Chapter 8: Pointers

WHAT IS A POINTER ?

Simply stated, a pointer is an address. Instead of being a variable, it is a pointer to a variable stored somewhere in the address space of the program. It is always best to use an example so load the file named pointer.c and display it on your monitor for an example of a program with some pointers in it.

For the moment, ignore the data declaration statement where we define **index** and two other fields beginning with a star. It is properly called an asterisk, but for reasons we will see later, let's agree to call it a star. If you observe the statement in line 8, it should be clear that we assign the value of 39 to the variable named **index**. This is no surprise, we have been doing it for several programs now. The statement in line 9 however, says to assign to **pt1** a strange looking value, namely the variable **index** with an ampersand in front of it. In this example, **pt1** and **pt2** are pointers, and the variable named **index** is a simple variable. Now we have a problem similar to the old chicken and egg problem. We need to learn how to use pointers in a program, but to do so requires that first we define the means of using the pointers in the program.

The following two rules will be somewhat confusing to you at first, but we need to state the definitions before we can use them. Take your time, and the whole thing will clear up very quickly.

TWO VERY IMPORTANT RULES

The following two rules are very important when using pointers and must be thoroughly understood.

1. A variable name with an ampersand in front of it defines the address of the variable and therefore points to the variable. You can therefore read line nine as "**pt1** is assigned the value of the address of **index**".
2. A pointer with a star in front of it refers to the value of the variable pointed to by the pointer. Line twelve of the program can be read as "The stored (starred) value to which the pointer **pt1** points is assigned the value 13". Now you can see why it is convenient to think of the asterisk as a star, it sort of sounds like the word store.

MEMORY AIDS

1. Think of & as an address.
2. Think of * as a star referring to stored.

Assume for the moment that **pt1** and **pt2** are pointers (we will see how to define pointers shortly). As pointers, they do not contain a variable value but an address of a variable and can be used to point to a variable. Figure 8-1 is a graphical representation of the data space as it is configured just prior to executing line 8. A box represents a variable, and a box with a dot in it represents a pointer. At this time the pointers are not pointing at anything, so they have no arrows emanating from the boxes. Executing line 8 stores the value 39 in index.

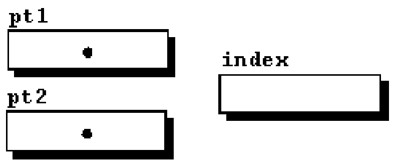


Figure 8-1

Continuing execution of the program, we come to line 9 which assigns the address of the variable **index** to the pointer **pt1**. Since we have a pointer to **index**, we can manipulate the value of **index** by using either the variable name itself, or the pointer. Figure 8-2 depicts the condition of the data space after executing line 9 of the program.

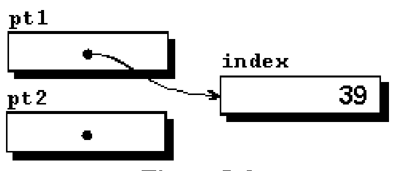


Figure 8-2

Jumping ahead a little in the program, line 12 modifies the value of **index** by using the pointer. Since the pointer **pt1** points to the variable named **index**, putting a star in front of the pointer name refers to the memory location to which it is pointing. Line 12 therefore assigns the value of 13 to **index**. Anyplace in the program where it is permissible to use the variable name **index**, it is also permissible to use the name ***pt1** since they are identical in meaning until the pointer is reassigned to some other variable.

ANOTHER POINTER

Just to add a little intrigue to the system, we have another pointer defined in this program, **pt2**. Since **pt2** has not been assigned a value prior to statement 10, it doesn't point to anything, it contains garbage. Of course, that is also true of any variable until a value is assigned to it. The statement in line 10 assigns **pt2** the same address as **pt1**, so that now **pt2** also points to the variable named **index**. We have copied the address from one pointer to another pointer. To continue the definition from the last paragraph, anyplace in the program where it is permissible to use the variable **index**, it is also permissible to use the name ***pt2** because they are now identical in meaning. This fact is illustrated in the **printf()** statement in line 11 since this statement uses the three means of identifying the same variable to print out the same variable three times. Refer to figure 8-3 for the representation of the data space at this time.

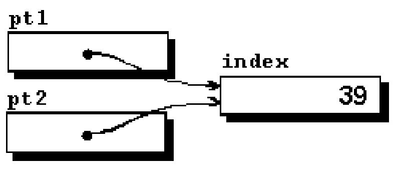


Figure 8-3

THERE IS ONLY ONE VARIABLE

Note carefully that, even though it appears that there are three variables, there is really only one variable. The two pointers point to the single variable. This is illustrated in the statement in line 12 which assigns the value of 13 to the variable **index**, because that is where the pointer **pt1** is pointing. The **printf()** statement in line 13 causes the new value of 13 to be printed out three times. Keep in mind that there is really only one variable to be changed or printed, not three. We do have three aliases for the one variable, **index**, ***pt1**, and ***pt2.** Figure 8-4 is the graphical representation of the data space at this time.

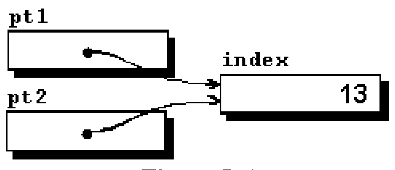


Figure 8-4

This is admittedly a very difficult concept, but since it is used extensively in all but the most trivial C programs, it is well worth your time to stay with this material until you understand it thoroughly.

HOW DO YOU DECLARE A POINTER ?

Now to keep a promise and tell you how to define a pointer. Refer to line 6 of the program and you will see our old familiar way of defining the variable **index**, followed by two more definitions. The second definition can be read as "the storage location to which **pt1** points will be an **int** type variable". Therefore, **pt1** is a pointer to an **int** type variable. Likewise, **pt2** is another pointer to an **int** type variable, because it has a star (asterisk) in front of it. These pointers can point to the same **int** variable or to two different **int** variables.

A pointer must be defined to point to a specific type of variable. Following a proper definition, it cannot be used to point to any other type of variable or it will result in a type incompatibility error. In the same manner that a **float** type of variable cannot be added to an **int** type variable, a pointer to a **float** variable cannot be used to point to an integer variable without some major problems.

Compile and run this program and observe that there is only one variable and the single statement in line 12 changes the one variable which is displayed three times. This material is so important that you should review it carefully if you do not fully understand it at this time. It would be a good exercise for you to draw the graphics yourself as you review the code for this program.

THE SECOND PROGRAM WITH POINTERS

In these few pages so far on pointers, we have covered a lot of territory, but it is important territory. We still have a lot of material to cover so stay in tune as we continue this important aspect of C. Load the next file named pointer2.c and display it on your monitor so we can continue our study.

In this program we have defined several variables and two pointers. The first pointer named **there** is a pointer to a **char** type variable and the second named **pt** points to an **int** type variable. Notice also that we have defined two array variables named **strg** and **list**. We will use them to show the correspondence between pointers and array names.

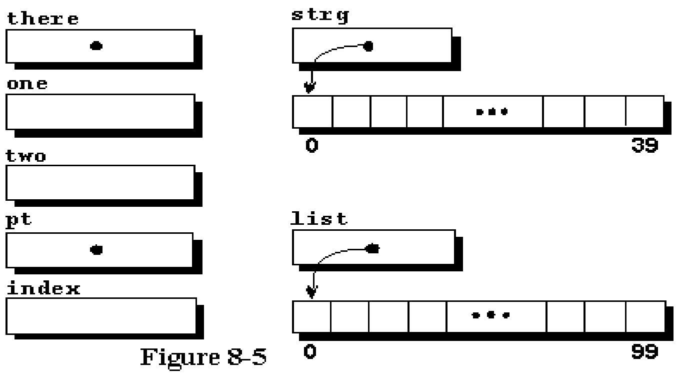


Figure 8-5

Figure 8-5 depicts the data just prior to executing line 10. There are three variables, two pointers, a string, and an array of ints, or we could say there are three variables, two pointers, and two arrays. Each array is composed of the array itself and a pointer which points to the beginning of the array according to the definition of an array in C. This will be completely defined in the next paragraph. Each array is composed of a number of identical elements of which only a few at the beginning and a few at the end are depicted graphically for convenience.

AN ARRAY VARIABLE IS ACTUALLY A POINTER

In the programming language C, an array variable is defined to be simply a pointer to the beginning of the array. This will take some explaining. Refer to the example program on your monitor. You will notice that in line 10 we assign a string constant to the string variable named **strg** so we will have some data to work with. Next, we assign the value of the first element to the variable **one**, a simple **char** variable. Next, since the string name is a pointer to the first element of the string, by definition of the C language, we can assign the same value to **two** by using the star and the string name (***strg**). Observe that the box with a dot pointing to a variable can be used to access the variable just like in the last program. The result of the two assignments are such that **one** now has the same value as **two**, and both contain the character T, the first character in the string. Note that it would be incorrect to write line 10 as `two = *strg[0];` because the star takes the place of the square brackets.

For all practical purposes, **strg** is a pointer to a **char** type variable. It does, however, have one restriction that a true pointer does not have. It cannot be changed like a variable, but must always contain the address of the first element of the string and therefore always points to its string. It could be thought of as a pointer constant. Even though it cannot be changed, it can be used to refer to other values than the one it is defined to point to, as we will see in the next section of the program.

Moving ahead to line 16, the variable **one** is assigned the value of the ninth character in the string (since the indexing starts at zero) and **two** is assigned the same value because we are allowed to index a pointer to get to values farther ahead in the string. Both variables now contain the character a. Line 17 says to add 8 to the value of the pointer strg, then get the value stored at that location and store it in the variable **two**.

POINTER INDEXING

The C programming language takes care of indexing for us automatically by adjusting the indexing for the type of variable the pointer is pointing to. In this case, the index of 8 is simply added to the pointer value before looking up the desired result because a **char** type variable is one byte long. If we were using a pointer to an **int** type variable, the index would be doubled and added to the pointer before looking up the value because an **int** type variable uses two bytes per value stored on most microcomputers. When we get to the chapter on structures, we will see that a variable can have many, even into the hundreds or thousands, of bytes per variable, but the indexing will be handled automatically for us by the system.

The data space is now in the state defined graphically in figure 8-6. The string named **strg** has been filled and the two variables named **one** and **two** have the letter "a" stored in them. Since the pointer variable **there** is already a pointer, it can be assigned the address of the 11th element of **strg** by the statement in line 20 of this program. Remember that since **there** is a pointer to type **char**, it can be assigned any value as long as that value represents a **char** type of address. It should be clear that the pointers must be typed in order to allow the pointer arithmetic described in the last paragraph to be done properly. The third and fourth outputs will be the same, namely the letter c.

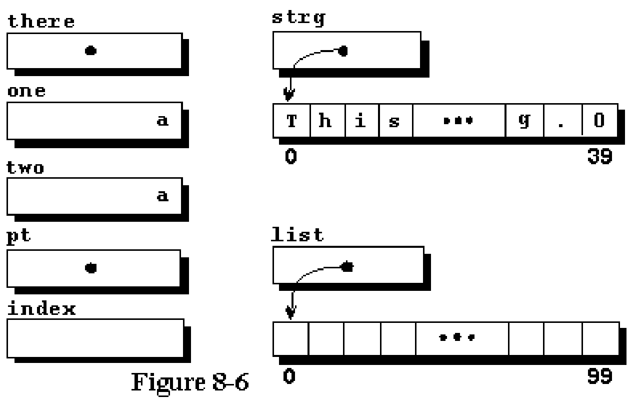


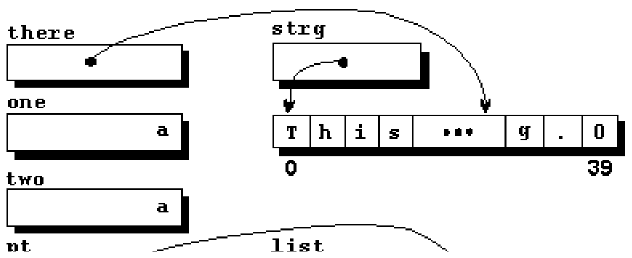
Figure 8-6

POINTER ARITHMETIC

Not all forms of arithmetic are permissible on a pointer. Only those things that make sense, considering that a pointer is an address somewhere in the computer. It would make sense to add a constant to an address, thereby moving it ahead in memory that number of places. Likewise, subtraction is permissible, moving it back some number of locations. Adding two pointers together would not make sense because absolute memory addresses are not additive. Pointer multiplication is also not allowed, as that would be a funny number. If you think about what you are actually doing, it will make sense to you what is allowed, and what is not.

NOW FOR AN INTEGER POINTER

The array named **list** is assigned a series of values from 100 to 199 in order to have some data to work with in lines 24 and 25. Next, we assign the pointer **pt** the address of the 28th element of the list and print out the same value both ways to illustrate that the system truly will adjust the index for the **int** type variable. You should spend some time in this



program until you feel you fairly well understand these lessons on pointers.

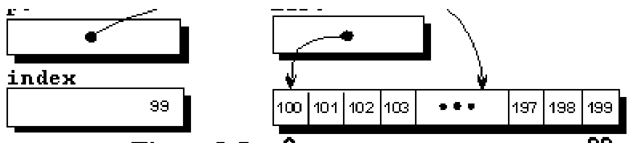


Figure 8-7

Compile and execute pointer2.c and study the output. At the termination of execution, the data space will be as depicted in figure 8-7. Once again, it would be a good exercise for you to attempt to draw the graphic for this program as you review the code.

FUNCTION DATA RETURN WITH A POINTER

You may recall that back in the lesson on functions we mentioned that there were two ways to get variable data back from a function. One way is through use of the array, and you should be right on the verge of guessing the other way. If your guess is through use of a pointer, you are correct. Load and display the example program named twoway.c for an example of this.

In twoway.c, there are two variables defined in the main program, **pecans** and **apples**. Notice that neither of these is defined as a pointer. We assign values to both of these and print them out, then call the function named **fixup()** taking both of these values along with us. The variable **pecans** is simply sent to the function, but the address of the variable **apples** is sent to the function. Now we have a problem. The two arguments are not the same, the second is a pointer to a variable. We must somehow alert the function to the fact that it is supposed to receive an integer variable and a pointer to an integer variable. This turns out to be very simple. Notice that the parameter definitions in line 20 defines **nuts** as an integer, and **fruit** as a pointer to an integer. The call in the main program therefore is now in agreement with the function heading and the program interface will work just fine.

In the body of the function, we print the two values sent to the function, then modify them and print the new values out. This should be perfectly clear to you by now. The surprise occurs when we return to the main program and print out the two values again. We will find that the value of **pecans** will be restored to the value it had prior to the function call because the C language makes a copy of the item in question and takes the copy to the called function, leaving the original intact as we explained earlier. In the case of the variable **apples**, we made a copy of a pointer to the variable and took the copy of the pointer to the function. Since we had a pointer to the original variable, even though the pointer was a local copy, it pointed to the original variable and we could change the value of **apples** from within the function. When we returned to the main program, we found a changed value in **apples** when we printed it out.

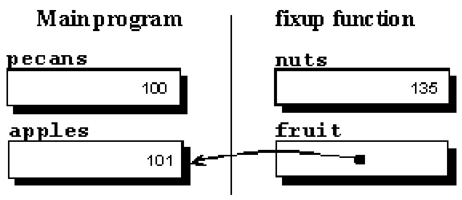


Figure 8-8

This is illustrated graphically in figure 8-8. The state of the system is illustrated following execution of line 25 of the program. The observant student will notice the prototype in line 3. This allows the compiler to check the type of both parameters when it gets to line 14 where the function is called.

By using a pointer in a function call, we can have access to the data in the function and change it in such a way that when we return to the calling program, we have a changed value of the original variable. In this example, there was no pointer in the main program because we simply sent the address to the function, but in many programs you will use pointers in function calls. One of the places you will find need for pointers in function calls will be when you request data input using standard input/output routines. These will be covered in the next two chapters. Compile and run twoway.c and observe the output.

POINTERS ARE VALUABLE

Even though you are probably somewhat intimidated by this time about the proper use of pointers, you will find that after you gain experience, you will use them profusely in many ways. You will also use pointers in every program you write other than the most trivial because they are so useful. You should probably go over this material carefully several times until you feel comfortable with it because it is very important in the area of input/output which is next on the agenda.

A POINTER TO A FUNCTION

Examine the example program named funcpnt.c for the most unusual pointer yet. This program contains a pointer to a function, and illustrates how to use it.

Line 8 of this program defines **function_pointer** as a pointer to a function and not to just any function, it points to a function with a single formal parameter of type **float**. The function must also return nothing because of the **void** before the pointer definition. The parentheses are required around the pointer name as illustrated or the system will think it is a prototype definition for a function that returns a pointer to **void**.

You will note the prototypes given in lines 4 through 6 that declare three functions that use the same parameter and return type as the pointer. Since they are the same as the pointer, the pointer can be used to refer to them as is illustrated in the executable part of the program. Line 15 contains a call to the **print_stuff()** function, and line 16 assigns the value of **print_stuff()** to **function_pointer**. Because the name of a function is defined as a pointer to that function, its name can be assigned to a function pointer variable. You will recall that the name of an array is actually a pointer constant to the first element of the array. In like manner, a function name is actually a pointer constant which is pointing to the function itself. The pointer is successively assigned the address of each of the three functions and each is called once or twice as an illustration of how a pointer to a function can be used.

A function pointer can be passed to another function as a parameter and can be used within the function to call the function which is pointed to. You are not permitted to increment or add a constant to a function pointer, it can only be assigned the value of a function with the same parameters and return with which it was initially declared. It may take you a little time to appreciate the value of this construct, but when you do understand it, you will see the flexibility built into the C programming language.

A pointer to a function is not used very often but it is a very powerful construct when needed. You should plan to do a lot of C programming before you find a need for this technique. I mention it here only to prevent you being unduly intimidated by this difficult concept. We will continue to study pointers by examining their use in additional example programs.

Programming Exercise:

1. Define a character array and use **strcpy()** to copy a string into it. Print the string out by using a loop with a pointer to print out one character at a time. Initialize the pointer to the first element and use the double plus sign to increment the pointer. Use a separate integer variable to count the characters to print.
2. Modify the program from programming exercise 1 to print out the string backwards by pointing to the end and using a decrementing pointer.

Index Previous Next . . . . *C Tutorial*

The Webwizard