Sams Teach Yourself C in 24 Hours

Hour 11 - An Introduction to Pointers

The duties of the Pointer were to point out, by calling their names, those in the congregation who should take note of some point made in the sermon.

—H. B. Otis, Simple Truth

You've learned about many important C data types, operators, functions, and loops in the last 10 hours. In this lesson you'll learn about one of the most important and powerful features in C: pointers. The topics covered in this
hour are

- Pointer variables
- Memory addresses
- The concept of indirection
- Declaring a pointer
- The address-of operator
- The dereference operator

What Is a Pointer?

A pointer is a variable whose value is used to point to another variable.

From this definition, you know two things: first, that a pointer is a variable, so you can assign different values to a pointer variable, and second, that the value contained by a pointer must be an address that indicates the location of another variable in the memory. That's why a pointer is also called an address variable.

Address (Left Value) Versus Content (Right Value)

As you might know, the memory inside your computer is used to hold the binary code of your program, which consists of statements and data, as well as the binary code of the operating system on your machine.

Each memory location must have a unique address so that the computer can read from or write to the memory location without any confusion. This is similar to the concept that each house in a city must have a unique address.

When a variable is declared, a piece of unused memory will be reserved for the variable, and the unique address to the memory will be associated with the name of the variable. The address associated with the variable name is usually called the left value of the variable.

Then, when the variable is assigned a value, the value is stored into the reserved memory location as the content. The content is also called the right value of the variable.

For instance, after the integer variable x is declared and assigned to a value like this:

```
int x;
x = 7;
```

the variable x now has two values:

Left value: 1000

Right value: 7

Here the left value, 1000, is the address of the memory location reserved for x. The right value, 7, is the content stored in the memory location. Note that depending on computers and operating systems, the right value of x can be different from one machine to another.

You can imagine that the variable x is the mailbox in front of your house, which has the address (normally the street number) 1000. The right value, 7, can be thought as a letter delivered to the mailbox.

Note that when your C program is being compiled and a value is being assigned to a variable, the C compiler has to check the left value of the variable. If the compiler cannot find the left value, it will issue an error message saying that the variable is undefined in your program. That's why, in C, you have to declare a variable before you can use it. (Imagine a postal worker complaining that he or she cannot drop the letters addressed to you because you haven't built a mailbox yet.)

By using a variable's left value, the C compiler can easily locate the appropriate memory storage reserved for a variable, and then read or write the right value of the variable.

The Address-of Operator (&)

The C language even provides you with an operator, &, in case you want to know the left value of a variable. This operator is called the address-of operator because it can return the address (that is, the left value) of a variable.

The following code, for example,

```
long int x, y;
y = &x;
```

assigns the address of the long integer variable x to another variable, y.

Listing 11.1 gives another example of obtaining addresses (that is, left values) of variables.

TYPE

Listing 11.1. Obtaining the left values of variables.

```
1:  /* 11L01.c: Obtaining addresses */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char  c;
7:     int   x;
8:     float y;
9:
10:    printf("c: address=0x%p, content=%c\n", &c, c);
11:    printf("x: address=0x%p, content=%d\n", &x, x);
12:    printf("y: address=0x%p, content=%5.2f\n", &y, y);
13:    c = `A';
14:    x = 7;
15:    y = 123.45;
16:    printf("c: address=0x%p, content=%c\n", &c, c);
17:    printf("x: address=0x%p, content=%d\n", &x, x);
18:    printf("y: address=0x%p, content=%5.2f\n", &y, y);
19:    return 0;
20: }
```

After the executable (11L01.exe) of this program is created and run from a DOS prompt, the following output is displayed on the screen (note that you might get a different result, depending on your computer and operating system):

OUTPUT

```
C:\app> 11L01

c: address=0x1AF4, content=@
x: address=0x1AF2, content=-32557
y: address=0x1AF6, content=0.00
c: address=0x1AF4, content=A
x: address=0x1AF2, content=7
y: address=0x1AF6, content=123.45
C:\app>
```

As you can see in Listing 11.1, there are three variables, c, x, and y, declared in lines 6_8, respectively.

ANALYSIS

The statement in line 10 displays the address (that is, the left value) and the content (that is, the right value) of the character variable c on the screen. Here the &c expression returns the address of c.

Note that the format specifier %p is used in the printf() function of line 10 for displaying the address returned from &c.

Likewise, lines 11 and 12 print out the addresses of x and y, as well as the contents of x and y.

From the first part of the output, you see that the addresses (in hex format) of c, x, and y are 0x1AF4, 0x1AF2, and 0x1AF6. Because these three variables have not been initialized yet, the contents contained in their memory locations are what is left there from the last memory writing.

However, after the initializations that are carried on in lines 13_15, the memory slots reserved for the three variables have the contents of the initial values. Lines 16_18 display the addresses and contents of c, x, and y after the initialization.

You can see in the second part of the output, the contents of c, x, and y are now `A', 7, and 123.45, respectively, with the same memory addresses.

NOTE

> The format specifier %p used in the printf() function is supported by the ANSI standard. If, somehow, your compiler does not support %p, you can try to use %u or %lu in the printf() function to convert and print out a left value (that is, an address).
>
> Also, the addresses printed out by the examples in this lesson are obtained by running the examples on my machine. The values may be different from what you can get by running the examples on your machine. This is because the address of a variable may vary from one type of computer to another.

Declaring Pointers

As mentioned at the beginning of this lesson, a pointer is a variable, which means that a pointer has a left value and a right value as well. However, both the left and right values are addresses. The left value of a pointer is used to refer to the pointer itself, whereas the right value of a pointer, which is the content of the pointer, is the address of another variable.

The general form of a pointer declaration is

```
data-type  *pointer-name;
```

Here data-type specifies the type of data to which the pointer points. pointer-name is the name of the pointer variable, which can be any valid variable name in C.

Note that right before the pointer name is an asterisk (*), which indicates that the variable is a pointer. When the compiler sees the asterisk in

the declaration, it makes a note in its symbol table so that the variable can be used as a pointer.

The following shows different types of pointers:

```
char *ptr_c;    /* declare a pointer to a character */

int  *ptr_int;  /* declare a pointer to an integer */

float *ptr_flt; /* declare a pointer to a floating-point */
```

The program in Listing 11.2 demonstrates how to declare pointers and assign values to them.

TYPE

Listing 11.2. Declaring and assigning values to pointers.

```
1:  /* 11L02.c: Declaring and assigning values to pointers */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char  c, *ptr_c;
7:     int   x, *ptr_x;
8:     float y, *ptr_y;
9:
10:    c = `A';
11:    x = 7;
12:    y = 123.45;
13:    printf("c: address=0x%p, content=%c\n", &c, c);
14:    printf("x: address=0x%p, content=%d\n", &x, x);
15:    printf("y: address=0x%p, content=%5.2f\n", &y, y);
16:    ptr_c = &c;
17:       printf("ptr_c: address=0x%p, content=0x%p\n", &ptr_c, ptr_c);
18:       printf("*ptr_c => %c\n", *ptr_c);
19:    ptr_x = &x;
20:       printf("ptr_x: address=0x%p, content=0x%p\n", &ptr_x, ptr_x);
21:       printf("*ptr_x => %d\n", *ptr_x);
22:    ptr_y = &y;
23:       printf("ptr_y: address=0x%p, content=0x%p\n", &ptr_y, ptr_y);
24:       printf("*ptr_y => %5.2f\n", *ptr_y);
25:    return 0;
26: }
```

OUTPUT

I get the following output displayed on the screen after running the executable 11L02.exe from a DOS prompt on my machine:

```
C:\app> 11L02
c: address=0x1B38, content=A
x: address=0x1B36, content=7
y: address=0x1B32, content=123.45
ptr_c: address=0x1B30, content=0x1B38
*ptr_c => A
ptr_x: address=0x1B2E, content=0x1B36
*ptr_x => 7
ptr_y: address=0x1B2C, content=0x1B32
*ptr_y => 123.45
C:\app>
```

ANALYSIS

In Listing 11.2, there are three variables, c, x, and y, and three pointer variables, ptr_c, ptr_x, and ptr_y, declared in lines 6_8, respectively.

The statements in lines 10_12 initialize the three variables c, x, and y. Then, lines 13_15 print out the addresses as well as the contents of the three variables.

In line 16, the left value of the character variable c is assigned to the pointer variable ptr_c. The output made by the statement in line 17 shows that the pointer variable ptr_c contains the address of c. In other words, the content (that is, the right value) of ptr_c is the address (that is, the left value) of c.

Then in line 18, the value referred to by the pointer *ptr_c is printed out. The output proves that the pointer *ptr_c does point to the memory location of c.

Line 19 assigns the left value of the integer x to the integer pointer variable ptr_x. The statements in lines 20 and 21 print out the left value and right value of the pointer variable ptr_x, as well as the value referred to by the pointer *ptr_x.

Similarly, the left value of the float variable y is assigned to the float pointer variable ptr_y in line 22. To prove that ptr_y contains the address of y, and *ptr_y gives the content held by y, lines 23 and 24 print out the right values of ptr_y and *ptr_y, respectively.

The statements in lines 16, 19, and 22 show you how to assign the value of a variable

to another—in an indirect way. In other words, the left value of a variable can be assigned to another variable so that the latter can be used as a pointer variable to obtain the right value of the former. In this case, the variable name and the pointer refer to the same memory location. Accordingly, if either the variable name or the pointer is used in an expression to change the contents of the memory location, the contents of the memory location has changed for the other.

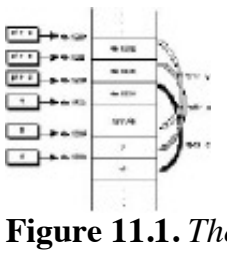

Figure 11.1. *The memory image of variables and their pointers.*

To help you understand the indirection of assigning values, Figure 11.1 demonstrates the memory image of the relationships between c and ptr_c, x and ptr_x, and y and ptr_y, based on the output obtained on my machine.

The Dereference Operator (*)

You've seen the asterisk (*) in the declaration of a pointer. In C, the asterisk is called the dereference operator when it is used as a unary operator. (Sometimes, it's also called the indirection operator.) The value of a variable can be referenced by the combination of the * operator and its operand, which contains the address of the variable.

For instance, in the program shown in Listing 11.2, after the address of the character variable c is assigned to the pointer variable ptr_c, the expression *ptr_c refers to the value contained by c. Therefore, you can use the *ptr_c expression, instead of calling the variable c directly, to obtain the value of c.

Likewise, given an integer variable x and x = 1234, you can declare an integer pointer variable, ptr_x, for instance, and assign the left value (address) of x to ptr_x—that is, ptr_x = &x. Then, the expression *ptr_x returns 1234, which is the right value (content) of x.

WARNING

Don't confuse the dereference operator with the multiplication operator, although they share the same symbol, *.

The dereference operator is a unary operator, which takes only one operand. The operand contains the address (that is, left value) of a variable.

On the other hand, the multiplication operator is a binary operator that requires two operands to perform the operation of multiplication.

Null Pointers

A pointer is said to be a null pointer when its right value is 0. Remember, a null pointer can never point to valid data.

To set a null pointer, simply assign 0 to the pointer variable. For example:

```
char *ptr_c;
int  *ptr_int;

ptr_c = ptr_int = 0;
```

Here ptr_c and ptr_int become null pointers after the integer value of 0 is assigned to them.

You'll see applications of null pointers used in control-flow statements and arrays later in this book.

Updating Variables via Pointers

As you learned in the previous section, as long as you link up a variable to a pointer variable, you can obtain the value of the variable by using the pointer variable. In other words, you can read the value by pointing to the memory location of the variable and using the dereferencing operator.

TYPE

This section shows you that you can write a new value to the memory location of a variable using a pointer that contains the left value of the variable. Listing 11.3 gives an example.

Listing 11.3. Changing variable values via pointers.

```
1:  /* 11L03.c: Changing values via pointers */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char  c,  *ptr_c;
7:
8:     c = `A';
9:     printf("c: address=0x%p, content=%c\n", &c, c);
10:    ptr_c = &c;
11:       printf("ptr_c: address=0x%p, content=0x%p\n", &ptr_c, ptr_c);
12:       printf("*ptr_c => %c\n", *ptr_c);
13:    *ptr_c = `B';
14:       printf("ptr_c: address=0x%p, content=0x%p\n", &ptr_c, ptr_c);
15:       printf("*ptr_c => %c\n", *ptr_c);
16:    printf("c: address=0x%p, content=%c\n", &c, c);
17:    return 0;
18: }
```

OUTPUT

After running the executable 11L03.exe from a DOS prompt on my machine, I get the following output displayed on the screen:

ANALYSIS

```
C:\app> 11L03
c: address=0x1828, content=A
ptr_c: address=0x1826, content=0x1828
*ptr_c => A
ptr_c: address=0x1826, content=0x1828
*ptr_c => B
c: address=0x1828, content=B
C:\app>
```

A char variable, c, and a char pointer variable, ptr_c, are declared in line 6 of List-ing 11.3.

The variable c is initialized with `A' in line 8, which is printed out, along with the address of the variable, by the printf() function in line 9.

Then, in line 10, the pointer variable ptr_c is assigned the left value (address) of c. It's not surprising to see the output printed out by the statements in lines 11 and 12, where the right value of ptr_c is the left value of c, and the pointer *ptr_c points to the right value of c.

In line 13 the expression *ptr_c = `B' asks the computer to write `B' to the location pointed to by the pointer *ptr_c. The output printed by the statement in line 15 proves that the content of the memory location pointed to by *ptr_c is updated. The statement in line 14 prints out the left and right values of the pointer variable ptr_c and shows that these values remain the same.

As you know, *ptr_c points to where the character variable c resides. Therefore, the expression *ptr_c = `B' actually updates the content (that is, the right value) of the variable c to `B'. To prove this, the statement in line 16 displays the left and right values of c on the screen. Sure enough, the output shows that the right value of c has been changed.

Pointing to the Same Thing

TYPE

A memory location can be pointed to by more than one pointer. For example, given that

c = `A' and that ptr_c1 and ptr_c2 are two character pointer variables, ptr_c1 = &c and ptr_c2 = &c set the two pointer variables to point to the same location in the memory.

The program in Listing 11.4 shows another example of pointing to the same thing with several pointers.

Listing 11.4. Pointing to the same thing with more than one pointer.

```
1:  /* 11L04.c: Pointing to the same thing */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int x;
7:     int *ptr_1, *ptr_2, *ptr_3;
8:
9:     x = 1234;
10:    printf("x: address=0x%p, content=%d\n", &x, x);
11:    ptr_1 = &x;
12:    printf("ptr_1: address=0x%p, content=0x%p\n", &ptr_1, ptr_1);
13:       printf("*ptr_1 => %d\n", *ptr_1);
14:    ptr_2 = &x;
15:    printf("ptr_2: address=0x%p, content=0x%p\n", &ptr_2, ptr_2);
16:       printf("*ptr_2 => %d\n", *ptr_2);
17:    ptr_3 = ptr_1;
18:    printf("ptr_3: address=0x%p, content=0x%p\n", &ptr_3, ptr_3);
19:       printf("*ptr_3 => %d\n", *ptr_3);
20:    return 0;
21: }
```

OUTPUT

The following output is displayed on the screen by running the executable 11L04.exe from a DOS prompt on my machine (note that you might get different address values if you run the program on your machine):

```
C:\app> 11L04
x: address=0x1838, content=1234
ptr_1: address=0x1834, content=0x1838
*ptr_1 => 1234
ptr_2: address=0x1836, content=0x1838
*ptr_2 => 1234
ptr_3: address=0x1832, content=0x1838
*ptr_3 => 1234
C:\app>
```

ANALYSIS

As shown in Listing 11.4, line 6 declares an integer variable, x, and line 7 declares three integer pointer variables, ptr_1, ptr_2, and ptr_3.

The statement in line 10 prints out the left and right values of x. On my machine, the left value (address) of x is 0x1838. The right value (content) of x is 1234, which is the initial value assigned to x in line 9.

Line 11 assigns the left value of x to the pointer variable ptr_1 so that ptr_1 can be used to refer to the right value of x. To make sure that the pointer variable ptr_1 now contains the address of x, line 12 prints out the right value of ptr_1, along with its left value. The output shows that ptr_1 does hold the address of x, 0x1838. Then, line 13 prints out the value 1234, which is referred to by the *ptr_1 expression. Note that the asterisk * in the expression is the dereference operator.

In line 14, the *ptr_2 = &x expression assigns the left value of x to another pointer variable, ptr_2; that is, the pointer variable ptr_2 is now linked with the address of x. The statement in line 16 displays the integer 1234 on the screen by using the dereference operator * and its operand, ptr_2. In other words, the memory location of x is referred to by the second pointer *ptr_2.

In line 17, the pointer variable ptr_3 is assigned with the right value of ptr_1. Because ptr_1 now holds the address of x, the expression ptr_3 = ptr_1 is equivalent to ptr_3 = &x. Then, from the output made by the statements in lines 18 and 19, you see the integer 1234 again on the screen. This time the integer is referred to by the third pointer, *ptr_3.

Summary

In this lesson you've learned the following:

- A pointer is a variable whose value is used to point to another variable.
- A variable declared in C has two values: the left value and the right value.
- The left value of a variable is the address; the right value is the content of the variable.
- The address-of operator (&) can be used to obtain the left value (address) of a variable.
- The asterisk (*) in a pointer declaration tells the compiler that the variable is a pointer variable.
- The dereference operator (*) is a unary operator; as such, it requires only one operand.
- The *ptr_name expression returns the value pointed to by the pointer variable ptr_name, where ptr_name can be any valid variable name in C.
- If the right value of a pointer variable is 0, the pointer is a null pointer. A null pointer cannot point to valid data.
- You can update the value of a variable referred by a pointer variable.
- Several pointers can point to the same location of a variable in the memory.

In the next lesson you'll learn about an aggregate type—an array, which is closely related to pointers in C.

Q&A

Q What are the left and right values?

A The left value refers to the address of a variable, and the right value refers to the content stored in the memory location of a variable. There are two ways to get the right value of a variable: use the variable name directly, or use the left value of the variable to refer to where the right value resides. The second way is also called the indirect way.

Q How can you obtain the address of a variable?

A By using the address-of operator, &. For instance, given an integer variable x, the &x expression returns the address of x. To print out the address of x, you can use the %p format specifier in the printf() function.

Q What is the concept of indirection in terms of using pointers?

A Before this hour, the only way you knew for reading from or writing to a variable was to invoke the variable directly. For instance, if you wanted to write a decimal, 16, to an integer variable x, you could call the statement x = 16;.

As you learned in this hour, C allows you to access a variable in another way—using pointers. Therefore, to write 16 to x, you can first declare an integer pointer (ptr) and assign the left value (address) of x to ptr—that is, ptr = &x;. Then, instead of executing the statement x = 16;, you can use another statement:

```
*ptr = 16;
```

Here the pointer *ptr refers to the memory location reserved by x, and the content stored in the memory location is updated to 16 after the statement is executed. So, you see, making use of pointers to access the memory locations of variables is a way of indirection.

Q Can a null pointer point to valid data?

A No. A null pointer cannot point to valid data. This is so because the value contained by a null pointer is 0. You'll see examples of using null pointers in arrays, strings, or memory allocation later in the book.

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. How can you obtain the left value of a character variable ch?
2. In the following expressions, does the asterisk (*) act as a dereference operator or as a multiplication operator?
    - *ptr
    - x * y
    - y *= x + 5
    - *y *= *x + 5
3. Given that x = 10, the address of x is 0x1A38, and ptr_int = &x, what will ptr_int and *ptr_int return, respectively?
4. Given that x = 123, and ptr_int = &x after the execution of *ptr_int = 456, what does x contain?

Exercises

1. Given three integer variables, x = 512, y = 1024, and z = 2048, write a program to print out their left values as well as their right values.

2. Write a program to update the value of the double variable flt_num from 123.45 to 543.21 by using a double pointer.
3. Given a character variable ch and ch = `A', write a program to update the value of ch to decimal 66 by using a pointer.
4. Given that x=5 and y=6, write a program to calculate the multiplication of the two integers and print out the result, which is saved in x, all in the way of indirection (that is, using pointers).

Previous | Table of Contents | Next