

Module 2

Introducing Data Types and Operators

Table of Contents

CRITICAL SKILL 2.1: The C++ Data Types	2
Project 2-1 Talking to Mars	10
CRITICAL SKILL 2.2: Literals	12
CRITICAL SKILL 2.3: A Closer Look at Variables	15
CRITICAL SKILL 2.4: Arithmetic Operators.....	17
CRITICAL SKILL 2.5: Relational and Logical Operators	20
Project 2-2 Construct an XOR Logical Operation	22
CRITICAL SKILL 2.6: The Assignment Operator	25
CRITICAL SKILL 2.7: Compound Assignments.....	25
CRITICAL SKILL 2.8: Type Conversion in Assignments.....	26
CRITICAL SKILL 2.9: Type Conversion in Expressions	27
CRITICAL SKILL 2.10: Casts.....	27
CRITICAL SKILL 2.11: Spacing and Parentheses.....	28
Project 2-3 Compute the Regular Payments on a Loan	29

At the core of a programming language are its data types and operators. These elements define the limits of a language and determine the kind of tasks to which it can be applied. As you might expect, C++ supports a rich assortment of both data types and operators, making it suitable for a wide range of programming. Data types and operators are a large subject. We will begin here with an examination of C++'s foundational data types and its most commonly used operators. We will also take a closer look at variables and examine the expression.

Why Data Types Are Important

The data type of a variable is important because it determines the operations that are allowed and the range of values that can be stored. C++ defines several types of data, and each type has unique characteristics. Because data types differ, all variables must be declared prior to their use, and a variable declaration always includes a type specifier. The compiler requires this information in order to generate correct code. In C++ there is no concept of a “type-less” variable.

A second reason that data types are important to C++ programming is that several of the basic types are closely tied to the building blocks upon which the computer operates: bytes and words. Thus, C++ lets you operate on the same types of data as does the CPU itself. This is one of the ways that C++ enables you to write very efficient, system-level code.

CRITICAL SKILL 2.1: The C++ Data Types

C++ provides built-in data types that correspond to integers, characters, floating-point values, and Boolean values. These are the ways that data is commonly stored and manipulated by a program. As you will see later in this book, C++ allows you to construct more sophisticated types, such as classes, structures, and enumerations, but these too are ultimately composed of the built-in types.

At the core of the C++ type system are the seven basic data types shown here:

Type	Meaning
char	Character
wchar_t	Wide character
int	Integer
float	Floating point
double	Double floating point
bool	Boolean
void	Valueless

C++ allows certain of the basic types to have modifiers preceding them. A modifier alters the meaning of the base type so that it more precisely fits the needs of various situations. The data type modifiers are listed here:

signed
unsigned
long
short

The modifiers signed, unsigned, long, and short can be applied to int. The modifiers signed and unsigned can be applied to the char type. The type double can be modified by long. Table 2-1 shows all valid

combinations of the basic types and the type modifiers. The table also shows the guaranteed minimum range for each type as specified by the ANSI/ISO C++ standard.

Type	Minimal Range
char	−127 to 127
unsigned char	0 to 255
signed char	−127 to 127
int	−32,767 to 32,767
unsigned int	0 to 65,535
signed int	Same as int
short int	−32,767 to 32,767
unsigned short int	0 to 65,535
signed short int	Same as short int
long int	−2,147,483,647 to 2,147,483,647
signed long int	Same as long int
unsigned long int	0 to 4,294,967,295
float	1E−37 to 1E+37, with six digits of precision
double	1E−37 to 1E+37, with ten digits of precision
long double	1E−37 to 1E+37, with ten digits of precision

Table 2-1 All Numeric Data Types Defined by C++ and Their Minimum Guaranteed Ranges as Specified by the ANSI/ISO C++ Standard

It is important to understand that minimum ranges shown in Table 2-1 are just that: minimum ranges. A C++ compiler is free to exceed one or more of these minimums, and most compilers do. Thus, the ranges of the C++ data types are implementation dependent. For example, on computers that use two's complement arithmetic (which is nearly all), an integer will have a range of at least −32,768 to 32,767. In all cases, however, the range of a short int will be a subrange of an int, which will be a subrange of a long int. The same applies to float, double, and long double. In this usage, the term subrange means a range narrower than or equal to. Thus, an int and long int can have the same range, but an int cannot be larger than a long int.

Since C++ specifies only the minimum range a data type must support, you should check your compiler's documentation for the actual ranges supported. For example, Table 2-2 shows typical bit widths and ranges for the C++ data types in a 32-bit environment, such as that used by Windows XP.

Let's now take a closer look at each data type.

Type	Bit Width	Typical Range
char	8	−128 to 127
unsigned char	8	0 to 255
signed char	8	−128 to 127
int	32	−2,147,483,648 to 2,147,483,647
unsigned int	32	−2,147,483,648 to 2,147,483,647
signed int	32	0 to 4,294,967,295
short int	16	−32,768 to 32,767
unsigned short int	16	−32,768 to 32,767
signed short int	16	0 to 65,535
long int	32	Same as int
signed long int	32	Same as signed int
unsigned long int	32	Same as unsigned int
float	32	1.8E−38 to 3.4E+38
double	32	2.2E−308 to 1.8E+308
long double	64	2.2E−308 to 1.8E+308
bool	N/A	True or false
wchar_t	16	0 to 65,535

Table 2-2 Typical Bit Widths and Ranges for the C++ Data Types in a 32-Bit Environment

Integers

As you learned in Module 1, variables of type `int` hold integer quantities that do not require fractional components. Variables of this type are often used for controlling loops and conditional statements, and for counting. Because they don't have fractional components, operations on `int` quantities are much faster than they are on floating-point types.

Because integers are so important to programming, C++ defines several varieties. As shown in Table 2-1, there are short, regular, and long integers. Furthermore, there are signed and unsigned versions of each. A signed integer can hold both positive and negative values. By default, integers are signed. Thus, the use of signed on integers is redundant (but allowed) because the default declaration assumes a signed value. An unsigned integer can hold only positive values. To create an unsigned integer, use the unsigned modifier.

The difference between signed and unsigned integers is in the way the high-order bit of the integer is interpreted. If a signed integer is specified, then the C++ compiler will generate code that assumes that the high-order bit of an integer is to be used as a sign flag. If the sign flag is 0, then the number is positive; if it is 1, then the number is negative. Negative numbers are almost always represented using

the two's complement approach. In this method, all bits in the number (except the sign flag) are reversed, and then 1 is added to this number. Finally, the sign flag is set to 1.

Signed integers are important for a great many algorithms, but they have only half the absolute magnitude of their unsigned relatives. For example, assuming a 16-bit integer, here is 32,767:

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

For a signed value, if the high-order bit were set to 1, the number would then be interpreted as -1 (assuming the two's complement format). However, if you declared this to be an unsigned int, then when the high-order bit was set to 1, the number would become 65,535.

To understand the difference between the way that signed and unsigned integers are interpreted by C++, try this short program:

```
#include <iostream>

/* This program shows the difference between
   signed and unsigned integers. */

using namespace std;

int main()
{
    short int i; // a signed short integer
    unsigned short j; // an unsigned short integer
    j = 60000;
    i = j;
    cout << i << " " << j;

    return 0;
}
```

60,000 is within the range of an **unsigned short int**, but is typically outside the range of a **signed short int**. Thus, it will be interpreted as a negative value when assigned to **i**.

The output from this program is shown here:

-5536 60000

These values are displayed because the bit pattern that represents 60,000 as a short unsigned integer is interpreted as -5,536 as short signed integer (assuming 16-bit short integers).

C++ allows a shorthand notation for declaring unsigned, short, or long integers. You can simply use the word unsigned, short, or long, without the int. The int is implied. For example, the following two statements both declare unsigned integer variables:

```
unsigned x;
unsigned int y;
```

Characters

Variables of type `char` hold 8-bit ASCII characters such as `A`, `z`, or `G`, or any other 8-bit quantity. To specify a character, you must enclose it between single quotes. Thus, this assigns `X` to the variable `ch`:

```
char ch;  
ch = 'X';
```

You can output a `char` value using a `cout` statement. For example, this line outputs the value in `ch`:

```
cout << "This is ch: " << ch;
```

This results in the following output:


```
This is ch: X
```

The `char` type can be modified with `signed` or `unsigned`. Technically, whether `char` is signed or unsigned by default is implementation-defined. However, for most compilers `char` is signed. In these environments, the use of `signed` on `char` is also redundant. For the rest of this book, it will be assumed that `chars` are signed entities.

The type `char` can hold values other than just the ASCII character set. It can also be used as a “small” integer with the range typically from `-128` through `127` and can be substituted for an `int` when the situation does not require larger numbers. For example, the following program uses a `char` variable to control the loop that prints the alphabet on the screen:

```
// This program displays the alphabet.  
  
#include <iostream>  
using namespace std;  
  
int main()  
{  
    char letter;  
    for(letter = 'A'; letter <= 'Z'; letter++)  
        cout << letter;  
  
    return 0;  
}
```

Use a `char` variable
to control a `for` loop.



The `for` loop works because the character `A` is represented inside the computer by the value `65`, and the values for the letters `A` to `Z` are in sequential, ascending order. Thus, `letter` is initially set to `'A'`. Each time through the loop, `letter` is incremented. Thus, after the first iteration, `letter` is equal to `'B'`.

The type `wchar_t` holds characters that are part of large character sets. As you may know, many human languages, such as Chinese, define a large number of characters, more than will fit within the 8 bits provided by the `char` type. The `wchar_t` type was added to C++ to accommodate this situation. While we

won't be making use of `wchar_t` in this book, it is something that you will want to look into if you are tailoring programs for the international market.



Progress Check

1. What are the seven basic types?
2. What is the difference between signed and unsigned integers?
3. Can a `char` variable be used like a little integer?

Answer Key:

1. The seven basic types are `char`, `wchar_t`, `int`, `float`, `double`, `bool`, and `void`.
2. A signed integer can hold both positive and negative values. An unsigned integer can hold only positive values.
3. Yes.

Ask the Expert

Q: Why does C++ specify only minimum ranges for its built-in types rather than stating these precisely?

A: By not specifying precise ranges, C++ allows each compiler to optimize the data types for the execution environment. This is part of the reason that C++ can create high-performance software. The ANSI/ISO C++ standard simply states that the built-in types must meet certain requirements. For example, it states that an `int` will “have the natural size suggested by the architecture of the execution environment.” Thus, in a 32-bit environment, an `int` will be 32 bits long. In a 16-bit environment, an `int` will be 16 bits long. It would be an inefficient and unnecessary burden to force a 16-bit compiler to implement `int` with a 32-bit range, for example. C++’s approach avoids this. Of course, the C++ standard does specify a minimum range for the built-in types that will be available in all environments. Thus, if you write your programs in such a way that these minimal ranges are not exceeded, then your program will be portable to other environments. One last point: Each C++ compiler specifies the range of the basic types in the header `<climits>`.

Floating-Point Types

Variables of the types `float` and `double` are employed either when a fractional component is required or when your application requires very large or small numbers. The difference between a `float` and a `double` variable is the magnitude of the largest (and smallest) number that each one can hold. Typically, a `double` can store a number approximately ten times larger than a `float`. Of the two, `double` is the most commonly used. One reason for this is that many of the math functions in the C++ function library use `double` values. For example, the `sqrt()` function returns a `double` value that is the square root of its `double` argument. Here, `sqrt()` is used to compute the length of the hypotenuse given the lengths of the two opposing sides.

```
/*
    Use the Pythagorean theorem to find
    the length of the hypotenuse given
    the lengths of the two opposing sides.
*/

#include <iostream>
#include <cmath>  ← The <cmath> header is needed for the sqrt( ) function.
using namespace std;

int main() {
    double x, y, z;

    x = 5.0;
    y = 4.0;

    z = sqrt(x*x + y*y); ← The sqrt( ) function is part of C++'s math library.

    cout << "Hypotenuse is " << z;

    return 0;
}
```

The output from the program is shown here:

```
Hypotenuse is 6.40312
```

One other point about the preceding example: Because `sqrt()` is part of the C++ standard function library, it requires the standard header `<cmath>`, which is included in the program.

The long `double` type lets you work with very large or small numbers. It is most useful in scientific programs. For example, the long `double` type might be useful when analyzing astronomical data.

The `bool` Type

The `bool` type is a relatively recent addition to C++. It stores Boolean (that is, true/false) values. C++ defines two Boolean constants, `true` and `false`, which are the only two values that a `bool` value can have. Before continuing, it is important to understand how `true` and `false` are defined by C++. One of the fundamental concepts in C++ is that any nonzero value is interpreted as `true` and zero is `false`. This

concept is fully compatible with the `bool` data type because when used in a Boolean expression, C++ automatically converts any nonzero value into `true`. It automatically converts zero into `false`. The reverse is also true; when used in a non-Boolean expression, `true` is converted into 1, and `false` is converted into zero. The convertibility of zero and nonzero values into their Boolean equivalents is especially important when using control statements, as you will see in Module 3. Here is a program that demonstrates the `bool` type:

```
// Demonstrate bool values.

#include <iostream>

using namespace std;

int main() {
    bool b;

    b = false;
    cout << "b is " << b << "\n";


    b = true;
    cout << "b is " << b << "\n";

    // a bool value can control the if statement
    if(b) cout << "This is executed.\n";

    b = false;
    if(b) cout << "This is not executed.\n";

    // outcome of a relational operator is a true/false value
    cout << "10 > 9 is " << (10 > 9) << "\n";

    return 0;
}
```



A single `bool` value can control an if statement.

The output generated by this program is shown here:

```
b is 0
b is 1
This is executed.
10 > 9 is 1
```

There are three interesting things to notice about this program. First, as you can see, when a `bool` value is output using `cout`, 0 or 1 is displayed. As you will see later in this book, there is an output option that causes the words “false” and “true” to be displayed.

Second, the value of a `bool` variable is sufficient, by itself, to control the if statement. There is no need to write an if statement like this:

```
if(b == true) ...
```

Third, the outcome of a relational operator, such as `<`, is a Boolean value. This is why the expression `10 > 9` displays the value 1. Further, the extra set of parentheses around `10 > 9` is necessary because the `<<` operator has a higher precedence than the `>`.

void

The void type specifies a valueless expression. This probably seems strange now, but you will see how void is used later in this book.



Progress Check

1. What is the primary difference between float and double?
2. What values can a bool variable have? To what Boolean value does zero convert?
3. What is void?

Answer Key:

1. The primary difference between float and double is in the magnitude of the values they can hold.
2. Variables of type bool can be either true or false. Zero converts to false.
3. void is a type that stands for valueless.

Project 2-1 Talking to Mars

At its closest point to Earth, Mars is approximately 34,000,000 miles away. Assuming there is someone on Mars that you want to talk with, what is the delay between the time a radio signal leaves Earth and the time it arrives on Mars? This project creates a program that answers this question. Recall that radio signals travel at the speed of light, approximately 186,000 miles per second. Thus, to compute the delay, you will need to divide the distance by the speed of light. Display the delay in terms of seconds and also in minutes.

Step by Step

1. Create a new file called Mars.cpp.
2. To compute the delay, you will need to use floating-point values. Why? Because the time interval will have a fractional component. Here are the variables used by the program:

```
double distance;
```

```
double lightspeed;
double delay;
double delay_in_min;
```

3. Give distance and lightspeed initial values, as shown here:

```
distance = 34000000.0; // 34,000,000 miles
lightspeed = 186000.0; // 186,000 per second
```

4. To compute the delay, divide distance by lightspeed. This yields the delay in seconds. Assign this value to delay and display the results. These steps are shown here:

```
delay = distance / lightspeed;
cout << "Time delay when talking to Mars: " << delay << " seconds.\n";
```

5. Divide the number of seconds in delay by 60 to obtain the delay in minutes; display that result using these lines of code:

```
delay_in_min = delay / 60.0;
```

6. Here is the entire Mars.cpp program listing:

```
/*
Project 2-1
Talking to Mars
*/

#include <iostream>
using namespace std;

int main()
{
    double distance;
    double lightspeed;
    double delay;
    double delay_in_min;

    distance = 34000000.0; // 34,000,000 miles
    lightspeed = 186000.0; // 186,000 per second

    delay = distance / lightspeed;

    cout << "Time delay when talking to Mars: " << delay << " seconds.\n";

    delay_in_min = delay / 60.0;

    cout << "This is " << delay_in_min << " minutes.";
```

```
    return 0;
}
```

7. Compile and run the program. The following result is displayed:

```
Time delay when talking to Mars: 182.796 seconds.
This is 3.04659 minutes.
```

8. On your own, display the time delay that would occur in a bidirectional conversation with Mars.

CRITICAL SKILL 2.2: Literals

Literals refer to fixed, human-readable values that cannot be altered by the program. For example, the value 101 is an integer literal. Literals are also commonly referred to as constants. For the most part, literals and their usage are so intuitive that they have been used in one form or another by all the preceding sample programs. Now the time has come to explain them formally.

C++ literals can be of any of the basic data types. The way each literal is represented depends upon its type. As explained earlier, character literals are enclosed between single quotes. For example, 'a' and '%' are both character literals.

Integer literals are specified as numbers without fractional components. For example, 10 and -100 are integer constants. Floating-point literals require the use of the decimal point followed by the number's fractional component. For example, 11.123 is a floating-point constant. C++ also allows you to use scientific notation for floating-point numbers.

All literal values have a data type, but this fact raises a question. As you know, there are several different types of integers, such as int, short int, and unsigned long int. There are also three different floating-point types: float, double, and long double. The question is: How does the compiler determine the type of a literal? For example, is 123.23 a float or a double? The answer to this question has two parts. First, the C++ compiler automatically makes certain assumptions about the type of a literal and, second, you can explicitly specify the type of a literal, if you like.

By default, the C++ compiler fits an integer literal into the smallest compatible data type that will hold it, beginning with int. Therefore, assuming 16-bit integers, 10 is int by default, but 103,000 is long. Even though the value 10 could be fit into a char, the compiler will not do this because it means crossing type boundaries.

By default, floating-point literals are assumed to be double. Thus, the value 123.23 is of type double.

For virtually all programs you will write as a beginner, the compiler defaults are perfectly adequate. In cases where the default assumption that C++ makes about a numeric literal is not what you want, C++ allows you to specify the exact type of numeric literal by using a suffix. For floating-point types, if you follow the number with an F, the number is treated as a float. If you follow it with an L, the number becomes a long double. For integer types, the U suffix stands for unsigned and the L for long. (Both the U and the L must be used to specify an unsigned long.) Some examples are shown here:

Data Type	Examples of Constants
int	1 123 21000 -234
long int	35000L -34L
unsigned int	10000U 987U 40000U
unsigned long	12323UL 900000UL
float	123.23F 4.34e-3F
double	23.23 123123.33 -0.9876324
long double	1001.2L

Hexadecimal and Octal Literals

As you probably know, in programming it is sometimes easier to use a number system based on 8 or 16 instead of 10. The number system based on 8 is called octal, and it uses the digits 0 through 7. In octal, the number 10 is the same as 8 in decimal. The base-16 number system is called hexadecimal and uses the digits 0 through 9 plus the letters A through F, which stand for 10, 11, 12, 13, 14, and 15. For example, the hexadecimal number 10 is 16 in decimal. Because of the frequency with which these two number systems are used, C++ allows you to specify integer literals in hexadecimal or octal instead of decimal. A hexadecimal literal must begin with 0x (a zero followed by an x). An octal literal begins with a zero. Here are some examples:

```
hex = 0xFF; // 255 in decimal
oct = 011; // 9 in decimal
```

String Literals

C++ supports one other type of literal in addition to those of the predefined data types: the string. A string is a set of characters enclosed by double quotes. For example, "this is a test" is a string. You have seen examples of strings in some of the cout statements in the preceding sample programs. Keep in mind one important fact: although C++ allows you to define string constants, it does not have a built-in string data type. Instead, as you will see a little later in this book, strings are supported in C++ as character arrays. (C++ does, however, provide a string type in its class library.)

Ask the Expert

Q: You showed how to specify a char literal. Is a wchar_t literal specified in the same way?

A: No. A wide-character constant (that is, one that is of type wchar_t) is preceded with the character L. For example:

```
wchar_t wc;
wc = L'A';
```

Here, `wc` is assigned the wide-character constant equivalent of `A`. You will not use wide characters often in your normal day-to-day programming, but they are something that might be of importance if you need to internationalize your program.

Character Escape Sequences

Enclosing character constants in single quotes works for most printing characters, but a few characters, such as the carriage return, pose a special problem when a text editor is used. In addition, certain other characters, such as the single and double quotes, have special meaning in C++, so you cannot use them directly. For these reasons, C++ provides the character escape sequences, sometimes referred to as backslash character constants, shown in Table 2-3, so that you can enter them into a program. As you can see, the `\n` that you have been using is one of the escape sequences.

Code	Meaning
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	Newline
<code>\r</code>	Carriage return
<code>\t</code>	Horizontal tab
<code>\"</code>	Double quote
<code>\'</code>	Single quote character
<code>\\</code>	Backslash
<code>\v</code>	Vertical tab
<code>\a</code>	Alert
<code>\?</code>	?
<code>\N</code>	Octal constant (where <i>N</i> is an octal constant)
<code>\xN</code>	Hexadecimal constant (where <i>N</i> is a hexadecimal constant)

Table 2-3 The Character Escape Sequences

Ask the Expert

Q: Is a string consisting of a single character the same as a character literal? For example, is `"k"` the same as `'k'`?

A: No. You must not confuse strings with characters. A character literal represents a single letter of type `char`. A string containing only one letter is still a string. Although strings consist of characters, they are not the same type.

The following sample program illustrates a few of the escape sequences:

```
// Demonstrate some escape sequences.

#include <iostream>
using namespace std;

int main()
{
    cout << "one\ttwo\tthree\n";
    cout << "123\b\b45"; ← The \b\b will backspace over the 2 and 3.

    return 0;
}
```

The output is shown here:

```
one      two      three
145
```

Here, the first cout statement uses tabs to position the words “two” and “three”. The second cout statement displays the characters 123. Next, two backspace characters are output, which deletes the 2 and 3. Finally, the characters 4 and 5 are displayed.



Progress Check

1. By default, what is the type of the literal 10? What is the type of the literal 10.0?
2. How do you specify 100 as a long int? How do you specify 100 as an unsigned int?
3. What is \b?

Answer Key:

1. 10 is an int and 10.0 is a double.
2. 100 as a long int is 100L. 100 as an unsigned int is 100U.
3. \b is the escape sequence that causes a backspace.

CRITICAL SKILL 2.3: A Closer Look at Variables

Variables were introduced in Module 1. Here we will take a closer look at them. As you learned, variables are declared using this form of statement:

type var-name;

where type is the data type of the variable and var-name is its name. You can declare a variable of any valid type. When you create a variable, you are creating an instance of its type. Thus, the capabilities of a variable are determined by its type. For example, a variable of type bool stores Boolean values. It cannot be used to store floating-point values. Furthermore, the type of a variable cannot change during its lifetime. An int variable cannot turn into a double variable, for example.

Initializing a Variable

You can assign a value to a variable at the same time that it is declared. To do this, follow the variable's name with an equal sign and the value being assigned. This is called a variable initialization. Its general form is shown here:

type var = value;

Here, value is the value that is given to var when var is created.

Here are some examples:

```
int count = 10; // give count an initial value of 10
char ch = 'X'; // initialize ch with the letter X
float f = 1.2F; // f is initialized with 1.2
```

When declaring two or more variables of the same type using a comma separated list, you can give one or more of those variables an initial value. For example,

```
int a, b = 8, c = 19, d; // b and c have initializations
```

In this case, only b and c are initialized.

Dynamic Initialization

Although the preceding examples have used only constants as initializers, C++ allows variables to be initialized dynamically, using any expression valid at the time the variable is declared. For example, here is a short program that computes the volume of a cylinder given the radius of its base and its height:


```
// Demonstrate dynamic initialization.

#include <iostream>
using namespace std;

int main() {
    double radius = 4.0, height = 5.0;

    // dynamically initialize volume
    double volume = 3.1416 * radius * radius * height;

    cout << "Volume is " << volume;

    return 0;
}
```

←
volume is dynamically
initialized at runtime.

Here, three local variables—radius, height, and volume—are declared. The first two, radius and height, are initialized by constants. However, volume is initialized dynamically to the volume of the cylinder. The key point here is that the initialization expression can use any element valid at the time of the initialization, including calls to functions, other variables, or literals.

Operators

C++ provides a rich operator environment. An operator is a symbol that tells the compiler to perform a specific mathematical or logical manipulation. C++ has four general classes of operators: arithmetic, bitwise, relational, and logical. C++ also has several additional operators that handle certain special situations. This chapter will examine the arithmetic, relational, and logical operators. We will also examine the assignment operator. The bitwise and other special operators are examined later.

CRITICAL SKILL 2.4: Arithmetic Operators

C++ defines the following arithmetic operators:

Operator	Meaning
+	Addition
−	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement

The operators +, −, *, and / all work the same way in C++ as they do in algebra. These can be applied to any built-in numeric data type. They can also be applied to values of type char.

The % (modulus) operator yields the remainder of an integer division. Recall that when / is applied to an integer, any remainder will be truncated; for example, 10/3 will equal 3 in integer division. You can obtain the remainder of this division by using the % operator. For example, 10 % 3 is 1. In C++, the % can be applied only to integer operands; it cannot be applied to floating-point types.

The following program demonstrates the modulus operator:

```
// Demonstrate the modulus operator.

#include <iostream>
using namespace std;

int main()
{
    int x, y;

    x = 10;
    y = 3;
    cout << x << " / " << y << " is " << x / y <<
        " with a remainder of " << x % y << "\n";

    x = 1;
    y = 2;
    cout << x << " / " << y << " is " << x / y << "\n" <<
        x << " % " << y << " is " << x % y;

    return 0;
}
```

The output is shown here:

```
10 / 3 is 3 with a remainder of 1
1 / 2 is 0
1 % 2 is 1
```

Increment and Decrement

Introduced in Module 1, the ++ and the -- are the increment and decrement operators. They have some special properties that make them quite interesting. Let's begin by reviewing precisely what the increment and decrement operators do.

The increment operator adds 1 to its operand, and the decrement operator subtracts 1. Therefore,

```
x = x + 1;
```

is the same as

```
x++;
```

and

```
x = x - 1;
```

is the same as

```
--x;
```

Both the increment and decrement operators can either precede (prefix) or follow (postfix) the operand. For example,

```
x = x + 1;
```

can be written as

```
++x; // prefix form
```

or as

```
x++; // postfix form
```

In this example, there is no difference whether the increment is applied as a prefix or a postfix. However, when an increment or decrement is used as part of a larger expression, there is an important difference. When an increment or decrement operator precedes its operand, C++ will perform the operation prior to obtaining the operand's value for use by the rest of the expression. If the operator follows its operand, then C++ will obtain the operand's value before incrementing or decrementing it. Consider the following:

```
x = 10; y = ++x;
```

In this case, y will be set to 11. However, if the code is written as

```
x = 10; y = x++;
```

then y will be set to 10. In both cases, x is still set to 11; the difference is when it happens. There are significant advantages in being able to control when the increment or decrement operation takes place.

The precedence of the arithmetic operators is shown here:

Highest	++ --
	- (unary minus)
	* / %
Lowest	+ -

Operators on the same precedence level are evaluated by the compiler from left to right. Of course, parentheses may be used to alter the order of evaluation. Parentheses are treated by C++ in the same way that they are by virtually all other computer languages: they force an operation, or a set of operations, to have a higher precedence level.

Ask the Expert

Q: Does the increment operator ++ have anything to do with the name C++?

A: Yes! As you know, C++ is built upon the C language. C++ adds to C several enhancements, most of which support object-oriented programming. Thus, C++ represents an incremental improvement to C, and the addition of the ++ (which is, of course, the increment operator) to the name C is a fitting way to describe C++.

Stroustrup initially named C++ “C with Classes,” but at the suggestion of Rick Mascitti, he later changed the name to C++. While the new language was already destined for success, the adoption of the name C++ virtually guaranteed its place in history because it was a name that every C programmer would instantly recognize!

CRITICAL SKILL 2.5: Relational and Logical Operators

In the terms relational operator and logical operator, relational refers to the relationships that values can have with one another, and logical refers to the ways in which true and false values can be connected together. Since the relational operators produce true or false results, they often work with the logical operators. For this reason, they will be discussed together here.

The relational and logical operators are shown in Table 2-4. Notice that in C++, not equal to is represented by != and equal to is represented by the double equal sign, ==. In C++, the outcome of a relational or logical expression produces a bool result. That is, the outcome of a relational or logical expression is either true or false.

NOTE: For older compilers, the outcome of a relational or logical expression will be an integer value of either 0 or 1. This difference is mostly academic, though, because C++ automatically converts **true** into 1 and **false** into 0, and vice versa as explained earlier.

The operands for a relational operator can be of nearly any type as long as they can be meaningfully compared. The operands to the logical operators must produce a true or false result. Since any nonzero value is true and zero is false, this means that the logical operators can be used with any expression that evaluates to a zero or nonzero result. Thus, any expression other than one that has a void result can be used.

Relational Operators	
Operator	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
Logical Operators	
Operator	Meaning
&&	AND
	OR
!	NOT

Table 2-4 The Relational and Logical Operators in C++

The logical operators are used to support the basic logical operations AND, OR, and NOT, according to the following truth table:

p	q	p AND q	p OR q	NOT p
False	False	False	False	True
False	True	False	True	True
True	True	True	True	False
True	False	False	True	False

Here is a program that demonstrates several of the relational and logical operators:

```
// Demonstrate the relational and logical operators.

#include <iostream>
using namespace std;

int main()
{
    int i, j;
    bool b1, b2;

    i = 10;
    j = 11;
    if(i < j) cout << "i < j\n";
    if(i <= j) cout << "i <= j\n";
}
```

```

    if(i != j) cout << "i != j\n";
    if(i == j) cout << "this won't execute\n";
    if(i >= j) cout << "this won't execute\n";
    if(i > j) cout << "this won't execute\n";

    b1 = true;
    b2 = false;
    if(b1 && b2) cout << "this won't execute\n";
    if(!(b1 && b2)) cout << "!(b1 && b2) is true\n";
    if(b1 || b2) cout << "b1 || b2 is true\n";

    return 0;
}

```

The output from the program is shown here:

```

i < j
i <= j
i != j
!(b1 && b2) is true
b1 || b2 is true

```

Both the relational and logical operators are lower in precedence than the arithmetic operators. This means that an expression like `10 > 1+12` is evaluated as if it were written `10 > (1+12)`. The result is, of course, false.

You can link any number of relational operations together using logical operators. For example, this expression joins three relational operations:

```
var > 15 || !(10 < count) && 3 <= item
```

The following table shows the relative precedence of the relational and logical operators:

Highest	!
	> >= < <=
	== !=
	&&
Lowest	

Project 2-2 Construct an XOR Logical Operation

C++ does not define a logical operator that performs an exclusive-OR operation, usually referred to as XOR. The XOR is a binary operation that yields true when one and only one operand is true. It has this truth table:

p	q	p XOR q
False	False	False
False	True	True
True	False	True
True	True	False

Some programmers have called the omission of the XOR a flaw. Others argue that the absence of the XOR logical operator is simply part of C++'s streamlined design, which avoids redundant features. They point out that it is easy to create an XOR logical operation using the three logical operators that C++ does provide.

In this project, you will construct an XOR operation using the `&&`, `||`, and `!` operators. You can decide for yourself if the omission of an XOR logical operator is a design flaw or an elegant feature!

Step by Step

1. Create a new file called `XOR.cpp`.
2. Assuming two Boolean values, `p` and `q`, a logical XOR is constructed like this:

```
(p || q) && !(p && q)
```

Let's go through this carefully. First, `p` is ORed with `q`. If this result is true, then at least one of the operands is true. Next, `p` is ANDed with `q`. This result is true if both operands are true. This result is then inverted using the NOT operator. Thus, the outcome of `!(p && q)` will be true when either `p`, `q`, or both are false. Finally, this result is ANDed with the result of `(p || q)`. Thus, the entire expression will be true when one but not both operands is true.

3. Here is the entire `XOR.cpp` program listing. It demonstrates the XOR operation for all four possible combinations of true/false values.

```
/*
   Project 2-2
   Create an XOR using the C++ logical operators.
*/

#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    bool p, q;

    p = true;
```

```

q = true;

cout << p << " XOR " << q << " is " <<
    ( (p || q) && !(p && q) ) << "\n";

p = false;
q = true;

cout << p << " XOR " << q << " is " <<
    ( (p || q) && !(p && q) ) << "\n";

p = true;
q = false;

cout << p << " XOR " << q << " is " <<
    ( (p || q) && !(p && q) ) << "\n";

p = false;
q = false;

cout << p << " XOR " << q << " is " <<
    ( (p || q) && !(p && q) ) << "\n";

return 0;
}

```

4. Compile and run the program. The following output is produced:

```

1 XOR 1 is 0
0 XOR 1 is 1
1 XOR 0 is 1
0 XOR 0 is 0

```

5. Notice the outer parentheses surrounding the XOR operation inside the cout statements. They are necessary because of the precedence of C++'s operators. The << operator is higher in precedence than the logical operators. To prove this, try removing the outer parentheses, and then attempt to compile the program. As you will see, an error will be reported.



Progress Check

1. What does the % operator do? To what types can it be applied?

2. How do you declare an int variable called index with an initial value of 10?
3. Of what type is the outcome of a relational or logical expression?

Answer Key:

1. The % is the modulus operator, which returns the remainder of an integer division. It can be applied to integer types.
2. `int index = 10;`
3. The result of a relational or logical expression is of type bool.

CRITICAL SKILL 2.6: The Assignment Operator

You have been using the assignment operator since Module 1. Now it is time to take a formal look at it. The assignment operator is the single equal sign, `=`. The assignment operator works in C++ much as it does in any other computer language. It has this general form:

var = expression;

Here, the value of the expression is given to var. The assignment operator does have one interesting attribute: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;  
x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables x, y, and z to 100 using a single statement. This works because the `=` is an operator that yields the value of the right-hand expression. Thus, the value of `z = 100` is 100, which is then assigned to y, which in turn is assigned to x. Using a “chain of assignment” is an easy way to set a group of variables to a common value.

CRITICAL SKILL 2.7: Compound Assignments

C++ provides special compound assignment operators that simplify the coding of certain assignment statements. Let’s begin with an example. The assignment statement shown here:

```
x = x + 10;
```

can be written using a compound assignment as

```
x += 10;
```

The operator pair `+=` tells the compiler to assign to x the value of x plus 10. Here is another example. The statement

```
x = x - 100;
```

is the same as

```
x -= 100;
```

Both statements assign to *x* the value of *x* minus 100. There are compound assignment operators for most of the binary operators (that is, those that require two operands). Thus, statements of the form

```
var = var op expression;
```

can be converted into this compound form:

```
var op = expression;
```

Because the compound assignment statements are shorter than their noncompound equivalents, the compound assignment operators are also sometimes called the shorthand assignment operators.

The compound assignment operators provide two benefits. First, they are more compact than their “longhand” equivalents. Second, they can result in more efficient executable code (because the operand is evaluated only once). For these reasons, you will often see the compound assignment operators used in professionally written C++ programs.

CRITICAL SKILL 2.8: Type Conversion in Assignments

When variables of one type are mixed with variables of another type, a type conversion will occur. In an assignment statement, the type conversion rule is easy: The value of the right side (expression side) of the assignment is converted to the type of the left side (target variable), as illustrated here:

```
int x;
char ch;
float f;

ch = x; /* line 1 */
x = f; /* line 2 */
f = ch; /* line 3 */
f = x; /* line 4 */
```

In line 1, the high-order bits of the integer variable *x* are lopped off, leaving *ch* with the lower 8 bits. If *x* were between -128 and 127 , *ch* and *x* would have identical values. Otherwise, the value of *ch* would reflect only the lower-order bits of *x*. In line 2, *x* will receive the nonfractional part of *f*. In line 3, *f* will convert the 8-bit integer value stored in *ch* to the same value in the floating-point format. This also happens in line 4, except that *f* will convert an integer value into floating-point format.

When converting from integers to characters and long integers to integers, the appropriate number of high-order bits will be removed. In many 32-bit environments, this means that 24 bits will be lost when going from an integer to a character, and 16 bits will be lost when going from an integer to a short integer. When converting from a floating-point type to an integer, the fractional part will be lost. If the target type is not large enough to store the result, then a garbage value will result.

A word of caution: Although C++ automatically converts any built-in type into another, the results won't always be what you want. Be careful when mixing types in an expression.

Expressions

Operators, variables, and literals are constituents of expressions. You might already know the general form of an expression from other programming experience or from algebra. However, a few aspects of expressions will be discussed now.

CRITICAL SKILL 2.9: Type Conversion in Expressions

When constants and variables of different types are mixed in an expression, they are converted to the same type. First, all char and short int values are automatically elevated to int. This process is called integral promotion. Next, all operands are converted “up” to the type of the largest operand, which is called type promotion. The promotion is done on an operation-by-operation basis. For example, if one operand is an int and the other a long int, then the int is promoted to long int. Or, if either operand is a double, the other operand is promoted to double. This means that conversions such as that from a char to a double are perfectly valid. Once a conversion has been applied, each pair of operands will be of the same type, and the result of each operation will be the same as the type of both operands.

Converting to and from bool

As mentioned earlier, values of type bool are automatically converted into the integers 0 or 1 when used in an integer expression. When an integer result is converted to type bool, 0 becomes false and nonzero becomes true. Although bool is a fairly recent addition to C++, the automatic conversions to and from integers mean that it has virtually no impact on older code. Furthermore, the automatic conversions allow C++ to maintain its original definition of true and false as zero and nonzero.

CRITICAL SKILL 2.10: Casts

It is possible to force an expression to be of a specific type by using a construct called a cast. A cast is an explicit type conversion. C++ defines five types of casts. Four allow detailed and sophisticated control over casting and are described later in this book after objects have been explained. However, there is one type of cast that you can use now. It is C++'s most general cast because it can transform any type into any other type. It was also the only type of cast that early versions of C++ supported. The general form of this cast is

(type) expression

where type is the target type into which you want to convert the expression. For example, if you wish to make sure the expression $x/2$ is evaluated to type float, you can write

```
(float) x / 2
```

Casts are considered operators. As an operator, a cast is unary and has the same precedence as any other unary operator.

There are times when a cast can be very useful. For example, you may wish to use an integer for loop control, but also perform computation on it that requires a fractional part, as in the program shown here:

```
// Demonstrate a cast.

#include <iostream>
using namespace std;

int main(){
    int i;

    for(i=1; i <= 10; ++i )
        cout << i << "/ 2 is: " << (float) i / 2 << '\n';

    return 0;
}
```

The cast to `float` causes a fractional component to be displayed.

Here is the output from this program:

```
1/ 2 is: 0.5
2/ 2 is: 1
3/ 2 is: 1.5
4/ 2 is: 2
5/ 2 is: 2.5
6/ 2 is: 3
7/ 2 is: 3.5
8/ 2 is: 4
9/ 2 is: 4.5
10/ 2 is: 5
```

Without the cast (`float`) in this example, only an integer division would be performed. The cast ensures that the fractional part of the answer will be displayed.

CRITICAL SKILL 2.11: Spacing and Parentheses

An expression in C++ can have tabs and spaces in it to make it more readable. For example, the following two expressions are the same, but the second is easier to read:

```
x=10/y*(127/x);
x = 10 / y * (127/x);
```

Parentheses increase the precedence of the operations contained within them, just like in algebra. Use of redundant or additional parentheses will not cause errors or slow down the execution of the expression. You are encouraged to use parentheses to make clear the exact order of evaluation, both for yourself and for others who may have to figure out your program later. For example, which of the following two expressions is easier to read?

```
x = y/3-34*temp+127;
x = (y/3) - (34*temp) + 127;
```

Project 2-3 Compute the Regular Payments on a Loan

In this project, you will create a program that computes the regular payments on a loan, such as a car loan. Given the principal, the length of time, number of payments per year, and the interest rate, the program will compute the payment. Since this is a financial calculation, you will need to use floating-point data types for the computations. Since double is the most commonly used floating-point type, we will use it in this project. This project also demonstrates another C++ library function: `pow()`.

To compute the payments, you will use the following formula:

$$\text{Payment} = \frac{\text{IntRate} * (\text{Principal} / \text{PayPerYear})}{1 - ((\text{IntRate} / \text{PayPerYear}) + 1)^{-\text{PayPerYear} * \text{NumYears}}}$$

where `IntRate` specifies the interest rate, `Principal` contains the starting balance, `PayPerYear` specifies the number of payments per year, and `NumYears` specifies the length of the loan in years.

Notice that in the denominator of the formula, you must raise one value to the power of another. To do this, you will use `pow()`. Here is how you will call it:

```
result = pow(base, exp);
```

`pow()` returns the value of `base` raised to the `exp` power. The arguments to `pow()` are double values, and `pow()` returns a value of type double.

Step by Step

1. Create a new file called `RegPay.cpp`.
2. Here are the variables that will be used by the program:

```
double Principal; // original principal
double IntRate; // interest rate, such as 0.075
double PayPerYear; // number of payments per year
double NumYears; // number of years
double Payment; // the regular payment
double numer, denom; // temporary work variables
double b, e; // base and exponent for call to pow()
```

Notice how each variable declaration is followed by a comment that describes its use. This helps anyone reading your program understand the purpose of each variable. Although we won't include

such detailed comments for most of the short programs in this book, it is a good practice to follow as your programs become longer and more complicated.

3. Add the following lines of code, which input the loan information:

```
cout << "Enter principal: ";
cin >> Principal;

cout << "Enter interest rate (i.e., 0.075): ";
cin >> IntRate;

cout << "Enter number of payments per year: ";
cin >> PayPerYear;

cout << "Enter number of years: ";
cin >> NumYears;
```

4. Add the lines that perform the financial calculation:

```
numer = IntRate * Principal / PayPerYear;

e = -(PayPerYear * NumYears);
b = (IntRate / PayPerYear) + 1;

denom = 1 - pow(b, e);

Payment = numer / denom;
```

5. Finish the program by outputting the regular payment, as shown here:

```
cout << "Payment is " << Payment;
```

6. Here is the entire RegPay.cpp program listing:

```
/*
    Project 2-3
    Compute the regular payments for a loan.
    Call this file RegPay.cpp
*/

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

int main() {
    double Principal; // original principal
    double IntRate; // interest rate, such as 0.075
    double PayPerYear; // number of payments per year
    double NumYears; // number of years
```

```

double Payment; // the regular payment
double numer, denom; // temporary work variables
double b, e; // base and exponent for call to pow()

cout << "Enter principal: ";
cin >> Principal;

cout << "Enter interest rate (i.e., 0.075): ";
cin >> IntRate;

cout << "Enter number of payments per year: ";
cin >> PayPerYear;

cout << "Enter number of years: ";
cin >> NumYears;

numer = IntRate * Principal / PayPerYear;

e = -(PayPerYear * NumYears);
b = (IntRate / PayPerYear) + 1;

denom = 1 - pow(b, e);

Payment = numer / denom;

cout << "Payment is " << Payment;

return 0;
}

```

Here is a sample run:

```

Enter principal: 10000
Enter interest rate (i.e., 0.075): 0.075
Enter number of payments per year: 12
Enter number of years: 5
Payment is 200.379

```

7. On your own, have the program display the total amount of interest paid over the life of the loan.

Module 2 Mastery Check

1. What type of integers are supported by C++?
2. By default, what type is 12.2?

3. What values can a bool variable have?
4. What is the long integer data type?
5. What escape sequence produces a tab? What escape sequence rings the bell?
6. A string is surrounded by double quotes. True or false?
7. What are the hexadecimal digits?
8. Show the general form for initializing a variable when it is declared.
9. What does the % do? Can it be used on floating-point values?
10. Explain the difference between the prefix and postfix forms of the increment operator.
11. Which of the following are logical operators in C++?
 - a. &&
 - b. ##
 - c. ||
 - d. \$\$
 - e. !
12. How can
 - `x = x + 12;`be rewritten?
13. What is a cast?
14. Write a program that finds all of the prime numbers between 1 and 100.