

Module 7

More Data Types and Operators

Table of Contents

CRITICAL SKILL 7.1: The const and volatile Qualifiers	2
CRITICAL SKILL 7.2: extern.....	5
CRITICAL SKILL 7.3: static Variables	6
CRITICAL SKILL 7.4: register Variables.....	10
CRITICAL SKILL 7.5: Enumerations	12
CRITICAL SKILL 7.6 typedef.....	16
CRITICAL SKILL 7.8: The Shift Operators	22
CRITICAL SKILL 7.9: The ? Operator	29
CRITICAL SKILL 7.10: The Comma Operator.....	31
CRITICAL SKILL 7.11: Compound Assignment	33
CRITICAL SKILL 7.12: Using sizeof.....	33

This module returns to the topics of data types and operators. In addition to the data types that you have been using so far, C++ supports several others. Some of these consist of modifiers added to the types you already know about. Other data types include enumerations and typedefs. C++ also provides several additional operators that greatly expand the range of programming tasks to which C++ can be applied. These operators include the bitwise, shift, ?, and sizeof operators.

CRITICAL SKILL 7.1: The const and volatile Qualifiers

C++ has two type qualifiers that affect the ways in which variables can be accessed or modified. These modifiers are `const` and `volatile`. Formally called the cv-qualifiers, they precede the base type when a variable is declared.

`const`

A variable declared with the `const` modifier cannot have its value changed during the execution of your program. Thus, a `const` “variable” isn’t really variable! You can give a variable declared as `const` an initial value, however. For example,

```
const int max_users = 9;
```

creates an `int` variable called `max_users` that contains the value 9. This variable can be used in expressions like any other variable, but its value cannot be modified by your program.

A common use of `const` is to create a named constant. Often programs require the same value for many different purposes. For example, a program might have several different arrays that are all the same size. In this case, you can specify the size of the arrays using a `const` variable. The advantage to this approach is that if the size needs to be changed at a later date, you need change only the value of the `const` variable and then recompile the program. You don’t need to change the size in each array declaration. This approach avoids errors and is easier, too. The following example illustrates this application of `const`:

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

const int num_employees = 100; ← This creates a named constant called
                                num_employees that has the value 100.

int main()
{
    int empNums[num_employees];
    double salary[num_employees]; ← num_employees is used here to
                                    specify the size of the arrays.

    char *names[num_employees];

    // ...
}
```

In this example, if you need to use a new size for the arrays, you need change only the declaration of `num_employees` and then recompile the program. All three arrays will then automatically be resized.

Another important use of `const` is to prevent an object from being modified through a pointer. For example, you might want to prevent a function from changing the value of the object pointed to by a parameter. To do this, declare a pointer parameter as `const`. This prevents the object pointed to by the parameter from being modified by a function. That is, when a pointer parameter is preceded by `const`,

no statement in the function can modify the variable pointed to by that parameter. For example, the `negate()` function in the following program returns the negation of the value pointed to by its parameter. The use of `const` in the parameter declaration prevents the code inside the function from modifying the value pointed to by the parameter.

```
// Use a const pointer parameter.

#include <iostream>
using namespace std;

int negate(const int *val);

int main()
{
    int result;
    int v = 10;

    result = negate(&v);

    cout << v << " negated is " << result;
    cout << "\n";

    return 0;
}

int negate(const int *val) ← This specifies val as
{                               a const pointer.
    return - *val;
}
```

Since `val` is declared as being a `const` pointer, the function can make no changes to the value pointed to by `val`. Since `negate()` does not attempt to change `val`, the program compiles and runs correctly. However, if `negate()` were written as shown in the next example, a compile-time error would result.

```
// This won't work!
int negate(const int *val)
{
    *val = - *val; // Error, can't change
    return *val;
}
```

In this case, the program attempts to alter the value of the variable pointed to by `val`, which is prevented because `val` is declared as `const`.

The `const` modifier can also be used on reference parameters to prevent a function from modifying the object referenced by a parameter. For example, the following version of `negate()` is incorrect because it attempts to modify the variable referred to by `val`:

```
// This won't work either!
int negate(const int &val)
{
    val = -val; // Error, can't change
    return val;
}
```

volatile

The volatile modifier tells the compiler that a variable's value may be changed in ways not explicitly specified by the program. For example, the address of a global variable might be passed to an interrupt-driven clock routine that updates the variable with each tick of the clock. In this situation, the contents of the variable are altered without the use of any explicit assignment statement in the program. The reason the external alteration of a variable may be important is that a C++ compiler is permitted to automatically optimize certain expressions, on the assumption that the content of a variable is unchanged if it does not occur on the left side of an assignment statement. However, if factors beyond program control change the value of a variable, then problems can occur. To prevent such problems, you must declare such variables volatile, as shown here:

```
volatile int current_users;
```

Because it is declared as volatile, the value of current_users will be obtained each time it is referenced.



1. Can the value of a const variable be changed by the program?
2. If a variable has its value changed by events outside the program, how should that variable be declared?

Storage Class Specifiers

There are five storage class specifiers supported by C++. They are

```
auto
extern
register
static
mutable
```

These are used to tell the compiler how a variable should be stored. The storage specifier precedes the rest of the variable declaration. The mutable specifier applies only to class objects, which are discussed later in this book.

Each of the other specifiers is examined here.

auto

The `auto` specifier declares a local variable. However, it is rarely (if ever) used, because local variables are `auto` by default. It is extremely rare to see this keyword used in a program. It is a holdover from the C language.

CRITICAL SKILL 7.2: `extern`

All the programs that you have worked with so far have been quite small. However, in reality, computer programs tend to be much larger. As a program file grows, the compilation time eventually becomes long enough to be annoying. When this happens, you should break your program into two or more separate files. Then, changes to one file will not require that the entire program be recompiled. Instead, you can simply recompile the file that changed, and link the existing object code for the other files. The multiple file approach can yield a substantial time savings with large projects. The `extern` keyword helps support this approach. Let's see how.

In programs that consist of two or more files, each file must know the names and types of the global variables used by the program. However, you cannot simply declare copies of the global variables in each file. The reason is that your program can only have one copy of each global variable. Therefore, if you try to declare the global variables needed by your program in each file, an error will occur when the linker tries to link the files. It will find the duplicated global variables and will not link your program. The solution to this dilemma is to declare all of the global variables in one file and use `extern` declarations in the others, as shown in Figure 7-1.

File One declares `x`, `y`, and `ch`. In File Two, the global variable list is copied from File One, and the `extern` specifier is added to the declarations. The `extern` specifier allows a variable to be made known to a module, but does not actually create that variable. In other words, `extern` lets the compiler know what the types and names are for these global variables without actually creating storage for them again. When the linker links the two modules together, all references to the external variables are resolved.

While we haven't yet worried about the distinction between the declaration and the definition of a variable, it is important here. A declaration declares the name and type of a variable. A definition causes storage to be allocated for the variable. In most cases, variable declarations are also definitions. However, by preceding a variable name with the `extern` specifier, you can declare a variable without defining it.

File One

```
int x, y;
char ch;

int main()
{
    // ...
}

void func1()
{
    x = 123;
}
```

File Two

```
extern int x, y;
extern char ch;

void func22()
{
    x = y/10;
}

void func23()
{
    y = 10;
}
```

A variation on `extern` provides a linkage specification, which is an instruction to the compiler about how a function is to be handled by the linker. By default, functions are linked as C++ functions, but a linkage specification lets you link a function for a different type of language. The general form of a linkage specifier is shown here:

`extern "language" function-prototype`

where `language` denotes the desired language. For example, this specifies that `myCfunc()` will have C linkage:

```
extern "C" void myCfunc();
```

All C++ compilers support both C and C++ linkage. Some may also allow linkage specifiers for FORTRAN, Pascal, or BASIC. (You will need to check the documentation for your compiler.) You can specify more than one function at a time using this form of the linkage specification:

```
extern "language" { prototypes
}
```

For most programming tasks, you won't need to use a linkage specification.

CRITICAL SKILL 7.3: static Variables

Variables of type `static` are permanent variables within their own function or file. They differ from global variables because they are not known outside their function or file. Because `static` affects local variables differently than it does global ones, local and global variables will be examined separately.

static Local Variables

When the `static` modifier is applied to a local variable, permanent storage for the variable is allocated in much the same way that it is for a global variable. This allows a static variable to maintain its value between function calls. (That is, its value is not lost when the function returns, unlike the value of a

normal local variable.) The key difference between a static local variable and a global variable is that the static local variable is known only to the block in which it is declared.

To declare a static variable, precede its type with the word `static`. For example, this statement declares `count` as a static variable:

```
static int count;
```

A static variable may be given an initial value. For example, this statement gives `count` an initial value of 200:

```
static int count = 200;
```

Local static variables are initialized only once, when program execution begins, not each time the block in which they are declared is entered.

The static local variable is important to functions that must preserve a value between calls. If static variables were not available, then global variables would have to be used—opening the door to possible side effects.

To see an example of a static variable, try this program. It keeps a running average of the numbers entered by the user.

```

// Compute a running average of numbers entered by the user.

#include <iostream>
using namespace std;

int running_avg(int i);

int main()
{
    int num;

    do {
        cout << "Enter numbers (-1 to quit): ";
        cin >> num;
        if(num != -1)
            cout << "Running average is: " << running_avg(num);
        cout << '\n';
    } while(num > -1);

    return 0;
}

int running_avg(int i)
{
    static int sum = 0, count = 0; ← Because sum and count are static,
    sum = sum + i;                  they will retain their values
    count++;                       between calls to running_avg().

    return sum / count;
}

```

Here, the local variables `sum` and `count` are both declared as static and initialized to 0. Remember, for static variables the initialization only occurs once—not each time the function is entered. The program uses `running_avg()` to compute and report the current average of the numbers entered by the user. Because both `sum` and `count` are static, they will maintain their values between calls, causing the program to work properly. To prove to yourself that the static modifier is necessary, try removing it and running the program. As you can see, the program no longer works correctly, because the running total is lost each time `running_avg()` returns.

static Global Variables

When the static specifier is applied to a global variable, it tells the compiler to create a global variable that is known only to the file in which the static global variable is declared. This means that even though the variable is global, other functions in other files have no knowledge of it and cannot alter its contents. Thus, it is not subject to side effects. Therefore, for the few situations where a local static variable cannot do the job, you can create a small file that contains only the functions that need the global static variable, separately compile that file, and use it without fear of side effects. For an example of global

static variables, we will rework the running average program from the preceding section. In this version, the program is broken into the two files shown here. The function `reset()`, which resets the average, is also added.

```
// ----- First File -----

#include <iostream>
using namespace std;

int running_avg(int i);
void reset();

int main()
{
    int num;

    do {
        cout << "Enter numbers (-1 to quit, -2 to reset): ";
        cin >> num;
        if(num == -2) {
            reset();
            continue;
        }
        if(num != -1)
            cout << "Running average is: " << running_avg(num);
        cout << '\n';
    } while(num != -1);

    return 0;
}

// ----- Second File -----

static int sum = 0, count = 0; ← These are known only in the file
                                in which they are declared.

int running_avg(int i)
{
    sum = sum + i;

    count++;

    return sum / count;
}

void reset()
{
    sum = 0;
    count = 0;
}
```

Here, `sum` and `count` are now global static variables that are restricted to the second file. Thus, they can be accessed by both `running_avg()` and `reset()` in the second file, but not elsewhere. This allows them

to be reset by a call to `reset()` so that a second set of numbers can be averaged. (When you run the program, you can reset the average by entering `-2`.) However, no functions outside the second file can access those variables. For example, if you try to access either `sum` or `count` from the first file, you will receive an error message.

To review: The name of a local static variable is known only to the function or block of code in which it is declared, and the name of a global static variable is known only to the file in which it resides. In essence, the static modifier allows variables to exist that are known to the scopes that need them, thereby controlling and limiting the possibility of side effects. Variables of type static enable you, the programmer, to hide portions of your program from other portions. This can be a tremendous advantage when you are trying to manage a very large and complex program.

Ask the Expert

Q: I have heard that some C++ programmers do not use static global variables. Is this true?

A: Although static global variables are still valid and widely used in C++ code, the C++ Standard discourages their use. Instead, it recommends another method of controlling access to global variables that involves the use of namespaces, which are described later in this book. However, static global variables are widely used by C programmers because C does not support namespaces. For this reason, you will continue to see static global variables for a long time to come.

CRITICAL SKILL 7.4: register Variables

Perhaps the most frequently used storage class specifier is `register`. The `register` modifier tells the compiler to store a variable in such a way that it can be accessed as quickly as possible. Typically, this means storing the variable either in a register of the CPU or in cache memory. As you probably know, accessing the registers of the CPU (or cache memory) is fundamentally faster than accessing the main memory of the computer. Thus, a variable stored in a register will be accessed much more quickly than if that variable had been stored in RAM. Because the speed by which variables can be accessed has a profound effect on the overall speed of your programs, the careful use of `register` is an important programming technique.

Technically, `register` is only a request to the compiler, which the compiler is free to ignore. The reason for this is easy to understand: there are a finite number of registers (or fast-access memory), and these may differ from environment to environment. Thus, if the compiler runs out of fast-access memory, it simply stores the variable normally. Generally, this causes no harm, but of course the `register` advantage is lost. You can usually count on at least two variables being optimized for speed. Since only a limited number of variables can actually be granted the fastest access, it is important to choose carefully those to which you apply the `register` modifier. (Only by choosing the right variables can you gain the greatest increase in performance.) In general, the more often a variable is accessed, the more benefit there will

be to optimizing it as a register variable. For this reason, variables that control or are accessed within loops are good candidates for the register specifier.

Here is an example that uses register variables to improve the performance of the summation() function, which computes the summation of the values in an array. This example assumes that only two variables will actually be optimized for speed.

```
// Demonstrate register.

#include <iostream>
using namespace std;

int summation(int nums[], int n);

int main()
{
    int vals[] = { 1, 2, 3, 4, 5 };
    int result;

    result = summation(vals, 5);

    cout << "Summation is " << result << "\n";

    return 0;
}

// Return summation of an array of ints.
int summation(int nums[], int n)
{
    register int i;
    register int sum = 0;
    for(i = 0; i < n; i++)
        sum = sum + nums[i];

    return sum;
}
```

These variables are optimized for speed.

Here, the variable `i`, which controls the for loop, and `sum`, which is accessed inside the loop, are specified as register. Since they are both used within the loop, both benefit from being optimized for fast access. This example assumed that only two variables could actually be optimized for speed, so `n` and `nums` were not specified as register because they are not accessed as often as `i` and `sum` within the loop. However, in environments in which more than two variables can be optimized, they too could be specified as register to further improve performance.



1. A static local variable _____ its value between function calls.

2. You use `extern` to declare a variable without defining that variable. True or false?
3. What specifier requests that the compiler optimize a variable for speed?

Ask the Expert

Q: When I tried adding the register specifier to a program, I saw no change in performance. Why not?

A: Because of advances in compiler technology, most compilers today will automatically optimize your code. Thus, in many cases, adding the register specifier to a declaration might not result in any performance increase because that variable is already optimized. However, in some cases, using register is still beneficial because it lets you tell the compiler which variables you think are the most important to optimize. This can be valuable for functions that use a large number of variables, all of which cannot be optimized. Thus, register still fulfills an important role despite advances in compiler design.

CRITICAL SKILL 7.5: Enumerations

In C++, you can define a list of named integer constants. Such a list is called an enumeration. These constants can then be used anywhere that an integer can. Enumerations are defined using the keyword `enum` and have this general format:

```
enum type-name { value-list } variable-list;
```

Here, `type-name` is the type name of the enumeration. The `value-list` is a comma-separated list of names that represent the values of the enumeration. The `variable-list` is optional because variables may be declared later using the enumeration type name.

The following fragment defines an enumeration called `transport` and two variables of type `transport` called `t1` and `t2`:

```
enum transport { car, truck, airplane, train, boat } t1, t2;
```

Once you have defined an enumeration, you can declare additional variables of its type using its name. For example, this statement declares one variable, called `how`, of enumeration `transport`:

```
transport how;
```

The statement can also be written like this:

```
enum transport how;
```

However, the use of `enum` here is redundant. In C (which also supports enumerations), this second form was required, so you may see it used in some programs.

Assuming the preceding declarations, the following gives how the value `airplane`:

```
how = airplane;
```

The key point to understand about an enumeration is that each of the symbols stands for an integer value. As such, they can be used in any integer expression. Unless initialized otherwise, the value of the first enumeration symbol is 0, the value of the second symbol is 1, and so forth. Therefore,

```
cout << car << ' ' << train;
```

displays 0 3.

Although enumerated constants are automatically converted to integers, integers are not automatically converted into enumerated constants. For example, the following statement is incorrect:

```
how = 1; // Error
```

This statement causes a compile-time error because there is no automatic conversion from integer to transport. You can fix the preceding statement by using a cast, as shown here:

```
how = (transport) 1; // now OK, but probably poor style
```

This causes how to contain the value truck, because it is the transport constant associated with the value 1. As the comment suggests, although this statement is now correct, it would be considered to be poor style except in unusual circumstances.

It is possible to specify the value of one or more of the enumerated constants by using an initializer. This is done by

car	0
truck	1
airplane	10
train	11
boat	12

following the symbol with an equal sign and an integer value. When an initializer is used, each symbol that appears after it is assigned a value 1 greater than the previous initialization value. For example, the following statement assigns the value of 10 to airplane:

```
enum transport { car, truck, airplane = 10, train, boat };
```

Now, the values of these symbols are as follows:

One common, but erroneous, assumption sometimes made about enumerations is that the symbols can be input and output as a string. This is not the case. For example, the following code fragment will not perform as desired:

```
// This will not print "train" on the screen. how = train; cout << how;
```

Remember, the symbol `train` is simply a name for an integer; it is not a string. Thus, the preceding code will display the numeric value of `train`, not the string “train”. Actually, to create code that inputs and outputs enumeration symbols as strings is quite tedious. For example, the following code is needed in order to display, in words, the kind of transportation that `how` contains:

```
switch(how) {
    case car:
        cout << "Automobile";
        break;
    case truck:
        cout << "Truck";
        break;
    case airplane:
        cout << "Airplane";
        break;
    case train:
        cout << "Train";
        break;
    case boat:
        cout << "Boat";
        break;
}
```

Sometimes it is possible to declare an array of strings and use the enumeration value as an index in order to translate the value into its corresponding string. For example, the following program prints the names of three types of transportation:

```
// Demonstrate an enumeration.

#include <iostream>
using namespace std;

enum transport { car, truck, airplane, train, boat };

char name[][20] = {
    "Automobile",
    "Truck",
    "Airplane",
    "Train",
    "Boat"
};

int main()
{
    transport how;

    how = car;
    cout << name[how] << '\n';

    how = airplane;
    cout << name[how] << '\n';

    how = train;
    cout << name[how] << '\n';

    return 0;
}
```

Use an enumeration value to index an array.

The output is shown here:

```
Automobile
Airplane
Train
```

The approach used by this program to convert an enumeration value into a string can be applied to any type of enumeration as long as that enumeration does not contain initializers. To properly index the array of strings, the enumerated constants must begin at zero and be in strictly ascending order, each precisely one greater than the previous. Given the fact that enumeration values must be converted manually to their human-readable string values, they find their greatest use in routines that do not make such conversions. It is common to see an enumeration used to define a compiler's symbol table, for example.

CRITICAL SKILL 7.6 typedef

C++ allows you to define new data type names with the typedef keyword. When you use typedef, you are not actually creating a new data type, but rather defining a new name for an existing type. This process can help make machine-dependent programs more portable; only the typedef statements have to be changed. It also helps you self-document your code by allowing descriptive names for the standard data types. The general form of the typedef statement is

```
typedef type name;
```

where type is any valid data type, and name is the new name for this type. The new name you define is in addition to, not a replacement for, the existing type name.

For example, you could create a new name for float using

```
typedef float balance;
```

This statement would tell the compiler to recognize balance as another name for float. Next, you could create a float variable using balance:

```
balance over_due;
```

Here, over_due is a floating-point variable of type balance, which is another name for float.



1. An enumeration is a list of named _____ constants.
2. Enumerated values begin with what integer value?
3. Show how to declare BigInt to be another name for long int.

CRITICAL SKILL 7.7: Bitwise Operators

Since C++ is designed to allow full access to the computer's hardware, it gives you the ability to operate directly upon the bits within a byte or word. Toward this end, C++ contains the bitwise operators.

Bitwise operations refer to the testing, setting, or shifting of the actual bits in a byte or word, which correspond to C++'s character and integer types. Bitwise operations cannot be used on bool, float, double, long double, void, or other more complex data types. Bitwise operations are important in a wide variety of systems-level programming, especially when status information from a device must be interrogated or constructed. Table 7-1 lists the bitwise operators. Each operator is examined in turn.

Operator	Action
&	AND
	OR
^	Exclusive OR (XOR)
~	One's complement (NOT)
>>	Shift right
<<	Shift left

Table 7-1 The Bitwise Operators

AND, OR, XOR, and NOT The bitwise AND, OR, and one's complement (NOT) are governed by the same truth table as their logical equivalents, except that they work on a bit-by-bit level. The exclusive OR (XOR) operates according to the following truth table:

p	q	p ^ q
0	0	0
1	0	1
1	1	0
0	1	1

As the table indicates, the outcome of an XOR is true only if exactly one of the operands is true; it is false otherwise.

In terms of its most common usage, you can think of the bitwise AND as a way to turn bits off. That is, any bit that is 0 in either operand will cause the corresponding bit in the outcome to be set to 0. For example:

```

  1 1 0 1 0 0 1 1
& 1 0 1 0 1 0 1 0
-----
  1 0 0 0 0 0 1 0

```

The following program demonstrates the & by turning any lowercase letter into uppercase by resetting the sixth bit to 0. As the ASCII character set is defined, the lowercase letters are the same as the uppercase ones except that the lowercase ones are greater in value by exactly 32. Therefore, to transform a lowercase letter to uppercase, just turn off the sixth bit, as this program illustrates:

```
// Uppercase letters using bitwise AND.

#include <iostream>
using namespace std;

int main()
{
    char ch;

    for(int i = 0 ; i < 10; i++) {
        ch = 'a' + i;
        cout << ch;

        // This statement turns off the 6th bit.
        ch = ch & 223; // ch is now uppercase ← Use the bitwise AND
                                                    to turn off bit 6.

        cout << ch << " ";
    }

    cout << "\n";

    return 0;
}
```

The output from this program is shown here:

```
aA bB cC dD eE fF gG hH iI jJ
```

The value 223 used in the AND statement is the decimal representation of 1101 1111. Thus, the AND operation leaves all bits in `ch` unchanged except for the sixth one, which is set to zero.

The AND operator is also useful when you want to determine whether a bit is on or off. For example, this statement checks to see if bit 4 in `status` is set:

```
if(status & 8) cout << "bit 4 is on";
```

The reason 8 is used is that in binary it is represented as 0000 1000. That is, the number 8 translated into binary has only the fourth bit set. Therefore, the if statement can succeed only when bit 4 of `status` is also on. An interesting use of this feature is the `show_binary()` function, shown next. It displays, in binary format, the bit pattern of its argument. You will use `show_binary()` later in this module to examine the effects of other bitwise operations.

```
// Display the bits within a byte.
void show_binary(unsigned int u)
{
    int t;

    for(t=128; t > 0; t = t/2)
        if(u & t) cout << "1 ";
        else cout << "0 ";

    cout << "\n";
}
```

The `show_binary()` function works by successively testing each bit in the low-order byte of `u`, using the bitwise AND, to determine if it is on or off. If the bit is on, the digit 1 is displayed; otherwise, 0 is displayed.

The bitwise OR, as the reverse of AND, can be used to turn bits on. Any bit that is set to 1 in either operand will cause the corresponding bit in the variable to be set to 1. For example,

```

1 1 0 1 0 0 1 1
1 0 1 0 1 0 1 0
-----
1 1 1 1 1 0 1 1
```

You can make use of the OR to change the uppercasing program used earlier into a lowercasing program, as shown here:

```
// Lowercase letters using bitwise OR.

#include <iostream>
using namespace std;

int main()
{
    char ch;

    for(int i = 0 ; i < 10; i++) {
        ch = 'A' + i;
        cout << ch;


        // This statement turns on the 6th bit.
        ch = ch | 32; // ch is now lowercase

        cout << ch << " ";
    }

    cout << "\n";

    return 0;
}
```

Use the bitwise OR to turn on bit 6.



The output is shown here:

Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj

When the sixth bit is set, each uppercase letter is transformed into its lowercase equivalent.

An exclusive OR, usually abbreviated XOR, will set a bit on only if the bits being compared are different, as illustrated here:

```
  0 1 1 1 1 1 1 1
^ 1 0 1 1 1 0 0 1
-----
  1 1 0 0 0 1 1 0
```

The XOR operator has an interesting property that makes it a simple way to encode a message. When some value X is XORed with another value Y, and then when that result is XORed with Y again, X is produced. That is, given the sequence

```
R1 = X ^ Y;
R2 = R1 ^ Y;
```

then R2 is the same value as X. Thus, the outcome of a sequence of two XORs using the same value produces the original value. You can use this principle to create a simple cipher program in which some integer is the key that is used to both encode and decode a message by XORing the characters in that message. To encode, the XOR operation is applied the first time, yielding the ciphertext. To decode, the XOR is applied a second time, yielding the plaintext. Here is a simple example that uses this approach to encode and decode a short message:

```
// Use XOR to encode and decode a message.

#include <iostream>
using namespace std;

int main()
{
    char msg[] = "This is a test";
```

```

char key = 88;

cout << "Original message: " << msg << "\n";

for(int i = 0 ; i < strlen(msg); i++)
    msg[i] = msg[i] ^ key; ← This constructs the encoded string.

cout << "Encoded message: " << msg << "\n";

for(int i = 0 ; i < strlen(msg); i++)
    msg[i] = msg[i] ^ key; ← This constructs a decoded string.

cout << "Decoded message: " << msg << "\n";

return 0;
}

```

Here is the output:

```

Original message: This is a test
Encoded message: 01+x1+x9x,=+,
Decoded message: This is a test

```

As the output proves, the result of two XORs using the same key produces the decoded message.

The unary one's complement (NOT) operator reverses the state of all the bits of the operand. For example, if some integer called A has the bit pattern 1001 0110, then $\sim A$ produces a result with the bit pattern 0110 1001. The following program demonstrates the NOT operator by displaying a number and its complement in binary, using the `show_binary()` function developed earlier:

```

#include <iostream>
using namespace std;

void show_binary(unsigned int u);

int main()
{
    unsigned u;

    cout << "Enter a number between 0 and 255: ";
    cin >> u;

    cout << "Here's the number in binary: ";
    show_binary(u);
}

```

```

    cout << "Here's the complement of the number: ";
    show_binary(~u);

    return 0;
}

// Display the bits within a byte.
void show_binary(unsigned int u)
{
    int t;

    for(t=128; t>0; t = t/2)
        if(u & t) cout << "1 ";
        else cout << "0 ";

    cout << "\n";
}

```

Here is a sample run produced by the program:

```

Enter a number between 0 and 255: 99
Here's the number in binary: 0 1 1 0 0 0 1 1
Here's the complement of the number: 1 0 0 1 1 1 0 0

```

In general, `&`, `|`, `^`, and `~` apply their operations directly to each bit in a value individually.

For this reason, bitwise operations are not usually used in conditional statements the way the relational and logical operators are. For example, if `x` equals 7, then `x && 8` evaluates to true, whereas `x & 8` evaluates to false.

CRITICAL SKILL 7.8: The Shift Operators

The shift operators, `>>` and `<<`, move all bits in a variable to the right or left as specified. The general form of the right-shift operator is

variable `>>` num-bits

and the left-shift operator is

variable `<<` num-bits

The value of num-bits determines how many bit places the bits are shifted. Each left-shift causes all bits within the specified variable to be shifted left one position and a zero bit to be brought in on the right. Each right-shift shifts all bits to the right one position and brings in a zero on the left. However, if the variable is a signed integer containing a negative value, then each right-shift brings in a 1 on the left, which preserves the sign bit. Remember, a shift is not a rotation. That is, the bits shifted off of one end do not come back around to the other.

The shift operators work only with integral types, such as int, char, long int, or short int. They cannot be applied to floating-point values, for example.

Bit shift operations can be very useful for decoding input from external devices such as D/A converters and for reading status information. The bitwise shift operators can also be used to perform very fast multiplication and division of integers. A shift left will effectively multiply a number by 2, and a shift right will divide it by 2.

The following program illustrates the effects of the shift operators:

```
// Example of bitshifting.

#include <iostream>
using namespace std;

void show_binary(unsigned int u);

int main()
{
    int i=1, t;

    // shift left
    for(t=0; t < 8; t++) {
        show_binary(i);
        i = i << 1; ← Left-shift i one position.
    }

    cout << "\n";

    // shift right
    for(t=0; t < 8; t++) {
        i = i >> 1; ← Right-shift i one position.
        show_binary(i);
    }

    return 0;
}

// Display the bits within a byte.
void show_binary(unsigned int u)
{
    int t;
```

```

for(t=128; t>0; t=t/2)
    if(u & t) cout << "1 ";
    else cout << "0 ";

    cout << "\n";
}

```

This program produces the following output:

```

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

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

```

Progress Check

1. What are the bitwise operators for AND, OR, NOT, and XOR?
2. A bitwise operator works on a bit-by-bit basis. True or false?
3. Given an integer called x, show how to left-shift x two places.

Project 7-1 Create Bitwise Rotation Functions

Although C++ provides two shift operators, it does not define a rotate operator.

A rotate is similar to a shift except that the bit shifted off one end is inserted onto the other end. Thus, bits are not lost, just moved. There are both left and right rotations. For example, 1010 0000 rotated left one place is 0100 0001. The same value rotated right one place is 0101 0000. In each case, the bit shifted out is inserted onto the other end. Although the lack of rotation operators may seem to be a

flaw in C++'s otherwise exemplary complement of bitwise operators, it really isn't, because you can easily create a left- and right-rotate by using the other bitwise operators.

This project creates two functions: `rrotate()` and `lrotate()`, which rotate a byte in the right or left direction. Each function takes two parameters. The first is the value to be rotated.

The second is the number of places to rotate. Each function returns the result. This project involves several bit manipulations and shows the bitwise operators in action.

Step by Step

1. Create a file called `rotate.cpp`.

2. Add the `lrotate()` function shown here. It performs a left-rotate.

```
// Left-rotate a byte n places.
unsigned char lrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;

    for(int i=0; i < n; i++) {
        t = t << 1;

        /* If a bit shifts out, it will be in bit 8
           of the integer t. If this is the case,
           put that bit on the right side. */
        if(t & 256)
            t = t | 1; // put a 1 on the right end
    }

    return t; // return the lower 8 bits.
}
```

Here is how `lrotate()` works. The function is passed the value to rotate in `val`, and the number of places to rotate is passed in `n`. The function assigns `val` to `t`, which is an unsigned int. Transferring the value to an unsigned int is necessary because it allows bits shifted off the left side to be recovered. Here's why. Because an unsigned int is larger than a byte, when a bit is shifted off the left side of a byte value, it simply moves to bit 8 of the integer value. The value of this bit can then be copied into bit 0 of the byte value, thus performing a rotation.

The actual rotation is performed as follows: A loop is established that performs the required number of rotations, one at a time. Inside the loop, the value of `t` is left-shifted one place. This causes a 0 to be

brought in on the right. However, if the value of bit 8 of the result (which is the bit shifted out of the byte value) is a 1, then bit 0 is set to 1. Otherwise, bit 0 remains 0.

The eighth bit is tested using the statement

```
if(t & 256)
```

The value 256 is the decimal value in which only bit 8 is set. Thus, `t & 256` will be true only when `t` has the value 1 in bit 8.

After the rotation has been completed, `t` is returned. Since `lrotate()` is declared to return an unsigned char value, only the lower 8 bits of `t` are returned.

3. Add the `rrotate()` function shown next. It performs a right rotate.

```
// Right-rotate a byte n places.
unsigned char rrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;

    // First, move the value 8 bits higher.
    t = t << 8;

    for(int i=0; i < n; i++) {
        t = t >> 1;

        /* If a bit shifts out, it will be in bit 7
           of the integer t. If this is the case,
           put that bit on the left side. */
        if(t & 128)
            t = t | 32768; // put a 1 on left end
    }

    /* Finally, move the result back to the
       lower 8 bits of t. */

    t = t >> 8;

    return t;
}
```

The right-rotate is slightly more complicated than the left-rotate because the value passed in `val` must be shifted into the second byte of `t` so that bits being shifted off the right side can be caught. Once the rotation is complete, the value must be shifted back into the low-order byte of `t` so that the value can be returned. Because the bit being shifted out moves to bit 7, the following statement checks whether that value is a 1:

```
if(t & 128)
```

The decimal value 128 has only bit 7 set. If it is set, then t is ORed with 32768, which is the decimal value in which bit 15 is set, and bits 14 through 0 are cleared. This causes bit 15 of t to be set and the other bits to remain unchanged.

4. Here is an entire program that demonstrates `lrotate()` and `rrotate()`. It uses the `show_binary()` function to display the results of each rotation.

```
/*
    Project 7-1

    Left and right rotate functions for byte values.
*/

#include <iostream>
using namespace std;

unsigned char rrotate(unsigned char val, int n);
unsigned char lrotate(unsigned char val, int n);
void show_binary(unsigned int u);

int main()
{
    char ch = 'T';

    cout << "Original value in binary:\n";
    show_binary(ch);

    cout << "Rotating right 8 times:\n";
    for(int i=0; i < 8; i++) {
        ch = rrotate(ch, 1);
        show_binary(ch);
    }
}
```

```

    cout << "Rotating left 8 times:\n";
    for(int i=0; i < 8; i++) {
        ch = lrotate(ch, 1);
        show_binary(ch);
    }

    return 0;
}

// Left-rotate a byte n places.
unsigned char lrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;

    for(int i=0; i < n; i++) {
        t = t << 1;

        /* If a bit shifts out, it will be in bit 8
           of the integer t. If this is the case,
           put that bit on the right side. */
        if(t & 256)
            t = t | 1; // put a 1 on the right end
    }

    return t; // return the lower 8 bits.
}

// Right-rotate a byte n places.
unsigned char rrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;

    // First, move the value 8 bits higher.
    t = t << 8;

    for(int i=0; i < n; i++) {
        t = t >> 1;

        /* If a bit shifts out, it will be in bit 7
           of the integer t. If this is the case,
           put that bit on the left side. */
        if(t & 128)

```

```

        t = t | 32768; // put a 1 on left end
    }

    /* Finally, move the result back to the
       lower 8 bits of t. */
    t = t >> 8;

    return t;
}

// Display the bits within a byte.
void show_binary(unsigned int u)
{
    int t;

    for(t=128; t>0; t = t/2)
        if(u & t) cout << "1 ";
        else cout << "0 ";

    cout << "\n";
}

```

5. The output from the program is shown here:

Original value in binary:

0 1 0 1 0 1 0 0

Rotating right 8 times:

0 0 1 0 1 0 1 0

0 0 0 1 0 1 0 1

1 0 0 0 1 0 1 0

0 1 0 0 0 1 0 1

1 0 1 0 0 0 1 0

0 1 0 1 0 0 0 1

1 0 1 0 1 0 0 0

0 1 0 1 0 1 0 0

Rotating left 8 times:

1 0 1 0 1 0 0 0

0 1 0 1 0 0 0 1

1 0 1 0 0 0 1 0

0 1 0 0 0 1 0 1

1 0 0 0 1 0 1 0

0 0 0 1 0 1 0 1

0 0 1 0 1 0 1 0

0 1 0 1 0 1 0 0

CRITICAL SKILL 7.9: The ? Operator

One of C++'s most fascinating operators is the ?. The ? operator is often used to replace if-else statements of this general form:

```
if (condition) var = expression1; else var = expression2;
```

Here, the value assigned to var depends upon the outcome of the condition controlling the if.

The ? is called a ternary operator because it requires three operands. It takes the general form

```
Exp1 ? Exp2 : Exp3;
```

where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression. If Exp1 is false, then Exp3 is evaluated, and its value becomes the value of the expression. Consider this example, which assigns absval the absolute value of val:

```
absval = val < 0 ? -val : val; // get absolute value of val
```

Here, absval will be assigned the value of val if val is zero or greater. If val is negative, then absval will be assigned the negative of that value (which yields a positive value). The same code written using an if-else statement would look like this:

```
if(val < 0) absval = -val; else absval = val;
```

Here is another example of the ? operator. This program divides two numbers, but will not allow a division by zero.

```
/* This program uses the ? operator to prevent
   a division by zero. */

#include <iostream>
using namespace std;

int div_zero();
```

```

int main()
{
    int i, j, result;

    cout << "Enter dividend and divisor: ";
    cin >> i >> j;

    // This statement prevents a divide by zero error.
    result = j ? i/j : div_zero(); ← Use the ? to prevent a
                                   divide-by-zero error.

    cout << "Result: " << result;

    return 0;
}

int div_zero()
{
    cout << "Cannot divide by zero.\n";
    return 0;
}

```

Here, if *j* is non-zero, then *i* is divided by *j*, and the outcome is assigned to *result*. Otherwise, the `div_zero()` error handler is called, and zero is assigned to *result*.

CRITICAL SKILL 7.10: The Comma Operator

Another interesting C++ operator is the comma. You have seen some examples of the comma operator in the for loop, where it has been used to allow multiple initialization or increment statements. However, the comma can be used as a part of any expression. It strings together several expressions. The value of a comma-separated list of expressions is the value of the right-most expression. The values of the other expressions will be discarded. This means that the expression on the right side will become the value of the total comma-separated expression. For example,

```
var = (count=19, incr=10, count+1);
```

first assigns *count* the value 19, assigns *incr* the value 10, then adds 1 to *count*, and finally assigns *var* the value produced by the entire comma expression, which is 20. The parentheses are necessary because the comma operator has a lower precedence than the assignment operator.

To actually see the effects of the comma operator, try running the following program:

```

#include <iostream>
using namespace std;

int main()
{
    int i, j;

    j = 10;

    i = (j++, j+100, 999+j); ← The comma means "do
                             this and this and this."

    cout << i;

    return 0;
}

```

This program prints "1010" on the screen. Here is why: *j* starts with the value 10. *j* is then incremented to 11. Next, *j* is added to 100. Finally, *j* (still containing 11) is added to 999, which yields the result 1010.

Essentially, the comma's effect is to cause a sequence of operations to be performed. When it is used on the right side of an assignment statement, the value assigned is the value of the last expression in the comma-separated list. You can, in some ways, think of the comma operator as having the same meaning that the word "and" has in English when used in the phrase "do this and this and this."

Progress Check

1. Given this expression:
`x = 10 > 11 ? 1 : 0;`
 what is the value of *x* after the expression evaluates?
2. The `?` operator is called a ternary operator because it has _____ operands.
3. What does the comma do?

Multiple Assignments

C++ allows a convenient method of assigning many variables the same value: using multiple assignments in a single statement. For example, this fragment assigns *count*, *incr*, and *index* the value 10:

```
count = incr = index = 10;
```

In professionally written programs, you will often see variables assigned a common value using this format.

CRITICAL SKILL 7.11: Compound Assignment

C++ has a special compound-assignment operator that simplifies the coding of a certain type of assignment statement. For example,

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

can be rewritten using a compound assignment operator, as shown next:

```
x += 10;
```

The operator pair `+=` tells the compiler to assign to `x` the value of `x` plus 10. Compound assignment operators exist for all the binary operators in C++ (that is, those that require two operands). Their general form is

```
var op = expression;
```

Here is another example:

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

is the same as

```
x -= 100;
```

Because it saves you some typing, compound assignment is also sometimes referred to as shorthand assignment. You will see shorthand notation used widely in professionally written C++ programs, so you should become familiar with it.

CRITICAL SKILL 7.12: Using `sizeof`

Sometimes it is helpful to know the size, in bytes, of a type of data. Since the sizes of C++'s built-in types can differ between computing environments, knowing the size of a variable in advance, in all situations, is not possible. To solve this problem, C++ includes the `sizeof` compile-time operator, which has these general forms:

```
sizeof (type) sizeof var-name
```

The first version returns the size of the specified data type, and the second returns the size of the specified variable. As you can see, if you want to know the size of a data type, such as `int`, you must enclose the type name in parentheses. If you want to know the size of a variable, no parentheses are needed, although you can use them if you desire.

To see how `sizeof` works, try the following short program. For many 32-bit environments, it displays the values 1, 4, 4, and 8.

```
// Demonstrate sizeof.

#include <iostream>
using namespace std;

int main()
{
    char ch;
    int i;

    cout << sizeof ch << ' '; // size of char
    cout << sizeof i << ' '; // size of int
    cout << sizeof (float) << ' '; // size of float
    cout << sizeof (double) << ' '; // size of double

    return 0;
}
```

You can apply `sizeof` to any data type. For example, when it is applied to an array, it returns the number of bytes used by the array. Consider this fragment:

Assuming 4-byte integers, this fragment displays the value 16 (that is, 4 bytes times 4 elements).

As mentioned earlier, `sizeof` is a compile-time operator. All information necessary for computing the size of a variable or data type is known during compilation. The `sizeof` operator primarily helps you to generate portable code that depends upon the size of the C++ data types. Remember, since the sizes of types in C++ are defined by the implementation, it is bad style to make assumptions about their sizes in code that you write.

Progress Check

1. Show how to assign the variables `t1`, `t2`, and `t3` the value 10 using one assignment statement.
2. How can
 $x = x + 100$
 be rewritten?
3. The `sizeof` operator returns the size of a variable or type in _____.
 More Data Types and Operators

Precedence Summary

Table 7-2 lists the precedence, from highest to lowest, of all C++ operators. Most operators associate from left to right. The unary operators, the assignment operators, and the `?` operator associate from right to left. Note that the table includes a few operators that you have not yet learned about, many of which are used in object-oriented programming.

Precedence	Operators
Highest	() [] -> :: .
	! ~ ++ -- * & sizeof new delete typeid type-casts
	.* ->*
	* / %
	+ -
	<< >>
	< <= > >=
	== !=
	&
	^
	&&
	?:
	= += -= *= /= %= >>= <<= &= ^= =
Lowest	,

Table 7-2 Precedence of the C++ Operators

Module 7 Mastery Check

1. Show how to declare an int variable called test that can't be changed by the program. Give it an initial value of 100.
2. The volatile specifier tells the compiler that a variable might be changed by forces outside the program. True or false?
3. In a multifile project, what specifier do you use to tell one file about a global variable declared in another file?
4. What is the most important attribute of a static local variable?
5. Write a program that contains a function called counter(), which simply counts how many times it is called. Have it return the current count.
6. Given this fragment, which variable would most benefit from being specified as register?

```

int myfunc()
{
    int x;
    int y;
    int z;

    z = 10;
    y = 0;

    for(x=z; x < 15; x++)
        y += x;

    return y;
}

```

7. How does & differ from &&?

8. What does this statement do?

```
x *= 10;
```

9. Using the `rrotate()` and `lrotate()` functions from Project 7-1, it is possible to encode and decode a string. To code the string, left-rotate each letter by some amount that is specified by a key. To decode, right-rotate each character by the same amount. Use a key that consists of a string of characters. There are many ways to compute the number of rotations from the key. Be creative. The solution shown in the online answers is only one of many.

10. On your own, expand `show_binary()` so that it shows all bits within an unsigned int rather than just the first eight.