CSC 111: Introduction to programming

Statements, Expressions, and Operators

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. 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.

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!");
```

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.

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.

Expressions

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

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.

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;

Figure 4.1. An assignment statement is itself an expression.

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.

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:
The assignment operator

Mathematical operators
Relational operators
Logical operators
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.

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.

Unary 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.

```
/* Demonstrates unary operator prefix and postfix modes */

#include <stdio.h>

int a, b;

main()
{
   /* Set a and b both equal to 5 */

    a = b = 5;

    /* 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.

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.

```
/* Illustrates the modulus operator. */
/* Inputs a number of seconds, and converts to hours, */
/* minutes, and seconds. */

#include <stdio.h>

/* Define constants */

#define SECS_PER_MIN 60
```

```
#define SECS_PER_HOUR 3600

unsigned seconds, minutes, hours, secs_left, mins_left;

main()
{
   /* Input the number of seconds */

   printf("Enter number of seconds (< 65000): ");
   scanf("%d", &seconds);

   hours = seconds / SECS_PER_HOUR;
   minutes = seconds / SECS_PER_MIN;
   mins_left = minutes % SECS_PER_MIN;
   secs_left = seconds % SECS_PER_MIN;

   printf("%u seconds is equal to ", seconds);
   printf("%u h, %u m, and %u s\n", hours, mins_left, secs_left);

   return 0;
}
```

```
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
```

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:
Unary increment and decrement
Multiplication, division, and modulus

Addition and subtraction

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.

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.

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 4.4. 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)

PROGRAM CONTROL

THE WHILE LOOP

The C programming language has several structures for looping and conditional branching. We will cover them all in this chapter and we will begin with the while loop.

The while loop continues to loop while some condition is true. When the condition becomes false, the looping is discontinued. It therefore does just what it says it does, the name of the loop being very descriptive.

Example program ------> WHILE.C

Load the program WHILE.C and display it for an example of a while loop. We begin with a comment and the program entry point main (), then go on to define an integer variable named count within the body of the program. The variable is set to zero and we come to the while loop itself. The syntax of a while loop is just as shown here. The keyword while is followed by an expression of something in parentheses, followed by a compound statement bracketed by braces. As long as the expression in the parenthesis is true, all statements within the braces will be repeatedly executed. In this case, since the variable count is incremented by one every time the statements are executed, it will eventually reach 6. At that time the statement will not be executed because count is not less than 6, and the loop will be terminated. The program control will resume at the statement following the statements in braces.

We will cover the compare expression, the one in parentheses, in the next chapter. Until then, simply accept the expressions for what you think they should do and you will be correct for these simple cases.

Several things must be pointed out regarding the while loop. First, if the variable count were initially set to any number greater than 5, the statements within the loop would not be executed at all, so it is possible to have a while loop that never is executed. Secondly, if the variable were not incremented in the loop, then in this case, the loop would never terminate, and the program would never complete. Finally, if there is only one statement to be executed within the loop, it does not need delimiting braces but can stand alone.

Compile and run this program after you have studied it enough to assure yourself that you understand its operation completely. Note that the result of execution is given for this program, (and will be given for the entire remaining example programs in this tutorial) so you do not need to compile and execute every program to see the results. Be sure to compile and execute some of the programs however, to gain experience with your compiler.

You should make some modifications to any programs that are not completely clear to you and compile them until you understand them completely. The best way to learn is to try various modifications yourself.

We will continue to ignore the #include statement and the return statement in the example programs in this chapter. We will define them completely later in this tutorial.

THE DO-WHILE LOOP

Example program ------> DOWHILE.C

A variation of the while loop is illustrated in the program DOWHILE.C, which you should load and display. This program is nearly identical to the last one except that the loop begins with the keyword do, followed by a compound statement in braces, then the keyword while, and finally an expression in parentheses. The statements in the braces are executed repeatedly as long as the expression in the parentheses is true. When the expression in parentheses becomes false, execution is terminated, and control passes to the statements following this statement.

THE FOR LOOP

Example program ------> FORLOOP.C

Load and display the file named FORLOOP.C on your monitor for an example of a program with a for loop. The for loop consists of the keyword for followed by a rather large expression in parentheses. This expression is really composed of three fields separated by semi-colons. The first field contains the expression "index = 0" and is an initializing field. Any expressions in this field are executed prior to the first pass through the loop. There is essentially no limit as to what can go here, but good programming practice would require it to be kept simple. Several initializing statements can be placed in this field, separated by commas.

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.

```
/* Demonstrates the use of if statements */

#include <stdio.h>

int x, y;

main()
{
    /* Input the two values to be tested */

     printf("\nInput an integer value for x: ");
    scanf("%d", &x);
    printf("\nInput an integer value for y: ");
    scanf("%d", &y);

    /* Test values and print result */

    if (x == y)
        printf("x is equal to y\n");

    if (x > y)
        printf("x is greater than y\n");

    if (x < y)
        printf("x is smaller than y\n");

    return 0;
}
```

```
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
```

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.

```
/* Demonstrates the use of if statement with else clause */

#include <stdio.h>

int x, y;

main()
{
   /* Input the two values to be tested */

    printf("\nInput an integer value for x: ");
    scanf("%d", &x);
    printf("\nInput an integer value for y: ");
    scanf("%d", &y);

    /* Test values and print result */

    if (x == y)
        printf("x is equal to y\n");
    else
        if (x > y)
            printf("x is greater than y\n");
        else
            printf("x is smaller than y\n");

    return 0;
}
```

```
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
```

The if Statement

```
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.

```
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.

```
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");
```

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.

```
/* Demonstrates the evaluation of relational expressions */

#include <stdio.h>

int a;

main()
{
   a = (5 == 5);            /* Evaluates to 1 */
   printf("\na = (5 == 5)\na = %d", a);

   a = (5 != 5);            /* Evaluates to 0 */
   printf("\na = (5 != 5)\na = %d", a);

   a = (12 == 12) + (5 != 1); /* Evaluates to 1 + 1 */
   printf("\na = (12 == 12) + (5 != 1)\na = %d\n", a);
   return 0;
}
```

```
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 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 are 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.

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)
```

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:
A value of zero represents false.

Any nonzero value represents true.

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)

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.

```
#include <stdio.h>

/* Initialize variables. Note that c is not less than d, */
/* which is one of the conditions to test for. */
/* Therefore, the entire expression should evaluate as false.*/

int a = 5, b = 6, c = 5, d = 1;
int x;

main()
{
    /* Evaluate the expression without parentheses */

    x = a < b || a < c && c < d;
    printf("\nWithout parentheses the expression evaluates as %d", x);

17:     /* Evaluate the expression with parentheses */
18:
19:     x = (a < b || a < c) && c < d;
20:     printf("\nWith parentheses the expression evaluates as %d\n", x);
21:     return 0;
22: }
Without parentheses the expression evaluates as 1
With parentheses the expression evaluates as 0
```

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.

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;
```

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) );
```

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:
Both expressions are evaluated, with the left expression being evaluated first.

The entire expression evaluates to the value of the right expression.

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.

Functions: The Basics

Functions are central to C programming and to the philosophy of C program design. You've already been introduced to some of C's library functions, which are complete functions supplied as part of your compiler. This chapter covers user-defined functions, which, as the name implies, are functions that you, the programmer, create. Today you will learn

What a function is and what its parts are

About the advantages of structured programming with functions

How to create a function

How to declare local variables in a function

How to return a value from a function to the program

How to pass arguments to a function

<u>**What Is a Function?**</u>

This chapter approaches the question "What is a function?" in two ways. First, it tells you what functions are, and then it shows you how they're used.

A Function Defined

First the definition: A function is a named, independent section of C code that performs a specific task and optionally returns a value to the calling program. Now let's look at the parts of this definition:

A function is named. Each function has a unique name. By using that name in another part of the program, you can execute the statements contained in the function. This is known as calling the function. A function can be called from within another function.

A function is independent. A function can perform its task without interference from or interfering with other parts of the program.

A function performs a specific task. This is the easy part of the definition. A task is a discrete job that your program must perform as part of its overall operation, such as sending a line of text to a printer, sorting an array into numerical order, or calculating a cube root.

A function can return a value to the calling program. When your program calls a function, the statements it contains are executed. If you want them to, these statements can pass information back to the calling program.

That's all there is to the "telling" part. Keep the previous definition in mind as you look at the next section.

A Function Illustrated

Listing 5.1 contains a user-defined function.

Listing 5.1. A program that uses a function to calculate the cube of a number.
```
/* Demonstrates a simple function */
#include <stdio.h>

long cube(long x);

long input, answer;

main()
{
   printf("Enter an integer value: ");
   scanf("%d", &input);
   answer = cube(input);
   /* Note: %ld is the conversion specifier for */
   /* a long integer */
   printf("\nThe cube of %ld is %ld.\n", input, answer);

17:     return 0;
18:  }
19:
20:  /* Function: cube() - Calculates the cubed value of a variable */
21:  long cube(long x)
22:  {
23:     long x_cubed;
24:
25:     x_cubed = x * x * x;
26:     return x_cubed;
27:  }
```

```
Enter an integer value: 100
The cube of 100 is 1000000.
Enter an integer value: 9
The cube of 9 is 729.
Enter an integer value: 3
The cube of 3 is 27.
```

NOTE: The following analysis focuses on the components of the program that relate directly to the function rather than explaining the entire program.

ANALYSIS: Line 4 contains the function prototype, a model for a function that will appear later in the program. A function's prototype contains the name of the function, a list of variables that must be passed to it, and the type of variable it returns, if any. Looking at line 4, you can tell that the function is named cube, that it requires a variable of the type long, and that it will return a value of type long. The variables to be passed to the function are called arguments, and they are enclosed in parentheses following the function's name. In this example, the function's argument is long x. The keyword before the name of the function indicates the type of variable the function returns. In this case, a type long variable is returned.

Line 12 calls the function cube and passes the variable input to it as the function's argument. The function's return value is assigned to the variable answer. Notice that both input and answer are declared on line 6 as long variables, in keeping with the function prototype on line 4.

The function itself is called the function definition. In this case, it's called cube and is contained in lines 21 through 27. Like the prototype, the function definition has several parts. The function starts out with a function header on line 21. The function header is at the start of a function, and it gives the function's name (in this case, the name is cube). The header also gives the function's return type and describes its arguments. Note that the function header is identical to the function prototype (minus the semicolon).

The body of the function, lines 22 through 27, is enclosed in braces. The body contains statements, such as on line 25, that are executed whenever the function is called. Line 23 is a variable declaration that looks like the declarations you have seen before, with one difference: it's local. Local variables are declared within a function body. (Local declarations are discussed further on Day 12, "Understanding Variable Scope.") Finally, the function concludes with a return statement on line 26, which signals the end of the function. A return statement also passes a value back to the calling program. In this case, the value of the variable x_cubed is returned.

If you compare the structure of the cube() function with that of the main() function, you'll see that they are the same. main() is also a function. Other functions that you already have used are printf() and scanf(). Although printf() and scanf() are library functions (as opposed to user-defined functions), they are functions that can take arguments and return values just like the functions you create.

Function Prototype

```
return_type function_name( arg-type name-1,...,arg-type name-n);
```

Function Definition

```
return_type function_name( arg-type name-1,...,arg-type name-n)
{
   /* statements; */
}
```

A function prototype provides the compiler with a description of a function that will be defined at a later point in the program. The prototype includes a return type indicating the type of variable that the function will return. It also includes the function name, which should describe what the function does. The prototype also contains the variable types of the arguments (arg type) that will be passed to the function. Optionally, it can contain the names of the variables that will be passed. A prototype should always end with a semicolon.

A function definition is the actual function. The definition contains the code that will be executed. The first line of a function definition, called the function header, should be identical to the function prototype, with the exception of the semicolon. A function header shouldn't end with a semicolon. In addition, although the argument variable names were optional in the prototype, they must be included in the function header. Following the header is the function body, containing the statements that the function will perform. The function body should start with an opening bracket and end with a closing bracket. If the function return type is anything other than void, a return statement should be included, returning a value matching the return type.

Function Prototype Examples

```
double squared( double number );
void print_report( int report_number );
int get_menu_choice( void );
```

Function Definition Examples

```
double squared( double number )         /* function header */
{                                    /* opening bracket */
   return(  number * number );        /* function body   */
}                                    /* closing bracket */
void print_report( int report_number )
{
   if( report_number == 1 )
      puts( "Printing Report 1" );
   else
      puts( "Not printing Report 1" );
}
```

Functions and Structured Programming

By using functions in your C programs, you can practice structured programming, in which individual program tasks are performed by independent sections of program code. "Independent sections of program code" sounds just like part of the definition of functions given earlier, doesn't it? Functions and structured programming are closely related.

The Advantages of Structured Programming

Why is structured programming so great? There are two important reasons:

It's easier to write a structured program, because complex programming problems are broken into a number of smaller, simpler tasks. Each task is performed by a function in which code and variables are isolated from the rest of the program. You can progress more quickly by dealing with these relatively simple tasks one at a time.

It's easier to debug a structured program. If your program has a bug (something that causes it to work improperly), a structured design makes it easy to isolate the problem to a specific section of code (a specific function).

A related advantage of structured programming is the time you can save. If you write a function to perform a certain task in one program, you can quickly and easily use it in another program that needs to execute the same task. Even if the new program needs to accomplish a slightly different task, you'll often find that modifying a function you created earlier is easier than writing a new one from scratch. Consider how much you've used the two functions printf() and scanf() even though you probably haven't seen the code they contain. If your functions have been created to perform a single task, using them in other programs is much easier.

Planning a Structured Program

If you're going to write a structured program, you need to do some planning first. This planning should take place before you write a single line of code, and it usually can be done with nothing more than pencil and paper. Your plan should be a list of the specific tasks your program performs. Begin with a global idea of the program's function. If you were planning a program to manage your name and address list, what would you want the program to do? Here are some obvious things:

Enter new names and addresses.

Modify existing entries.

Sort entries by last name.

Print mailing labels.

With this list, you've divided the program into four main tasks, each of which can be assigned to a function. Now you can go a step further, dividing these tasks into subtasks. For example, the "Enter new names and addresses" task can be subdivided into these subtasks:

Read the existing address list from disk.

Prompt the user for one or more new entries.

Add the new data to the list.

Save the updated list to disk.

Likewise, the "Modify existing entries" task can be subdivided as follows:

Read the existing address list from disk.

Modify one or more entries.

Save the updated list to disk.

You might have noticed that these two lists have two subtasks in common--the ones dealing with reading from and saving to disk. You can write one function to "Read the existing address list from disk," and that function can be called by both the "Enter new names and addresses" function and the "Modify existing entries" function. The same is true for "Save the updated list to disk."

Already you should see at least one advantage of structured programming. By carefully dividing the program into tasks, you can identify parts of the program that share common tasks. You can write "double-duty" disk access functions, saving yourself time and making your program smaller and more efficient.

This method of programming results in a hierarchical, or layered, program structure. Figure 5.2 illustrates hierarchical programming for the address list program.

Figure 5.2. A structured program is organized hierarchically.

When you follow this planned approach, you quickly make a list of discrete tasks that your program needs to perform. Then you can tackle the tasks one at a time, giving all your attention to one relatively simple task. When that function is written and working properly, you can move on to the next task. Before you know it, your program starts to take shape.

The Top-Down Approach

By using structured programming, C programmers take the top-down approach. You saw this illustrated in Figure 5.2, where the program's structure resembles an inverted tree. Many times, most of the real work of the program is performed by the functions at the tips of the "branches." The functions closer to the "trunk" primarily direct program execution among these functions.

As a result, many C programs have a small amount of code in the main body of the program--that is, in main(). The bulk of the program's code is found in functions. In main(), all you might find are a few dozen lines of code that direct program execution among the functions. Often, a menu is presented to the person using the program. Program execution is branched according to the user's choices. Each branch of the menu uses a different function.

This is a good approach to program design. Day 13, "Advanced Program Control," shows how you can use the switch statement to create a versatile menu-driven system.

Now that you know what functions are and why they're so important, the time has come for you to learn how to write your own.

DO plan before starting to code. By determining your program's structure ahead of time, you can save time writing the code and debugging it.

DON'T try to do everything in one function. A single function should perform a single task, such as reading information from a file.
Writing a Function

The first step in writing a function is knowing what you want the function to do. Once you know that, the actual mechanics of writing the function aren't particularly difficult.

The Function Header

The first line of every function is the function header, which has three components, each serving a specific function. They are shown in Figure 5.3 and explained in the following sections.

Figure 5.3. The three components of a function header.

The Function Return Type

The function return type specifies the data type that the function returns to the calling program. The return type can be any of C's data types: char, int, long, float, or double. You can also define a function that doesn't return a value by using a return type of void. Here are some examples:

```
int func1(...)          /* Returns a type int.   */
float func2(...)        /* Returns a type float. */
void func3(...)         /* Returns nothing.      */
```
The Function Name

You can name a function anything you like, as long as you follow the rules for C variable names (given in Day 3, "Storing Data: Variables and Constants"). A function name must be unique (not assigned to any other function or variable). It's a good idea to assign a name that reflects what the function does.

The Parameter List

Many functions use arguments, which are values passed to the function when it's called. A function needs to know what kinds of arguments to expect--the data type of each argument. You can pass a function any of C's data types. Argument type information is provided in the function header by the parameter list.

For each argument that is passed to the function, the parameter list must contain one entry. This entry specifies the data type and the name of the parameter. For example, here's the header from the function in Listing 5.1:

```
long cube(long x)
```

The parameter list consists of long x, specifying that this function takes one type long argument, represented by the parameter x. If there is more than one parameter, each must be separated by a comma. The function header

```
void func1(int x, float y, char z)
```

specifies a function with three arguments: a type int named x, a type float named y, and a type char named z. Some functions take no arguments, in which case the parameter list should consist of void, like this:

```
void func2(void)
```

NOTE: You do not place a semicolon at the end of a function header. If you mistakenly include one, the compiler will generate an error message.

Sometimes confusion arises about the distinction between a parameter and an argument. A parameter is an entry in a function header; it serves as a "placeholder" for an argument. A function's parameters are fixed; they do not change during program execution.

An argument is an actual value passed to the function by the calling program. Each time a function is called, it can be passed different arguments. In C, a function must be passed the same number and type of arguments each time it's called, but the argument values can be different. In the function, the argument is accessed by using the corresponding parameter name.

An example will make this clearer. Listing 5.2 presents a very simple program with one function that is called twice.

Listing 5.2. The difference between arguments and parameters.
```
/* Illustrates the difference between arguments and parameters. */

#include <stdio.h>

float x = 3.5, y = 65.11, z;

float half_of(float k);

main()
{
    /* In this call, x is the argument to half_of(). */
    z = half_of(x);
    printf("The value of z = %f\n", z);

    /* In this call, y is the argument to half_of(). */
    z = half_of(y);
17:     printf("The value of z = %f\n", z);
18:
19:     return 0;
20:  }
21:
22:  float half_of(float k)
23:  {
24:     /* k is the parameter. Each time half_of() is called, */
25:     /* k has the value that was passed as an argument. */
26:
27:     return (k/2);
28:  }
```

```
The value of z = 1.750000
The value of z = 32.555000
```

Figure 5.4 shows the relationship between arguments and parameters.

Figure 5.4. Each time a function is called, the arguments are passed to the function's parameters.

ANALYSIS: Looking at Listing 5.2, you can see that the half_of() function prototype is declared on line 7. Lines 12 and 16 call half_of(), and lines 22 through 28 contain the actual function. Lines 12 and 16 each send a different argument to half_of(). Line 12 sends x, which contains a value of 3.5, and line 16 sends y, which contains a value of 65.11. When the program runs, it prints the correct number for each. The values in x and y are passed into the argument k of half_of(). This is like copying the values from x to k, and then from y to k. half_of() then returns this value after dividing it by 2 (line 27).

DO use a function name that describes the purpose of the function.

DON'T pass values to a function that it doesn't need.

DON'T try to pass fewer (or more) arguments to a function than there are parameters. In C programs, the number of arguments passed must match the number of parameters.
The Function Body

The function body is enclosed in braces, and it immediately follows the function header. It's here that the real work is done. When a function is called, execution begins at the start of the function body and terminates (returns to the calling program) when a return statement is encountered or when execution reaches the closing brace.

Local Variables

You can declare variables within the body of a function. Variables declared in a function are called local variables. The term local means that the variables are private to that particular function and are distinct from other variables of the same name declared elsewhere in the program. This will be explained shortly; for now, you should learn how to declare local variables.

Here is an example of four local variables being declared within a function:

```
int func1(int y)
{
   int a, b = 10;
   float rate;
   double cost = 12.55;
   /* function code goes here... */
}
```

The preceding declarations create the local variables a, b, rate, and cost, which can be used by the code in the function. Note that the function parameters are considered to be variable declarations, so the variables, if any, in the function's parameter list also are available.

When you declare and use a variable in a function, it is totally separate and distinct from any other variables that are declared elsewhere in the program. This is true even if the variables have the same name. Listing 5.3 demonstrates this independence.

Listing 5.3. A demonstration of local variables.

```
/* Demonstrates local variables. */

#include <stdio.h>

int x = 1, y = 2;

void demo(void);

main()
{
  printf("\nBefore calling demo(), x = %d and y = %d.", x, y);
  demo();
  printf("\nAfter calling demo(), x = %d and y = %d\n.", x, y);

  return 0;
}

void demo(void)
{
   /* Declare and initialize two local variables. */

   int x = 88, y = 99;

   /* Display their values. */

   printf("\nWithin demo(), x = %d and y = %d.", x, y);
}
```

```
Before calling demo(), x = 1 and y = 2.
Within demo(), x = 88 and y = 99.
After calling demo(), x = 1 and y = 2.
```

Returning a Value

To return a value from a function, you use the return keyword, followed by a C expression. When execution reaches a return statement, the expression is evaluated, and execution passes the value back to the calling program. The return value of the function is the value of the expression. Consider this function:

```
int func1(int var)
{
   int x;
   /* Function code goes here... */
   return x;
}
```

When this function is called, the statements in the function body execute up to the return statement. The return terminates the function and returns the value of x to the calling program. The expression that follows the return keyword can be any valid C expression.

A function can contain multiple return statements. The first return executed is the only one that has any effect. Multiple return statements can be an efficient way to return different values from a function, as demonstrated in Listing 5.4.

Listing 5.4. Using multiple return statements in a function.

```
/* Demonstrates using multiple return statements in a function. */

#include <stdio.h>

int x, y, z;

int larger_of( int , int );

main()
{
    puts("Enter two different integer values: ");
    scanf("%d%d", &x, &y);

    z = larger_of(x,y);

    printf("\nThe larger value is %d.", z);

    return 0;
}

int larger_of( int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

```
Enter two different integer values:
200 300
The larger value is 300.
Enter two different integer values:
300
200
The larger value is 300.
```

The Function Prototype

A program must include a prototype for each function it uses. You saw an example of a function prototype on line 4 of Listing 5.1, and there have been function prototypes in the other listings as well. What is a function prototype, and why is it needed?

The prototype's job is to tell the compiler about the function's return type, name, and parameters. With this information, the compiler can check every time your source code calls the function and verify that you're passing the correct number and type of arguments to the function and using the return value correctly. If there's a mismatch, the compiler generates an error message.

Strictly speaking, a function prototype doesn't need to exactly match the function header. The parameter names can be different, as long as they are the same type, number, and in the same order. There's no reason for the header and prototype not to match; having them identical makes source code easier to understand. Matching the two also makes *writing a program easier. When you complete a function definition, use your editor's cut-and-paste feature to copy the function header and create the prototype. Be sure to add a semicolon at the end***.**

Where should function prototypes be placed in your source code? They should be placed before the start of main () or before the first function is defined. For readability, it's best to group all prototypes in one location.

Passing Arguments to a Function

To pass arguments to a function, you list them in parentheses following the function name. The number of arguments and the type of each argument must match the parameters in the function header and prototype. For example, if a function is defined to take two type int arguments, you must pass it exactly two int arguments--no more, no less--and no other type. If you try to pass a function an incorrect number and/or type of argument, the compiler will detect it, based on the information in the function prototype.

If the function takes multiple arguments, the arguments listed in the function call are assigned to the function parameters in order: the first argument to the first parameter, the second argument to the second parameter, and so on, as shown in Figure 5.5.

Figure 5.5. Multiple arguments are assigned to function parameters in order.

Each argument can be any valid C expression: a constant, a variable, a mathematical or logical expression, or even another function (one with a return value). For example, if half(), square(), and third() are all functions with return values, you could write

```
x = half(third(square(half(y))));
```

The program first calls half(), passing it y as an argument. When execution returns from half(), the program calls square(), passing half()'s return value as an argument. Next, third() is called with square()'s return value as the argument. Then, half() is called again, this time with third()'s return value as an argument. Finally, half()'s return value is assigned to the variable x. The following is an equivalent piece of code:

```
a = half(y);
b = square(a);
c = third(b);
x = half(c);
```

Calling Functions

There are two ways to call a function. Any function can be called by simply using its name and argument list alone in a statement, as in the following example. If the function has a return value, it is discarded.

```
wait(12);
```

The second method can be used only with functions that have a return value. Because these functions evaluate to a value (that is, their return value), they are valid C expressions and can be used anywhere a C expression can be used. You've already seen an expression with a return value used as the right side of an assignment statement. Here are some more examples.

In the following example, half_of() is a parameter of a function:

```
printf("Half of %d is %d.", x, half_of(x));
```

First, the function half_of() is called with the value of x, and then printf() is called using the values x and half_of(x).

In this second example, multiple functions are being used in an expression:

```
y = half_of(x) + half_of(z);
```

Although half_of() is used twice, the second call could have been any other function. The following code shows the same statement, but not all on one line:

```
a = half_of(x);
b = half_of(z);
y = a + b;
```

The final two examples show effective ways to use the return values of functions. Here, a function is being used with the if statement:

```
if ( half_of(x) > 10 )
{
   /* statements; */          /* these could be any statements! */
}
```

If the return value of the function meets the criteria (in this case, if half_of() returns a value greater than 10), the if statement is true, and its statements are executed. If the returned value doesn't meet the criteria, the if's statements are not executed.

The following example is even better:

```
if ( do_a_process() != OKAY )
{
   /* statements; */          /* do error routine */
}
```

Recursion

The term recursion refers to a situation in which a function calls itself either directly or indirectly. Indirect recursion occurs when one function calls another function that then calls the first function. C allows recursive functions, and they can be useful in some situations.

For example, recursion can be used to calculate the factorial of a number. The factorial of a number x is written x! and is calculated as follows:

x! = x * (x-1) * (x-2) * (x-3) * ... * (2) * 1

However, you can also calculate x! like this:

x! = x * (x-1)!

Going one step further, you can calculate (x-1)! using the same procedure:

(x-1)! = (x-1) * (x-2)!

You can continue calculating recursively until you're down to a value of 1, in which case you're finished. The program in Listing 5.5 uses a recursive function to calculate factorials. Because the program uses unsigned integers, it's limited to an input value of 8; the factorial of 9 and larger values are outside the allowed range for integers.

Listing 5.5. Using a recursive function to calculate factorials.

```
/* Demonstrates function recursion. Calculates the */
/* factorial of a number. */

#include <stdio.h>
```

```
5:
6:   unsigned int f, x;
7:   unsigned int factorial(unsigned int a);
8:
9:   main()
10:  {
11:      puts("Enter an integer value between 1 and 8: ");
12:      scanf("%d", &x);
13:
14:      if( x > 8 || x < 1)
15:      {
16:          printf("Only values from 1 to 8 are acceptable!");
17:      }
18:      else
19:      {
20:          f = factorial(x);
21:          printf("%u factorial equals %u\n", x, f);
22:      }
23:
24:      return 0;
25:  }
26:
27:  unsigned int factorial(unsigned int a)
28:  {
29:      if (a == 1)
30:          return 1;
31:      else
32:      {
33:          a *= factorial(a-1);
34:          return a;
35:      }
36:  }
```

```
Enter an integer value between 1 and 8:
6
6 factorial equals 720
```

ANALYSIS: The first half of this program is like many of the other programs you have worked with so far. It starts with comments on lines 1 and 2. On line 4, the appropriate header file is included for the input/output routines. Line 6 declares a couple of unsigned integer values. Line 7 is a function prototype for the factorial function. Notice that it takes an unsigned int as its parameter and returns an unsigned int. Lines 9 through 25 are the main() function. Lines 11 and 12 print a message asking for a value from 1 to 8 and then accept an entered value.

Lines 14 through 22 show an interesting if statement. Because a value greater than 8 causes a problem, this if statement checks the value. If it's greater than 8, an error message is printed; otherwise, the program figures the factorial on line 20 and prints the result on line 21. When you know there could be a problem, such as a limit on the size of a number, add code to detect the problem and prevent it.

Our recursive function, factorial(), is located on lines 27 through 36. The value passed is assigned to a. On line 29, the value of a is checked. If it's 1, the program returns the value of 1. If the value isn't 1, a is set equal to itself times the value of factorial(a-1). The program calls the factorial function again, but this time the value of a is (a-1). If (a-1) isn't equal to 1, factorial() is called again with ((a-1)-1), which is the same as (a-2). This process continues until the if statement on line 29 is true. If the value of the factorial is 3, the factorial is evaluated to the following:

3 * (3-1) * ((3-1)-1)

Summary

This chapter introduced you to functions, an important part of C programming. Functions are independent sections of code that perform specific tasks. When your program needs a task performed, it calls the function that performs that task. The use of functions is essential for structured programming--a method of program design that emphasizes a modular, top-down approach. Structured programming creates more efficient programs and also is much easier for you, the programmer, to use.

You also learned that a function consists of a header and a body. The header includes information about the function's return type, name, and parameters. The body contains local variable declarations and the C statements that are executed when the function is called. Finally, you saw that local variables--those declared within a function--are totally independent of any other program variables declared elsewhere.

Q&A

Q What if I need to return more than one value from a function?

A Many times you will need to return more than one value from a function, or, more commonly, you will want to change a value you send to the function and keep the change after the function ends. This process is covered on Day 18, "Getting More from Functions."

Q How do I know what a good function name is?

A A good function name describes as specifically as possible what the function does.

Q When variables are declared at the top of the listing, before main(), they can be used anywhere, but local variables can be used only in the specific function. Why not just declare everything before main()?

A Variable scope is discussed in more detail on Day 12.

Q What other ways are there to use recursion?

A The factorial function is a prime example of using recursion. The factorial number is needed in many statistical calculations. Recursion is just a loop; however, it has one difference from other loops. With recursion, each time a recursive function is called, a new set of variables is created. This is not true of the other loops that you will learn about in the next chapter.

Q Does main() have to be the first function in a program?

A No. It is a standard in C that the main() function is the first function to execute; however, it can be placed anywhere in your source file. Most people place it first so that it's easy to locate.

Q What are member functions?

A Member functions are special functions used in C++ and Java. They are part of a class--which is a special type of structure used in C++ and Java.