

## Unit 8

## Pointers – I

### Structure:

- 8.1 Introduction
  - Objectives
- 8.2 Basics of Pointers
- 8.3 Pointers and One-dimensional Arrays
  - Pointer Arithmetic
  - Pointer Subtraction and Comparison
  - Similarities between Pointers and One-dimensional Arrays
- 8.4 Summary
- 8.5 Terminal Questions
- 8.6 Answers to Self Assessment Questions
- 8.7 Answers to Terminal Questions
- 8.8 Exercises

### 8.1 Introduction

In the previous unit, you studied about the arrays and strings. You studied how the arrays and strings are declared and manipulated in making C a more easy and powerful programming language. In this unit, you will study about pointer which is another useful topic that makes C a most powerful programming language. You will study about the basics of pointer and pointer arithmetic.

A pointer is a variable that points at, or refers to, another variable. That is, if we have a pointer variable of type “pointer to int, “it might point to the int variable i, or to any one of the locations of the int array a. Given a pointer variable, we can ask questions like, “what’s the value of the variable that this pointer points to?”.

Why would we want to have a variable that refers to another variable? Why not just use that other variable directly? Pointers are used frequently in C, as they have number of useful applications. For example, pointers can be used to pass information back and forth between a function and its reference point. In particular, pointers provide a way to return multiple data items from a function via function arguments. Pointers also permit references to other functions to be specified as arguments to a given

function. This has the effect of passing functions as arguments to the given function.

Pointers are also closely associated with the arrays and therefore provide an alternate way to access individual array elements. Pointers make the program efficient when we use pointers for accessing the individual array elements. In this unit, you will study about the basics of pointers and its usage in one dimensional array.

**Objectives:**

After studying this unit, you should be able to:

- implement pointers in your program
- write a program related to arrays and using a pointer with it
- solve and illustrate the use of pointer arithmetics in C
- point out the similarities between pointers and one dimensional arrays

**8.2 Basics of Pointers**

A pointer is a variable which contains the address in memory of another variable. We can have a pointer to any variable type. Before studying pointers, you should mind certain symbols that are frequently used in implementing the pointers. We use the symbol **&** that gives the “address of a variable” and is known as the **unary** or **monadic** operator. We use other symbol **\*** that gives the “contents of an object *pointed to* by a pointer” and we call it the **indirection** or **dereference** operator.

The first things to do with pointers are to declare a pointer variable, set it to point somewhere, and finally manipulate the value that it points to. A simple pointer declaration has the following general format:

*datatype \*variablename*

where *datatype* represents the type of the data to which the pointer *variablename* points to. In simple terms, the *variablename* holds the address of the value of type *datatype*.

For example,

int \*ip;

This declaration looks like our earlier declarations, with one obvious difference: that is the asterisk. The asterisk means that ip, the variable we're

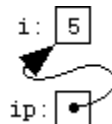
declaring, is not of type `int`, but rather of type `pointer-to-int`. (Another way of looking at it is that `*ip`, which as we'll see is the value pointed to by `ip`, will be an `int`.)

We may think of setting a pointer variable to point to another variable as a two-step process: first we generate a pointer to that other variable, and then we assign this new pointer to the pointer variable. We can say (but we have to be careful when we're saying it) that a pointer variable has a value, and that its value is "pointer to that other variable". This will make more sense when we see how to generate pointer values.

Pointers (that is, pointer values) are generated with the "address-of" operator `&`, which we can also think of as the "pointer-to" operator. We demonstrate this by declaring (and initializing) an `int` variable `i`, and then setting `ip` to point to it:

```
int i = 5;  
ip = &i;
```

The assignment expression `ip = &i`; contains both parts of the "two-step process": `&i` generates a pointer to `i`, and the assignment operator assigns the new pointer to (that is, places it "in") the variable `ip`. Now `ip` "points to" `i`, which we can illustrate with this picture:



`i` is a variable of type `int`, so the value in its box is a number, 5. `ip` is a variable of type `pointer-to-int`, so the "value" in its box is an arrow pointing at another box. Referring once again back to the "two-step process" for setting a pointer variable: the `&` operator draws us the arrowhead pointing at `i`'s box, and the assignment operator `=`, with the pointer variable `ip` on its left, anchors the other end of the arrow in `ip`'s box.

We discover the value pointed to by a pointer using the "contents-of" operator, `*`. Placed in front of a pointer, the `*` operator accesses the value pointed to by that pointer. In other words, if `ip` is a pointer, then the expression `*ip` gives us whatever it is that's in the variable or location pointed to by `ip`. For example, we could write something like

```
printf("%d\n", *ip);
```

which would print 5, since `ip` points to `i`, and `i` is (at the moment) 5.

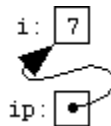
(You may wonder how the asterisk `*` can be the pointer contents-of operator when it is also the multiplication operator. There is no ambiguity here: it is the multiplication operator when it sits between two variables, and it is the contents-of operator when it sits in front of a single variable. The situation is analogous to the minus sign: between two variables or expressions it's the subtraction operator, but in front of a single operator or expression it's the negation operator. Technical terms you may hear for these distinct roles are *unary* and *binary*: a *binary* operator applies to two operands, usually on either side of it, while a *unary* operator applies to a single operand.)

The contents-of operator `*` does not merely fetch values through pointers; it can also *set* values through pointers. We can write something like

```
*ip = 7;
```

which means “set whatever `ip` points to 7.” Again, the `*` tells us to go to the location pointed to by `ip`, but this time, the location isn't the one to fetch from -- we're on the left-hand side of an assignment operator, so `*ip` tells us the location to store *to*. (The situation is no different from array subscripting expressions such as `a[3]` which we've already seen appearing on both sides of assignments.)

The result of the assignment `*ip = 7` is that `i`'s value is changed to 7, and the picture changes to:



If we called `printf("%d\n", *ip)` again, it would now print 7.

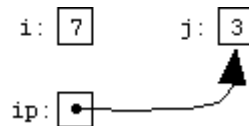
At this point, you may wonder, if we wanted to set `i` to 7, why didn't we do it directly? We'll begin to explore that next, but first let's notice the difference between changing a pointer (that is, changing what variable it points to) and changing the value at the location it points to. When we wrote `*ip = 7`, we changed the value pointed to by `ip`, but if we declare another variable `j`:

```
int j = 3;
```

and write

```
ip = &j;
```

we've changed ip itself. The picture now looks like this:



We have to be careful when we say that a pointer assignment changes “what the pointer points to.” Our earlier assignment

```
*ip = 7;
```

changed the value pointed to by ip, but this more recent assignment

```
ip = &j;
```

has changed what *variable* ip points to. It's true that “what ip points to” has changed, but this time, it has changed for a different reason. Neither i (which is still 7) nor j (which is still 3) has changed. (What has changed is ip's value.) If we again call

```
printf("%d\n", *ip);
```

this time it will print 3.

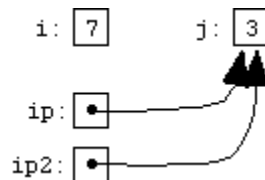
We can also assign pointer values to other pointer variables. If we declare a second pointer variable:

```
int *ip2;
```

then we can say

```
ip2 = ip;
```

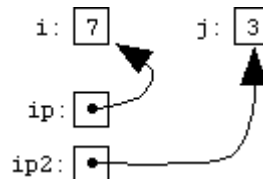
Now ip2 points where ip does; we've essentially made a “copy” of the arrow:



Now, if we set ip to point back to i again:

```
ip = &i;
```

the two arrows point to different places:



We can now see that the two assignments

```
ip2 = ip;
```

and

```
*ip2 = *ip;
```

do two very different things. The first would make ip2 again point to where ip points (in other words, back to i again). The second would store, at the location pointed to by ip2, a copy of the value pointed to by ip; in other words (if ip and ip2 still point to i and j respectively) it would set j to i's value, or 7.

It's important to keep very clear in your mind the distinction between a *pointer* and *what it points to*. You can't mix them. You can't “set ip to 5” by writing something like

```
ip = 5;          /* WRONG */
```

5 is an integer, but ip is a pointer. You probably wanted to “set *the value pointed to by* ip to 5,” which you express by writing

```
*ip = 5;
```

Similarly, you can't “see what ip is” by writing

```
printf("%d\n", ip);    /* WRONG */
```

Again, ip is a pointer-to-int, but %d expects an int. To print *what ip points to*, use

```
printf("%d\n", *ip);
```

Finally, a few more notes about pointer declarations. The `*` in a pointer declaration is related to, but different from, the contents-of operator `*`. After we declare a pointer variable

```
int *ip;
```

the expression

```
ip = &i
```

sets what `ip` points to (that is, which location it points to), while the expression

```
*ip = 5
```

sets the value of the location pointed to by `ip`. On the other hand, if we declare a pointer variable and include an initializer:

```
int *ip3 = &i;
```

we're setting the initial value for `ip3`, which is where `ip3` will point, so that initial value is a pointer. (In other words, the `*` in the declaration `int *ip3 = &i;` is not the contents-of operator, it's the indicator that `ip3` is a pointer.)

If you have a pointer declaration containing an initialization, and you ever have occasion to break it up into a simple declaration and a conventional assignment, do it like this:

```
int *ip3;  
ip3 = &i;
```

Don't write

```
int *ip3;  
*ip3 = &i;
```

or you'll be trying to mix pointer and the value to which it points

Also, when we write

```
int *ip;
```

although the asterisk affects `ip`'s type, it goes with the identifier name `ip`, not with the type `int` on the left. To declare two pointers at once, the declaration looks like

```
int *ip1, *ip2;
```

Some people write pointer declarations like this:

```
int* ip;
```

This works for one pointer, because C essentially ignores whitespace. But if you ever write

```
int* ip1, ip2;          /* PROBABLY WRONG */
```

it will declare one pointer-to-int ip1 and one *plain* int ip2, which is probably not what you meant.

What is all of this good for? If it was just for changing variables like i from 5 to 7, it would not be good for much. What it's good for, among other things, is when for various reasons we don't know exactly which variable we want to change.

**Program 8.1: A simple program to illustrate the relationship between two integer variables, their corresponding addresses and their associated pointers.**

```
#include<stdio.h>
main()
{
    int x=5;
    int y;
    int *px;      /* pointer to an integer */
    int *py;      /* pointer to an integer */

    px=&x;        /* assign address of x to px */
    y=*px;        /* assign value of x to y */
    py=&y;        /* assign address of y to py */

    printf("\nx=%d  &x=%u px=%u *px=%d", x, &x, px, *px);
    printf("\ny=%d  &y=%u py=%u *py=%d", y, &y, py, *py);
}
```

Execute this program and observe the result.

### Self Assessment Questions

1. Dereference operator is also known as \_\_\_\_\_.
2. Pointer is a variable containing address of another variable. (True/False)
3. State whether the following statements are correct: (Correct/Incorrect)  
int a, b;  
b=&a;



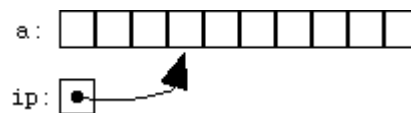
4. The contents-of operator `*` does not merely fetch values through pointers; it can also *set* values through pointers. (True/False)

### 8.3 Pointers and One-dimensional Arrays

Pointers do not have to point to single variables. They can also point at the cells of an array. For example, we can write

```
int *ip;  
int a[10];  
ip = &a[3];
```

and we would end up with `ip` pointing at the fourth cell of the array `a` (remember, arrays are 0-based, so `a[0]` is the first location). We could illustrate the situation like this:



We'd use this `ip` just like the one in the previous section: `*ip` gives us what `ip` points to, which in this case will be the value in `a[3]`.

#### 8.3.1 Pointer Arithmetic

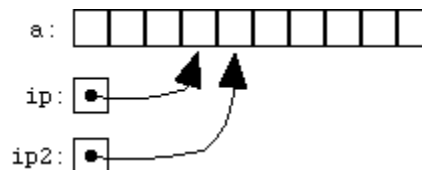
Once we have a pointer pointing into an array, we can start doing *pointer arithmetic*. Given that `ip` is a pointer to `a[3]`, we can add 1 to `ip`:

```
ip + 1
```

What does it mean to add one to a pointer? In C, it gives a pointer to the cell one farther on, which in this case is `a[4]`. To make this clear, let's assign this new pointer to another pointer variable:

```
ip2 = ip + 1;
```

Now the picture looks like this:



If we now do

```
*ip2 = 4;
```

we've set `a[4]` to 4. But it's not necessary to assign a new pointer value to a pointer variable in order to use it; we could also compute a new pointer value and use it immediately:

```
*(ip + 1) = 5;
```

In this last example, we've changed `a[4]` again, setting it to 5. The parentheses are needed because the unary “contents of” operator `*` has higher *precedence* (i.e., binds more tightly than) the addition operator. If we wrote `*ip + 1`, without the parentheses, we'd be fetching the value pointed to by `ip`, and adding 1 to that value. The expression `*(ip + 1)`, on the other hand, accesses the value one past the one pointed to by `ip`.

Given that we can add 1 to a pointer, it's not surprising that we can add and subtract other numbers as well. If `ip` still points to `a[3]`, then

```
*(ip + 3) = 7;
```

sets `a[6]` to 7, and

```
*(ip - 2) = 4;
```

sets `a[1]` to 4.

Up above, we added 1 to `ip` and assigned the new pointer to `ip2`, but there's no reason we can't add one to a pointer, and change the same pointer:

```
ip = ip + 1;
```

Now `ip` points one past where it used to (to `a[4]`, if we hadn't changed it in the meantime). The shortcuts work for pointers, too: we could also increment a pointer using

```
ip += 1;
```

or

```
ip++;
```

Of course, pointers are not limited to ints. It's quite common to use pointers to other types, especially `char`.

**Example 8.1:** Here is a program segment to compare two strings, character by character using pointers.

```
char *p1 = &str1[0], *p2 = &str2[0];
while(1)
{
    if(*p1 != *p2)
        return *p1 - *p2;
    if(*p1 == '\0' || *p2 == '\0')
        return 0;
    p1++;
    p2++;
}
```

The auto increment operator ++ (like its companion, --) makes it easy to do two things at once. We've seen idioms like `a[i++]` which accesses `a[i]` and simultaneously increments `i`, leaving it referencing the next cell of the array `a`. We can do the same thing with pointers: an expression like `*ip++` lets us access what `ip` points to, while simultaneously incrementing `ip` so that it points to the next element. The preincrement form works, too: `*++ip` increments `ip`, then accesses what it points to. Similarly, we can use notations like `*ip--` and `*--ip`.

**Example 8.2:** Here is a program segment to copy a string to another using pointer:

```
char *dp = &dest[0], *sp = &src[0];
while(*sp != '\0')
    *dp++ = *sp++;
*dp = '\0';
```

(One question that comes up is whether the expression `*p++` increments `p` or what it points to. The answer is that it increments `p`. To increment what `p` points to, you can use `(*p)++`.)

When you're doing pointer arithmetic, you have to remember how big the array the pointer points into is, so that you don't ever point outside it. If the array `a` has 10 elements, you can't access `a[50]` or `a[-1]` or even `a[10]` (remember, the valid subscripts for a 10-element array run from 0 to 9). Similarly, if `a` has 10 elements and `ip` points to `a[3]`, you can't compute or

access `ip + 10` or `ip - 5`. (There is one special case: you can, in this case, compute, but not access, a pointer to the nonexistent element just beyond the end of the array, which in this case is `&a[10]`. This becomes useful when you're doing pointer comparisons, which we'll look at next.)

**Program 8.2: Program to illustrate the relationship between an array and pointer.**

```
#include<stdio.h>
main()
{
    int a[10];
    int i;
    for(i=0;i<10;i++)
        scanf("%d",&a[i]);
    for(i=0;i<10;i++)
        printf("\ni=%d  a[i]=%d *(a+i)=%d &a[i]=%u a+i=%u", i, a[i], *(a+i), &a[i], a+i);
}
```

Execute this program and observe the result.

### 8.3.2 Pointer Subtraction and Comparison

As we've seen, you can add an integer to a pointer to get a new pointer, pointing somewhere beyond the original (as long as it is in the same array). For example, you might write

`ip2 = ip1 + 3;`

Applying a little algebra, you might wonder whether

`ip2 - ip1 = 3`

and the answer is, yes. When you subtract two pointers, as long as they point into the same array, the result is the number of elements separating them. You can also ask (again, as long as they point into the same array) whether one pointer is greater or less than another: one pointer is "greater than" another if it points beyond where the other one points. You can also compare pointers for equality and inequality: two pointers are equal if they point to the same variable or to the same cell in an array, and are (obviously) unequal if they don't. (When testing for equality or inequality, the two pointers do not have to point into the same array.)

One common use of pointer comparisons is when copying arrays using pointers.

**Example 8.3:** Here is a code fragment which copies 10 elements from array1 to array2, using pointers. It uses an end pointer, endptr, to keep track of when it should stop copying.

```
int array1[10], array2[10];
int *ip1, *ip2 = &array2[0];
int *endptr = &array1[10];
for(ip1 = &array1[0]; ip1 < endptr; ip1++)
    *ip2++ = *ip1;
```

As we mentioned, there is no element array1[10], but it is legal to compute a pointer to this (nonexistent) element, as long as we only use it in pointer comparisons like this (that is, as long as we never try to fetch or store the value that it points to.)

**Program 8.3:** In the following program, two different pointer variables point to the first and the last element of an integer array.

```
#include<stdio.h>
main()
{
    int *px, *py;
    int a[5]={1, 2, 3, 4, 5};
    px=&a[0];
    py=&a[4];
    printf("px=%u py=%u", px, py);
    printf("\n\n py-px=%u", py-px);
}
```

Execute this program and observe the result.

### 8.3.3 Similarities between Pointers and One-dimensional Arrays

There are a number of similarities between arrays and pointers in C. If you have an array

```
int a[10];
```

you can refer to a[0], a[1], a[2], etc., or to a[i] where i is an int. If you declare a pointer variable ip and set it to point to the beginning of an array:

```
int *ip = &a[0];
```

you can refer to `*ip`, `*(ip+1)`, `*(ip+2)`, etc., or to `*(ip+i)` where `i` is an `int`.

There are also differences, of course. You cannot assign two arrays; the code

```
int a[10], b[10];
a = b;                      /* WRONG */
```

is illegal. As we've seen, though, you *can* assign two pointer variables:

```
int *ip1, *ip2;
ip1 = &a[0];
ip2 = ip1;
```

Pointer assignment is straightforward; the pointer on the left is simply made to point wherever the pointer on the right does. We haven't copied the data pointed to (there's still just one copy, in the same place); we've just made two pointers point to that one place.

The similarities between arrays and pointers end up being quite useful, and in fact C builds on the similarities, leading to what is called "the equivalence of arrays and pointers in C." When we speak of this "equivalence" we do not mean that arrays and pointers are the same thing (they are in fact quite different), but rather that they can be used in related ways, and that certain operations may be used between them.

The first such operation is that it is possible to (apparently) assign an array to a pointer:

```
int a[10];
int *ip;
ip = a;
```

What can this mean? In that last assignment `ip = a`, C defines the result of this assignment to be that `ip` receives a pointer to the first element of `a`. In other words, it is as if you had written

```
ip = &a[0];
```

The second facet of the equivalence is that you can use the "array subscripting" notation `[i]` on pointers, too. If you write `ip[3]`

it is just as if you had written

```
*(ip + 3)
```

So when you have a pointer that points to a block of memory, such as an array or a part of an array, you can treat that pointer “as if” it *were* an array, using the convenient `[i]` notation. In other words, at the beginning of this section when we talked about `*ip`, `*(ip+1)`, `*(ip+2)`, and `*(ip+i)`, we could have written `ip[0]`, `ip[1]`, `ip[2]`, and `ip[i]`. As we'll see, this can be quite useful (or at least convenient).

The third facet of the equivalence (which is actually a more general version of the first one we mentioned) is that *whenever* you mention the name of an array in a context where the “value” of the array would be needed, C automatically generates a pointer to the first element of the array, as if you had written `&array[0]`. When you write something like

```
int a[10];
int *ip;
ip = a + 3;
```

it is as if you had written

```
ip = &a[0] + 3;
```

which (and you might like to convince yourself of this) gives the same result as if you had written

```
ip = &a[3];
```

For example, if the character array

```
char string[100];
```

contains some string, here is another way to find its length:

```
int len;
char *p;
for(p = string; *p != '\0'; p++)
    ;
len = p - string;
```

After the loop, `p` points to the `'\0'` terminating the string. The expression `p - string` is equivalent to `p - &string[0]`, and gives the length of the string. (Of course, we could also call `strlen`; in fact here we've essentially written another implementation of `strlen`.)

**Self Assessment Questions**

5. You can perform any type of arithmetic operation on pointers. (True or False)
6. For an int array a[10], If you declare a pointer variable ip then you can set it to point to the beginning of an array by assigning: `int *ip = _____`.
7. One common use of pointer comparisons is when copying arrays using pointers. (True/False)
8. To increment what p points to, you can use the expression \_\_\_\_\_.

**8.4 Summary**

A pointer is a variable that points at, or refers to, another variable. The general format of a simple pointer declaration is: `datatype *variablename`. We discover the value pointed to by a pointer using the “contents-of” operator, `*`. Pointers are very useful when you want to refer to some other variable by pointing at it. Pointers not only point to single variables but can also point at the cells of an array. Pointers are also closely associated with the arrays and therefore provide an alternate way to access individual array elements. Once we have a pointer pointing into an array, we can do some pointer arithmetic-add a constant to a variable or subtract a constant from a pointer, or you can subtract two pointers. There are similarities between pointers and one dimension arrays in some aspects although they are entirely different concepts and there are differences also in other aspects.

**8.5 Terminal Questions**

1. How do you declare a pointer to a floating point quantity and a double precision quantity? Illustrate it.
2. What is the significance of the following declaration? Explain with an example.  
`int *p(int a)`
3. What is the significance of the following declaration? Explain with an example.  
`int (*p) (int a)`
4. Illustrate with an example how you use pointer arithmetic.



**8.6 Answers to Self Assessment Questions**

1. indirection operator.
2. True
3. Incorrect
4. True
5. False
6. &a[0];
7. True
8. (\*p)++.)

**8.7 Answers to Terminal Questions**

1. We declare a pointer to a floating point quantity and double precision quantity as follows:  
float \*fptr;  
double \*dptr;  
(Refer section 8. 2 and 8.3 for more details)
2. int \*p(int a) defines the function that accepts an integer argument and returns a pointer to an integer. (Refer section 8. 2 and 8.3 for more details)
3. int (\*p) (int a) indicates a pointer to a function that accepts an integer argument and returns an integer. (Refer section 8. 2 and 8.3 for more details)
4. You can apply addition, subtraction or comparison with pointers. (Refer section 8. 2 and 8.3 for more details)

**8.8 Exercise**

1. Write a program that uses the pointer arithmetic.
2. Write a program that can accept any 5 data and sort it in ascending order using pointers.
3. With the help of pointers, write a program that uses functions to swap the data.
4. What are the various operators that are used when implementing a pointer in a program? Explain with an example.