employee[]>(10);
cout << "Salary: " << employees[0].salary << endl;
```

`shared_ptr` allows for distributed ownership of the data. Each time a `shared_ptr` is assigned, a reference count is incremented indicating there is one more owner of the data. When a `shared_ptr` goes out of scope, the reference count is decremented. When the reference count goes to zero, it means there is no longer any owner of the data, and the object referenced by the pointer is freed.

To create a `shared_ptr`, you should use `std::make_shared<>()`, which is similar to `make_unique<>()`:

```
auto anEmployee = make_shared<Employee>();
if (anEmployee) {
    cout << "Salary: " << anEmployee->salary << endl;
}
```

Starting with C++17, you can also store an array in a `shared_ptr`, whereas older versions of C++ did not allow this. Note however that `make_shared<>()` of C++17 cannot be used in this case. Here is an example:

```
shared_ptr<Employee[]> employees(new Employee[10]);
cout << "Salary: " << employees[0].salary << endl;
```

Chapter 7 discusses memory management and smart pointers in more details, but because the basic use of `unique_ptr` and `shared_ptr` is straightforward, they are already used in examples throughout this book.

> **NOTE** *Raw pointers are only allowed if there is no ownership involved. Otherwise, use* `unique_ptr` *by default, and* `shared_ptr` *if you need shared ownership. If you know about* `auto_ptr`*, forget it; it was deprecated in C++11/14, and has been removed from C++17.*

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. The subtleties of `const` make for excellent interview questions! Chapter 11 explains in detail all the ways that `const` can be used. This section outlines two common use-cases.

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 int versionNumberMajor = 2;
const int versionNumberMinor = 1;
const std::string productName = "Super Hyper Net Modulator";
```

const to Protect Parameters

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

```
void mysteryFunction(const std::string* someString)
{
    *someString = "Test";  // Will not compile.
}

int main()
{
    std::string myString = "The string";
    mysteryFunction(&myString);
    return 0;
}
```

References

A reference in C++ allows you to give another name to an existing variable. For example:

```
int x = 42;
int& xReference = x;
```

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. Both the variable `x` and the reference variable `xReference` point to exactly the same value. If you change the value through either one of them, the change is visible through the other one as well.

Pass By Reference

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, so you cannot modify the value of the original variable. Pointers to stack variables are often used in C to allow functions to modify variables in other stack frames. 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. The problem with this approach is that it brings the messiness of pointer syntax into what is really a simple task.

Instead of passing pointers to functions, C++ offers a better mechanism, called *pass by reference*, where parameters are references instead of pointers. Following are two implementations of an `addOne()` function. The first one has no effect on the variable that is passed in because it is passed by value and thus the function receives a copy of the value passed to it. The second one 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);
```

> **NOTE** *There is a subtle difference between the two* `addOne()` *implementations. The version using pass-by-value accepts literals without a problem; for example, "*`addOne(3);`*" is legal. However, doing the same with the pass-by-reference version of* `addOne()` *will result in a compiler error. This can be solved by using* `const` *references, discussed in the next section, or rvalue references, an advanced C++ feature explained in Chapter 9.*

If you have a function that needs to return a big structure or class (discussed later in this chapter) that is expensive to copy, you'll often see the function taking a non-`const` reference to such a structure or class which the function then modifies, instead of directly returning it. This was the recommended way a long time ago to prevent the performance penalty of creating a copy when you return the structure or class from the function. Since C++11, this is not necessary anymore. Thanks to

move semantics, directly returning structures or classes from functions is efficient without any copying. Move semantics is discussed in detail in Chapter 9.

Pass By const Reference

You will often find code that uses `const` reference parameters for functions. 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 value 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 a 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 11. The following example shows how to pass an `std::string` to a function as a `const` reference:

```
void printString(const std::string& myString)
{
    std::cout << myString << std::endl;
}

int main()
{
    std::string someString = "Hello World";
    printString(someString);
    printString("Hello World");  // Passing literals works
    return 0;
}
```

> **NOTE** *If you need to pass an object to a function, prefer to pass it by* `const` *reference instead of by value. This prevents unnecessary copying. Pass it by non-*`const` *reference if the function needs to modify the object.*

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 language feature 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. One way you could handle this situation is by returning a special value from the function, such as `nullptr` or an error code. 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. The use of `std::invalid_argument` requires `<stdexcept>`.

```
double divideNumbers(double numerator, double denominator)
{
    if (denominator == 0) {
        throw invalid_argument("Denominator cannot be 0.");
    }
    return numerator / denominator;
}
```

When the `throw` line is executed, the function immediately ends without returning a value. If the caller surrounds the function call with a `try/catch` block, as shown in the following code, it receives the exception and is able to handle it:

```
try {
    cout << divideNumbers(2.5, 0.5) << endl;
    cout << divideNumbers(2.3, 0) << endl;
    cout << divideNumbers(4.5, 2.5) << endl;
} catch (const invalid_argument& exception) {
    cout << "Exception caught: " << exception.what() << 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 third call is never executed because the second call throws an exception, causing the program to jump to the `catch` block. The output for the preceding block of code is as follows:

```
5
An exception was caught: Denominator cannot be 0.
```

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. Also, the preceding example uses the built-in `std::invalid_argument` type, but it is preferable to write your own exception types that are more specific to the error being thrown. Lastly, 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 14.

Type Inference

Type inference allows the compiler to automatically deduce the type of an expression. There are two keywords for type inference: `auto` and `decltype`.

The auto Keyword

The `auto` keyword has a number of completely different uses:

- Deducing a function's return type, as explained earlier in this chapter.
- Structured bindings, as explained earlier in this chapter.
- Deducing the type of an expression, as discussed later in this section.
- Deducing the type of non-type template parameters, see Chapter 12.
- `decltype(auto)`, see Chapter 12.
- Alternative function syntax, see Chapter 12.
- Generic lambda expressions, see Chapter 18.

`auto` can be used to tell the compiler to automatically deduce the type of a variable at compile time. The following line shows the simplest use of the `auto` keyword in that context:

```
auto x = 123;    // x will be of type int
```

In this example, you don't win much by typing `auto` instead of `int`; however, it becomes useful for more complicated types. Suppose you have a function called `getFoo()` that has a complicated return type. If you want to assign the result of calling `getFoo()` to a variable, you can spell out the complicated type, or you can simply use `auto` and let the compiler figure it out:

```
auto result = getFoo();
```

This has the added benefit that you can easily change the function's return type without having to update all the places in the code where that function is called.

However, using `auto` to deduce the type of an expression strips away reference and `const` qualifiers. Suppose you have the following function:

```
#include <string>

const std::string message = "Test";

const std::string& foo()
{
    return message;
}
```

You can call `foo()` and store the result in a variable with the type specified as `auto`, as follows:

```
auto f1 = foo();
```

Because `auto` strips away reference and `const` qualifiers, `f1` is of type `string`, and thus a *copy* is made. If you want a `const` reference, you can explicitly make it a reference and mark it `const`, as follows:

```
const auto& f2 = foo();
```

> **WARNING** *Always keep in mind that* `auto` *strips away reference and* `const` *qualifiers, and thus creates a copy! If you do not want a copy, use* `auto&` *or* `const auto&`.

The decltype Keyword

The `decltype` keyword takes an expression as argument, and computes the type of that expression, as shown here:

```
int x = 123;
decltype(x) y = 456;
```

In this example, the compiler deduces the type of `y` to be `int` because that is the type of `x`.

The difference between `auto` and `decltype` is that `decltype` does not strip reference and `const` qualifiers. Take again the function `foo()` returning a `const` reference to a `string`. Defining `f2` using `decltype` as follows results in `f2` being of type `const string&`, and thus no copy is made.

```
decltype(foo()) f2 = foo();
```

On first sight, `decltype` doesn't seem to add much value. However, it is pretty powerful in the context of templates, discussed in Chapters 12 and 22.

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 5 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.

Defining Classes

A *class* defines the characteristics of an object. In C++, classes are usually defined in a header file (.h), while their definitions usually are in a corresponding source file (.cpp).

A basic class definition for an airline ticket class is shown in the following example. 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. Members that are `public` can be accessed from outside the class, while members that are `private` cannot be

accessed from outside the class. It's recommended to make all your data members `private`, and if needed, to give access to them with `public` getters and setters. This way, you can easily change the representation of your data while keeping the `public` interface the same. The use of `protected` is explained in the context of inheritance in Chapters 5 and 10.

```
#include <string>

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

        double calculatePriceInDollars() const;

        const std::string& getPassengerName() const;
        void setPassengerName(const std::string& name);

        int getNumberOfMiles() const;
        void setNumberOfMiles(int miles);

        bool hasEliteSuperRewardsStatus() const;
        void setHasEliteSuperRewardsStatus(bool status);
    private:
        std::string mPassengerName;
        int mNumberOfMiles;
        bool mHasEliteSuperRewardsStatus;
};
```

This book follows the convention to prefix each data member of a class with a lowercase 'm', such as `mPassengerName`.

> **NOTE** *To follow the* `const`*-correctness principle, it's always a good idea to declare member functions that do not change any data member of the object as being* `const`*. These member functions are also called "inspectors," compared to "mutators" for non-*`const` *member functions.*

The method that has the same name as 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.

There are two ways of initializing data members with a constructor. The recommended way is to use a *constructor initializer*, which follows a colon after the constructor name. Here is the `AirlineTicket` constructor with a constructor initializer:

```
AirlineTicket::AirlineTicket()
    : mPassengerName("Unknown Passenger")
    , mNumberOfMiles(0)
    , mHasEliteSuperRewardsStatus(false)
{
}
```

A second way is to put the initializations in the body of the constructor, as shown here:

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

If the constructor is only initializing data members without doing anything else, then there is no real need for a constructor because data members can be initialized directly inside the class definition. For example, instead of writing an `AirlineTicket` constructor, you can modify the definition of the data members in the class definition as follows:

```
private:
    std::string mPassengerName = "Unknown Passenger";
    int mNumberOfMiles = 0;
    bool mHasEliteSuperRewardsStatus = false;
```

If your class additionally needs to perform some other types of initialization, such as opening a file, allocating memory, and so on, then you still need to write a constructor to handle those.

Here is the destructor for the `AirlineTicket` class:

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

This destructor doesn't do anything, and can simply be removed from this class. It is just shown here so you know the syntax of destructors. Destructors are required if you need to perform some cleanup, such as closing files, freeing memory, and so on. Chapters 8 and 9 discuss destructors in more detail.

The definitions of some of the `AirlineTicket` class methods are shown here:

```
double AirlineTicket::calculatePriceInDollars() const
{
    if (hasEliteSuperRewardsStatus()) {
        // 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 getNumberOfMiles() * 0.1;
}

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

void AirlineTicket::setPassengerName(const string& name)
{
    mPassengerName = name;
}
```

```
// Other get and set methods omitted for brevity.
```

Using Classes

The following sample program makes use of the `AirlineTicket` class. This example shows the creation of a stack-based `AirlineTicket` object as well as a heap-based one:

```
AirlineTicket myTicket;  // Stack-based AirlineTicket
myTicket.setPassengerName("Sherman T. Socketwrench");
myTicket.setNumberOfMiles(700);
double cost = myTicket.calculatePriceInDollars();
cout << "This ticket will cost $" << cost << endl;

// Heap-based AirlineTicket with smart pointer
auto myTicket2 = make_unique<AirlineTicket>();
myTicket2->setPassengerName("Laudimore M. Hallidue");
myTicket2->setNumberOfMiles(2000);
myTicket2->setHasEliteSuperRewardsStatus(true);
double cost2 = myTicket2->calculatePriceInDollars();
cout << "This other ticket will cost $" << cost2 << endl;
// No need to delete myTicket2, happens automatically

// Heap-based AirlineTicket without smart pointer (not recommended)
AirlineTicket* myTicket3 = new AirlineTicket();
// ... Use ticket 3
delete myTicket3;  // delete the heap object!
```

The preceding example exposes you to the general syntax for creating and using classes. Of course, there is much more to learn. Chapters 8, 9, and 10 go into more depth about the specific C++ mechanisms for defining classes.

UNIFORM INITIALIZATION

Before C++11, initialization of types was not always uniform. For example, take the following definition of a circle, once as a structure, and once as a class:

```
struct CircleStruct
{
    int x, y;
    double radius;
};

class CircleClass
{
    public:
        CircleClass(int x, int y, double radius)
            : mX(x), mY(y), mRadius(radius) {}
    private:
        int mX, mY;
        double mRadius;
};
```

In pre-C++11, initialization of a variable of type `CircleStruct` and a variable of type `CircleClass` looks different:

```
CircleStruct myCircle1 = {10, 10, 2.5};
CircleClass myCircle2(10, 10, 2.5);
```

For the structure version, you can use the `{ . . . }` syntax. However, for the class version, you need to call the constructor using function notation `(...)`.

Since C++11, you can more uniformly use the `{ . . . }` syntax to initialize types, as follows:

```
CircleStruct myCircle3 = {10, 10, 2.5};
CircleClass myCircle4 = {10, 10, 2.5};
```

The definition of `myCircle4` automatically calls the constructor of `CircleClass`. Even the use of the equal sign is optional, so the following is identical:

```
CircleStruct myCircle5{10, 10, 2.5};
CircleClass myCircle6{10, 10, 2.5};
```

Uniform initialization is not limited to structures and classes. You can use it to initialize anything in C++. For example, the following code initializes all four variables with the value 3:

```
int a = 3;
int b(3);
int c = {3};  // Uniform initialization
int d{3};     // Uniform initialization
```

Uniform initialization can be used to perform zero-initialization* of variables; you just specify an empty set of curly braces, as shown here:

```
int e{};      // Uniform initialization, e will be 0
```

Using uniform initialization prevents *narrowing*. C++ implicitly performs narrowing, as shown here:

```
void func(int i) { /* ... */ }

int main()
{
    int x = 3.14;
    func(3.14);
}
```

In both cases, C++ automatically truncates 3.14 to 3 before assigning it to `x` or calling `func()`. Note that some compilers *might* issue a warning about this narrowing, while others won't. With uniform initialization, both the assignment to `x` and the call to `func()` *must* generate a compiler error if your compiler fully conforms to the C++11 standard:

```
void func(int i) { /* ... */ }

int main()
{
    int x = {3.14};    // Error because narrowing
    func({3.14});      // Error because narrowing
}
```

**Zero-initialization* constructs objects with the default constructor, and initializes primitive integer types (such as `char`, `int`, and so on) to zero, primitive floating-point types to 0.0, and pointer types to `nullptr`.

Uniform initialization can be used to initialize dynamically allocated arrays, as shown here:

```
int* pArray = new int[4]{0, 1, 2, 3};
```

It can also be used in the constructor initializer to initialize arrays that are members of a class.

```
class MyClass
{
    public:
        MyClass() : mArray{0, 1, 2, 3} {}
    private:
        int mArray[4];
};
```

Uniform initialization can be used with the Standard Library containers as well—such as the `std::vector`, as demonstrated later in this chapter.

Direct List Initialization versus Copy List Initialization

There are two types of initialization that use braced initializer lists:

- Copy list initialization. `T obj = {arg1, arg2, ...};`
- Direct list initialization. `T obj {arg1, arg2, ...};`

In combination with auto type deduction, there is an important difference between copy- and direct list initialization introduced with C++17.

Starting with C++17, you have the following results:

```
// Copy list initialization
auto a = {11};          // initializer_list<int>
auto b = {11, 22};      // initializer_list<int>

// Direct list initialization
auto c {11};            // int
auto d {11, 22};        // Error, too many elements.
```

Note that for copy list initialization, all the elements in the braced initializer must be of the same type. For example, the following does not compile:

```
auto b = {11, 22.33};  // Compilation error
```

In earlier versions of the standard (C++11/14), both copy- and direct list initialization deduce an `initializer_list<>`:

```
// Copy list initialization
auto a = {11};          // initializer_list<int>
auto b = {11, 22};      // initializer_list<int>

// Direct list initialization
auto c {11};            // initializer_list<int>
auto d {11, 22};        // initializer_list<int>
```

THE STANDARD LIBRARY

C++ comes with a Standard Library, which contains a lot of useful classes that can easily be used in your code. The benefit of using these classes is that you don't need to reinvent certain classes and you don't need to waste time on implementing things that have already been implemented for you. Another benefit is that the classes available in the Standard Library are heavily tested and verified for correctness by thousands of users. The Standard Library classes are also tuned for high performance, so using them will most likely result in better performance compared to making your own implementation.

A lot of functionality is available to you in the Standard Library. Chapters 16 to 20 provide more details; however, when you start working with C++ it is a good idea to understand what the Standard Library can do for you from the very beginning. This is especially important if you are a C programmer. As a C programmer, you might try to solve problems in C++ the same way you would solve them in C. However, in C++ there is probably an easier and safer solution to the problem that involves using Standard Library classes.

You already saw some Standard Library classes earlier in this chapter—for example, `std::string`, `std::array`, `std::vector`, `std::unique_ptr`, and `std::shared_ptr`. Many more classes are introduced in Chapters 16 to 20.

YOUR FIRST USEFUL C++ PROGRAM

The following program builds on the employee database example used earlier in the discussion on 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, `vector`s, 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 abilities:

- To add an employee
- To fire an employee
- To promote an employee
- To view all employees, past and present
- To view all current employees
- 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. `Employees` 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 defines the `Employee` class. The sections of this file are described individually in the text that follows.

The first line contains a `#pragma once` to prevent the file from being included multiple times, followed by the inclusion of the `string` functionality.

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

```
#pragma once
#include <string>
namespace Records {
```

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

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

Note that this book uses the convention to prefix constants with a lowercase 'k', from the German "Konstant," meaning "Constant."

The `Employee` class is defined, 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 setters and getters provide mechanisms to change the information about an employee or to query the current information about an employee.

The `Employee` class includes an explicitly defaulted constructor, as discussed in Chapter 8. It also includes a constructor that accepts a first and last name.

```
class Employee
{
    public:
        Employee() = default;
        Employee(const std::string& firstName,
                 const std::string& lastName);

        void promote(int raiseAmount = 1000);
        void demote(int demeritAmount = 1000);
        void hire(); // Hires or rehires the employee
        void fire(); // Dismisses the employee
        void display() const;// Outputs employee info to console
```

```
        // Getters and setters
        void setFirstName(const std::string& firstName);
        const std::string& getFirstName() const;

        void setLastName(const std::string& lastName);
        const std::string& getLastName() const;

        void setEmployeeNumber(int employeeNumber);
        int getEmployeeNumber() const;

        void setSalary(int newSalary);
        int getSalary() const;

        bool isHired() const;
```

Finally, the data members are declared as `private` so that other parts of the code cannot modify them directly. The setters and getters provide the only public way of modifying or querying those values. The data members are also initialized here instead of in a constructor. By default, new employees have no name, an employee number of -1, the default starting salary, and a status of not hired.

```
    private:
        std::string mFirstName;
        std::string mLastName;
        int mEmployeeNumber = -1;
        int mSalary = kDefaultStartingSalary;
        bool mHired = false;
};
}
```

Employee.cpp

The constructor accepting a first and last name just sets the corresponding data members:

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

using namespace std;

namespace Records {
    Employee::Employee(const std::string& firstName,
                       const std::string& lastName)
        : mFirstName(firstName), mLastName(lastName)
    {
    }
```

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 are only allowed in a function definition.

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

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

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

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

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

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 getters, such as `getSalary()`. However, it is considered good style to make use of getters and setters when they exist, even within the class.

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

A number of getters and setters perform the task of getting and setting values. Even though these methods seem trivial, it's better to have trivial getters and setters than to make your data members `public`. For example, in the future, you may want to perform bounds checking in the `setSalary()` method. Getters and setters also make debugging easier because you can insert a breakpoint in them to inspect values when they are retrieved or set. Another reason is that when you decide to change how you are storing the data in your class, you only need to modify these getters and setters.

```
    // Getters and setters
    void Employee::setFirstName(const string& firstName)
    {
        mFirstName = firstName;
    }

    const string& Employee::getFirstName() const
    {
        return mFirstName;
    }
    // ... other getters and setters omitted for brevity
}
```

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.

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

using namespace std;
using namespace Records;

int main()
{
    cout << "Testing the Employee class." << endl;
    Employee emp;
    emp.setFirstName("John");
    emp.setLastName("Doe");
    emp.setEmployeeNumber(71);
    emp.setSalary(50000);
    emp.promote();
    emp.promote(50);
    emp.hire();
    emp.display();
    return 0;
}
```

Another way of testing individual classes is with *unit testing*, which is discussed in Chapter 26.

The Database Class

The `Database` class uses the `std::vector` class from the Standard Library to store `Employee` objects.

Database.h

Because the database will take care of automatically assigning an employee number to a new employee, a constant defines where the numbering begins.

```
#pragma once
#include <iostream>
#include <vector>
#include "Employee.h"

namespace Records {
    const int kFirstEmployeeNumber = 1000;
```

The database provides an easy way to add a new employee by providing a first and last name. For convenience, this method returns 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.

```
    class Database
    {
        public:
```

```
        Employee& addEmployee(const std::string& firstName,
                              const std::string& lastName);
        Employee& getEmployee(int employeeNumber);
        Employee& getEmployee(const std::string& firstName,
                              const std::string& lastName);
```

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

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

`mEmployees` contains the `Employee` objects. The `mNextEmployeeNumber` data member keeps track of what employee number is assigned to a new employee, and is initialized with the `kFirstEmployeeNumber` constant.

```
    private:
        std::vector<Employee> mEmployees;
        int mNextEmployeeNumber = kFirstEmployeeNumber;
};
}
```

Database.cpp

The `addEmployee()` method creates a new `Employee` object, fills in its information, and adds it to the `vector`. The `mNextEmployeeNumber` data member is incremented after its use so that the next employee will get a new number.

```
#include <iostream>
#include <stdexcept>
#include "Database.h"

using namespace std;

namespace Records {
    Employee& Database::addEmployee(const string& firstName,
                                    const string& lastName)
    {
        Employee theEmployee(firstName, lastName);
        theEmployee.setEmployeeNumber(mNextEmployeeNumber++);
        theEmployee.hire();
        mEmployees.push_back(theEmployee);
        return mEmployees[mEmployees.size() - 1];
    }
```

Only one version of `getEmployee()` is shown. Both versions work in similar ways. The methods loop over all employees in `mEmployees` using range-based `for` loops, and check to see if an `Employee` is a match for the information passed to the method. An exception is thrown if no match is found.

```
    Employee& Database::getEmployee(int employeeNumber)
    {
        for (auto& employee : mEmployees) {
            if (employee.getEmployeeNumber() == employeeNumber) {
```

```
                return employee;
            }
        }
        throw logic_error("No employee found.");
    }
```

The display methods all use a similar algorithm. They loop through all employees and tell each employee to display itself to the console if the criterion for display matches. `displayFormer()` is similar to `displayCurrent()`.

```
    void Database::displayAll() const
    {
        for (const auto& employee : mEmployees) {
            employee.display();
        }
    }

    void Database::displayCurrent() const
    {
        for (const auto& employee : mEmployees) {
            if (employee.isHired())
                employee.display();
        }
    }
}
```

DatabaseTest.cpp

A simple test for the basic functionality of the database is shown here:

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

using namespace std;
using namespace Records;

int main()
{
    Database myDB;
    Employee& emp1 = myDB.addEmployee("Greg", "Wallis");
    emp1.fire();

    Employee& emp2 = myDB.addEmployee("Marc", "White");
    emp2.setSalary(100000);

    Employee& emp3 = myDB.addEmployee("John", "Doe");
    emp3.setSalary(10000);
    emp3.promote();

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

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

```
    cout << endl << "former employees: " << endl << 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.

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.

```
#include <iostream>
#include <stdexcept>
#include <exception>
#include "Database.h"

using namespace std;
using namespace Records;

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

int main()
{
    Database employeeDB;
    bool done = false;
    while (!done) {
        int selection = displayMenu();
        switch (selection) {
        case 0:
            done = true;
            break;
        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;
        default:
            cerr << "Unknown command." << endl;
            break;
        }
    }
    return 0;
}
```

The `displayMenu()` function 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 13, you will learn how to protect against bad input.

```
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 former employees" << endl;
    cout << "0) Quit" << endl;
    cout << endl;
    cout << "---> ";
    cin >> selection;
    return selection;
}
```

The `doHire()` function gets the new employee's name from the user and tells the database to add the employee.

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

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

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

    db.addEmployee(firstName, lastName);
}
```

`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.

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

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

    try {
        Employee& emp = db.getEmployee(employeeNumber);
        emp.fire();
        cout << "Employee " << employeeNumber << " terminated." << endl;
    } catch (const std::logic_error& exception) {
        cerr << "Unable to terminate employee: " << exception.what() << endl;
    }
}

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

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

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

    try {
        Employee& emp = db.getEmployee(employeeNumber);
        emp.promote(raiseAmount);
    } catch (const std::logic_error& exception) {
        cerr << "Unable to promote employee: " << exception.what() << endl;
    }
}
```

Evaluating the Program

The preceding program covers a number of topics from the very simple to the relatively complex. 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 `mEmployees`.

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 conditional operator, write a short `main()` function that uses it.

SUMMARY

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

The next chapter goes deeper in on how strings are handled in C++, because every program you write will have to work with strings one way or another.