Sams Teach Yourself C in 24 Hours

Hour 6 - Manipulating Data with Operators

"The question is," said Humpty Dumpty, "which is to be master—that's all."

—L. Carroll

You can think of operators as verbs in C that let you manipulate data. In fact, you've learned some operators, such as + (addition), - (subtraction), * (multiplication), / (division), and % (remainder), in Hour 3, "The Essentials of C Programs." The C language has a rich set of operators. In this hour, you'll learn about more operators, such as

- Arithmetic assignment operators
- Unary minus operators
- Increment and decrement operators
- Relational operators
- Cast operator

Arithmetic Assignment Operators

Before jumping into the arithmetic assignment operators, you first need to learn more about the assignment operator.

The Assignment Operator (=)

In the C language, the = operator is called an assignment operator, which you've seen and used for several hours.

The general statement form to use an assignment operator is

left-hand-operand = right-hand-operand;

Here the statement causes the value of the right-hand-operand to be assigned (or written) to the memory location of the left-hand-operand. Additionally, the assignment statement itself returns the same value that is assigned to the left-hand-operand.

For example, the a = 5; statement writes the value of the right-hand operand (5) into the memory location of the integer variable a (which is the left-hand operand in this case).

Similarly, the b = a = 5; statement assigns 5 to the integer variable a first, and then to the integer variable b. After the execution of the statement, both a and b contain the value of 5.

WARNING

> Don't confuse the assignment operator (=) with the relational operator, == (called the *equal-to operator*). The == operator is introduced later in this hour.

Combining Arithmetic Operators with =

Consider this example: Given two integer variables, x and y, how do you assign the sum of x and y to another integer variable, z?

By using the assignment operator (=) and the addition operator (+), you get the following statement:

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

As you can see, it's pretty simple. Now, consider the same example again. This time, instead of assigning the result to the third variable, z, let's write the sum back to the integer variable, x:

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

Here, on the right side of the assignment operator (=), the addition of x and y is executed; on the left side of =, the previous value saved by x is replaced with the result of the addition from the right side.

The C language gives you a new operator, +=, to do the addition and the assignment together. Therefore, you can rewrite the x = x + y; statement to

```
x += y;
```

The combinations of the assignment operator (=) with the arithmetic operators, +, -, *, /, and %, give you another type of operators—arithmetic assignment operators:

Operator	Description
+=	Addition assignment operator
-=	Subtraction assignment operator
*=	Multiplication assignment operator
/=	Division assignment operator
%=	Remainder assignment operator

The following shows the equivalence of statements:

x += y; is equivalent to x = x + y;
x -= y; is equivalent to x = x - y;
x *= y; is equivalent to x = x * y;
x /= y; is equivalent to x = x / y;
x %= y; is equivalent to x = x % y;

Note that the statement

z = z * x + y;

is not equivalent to the statement

z *= x + y;

because

z *= x + y

is indeed the same as

z = z * (x + y);

Listing 6.1 gives an example of using some of the arithmetic assignment operators.

TYPE
Listing 6.1. Using arithmetic assignment operators.

```
1:  /* 06L01.c: Using arithmetic assignment operators */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int x, y, z;
7:
8:     x = 1;   /* initialize x */
9:     y = 3;   /* initialize y */
10:    z = 10;  /* initialize z */
11:    printf("Given x = %d, y = %d, and z = %d,\n", x, y, z);
12:
13:    x = x + y;
14:    printf("x = x + y  assigns %d to x;\n", x);
15:
16:    x = 1;  /* reset x */
17:    x += y;
18:    printf("x += y  assigns %d to x;\n", x);
19:
20:    x = 1;  /* reset x */
21:    z = z * x + y;
22:    printf("z = z * x + y  assigns %d to z;\n", z);
23:
24:    z = 10;  /* reset z */
25:    z = z * (x + y);
26:    printf("z = z * (x + y) assigns %d to z;\n", z);
27:
28:    z = 10;  /* reset z */
29:    z *= x + y;
30:    printf("z *= x + y assigns %d to z.\n", z);
31:
32:    return 0;
33: }
```

OUTPUT
After this program is compiled and linked, an executable file is created. On my machine, this executable file is named as 06L01.exe. The following is the output printed on the screen after I run the executable from a DOS prompt:

```
C:\app> 06L01
Given x = 1, y = 3, and z = 10,
x = x + y  assigns 4 to x;
x += y  assigns 4 to x;
z = z * x + y  assigns 13 to z;
z = z * (x + y) assigns 40 to z;
z *= x + y assigns 40 to z.
C:\app>
```

ANALYSIS
Line 2 in Listing 6.1 includes the header file stdio.h by using the include directive in C. The stdio.h header file is needed for the printf() function used in the main() function body in lines 4_33.

Lines 8_10 initialize three integer variables, x, y, and z, which are declared in line 6. Line 11 then prints out the initial values assigned to x, y, and z.

The statement in line 13 uses the one addition operator and one assignment operator to add the values contained by x and y, and then assigns the result to x. Line 14 displays the result on the screen.

Similarly, lines 17 and 18 do the same addition and display the result again, after the variable x is reset in line 16. This time, the arithmetic assignment operator, +=, is used. Also, line 16 in Listing 6.1 resets the value of x to 1, before the addition.

The value of x is reset again in line 20. Line 21 performs a multiplication and an addition and saves the result to the integer variable z; that is, z = z * x + y;. The printf() function in line 22 displays the result, 13, on the screen. Again, the x = 1; statement in line 20 resets the integer

variable, x.

Lines 24_30 display two results from two computations. The two results are actually the same (that is, 40), because the two computations in lines 25 and 29 are equivalent. The only difference between the two statements in lines 25 and 29 is that the arithmetic assignment operator, *=, is used in line 29.

Getting Negations of Numeric Numbers

If you want to change the sign of a numeric number, you can put the minus operator (-) right before the number. For instance, given an integer of 7, you can get its negation by changing the sign of the integer like this: -7. Here, - is the minus operator.

Precisely, - is called the unary minus operator in C. This is because the operator takes only one operand. The type of the operand can be any integer or floating-point number.

You can apply the unary minus operator to an integer or a floating-point variable as well. For example, given x = 1.234, -x equals -1.234. Or, given x = -1.234, -x equals 1.234.

WARNING

Don't confuse the unary minus operator with the subtraction operator, although both operators use the same symbol. For instance, the following statement:

```
z = x - -y;
```

```
z = x - (-y);
```

or this one:

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

Here, in both statements, the first - symbol is used as the subtraction operator, while the second - symbol is the unary minus operator.

Incrementing or Decrementing by One

The increment and decrement operators are very handy to use when you want to add or subtract 1 from a variable. The symbol for the increment operator is ++. The decrement operator is --.

For instance, you can rewrite the statement x = x + 1; as ++x;, or you can replace x = x 1; with --x;.

Actually, there are two versions of the increment operator and of the decrement operator. In the ++x; statement, the increment operator is called the pre-increment operator, because the operator adds 1 to x first and then gets the value of x. Likewise, in the --x; statement, the pre-decrement operator first subtracts 1 from x and then gets the value of x.

If you have an expression like x++, you're using the post-increment operator. Similarly, in x--, the decrement operator is called the post-decrement operator.

For example, in the y = x++; statement, y is assigned the original value of x, not the one after x is increased by 1. In other words, the post-increment operator makes a copy of the original value of x and stores the copy in a temporary location. Then, x is increased by 1. However, instead of the modified value of x, the copy of the unmodified value of x is returned and assigned to y.

The post-decrement operator has a similar story. This operator returns a copy of the original value of a variable, rather than the current value of the variable (which has been decreased by 1).

The program in Listing 6.2 shows the differences between the two versions of increment operators and decrement operators.

TYPE
Listing 6.2. Using pre- or post-increment and decrement operators.

```
1:  /* 06L02.c: pre- or post-increment(decrement) operators */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int w, x, y, z, result;
7:
8:     w = x = y = z = 1;   /* initialize x and y */
9:     printf("Given w = %d, x = %d, y = %d, and z = %d,\n", w, x, y, z);
10:
11:    result = ++w;
12:    printf("++w gives: %d\n", result);
13:    result = x++;
14:    printf("x++ gives: %d\n", result);
15:    result = --y;
16:    printf("--y gives: %d\n", result);
17:    result = z--;
18:    printf("z-- gives: %d\n", result);
19:    return 0;
20: }
```

OUTPUT
The following result is obtained by running the executable file 06L02.exe:

```
C:\app> 06L02
```

```
Given w = 1, x = 1, y = 1, and z = 1,
++w gives: 2
x++ gives: 1
--y gives: 0
z-- gives: 1
C:\app>
```

ANALYSIS
Inside the main() function, line 8 in Listing 6.2 assigns 1 to each of the integer variables, w, x, y, and z. The printf() function in line 9 displays the values contained by the four integer variables.

Then, the statement in line 11 is executed and the result of the pre-increment of w is given to the integer variable result. In line 12, the value of result, which is 2, is printed out to the screen.

Lines 13 and 14 get the post-increment of x and print out the result. As you know, the result is obtained before the value of x is increased. Therefore, you see the numeric value 1 from the result of x++ on the screen.

The pre-decrement operator in line 15 causes the value of y to be reduced by 1 before the value is assigned to the integer variable result. Therefore, you see 0 as the result of --y shown on the screen.

In line 17, however, the post-decrement operator has no effect on the assignment because the original value of z is given to the integer variable result before z is decreased by 1. Line 18 thus prints out 1, which is the original value of z.

Greater Than or Less Than?

There are six types of relationships between two expressions: equal to, not equal to, greater than, less than, greater than or equal to, and less than or equal to. Accordingly, the C language provides six relational operators:

Operator	Description
==	Equal to
!=	Not equal to
>	Greater than

continues

Operator	Description
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

All the relational operators have lower precedence than the arithmetic operators. Therefore, all arithmetic operations are carried out before any comparison is made. You should use parentheses to enclose operations of operators that have to be performed first.

Among the six operators, the >, <, >=, and <= operators have higher precedence than the == and != operators.

For example, the expression

x * y < z + 3

is interpreted as

(x * y) < (z + 3)

Another important thing is that all relational expressions produce a result of either 0 or 1. In other words, a relational expression returns 1 if the specified relationship holds. Otherwise, 0 is returned.

Given x = 3 and y = 5, for instance, the relational expression x < y gives a result of 1.

Listing 6.3 shows more examples of using relational operators.

TYPE
Listing 6.3. Results produced by relational expressions.

```
1:  /* 06L03.c: Using relational operators */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int x, y;
7:     double z;
8:
9:     x = 7;
10:    y = 25;
11:    z = 24.46;
12:    printf("Given x = %d, y = %d, and z = %.2f,\n", x, y, z);
13:    printf("x >= y  produces: %d\n", x >= y);
14:    printf("x == y  produces: %d\n", x == y);
15:    printf("x < z   produces: %d\n", x < z);
16:    printf("y > z   produces: %d\n", y > z);
17:    printf("x != y - 18  produces: %d\n", x != y - 18);
18:    printf("x + y != z   produces: %d\n", x + y != z);
19:    return 0;
20: }
```

OUTPUT

After the executable 06L03.exe is executed from a DOS prompt, the following output is displayed on the screen:

```
C:\app> 06L03
Given x = 7, y = 25, and z = 24.46,
x >= y  produces: 0
x == y  produces: 0
x < z   produces: 1
y > z   produces: 1
x != y - 18  produces: 0
x + y != z   produces: 1
C:\app>
```

ANALYSIS
There are two integer variables, x and y, and one floating-point variable z, declared in lines 6 and 7, respectively.

Lines 9_11 initialize the three variables. Line 12 prints out the values assigned to the variables.

Because the value of x is 7 and the value of y is 25, y is greater than x. Therefore, line 13 prints out 0, which is returned from the relational expression, x >= y.

Likewise, in line 14, the relational expression x == y returns 0.

Lines 15 and 16, however, print out the result of 1, returned from either x < z or y > z.

The statement in line 17 displays 0, which is the result of the relational expression x != y --18. In line 18, the expression x + y != z produces 1, which is output on the screen.

WARNING
Be careful when you compare two values for equality. Because of the truncation, or rounding up, some relational expressions, which are algebraically true, may return 0 instead of 1. For example, look at the following relational expression:

```
1 / 2 + 1 / 2 == 1
```

this is algebraically true and is supposed to return
The expression, however, returns 0, which means that the equal-to relationship does not hold. This is because the truncation of the integer division—that is, 1 / 2—produces 0, not 0.5.
Another example is 1.0 / 3.0, which produces 0.33333.... This is a number with an infinite number of decimal places. But the computer can only hold a limited number of decimal places. Therefore, the expression

```
1.0 / 3.0 + 1.0 / 3.0 + 1.0 / 3.0 == 1.0
```

might not return 1 on some computers, although the expression is theoretically true.

Playing with the Cast Operator

In C, you can convert one data type to a different one by prefixing the cast operator to the operand.

The general form of the cast operator is

(data-type)x

Here data-type specifies the data type you want to convert to. x is a variable (or, expression) that contains the value of the current data type. You have to include the parentheses (and) to make up a cast operator.

For example, the (float)5 expression converts the integer 5 to a floating-point number, 5.0.

The program in Listing 6.4 shows another example of using the cast operator.

TYPE
Listing 6.4. Using the cast operator.

```
1:  /* 06L04.c: Using the cast operator */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int x, y;
7:
8:     x = 7;
9:     y = 5;
10:    printf("Given x = %d, y = %d\n", x, y);
11:    printf("x / y produces: %d\n",  x / y);
12:    printf("(float)x / y produces: %f\n",  (float)x / y);
13:    return 0;
14: }
```

OUTPUT
The following output is obtained by running the executable 06L04.exe from a DOS prompt:

```
C:\app> 06L04
Given x = 7, y = 5
x / y produces: 1
(float)x / y produces: 1.400000
C:\app>
```

ANALYSIS
In Listing 6.4, there are two integer variables, x and y, declared in line 6, and initialized in lines 8 and 9, respectively. Line 10 then displays the values contained by the integer variables x and y.

The statement in line 11 prints out the integer division of x/y. Because the fractional part is truncated, the result of the integer division is 1.

However, in line 12, the cast operator (float) converts the value of x to a floating-point value. Therefore, the (float)x/y expression becomes a floating-point division that returns a floating-point number. That's why you see the floating-point number 1.400000 shown on the screen after the statement in line 12 is executed.

Summary

In this lesson you've learned about the following:

- The assignment operator (=), which has two operands on each side. The value of the right-side operand is assigned to the operand on the left side.
- The arithmetic assignment operators, +=, -=, *=, /=, and %=, which are the combinations of the arithmetic operators with the assignment operator.
- The unary minus operator (-), which returns the negation of a numeric value.
- The two versions of the increment operator, ++. You know that in ++x, the ++ operator is called the pre-increment operator; and in x++, ++ is the post-increment operator.
- The two versions of decrement operator, --. You have learned that, for example, in --x, the -- operator is the pre-decrement operator, while in x--, -- is called the post-decrement operator.
- The six relational operators in C: == (equal to), != (not equal to), > (greater than), < (less than), >= (greater than or equal to), and <= (less than or equal to).
- How to change the type of data by prefixing a cast operator to the data.

In the next lesson you'll learn about loops in the C language.

Q&A

Q What is the difference between the pre-increment operator and the post-increment operator?

A The pre-increment operator increases the operand's value by 1 first, and then returns the modified value. On the other hand, the post-increment operator stores a copy of the operand value in a temporary location and then increases the operand value by 1. However, the copy of the unmodified operand value is returned in the expression. For instance, given x = 1, the ++x expression returns 2, while the x++ expression returns 1.

Q Is the unary minus operator (-) the same as the subtraction operator (-)?

A No, they are not the same, although the two operators share the same symbol. The unary minus operator is used to change the sign of a value. In other words, the unary minus operator returns the negation of the value. The subtraction operator is an arithmetic operator that performs subtraction between its two operands.

Q Which one has a higher precedence, a relational operator or an arithmetic operator?

A An arithmetic operator has a higher precedence than a relational operator. For instance, the x * y + z > x + y expression is interpreted as ((x * y) + z) > (x + y).

Q What does a relational expression return?

A A relational expression returns either 0 or 1. If the relationship indicated by a relational operator in an expression is true, the expression returns 1; otherwise, the expression returns 0.

Workshop

To help solidify your understanding of this hour's lesson, you are encouraged to answer the quiz questions and finish the exercises provided in the Workshop before you move to the next lesson. The answers and hints to the questions and exercises are given in Appendix E, "Answers to Quiz Questions and Exercises."

Quiz

1. What is the difference between the = operator and the == operator?
2. In the x + - y - - z expression, which operator is the subtraction operator, and which one is the unary minus operator?
3. Given x = 15 and y = 4, what do the x / y and (float)x / y expressions return, respectively?
4. Is the y *= x + 5 expression equivalent to the y = y * x + 5 expression?

Exercises

1. Given x = 1 and y = 3, write a program to print out the results of these expressions: x += y, x += -y, x -= y, x -= -y, x *= y, and x *= -y.
2. Given x = 3 and y = 6, what is the value of z after the expression

```
z = x * y == 18 is executed?
```

3. Write a program that initializes the integer variable x with 1 and outputs results with the following two statements:

```
printf("x++ produces:   %d\n", x++);
printf("Now x contains: %d\n", x);
```

4. Rewrite the program you wrote in exercise 3. This time, include the following two statements:

```
printf("x = x++ produces: %d\n", x = x++);
printf("Now x contains:   %d\n", x);
```

What do you get after running the executable of the program? Can you explain why you get such a result?

5. The following program is supposed to compare the two variables, x and y, for equality. What's wrong with the program? (Hint: Run the program to see what it prints out.)

```
#include <stdio.h>

main()
{
   int x, y;

   x = y = 0;
   printf("The comparison result is: %d\n",  x = y);
   return 0;
}
```