Chapter 4 Statements, Expressions, and Operators

4-1. Statements

A *statement* is a complete direction instructing the computer to carry out some task. In C, statements are usually written one per line, although some statements span multiple lines. C statements always end with a semicolon (except for preprocessor directives such as #define and #include, which are discussed on Day 21, "Advanced Compiler Use"). You've already been introduced to some of C's statement types. For example:

```
x = 2 + 3;
```

is an assignment statement. It instructs the computer to add 2 and 3 and to assign the result to the variable x. Other types of statements will be introduced as needed throughout this book.

4-1-1. Statements and White Space

The term *white space* refers to spaces, tabs, and blank lines in your source code. The C compiler isn't sensitive to white space. When the compiler reads a statement in your source code, it looks for the characters in the statement and for the terminating semicolon, but it ignores white space. Thus, the statement

```
x=2+3;
```

is equivalent to this statement:

```
x = 2 + 3;
```

It is also equivalent to this:

```
x         =
2
+
3;
```

This gives you a great deal of flexibility in formatting your source code. You shouldn't use formatting like the previous example, however. Statements should be entered one per line with a standardized scheme for spacing around variables and operators. If you follow the formatting conventions used in this book, you should be in good shape. As you become more experienced, you might discover that you prefer slight variations. The point is to keep your source code readable.

However, the rule that C doesn't care about white space has one exception: Within literal string constants, tabs and spaces aren't ignored; they are considered part of the string. A *string* is a series of characters. Literal string constants are strings that are enclosed within quotes and interpreted literally by the compiler, space for space. Although it's extremely bad form, the following is legal:

```
printf(
"Hello, world!"
);
```

This, however, is not legal:

```
printf("Hello,
world!");
```

To break a literal string constant line, you must use the backslash character (\) just before the break. Thus, the following is legal:

```
printf("Hello,world!");
```

4-1-2. Null Statements

If you place a semicolon by itself on a line, you create a *null statement*--a statement that doesn't perform any action. This is perfectly legal in C. Later in this book, you will learn how the null statement can be useful.

4-1-3. Compound Statements

A compound statement, also called a *block,* is a group of two or more C statements enclosed in braces. Here's an example of a block:

```
{
    printf("Hello, ");
    printf("world!");
}
```

In C, a block can be used anywhere a single statement can be used. Many examples of this appear throughout this book. Note that the enclosing braces can be positioned in different ways. The following is equivalent to the preceding example:

```
{printf("Hello, ");
printf("world!");}
```

It's a good idea to place braces on their own lines, making the beginning and end of blocks clearly visible. Placing braces on their own lines also makes it easier to see whether you've left one out.

DO stay consistent with how you use white space in statements.

DO put block braces on their own lines. This makes the code easier to read.

DO line up block braces so that it's easy to find the beginning and end of a block.

DON'T spread a single statement across multiple lines if there's no need to do so. Limit statements to one line if possible.

4-2. Expressions

In C, an *expression* is anything that evaluates to a numeric value. C expressions come in all levels of complexity.

4-2-1. Simple Expressions

The simplest C expression consists of a single item: a simple variable, literal constant, or symbolic constant. Here are four expressions:

Expression	*Description*

PI	A symbolic constant (defined in the program)
20	A literal constant
rate	A variable
-1.25	Another literal constant

A *literal constant* evaluates to its own value. A *symbolic constant* evaluates to the value it was given when you created it using the #define directive. A variable evaluates to the current value assigned to it by the program.

4-2-2. Complex Expressions

Complex expressions consist of simpler expressions connected by operators. For example:

```
2 + 8
```

is an expression consisting of the sub-expressions 2 and 8 and the addition operator +. The expression 2 + 8 evaluates, as you know, to 10. You can also write C expressions of great complexity:

```
1.25 / 8 + 5 * rate + rate * rate / cost
```

When an expression contains multiple operators, the evaluation of the expression depends on operator precedence. This concept is covered later in this chapter, as are details about all of C's operators.

C expressions get even more interesting. Look at the following assignment statement:

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

This statement evaluates the expression a + 10 and assigns the result to x. In addition, the entire statement x = a + 10 is itself an expression that evaluates to the value of the variable on the left side of the equal sign. This is illustrated in Figure 4.1.

Thus, you can write statements such as the following, which assigns the value of the expression a + 10 to both variables, x and y:

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

You can also write statements such as this:

```
x = 6 + (y = 4 + 5);
```

The result of this statement is that y has the value 9 and x has the value 15. Note the parentheses, which are required in order for the statement to compile. The use of parentheses is covered later in this chapter.

4-3. Operators

An *operator* is a symbol that instructs C to perform some operation, or action, on one or more operands. An *operand* is something that an operator acts on. In C, all operands are expressions. C operators fall into several categories:

4-3-1. The Assignment Operator

The *assignment operator* is the equal sign (=). Its use in programming is somewhat different from its use in regular math. If you write

```
x = y;
```

in a C program, it doesn't mean "x is equal to y." Instead, it means "assign the value of y to x." In a C assignment statement, the right side can be any expression, and the left side must be a variable name. Thus, the form is as follows:

```
variable = expression;
```

When executed, *expression* is evaluated, and the resulting value is assigned to *variable*.

4-3-2. Mathematical Operators

C's mathematical operators perform mathematical operations such as addition and subtraction. C has two unary mathematical operators and five binary mathematical operators.

The *unary* mathematical operators are so named because they take a single operand. C has two unary mathematical operators, listed in Table 4.1.

Table 4.1. C's unary mathematical operators.

Operator	Symbol	Action	Examples
Increment	++	Increments the operand by one	++x, x++
Decrement	--	Decrements the operand by one	--x, x--

The increment and decrement operators can be used only with variables, not with constants. The operation performed is to add one to or subtract one from the operand. In other words, the statements

```
++x;
--y;
```

are the equivalent of these statements:

```
x = x + 1;
y = y - 1;
```

You should note from Table 4.1 that either unary operator can be placed before its operand (*prefix* mode) or after its operand (*postfix* mode). These two modes are not equivalent. They differ in terms of when the increment or decrement is performed:

- When used in prefix mode, the increment and decrement operators modify their operand before it's used.
- When used in postfix mode, the increment and decrement operators modify their operand after it's used.

An example should make this clearer. Look at these two statements:

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

After these statements are executed, x has the value 11, and y has the value 10. The value of x was assigned to y, and then x was incremented. In contrast, the following statements result in both y and x having the value 11. x is incremented, and then its value is assigned to y.

```
x = 10;
```

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

Remember that = is the assignment operator, not a statement of equality. As an analogy, think of = as the "photocopy" operator. The statement y = x means to copy x into y. Subsequent changes to x, after the copy has been made, have no effect on y.

Listing 4.1 illustrates the difference between prefix mode and postfix mode.

Listing 4.1. UNARY.C: Demonstrates prefix and postfix modes.

```
1:   /* Demonstrates unary operator prefix and postfix modes */
2:
3:   #include <stdio.h>
4:
5:   int a, b;
6:
7:   main()
8:   {
9:       /* Set a and b both equal to 5 */
10:
11:      a = b = 5;
12:
13:      /* Print them, decrementing each time. */
14:      /* Use prefix mode for b, postfix mode for a */
15:
16:      printf("\n%d   %d", a--, --b);
17:      printf("\n%d   %d", a--, --b);
18:      printf("\n%d   %d", a--, --b);
19:      printf("\n%d   %d", a--, --b);
20:      printf("\n%d   %d\n", a--, --b);
21:
22:      return 0;
23:  }
5    4
4    3
3    2
2    1
1    0
```

ANALYSIS: This program declares two variables, a and b, in line 5. In line 11, the variables are set to the value of 5. With the execution of each printf() statement (lines 16 through 20), both a and b are decremented by 1. After a is printed, it is decremented, whereas b is decremented before it is printed.

4-3-3. Binary Mathematical Operators

C's binary operators take two operands. The binary operators, which include the common mathematical operations found on a calculator, are listed in Table 4.2.

Table 4.2. C's binary mathematical operators.

Operator	Symbol	Action	Example
Addition	+	Adds two operands	x + y

Subtraction	-	Subtracts the second operand from the first operand	x - y
Multiplication	*	Multiplies two operands	x * y
Division	/	Divides the first operand by the second operand	x / y
Modulus	%	Gives the remainder when the first operand is divided by the second operand	x % y

The first four operators listed in Table 4.2 should be familiar to you, and you should have little trouble using them. The fifth operator, modulus, might be new. Modulus returns the remainder when the first operand is divided by the second operand. For example, 11 modulus 4 equals 3 (that is, 4 goes into 11 two times with 3 left over). Here are some more examples:

```
100 modulus 9 equals 1
10 modulus 5 equals 0
40 modulus 6 equals 4
```

Listing 4.2 illustrates how you can use the modulus operator to convert a large number of seconds into hours, minutes, and seconds.

Listing 4.2. SECONDS.C: Demonstrates the modulus operator.

```
1:   /* Illustrates the modulus operator. */
2:   /* Inputs a number of seconds, and converts to hours, */
3:   /* minutes, and seconds. */
4:
5:   #include <stdio.h>
6:
7:   /* Define constants */
8:
9:   #define SECS_PER_MIN 60
10:  #define SECS_PER_HOUR 3600
11:
12:  unsigned seconds, minutes, hours, secs_left, mins_left;
13:
14:  main()
15:  {
16:      /* Input the number of seconds */
17:
18:      printf("Enter number of seconds (< 65000): ");
19:      scanf("%d", &seconds);
20:
21:      hours = seconds / SECS_PER_HOUR;
22:      minutes = seconds / SECS_PER_MIN;
23:      mins_left = minutes % SECS_PER_MIN;
24:      secs_left = seconds % SECS_PER_MIN;
25:
26:      printf("%u seconds is equal to ", seconds);
27:      printf("%u h, %u m, and %u s\n", hours, mins_left,
secs_left);
28:
29:      return 0;
30:  }
```

```
Enter number of seconds (< 65000): 60
60 seconds is equal to 0 h, 1 m, and 0 s
Enter number of seconds (< 65000): 10000
10000 seconds is equal to 2 h, 46 m, and 40 s
```

ANALYSIS: SECONDS.C follows the same format that all the previous programs have followed. Lines 1 through 3 provide some comments to state what the program does. Line 4 is white space to make the program more readable. Just like the white space in statements and expressions, blank lines are ignored by the compiler. Line 5 includes the necessary header file for this program. Lines 9 and 10 define two constants, SECS_PER_MIN and SECS_PER_HOUR, that are used to make the statements in the program easier to read. Line 12 declares all the variables that will be used. Some people choose to declare each variable on a separate line rather than all on one. As with many elements of C, this is a matter of style. Either method is correct.

Line 14 is the main() function, which contains the bulk of the program. To convert seconds to hours and minutes, the program must first get the values it needs to work with. To do this, line 18 uses the printf() function to display a statement on-screen, followed by line 19, which uses the scanf() function to get the number that the user entered. The scanf() statement then stores the number of seconds to be converted into the variable seconds. The printf() and scanf() functions are covered in more detail on Day 7, "Fundamentals of Input and Output." Line 21 contains an expression to determine the number of hours by dividing the number of seconds by the constant SECS_PER_HOUR. Because hours is an integer variable, the remainder value is ignored. Line 22 uses the same logic to determine the total number of minutes for the seconds entered. Because the total number of minutes figured in line 22 also contains minutes for the hours, line 23 uses the modulus operator to divide the hours and keep the remaining minutes. Line 24 carries out a similar calculation for determining the number of seconds that are left. Lines 26 and 27 are similar to what you have seen before. They take the values that have been calculated in the expressions and display them. Line 29 finishes the program by returning 0 to the operating system before exiting.

4-3-4. Operator Precedence and Parentheses

In an expression that contains more than one operator, what is the order in which operations are performed? The importance of this question is illustrated by the following assignment statement:

```
x = 4 + 5 * 3;
```

Performing the addition first results in the following, and x is assigned the value 27:

```
x = 9 * 3;
```

In contrast, if the multiplication is performed first, you have the following, and x is assigned the value 19:

```
x = 4 + 15;
```

Clearly, some rules are needed about the order in which operations are performed. This order, called *operator precedence,* is strictly spelled out in C. Each operator has a specific precedence. When an expression is evaluated, operators with higher precedence are performed first. Table 4.3 lists the precedence of C's mathematical operators. Number 1 is the highest precedence and thus is evaluated first.

Table 4.3. The precedence of C's mathematical operators.

Operators	Relative Precedence
++ --	1
* / %	2

+ -	3

Looking at Table 4.3, you can see that in any C expression, operations are performed in the following order:

If an expression contains more than one operator with the same precedence level, the operators are performed in left-to-right order as they appear in the expression. For example, in the following expression, the % and * have the same precedence level, but the % is the leftmost operator, so it is performed first:

```
12 % 5 * 2
```

The expression evaluates to 4 (12 % 5 evaluates to 2; 2 times 2 is 4).

Returning to the previous example, you see that the statement x = 4 + 5 * 3; assigns the value 19 to x because the multiplication is performed before the addition.

What if the order of precedence doesn't evaluate your expression as needed? Using the previous example, what if you wanted to add 4 to 5 and then multiply the sum by 3? C uses parentheses to modify the evaluation order. A subexpression enclosed in parentheses is evaluated first, without regard to operator precedence. Thus, you could write

```
x = (4 + 5) * 3;
```

The expression 4 + 5 inside parentheses is evaluated first, so the value assigned to x is 27.

You can use multiple and nested parentheses in an expression. When parentheses are nested, evaluation proceeds from the innermost expression outward. Look at the following complex expression:

```
x = 25 - (2 * (10 + (8 / 2)));
```

The evaluation of this expression proceeds as follows:

1. The innermost expression, 8 / 2, is evaluated first, yielding the value 4:
```
25 - (2 * (10 + 4))
```
2. Moving outward, the next expression, 10 + 4, is evaluated, yielding the value 14:
```
25 - (2 * 14)
```
3. The last, or outermost, expression, 2 * 14, is evaluated, yielding the value 28:
```
25 - 28
```
4. The final expression, 25 - 28, is evaluated, assigning the value -3 to the variable x:
```
x = -3
```

You might want to use parentheses in some expressions for the sake of clarity, even when they aren't needed for modifying operator precedence. Parentheses must always be in pairs, or the compiler generates an error message.

4-3-5. Order of Subexpression Evaluation

As was mentioned in the previous section, if C expressions contain more than one operator with the same precedence level, they are evaluated left to right. For example, in the expression

```
w * x / y * z
```

w is multiplied by x, the result of the multiplication is then divided by y, and the result of the division is then multiplied by z.

Across precedence levels, however, there is no guarantee of left-to-right order. Look at this expression:

```
w * x / y + z / y
```

Because of precedence, the multiplication and division are performed before the addition. However, C doesn't specify whether the subexpression w * x / y is to be evaluated before or after z / y. It might not be clear to you why this matters. Look at another example:

```
w * x / ++y + z / y
```

If the left subexpression is evaluated first, y is incremented when the second expression is evaluated. If the right expression is evaluated first, y isn't incremented, and the result is different. Therefore, you should avoid this sort of indeterminate expression in your programming.

Near the end of this chapter, the section "Operator Precedence Revisited" lists the precedence of all of C's operators.

DO use parentheses to make the order of expression evaluation clear.

DON'T overload an expression. It is often more clear to break an expression into two or more statements. This is especially true when you're using the unary operators (--) or (++).

4-3-6. Relational Operators

C's relational operators are used to compare expressions, asking questions such as, "Is x greater than 100?" or "Is y equal to 0?" An expression containing a relational operator evaluates to either true (1) or false (0). C's six relational operators are listed in Table 4.4.

Table 4.5 shows some examples of how relational operators might be used. These examples use literal constants, but the same principles hold with variables.

NOTE: "True" is considered the same as "yes," which is also considered the same as 1. "False" is considered the same as "no," which is considered the same as 0.

Table 4.4. C's relational operators.

Operator	Symbol	Question Asked	Example
Equal	==	Is operand 1 equal to operand 2?	x == y
Greater than	>	Is operand 1 greater than operand 2?	x > y
Less than	<	Is operand 1 less than operand 2?	x < y
Greater than or equal to	>=	Is operand 1 greater than or equal to operand 2?	x >= y
Less than or equal to	<=	Is operand 1 less than or equal to operand 2?	x <= y

Not equal	!=	Is operand 1 not equal to operand 2?	x != y

Table 4.5. Relational operators in use.

Expression	How It Reads	What It Evaluates To
5 == 1	Is 5 equal to 1?	0 (false)
5 > 1	Is 5 greater than 1?	1 (true)
5 != 1	Is 5 not equal to 1?	1 (true)
(5 + 10) == (3 * 5)	Is (5 + 10) equal to (3 * 5)?	1 (true)

DO learn how C interprets true and false. When working with relational operators, true is equal to 1, and false is equal to 0.

DON'T confuse ==, the relational operator, with =, the assignment operator. This is one of the most common errors that C programmers make.

4-4. The if Statement

Relational operators are used mainly to construct the relational expressions used in if and while statements, covered in detail on Day 6, "Basic Program Control." For now, I'll explain the basics of the if statement to show how relational operators are used to make *program control statements*.

You might be wondering what a program control statement is. Statements in a C program normally execute from top to bottom, in the same order as they appear in your source code file. A program control statement modifies the order of statement execution. Program control statements can cause other program statements to execute multiple times or to not execute at all, depending on the circumstances. The if statement is one of C's program control statements. Others, such as do and while, are covered on Day 6.

In its basic form, the if statement evaluates an expression and directs program execution depending on the result of that evaluation. The form of an if statement is as follows:

```
if (expression)
    statement;
```

If *expression* evaluates to true, *statement* is executed. If *expression* evaluates to false, *statement* is not executed. In either case, execution then passes to whatever code follows the if statement. You could say that execution of *statement* depends on the result of *expression*. Note that both the line if (*expression*) and the line *statement*; are considered to comprise the complete if statement; they are not separate statements.

An if statement can control the execution of multiple statements through the use of a compound statement, or block. As defined earlier in this chapter, a block is a group of two or more statements enclosed in braces. A block can be used anywhere a single statement can be used. Therefore, you could write an if statement as follows:

```
if (expression)
{
    statement1;
```

```
    statement2;
    /* additional code goes here */
    statementn;
}
```

DO remember that if you program too much in one day, you'll get C sick.

DO indent statements within a block to make them easier to read. This includes the statements within a block in an if statement.

DON'T make the mistake of putting a semicolon at the end of an if statement. An if statement should end with the conditional statement that follows it. In the following, *statement1* executes whether or not x equals 2, because each line is evaluated as a separate statement, not together as intended:

```
if( x == 2);          /* semicolon does not belong!  */
statement1;
```

In your programming, you will find that if statements are used most often with relational expressions; in other words, "Execute the following statement(s) only if such-and-such a condition is true." Here's an example:

```
if (x > y)
    y = x;
```

This code assigns the value of x to y only if x is greater than y. If x is not greater than y, no assignment takes place. Listing 4.3 illustrates the use of if statements.

Listing 4.3. LIST0403.C: Demonstrates if statements.

```
1:   /* Demonstrates the use of if statements */
2:
3:   #include <stdio.h>
4:
5:   int x, y;
6:
7:   main()
8:   {
9:       /* Input the two values to be tested */
10:
11:      printf("\nInput an integer value for x: ");
12:      scanf("%d", &x);
13:      printf("\nInput an integer value for y: ");
14:      scanf("%d", &y);
15:
16:      /* Test values and print result */
17:
18:      if (x == y)
```

```
19:          printf("x is equal to y\n");
20:
21:      if (x > y)
22:          printf("x is greater than y\n");
23:
24:      if (x < y)
25:          printf("x is smaller than y\n");
26:
27:      return 0;
28:  }
Input an integer value for x: 100
Input an integer value for y: 10
x is greater than y
Input an integer value for x: 10
Input an integer value for y: 100
x is smaller than y
Input an integer value for x: 10
Input an integer value for y: 10
x is equal to y
```

LIST0403.C shows three if statements in action (lines 18 through 25). Many of the lines in this program should be familiar. Line 5 declares two variables, x and y, and lines 11 through 14 prompt the user for values to be placed into these variables. Lines 18 through 25 use if statements to determine whether x is greater than, less than, or equal to y. Note that line 18 uses an if statement to see whether x is equal to y. Remember that ==, the equal operator, means "is equal to" and should not be confused with =, the assignment operator. After the program checks to see whether the variables are equal, in line 21 it checks to see whether x is greater than y, followed by a check in line 24 to see whether x is less than y. If you think this is inefficient, you're right. In the next program, you will see how to avoid this inefficiency. For now, run the program with different values for x and y to see the results.

NOTE: You will notice that the statements within an if clause are indented. This is a common practice to aid readability.

4-4-1. The else Clause

An if statement can optionally include an else clause. The else clause is included as follows:

```
if (expression)
    statement1;
else
    statement2;
```

If *expression* evaluates to true, *statement1* is executed. If *expression* evaluates to false, *statement2* is executed. Both *statement1* and *statement2* can be compound statements or blocks.

Listing 4.4 shows Listing 4.3 rewritten to use an if statement with an else clause.

Listing 4.4. An if statement with an else clause.

```
1:   /* Demonstrates the use of if statement with else clause */
2:
3:   #include <stdio.h>
4:
5:   int x, y;
6:
7:   main()
8:   {
9:       /* Input the two values to be tested */
10:
11:      printf("\nInput an integer value for x: ");
12:      scanf("%d", &x);
13:      printf("\nInput an integer value for y: ");
14:      scanf("%d", &y);
15:
16:      /* Test values and print result */
17:
18:      if (x == y)
19:          printf("x is equal to y\n");
20:      else
21:          if (x > y)
22:              printf("x is greater than y\n");
23:          else
24:              printf("x is smaller than y\n");
25:
26:      return 0;
27:  }
Input an integer value for x: 99
Input an integer value for y: 8
x is greater than y
Input an integer value for x: 8
Input an integer value for y: 99
x is smaller than y
Input an integer value for x: 99
Input an integer value for y: 99
x is equal to y
```

ANALYSIS: Lines 18 through 24 are slightly different from the previous listing. Line 18 still checks to see whether x equals y. If x does equal y, x is equal to y appears on-screen, just as in Listing 4.3 (LIST0403.C). However, the program then ends, and lines 20 through 24 aren't executed. Line 21 is executed only if x is not equal to y , or, to be more accurate, if the expression "x equals y" is false. If x does not equal y, line 21 checks to see whether x is greater than y. If so, line 22 prints x is greater than y; otherwise (else), line 24 is executed.

Listing 4.4 uses a nested if statement. Nesting means to place (nest) one or more C statements inside another C statement. In the case of Listing 4.4, an if statement is part of the first if statement's else clause.

The if Statement

Form 1

```
if( expression )
    statement1;
next_statement;
```

This is the if statement in its simplest form. If *expression* is true, *statement1* is executed. If *expression* is not true, *statement1* is ignored.

Form 2

```
if( expression )
    statement1;
else
    statement2;
next_statement;
```

This is the most common form of the if statement. If *expression* is true, *statement1* is executed; otherwise, *statement2* is executed.

Form 3

```
if( expression1 )
    statement1;
else if( expression2 )
    statement2;
else
    statement3;
next_statement;
```

This is a nested if. If the first expression, *expression1*, is true, *statement1* is executed before the program continues with the *next_statement*. If the first expression is not true, the second expression, *expression2*, is checked. If the first expression is not true, and the second is true, *statement2* is executed. If both expressions are false, *statement3* is executed. Only one of the three statements is executed.

Example 1

```
if( salary > 45,0000 )
    tax = .30;
else
    tax = .25;
```

Example 2

```
if( age < 18 )
    printf("Minor");
else if( age < 65 )
    printf("Adult");
else
    printf( "Senior Citizen");
```

4-5. Evaluating Relational Expressions

Remember that expressions using relational operators are true C expressions that evaluate, by definition, to a value. Relational expressions evaluate to a value of either false (0) or true (1). Although the most common use of relational expressions is within if statements and other conditional constructions, they can be used as purely numeric values. This is illustrated in Listing 4.5.

Listing 4.5. Evaluating relational expressions.

```
1:   /* Demonstrates the evaluation of relational expressions */
2:
3:   #include <stdio.h>
4:
5:   int a;
6:
7:   main()
8:   {
9:       a = (5 == 5);           /* Evaluates to 1 */
10:      printf("\na = (5 == 5)\na = %d", a);
11:
12:      a = (5 != 5);           /* Evaluates to 0 */
13:      printf("\na = (5 != 5)\na = %d", a);
14:
15:      a = (12 == 12) + (5 != 1); /* Evaluates to 1 + 1 */
16:      printf("\na = (12 == 12) + (5 != 1)\na = %d\n", a);
17:      return 0;
18:  }
a = (5 == 5)
a = 1
a = (5 != 5)
a = 0
a = (12 == 12) + (5 != 1)
a = 2
```

ANNALYSIS: The output from this listing might seem a little confusing at first. Remember, the most common mistake people make when using the relational operators is to use a single equal sign--the assignment operator--instead of a double equal sign. The following expression evaluates to 5 (and also assigns the value 5 to x):

```
x = 5
```

In contrast, the following expression evaluates to either 0 or 1 (depending on whether x is equal to 5) and doesn't change the value of x:

```
x == 5
```

If by mistake you write

```
if (x = 5)
   printf("x is equal to 5");
```

the message always prints because the expression being tested by the if statement always evaluates to true, no matter what the original value of x happens to be.

Looking at Listing 4.5, you can begin to understand why a takes on the values that it does. In line 9, the value 5 does equal 5, so true (1) is assigned to a. In line 12, the statement "5 does not equal 5" is false, so 0 is assigned to a.

To reiterate, the relational operators are used to create relational expressions that ask questions about relationships between expressions. The answer returned by a relational expression is a numeric value of either 1 (representing true) or 0 (representing false).

The Precedence of Relational Operators

Like the mathematical operators discussed earlier in this chapter, the relational operators each have a precedence that determines the order in which they are performed in a multiple-operator expression. Similarly, you can use parentheses to modify precedence in expressions that use relational operators. The section "Operator Precedence Revisited" near the end of this chapter lists the precedence of all of C's operators.

First, all the relational operators have a lower precedence than the mathematical operators. Thus, if you write the following, 2 is added to x, and the result is compared to y:

```
if (x + 2 > y)
```

This is the equivalent of the following line, which is a good example of using parentheses for the sake of clarity:

```
if ((x + 2) > y)
```

Although they aren't required by the C compiler, the parentheses surrounding (x + 2) make it clear that it is the sum of x and 2 that is to be compared with y.

There is also a two-level precedence within the relational operators, as shown in Table 4.6.

Table 4.6. The order of precedence of C's relational operators.

Operators	Relative Precedence
< <= > >=	1
!= ==	2

Thus, if you write

```
x == y > z
```

it is the same as

```
x == (y > z)
```

because C first evaluates the expression y > z, resulting in a value of 0 or 1. Next, C determines whether x is equal to the 1 or 0 obtained in the first step. You will rarely, if ever, use this sort of construction, but you should know about it.

DON'T put assignment statements in if statements. This can be confusing to other people who look at your code. They might think it's a mistake and change your assignment to the logical equal statement.

DON'T use the "not equal to" operator (!=) in an if statement containing an else. It's almost always clearer to use the "equal to" operator (==) with an else. For instance, the following code:

```
if ( x != 5 )
   statement1;
else
   statement2;
```

would be better written as this:

```
if (x == 5 )
   statement2;
else
   statement1;
```

4-6. Logical Operators

Sometimes you might need to ask more than one relational question at once. For example, "If it's 7:00 a.m. and a weekday and not my vacation, ring the alarm." C's logical operators let you combine two or more relational expressions into a single expression that evaluates to either true or false. Table 4.7 lists C's three logical operators.

Table 4.7. C's logical operators.

Operator	Symbol	Example
AND	&&	*exp1* && *exp2*
OR	||	*exp1* || *exp2*
NOT	!	!*exp1*

The way these logical operators work is explained in Table 4.8.

Table 4.8. C's logical operators in use.

Expression	What It Evaluates To
(*exp1* && *exp2*)	True (1) only if both *exp1* and *exp2* are true; false (0) otherwise
(*exp1* || *exp2*)	True (1) if either *exp1* or *exp2* is true; false (0) only if both are false
(!*exp1*)	False (0) if *exp1* is true; true (1) if *exp1* is false

You can see that expressions that use the logical operators evaluate to either true or false, depending on the true/false value of their operand(s). Table 4.9 shows some actual code examples.

Table 4.9. Code examples of C's logical operators.

Expression	What It Evaluates To
(5 == 5) && (6 != 2)	True (1), because both operands are true
(5 > 1) || (6 < 1)	True (1), because one operand is true
(2 == 1) && (5 == 5)	False (0), because one operand is false
!(5 == 4)	True (1), because the operand is false

You can create expressions that use multiple logical operators. For example, to ask the question "Is x equal to 2, 3, or 4?" you would write

```
(x == 2) || (x == 3) || (x == 4)
```

The logical operators often provide more than one way to ask a question. If x is an integer variable, the preceding question also could be written in either of the following ways:

```
(x > 1) && (x < 5)
(x >= 2) && (x <= 4)
```

4-7. More on True/False Values

You've seen that C's relational expressions evaluate to 0 to represent false and to 1 to represent true. It's important to be aware, however, that any numeric value is interpreted as either true or false when it is used in a C expression or statement that is expecting a logical value (that is, a true or false value). The rules are as follows:

This is illustrated by the following example, in which the value of x is printed:

```
x = 125;
if (x)
   printf("%d", x);
```

Because x has a nonzero value, the if statement interprets the expression (x) as true. You can further generalize this because, for any C expression, writing

```
(expression)
```

is equivalent to writing

```
(expression != 0)
```

Both evaluate to true if *expression* is nonzero and to false if *expression* is 0. Using the not (!) operator, you can also write

```
(!expression)
```

which is equivalent to

```
(expression == 0)
```

4-7-1. The Precedence of Operators

As you might have guessed, C's logical operators also have a precedence order, both among themselves and in relation to other operators. The ! operator has a precedence equal to the unary mathematical operators ++ and --. Thus, ! has a higher precedence than all the relational operators and all the binary mathematical operators.

In contrast, the && and || operators have much lower precedence, lower than all the mathematical and relational operators, although && has a higher precedence than ||. As with all of C's operators, parentheses can be used to modify the evaluation order when using the logical operators. Consider the following example:

You want to write a logical expression that makes three individual comparisons:

1. Is a less than b?
2. Is a less than c?
3. Is c less than d?

You want the entire logical expression to evaluate to true if condition 3 is true and if either condition 1 or condition 2 is true. You might write

```
a < b || a < c && c < d
```

However, this won't do what you intended. Because the && operator has higher precedence than ||, the expression is equivalent to

```
a < b || (a < c && c < d)
```

and evaluates to true if (a < b) is true, whether or not the relationships (a < c) and (c < d) are true. You need to write

```
(a < b || a < c) && c < d
```

which forces the || to be evaluated before the &&. This is shown in Listing 4.6, which evaluates the expression written both ways. The variables are set so that, if written correctly, the expression should evaluate to false (0).

Listing 4.6. Logical operator precedence.

View Code

ANALYSIS: Enter and run this listing. Note that the two values printed for the expression are different. This program initializes four variables, in line 7, with values to be used in the comparisons. Line 8 declares x to be used to store and print the results. Lines 14 and 19 use the logical operators. Line 14 doesn't use parentheses, so the results are determined by operator precedence. In this case, the results aren't what you wanted. Line 19 uses parentheses to change the order in which the expressions are evaluated.

4-7-2. Compound Assignment Operators

C's compound assignment operators provide a shorthand method for combining a binary mathematical operation with an assignment operation. For example, say you want to increase the value of x by 5, or, in other words, add 5 to x and assign the result to x. You could write

```
x = x + 5;
```

Using a compound assignment operator, which you can think of as a shorthand method of assignment, you would write

```
x += 5;
```

In more general notation, the compound assignment operators have the following syntax (where op represents a binary operator):

```
exp1 op= exp2
```

This is equivalent to writing

```
exp1 = exp1 op exp2;
```

You can create compound assignment operators using the five binary mathematical operators discussed earlier in this chapter. Table 4.10 lists some examples.

Table 4.10. Examples of compound assignment operators.

When You Write This...	It Is Equivalent To This
x *= y	x = x * y
y -= z + 1	y = y - z + 1
a /= b	a = a / b
x += y / 8	x = x + y / 8
y %= 3	y = y % 3

The compound operators provide a convenient shorthand, the advantages of which are particularly evident when the variable on the left side of the assignment operator has a long name. As with all other assignment statements, a compound assignment statement is an expression and evaluates to the value assigned to the left side. Thus, executing the following statements results in both x and z having the value 14:

```
x = 12;
z = x += 2;
```

4-7-3. The Conditional Operator

The conditional operator is C's only *ternary* operator, meaning that it takes three operands. Its syntax is

```
exp1 ? exp2 : exp3;
```

If *exp1* evaluates to true (that is, nonzero), the entire expression evaluates to the value of *exp2*. If *exp1* evaluates to false (that is, zero), the entire expression evaluates as the value of *exp3*. For example, the following statement assigns the value 1 to x if y is true and assigns 100 to x if y is false:

```
x = y ? 1 : 100;
```

Likewise, to make z equal to the larger of x and y, you could write

```
z = (x > y) ? x : y;
```

Perhaps you've noticed that the conditional operator functions somewhat like an if statement. The preceding statement could also be written like this:

```
if (x > y)
z = x;
else
z = y;
```

The conditional operator can't be used in all situations in place of an if...else construction, but the conditional operator is more concise. The conditional operator can also be used in places you can't use an if statement, such as inside a single printf() statement:

```
printf( "The larger value is %d", ((x > y) ? x : y) );
```

4-7-4. The Comma Operator

The comma is frequently used in C as a simple punctuation mark, serving to separate variable declarations, function arguments, and so on. In certain situations, the comma acts as an operator rather than just as a separator. You can form an expression by separating two subexpressions with a comma. The result is as follows:

For example, the following statement assigns the value of b to x, then increments a, and then increments b:

```
x = (a++ , b++);
```

Because the ++ operator is used in postfix mode, the value of b--before it is incremented--is assigned to x. Using parentheses is necessary because the comma operator has low precedence, even lower than the assignment operator.

As you'll learn in the next chapter, the most common use of the comma operator is in for statements.

DO use (expression == 0) instead of (!expression). When compiled, these two expressions evaluate the same; however, the first is more readable.

DO use the logical operators && and || instead of nesting if statements.

DON'T confuse the assignment operator (=) with the equal to (==) operator.

4-8. Operator Precedence Revisited

Table 4.11 lists all the C operators in order of decreasing precedence. Operators on the same line have the same precedence.

Table 4.11. C operator precedence.

Level	Operators
1	() [] -> .
2	! ~ ++ -- * *(indirection)* & *(address-of)* (*type*)
	sizeof + *(unary)* - *(unary)*
3	* *(multiplication)* / %
4	+ -
5	<< >>
6	< <= > >=
7	== !=
8	& *(bitwise* AND*)*
9	^
10	|
11	&&
12	||
13	?:
14	= += -= *= /= %= &= ^= |= <<= >>=
15	,

() is the function operator; [] is the array operator.	

TIP: This is a good table to keep referring to until you become familiar with the order of precedence. You might find that you need it later.

4-9. Summary

This chapter covered a lot of material. You learned what a C statement is, that white space doesn't matter to a C compiler, and that statements always end with a semicolon. You also learned that a compound statement (or block), which consists of two or more statements enclosed in braces, can be used anywhere a single statement can be used.

Many statements are made up of some combination of expressions and operators. Remember that an expression is anything that evaluates to a numeric value. Complex expressions can contain many simpler expressions, which are called subexpressions.

Operators are C symbols that instruct the computer to perform an operation on one or more expressions. Some operators are unary, which means that they operate on a single operand. Most of C's operators are binary, however, operating on two operands. One operator, the conditional operator, is ternary. C's operators have a defined hierarchy of precedence that determines the order in which operations are performed in an expression that contains multiple operators.

You've also been introduced to C's if statement, which lets you control program execution based on the evaluation of relational expressions.

Q&A

Q What effect do spaces and blank lines have on how a program runs?
A White space (lines, spaces, tabs) makes the code listing more readable. When the program is compiled, white space is stripped and thus has no effect on the executable program. For this reason, you should use white space to make your program easier to read.
Q Is it better to code a compound if statement or to nest multiple if statements?
A You should make your code easy to understand. If you nest if statements, they are evaluated as shown in this chapter. If you use a single compound statement, the expressions are evaluated only until the entire statement evaluates to false.
Q What is the difference between unary and binary operators?
A As the names imply, unary operators work with one variable, and binary operators work with two.
Q Is the subtraction operator (-) binary or unary?
A It's both! The compiler is smart enough to know which one you're using. It knows which form to use based on the number of variables in the expression that is used. In the following statement, it is unary:

```
x = -y;
```

versus the following binary use:

```
x = a - b;
```

Q Are negative numbers considered true or false?
A Remember that 0 is false, and any other value is true. This includes negative numbers.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the following C statement called, and what is its meaning?

```
x = 5 + 8;
```

2. What is an expression?
3. In an expression that contains multiple operators, what determines the order in which operations are performed?
4. If the variable x has the value 10, what are the values of x and a after each of the following statements is executed separately?

```
a = x++;
a = ++x;
```

5. To what value does the expression 10 % 3 evaluate?
6. To what value does the expression 5 + 3 * 8 / 2 + 2 evaluate?
7. Rewrite the expression in question 6, adding parentheses so that it evaluates to 16.
8. If an expression evaluates to false, what value does the expression have?
9. In the following list, which has higher precedence?
a. == or <
b. * or +
c. != or ==
d. >= or >
10. What are the compound assignment operators, and how are they useful?

Exercises

1. The following code is not well-written. Enter and compile it to see whether it works.

```
#include <stdio.h>
int x,y;main(){ printf(
"\nEnter two numbers");scanf(
"%d %d",&x,&y);printf(
"\n\n%d is bigger",(x>y)?x:y);return 0;}
```

2. Rewrite the code in exercise 1 to be more readable.
3. Change Listing 4.1 to count upward instead of downward.
4. Write an if statement that assigns the value of x to the variable y only if x is between 1 and 20. Leave y unchanged if x is not in that range.
5. Use the conditional operator to perform the same task as in exercise 4.
6. Rewrite the following nested if statements using a single if statement and compound operators.

```
if (x < 1)
   if ( x > 10 )
      statement;
```

7. To what value do each of the following expressions evaluate?
a. (1 + 2 * 3)
b. 10 % 3 * 3 - (1 + 2)
c. ((1 + 2) * 3)
d. (5 == 5)
e. (x = 5)
8. If x = 4, y = 6, and z = 2, determine whether each of the following evaluates to true or false.
a. if(x == 4)
b. if(x != y - z)
c. if(z = 1)
d. if(y)
9. Write an if statement that determines whether someone is legally an adult (age 21), but not a senior citizen (age 65).
10. BUG BUSTER: Fix the following program so that it runs correctly.

```
/* a program with problems... */
```

```
#include <stdio.h>
int x= 1:
main()
{
   if( x = 1);
      printf(" x equals 1" );
   otherwise
      printf(" x does not equal 1");
   return 0;
}
```