17

POINTERS

17.1 INTRODUCTION

As you know by now, variables are stored in memory. Each memory location has a numeric address, in much the same way that each element of an array has its own subscript. Variable names in C and other high-level languages enable the programmer to refer to memory locations by name, but the compiler must translate these names into addresses. A pointer is a variable that represents the location of a data item, such as a variable or an array element. Pointers are used frequently in C. 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.

17.2 OBJECTIVES

After going through this lesson you will be in a position to

- explain address operator
- define pointer declarations

- describe pointers and one-dimensional arrays
- define pointers and multidimensional arrays
- explain arrays of pointers

17.3 THE ADDRESS OPERATOR

Within a computer's memory, every stored data item occupies one or more contiguous memory cells. The number of memory cells required to store a data item depends on the type of data item. For example, a single character will typically be stored in 1 byte of memory, an integer requires two contiguous bytes. A floating point number may require four contigueous bytes, and a double precision quantity may require eight contiguous bytes.

Suppose 'a' is a variable that represents some particular data item. The compiler will automatically assign memory cells for this data item. The data item can be accessed if we know the location of the first memory cell. The address of a's memory location can be determined by the expression &v, where & is a unary operator called the **address operator**, that evaluates the address of its operand. Now let us assign the address of 'a' to another variable,

Thus

pa= &a;

This new variable is called a pointer to 'a', since it "points" to the locations where 'a' is stored in memory. However, pa represents a's address, not its value. Thus, pa is referred to as a pointer variable.

Address of a $\rightarrow$ value of a

pa a

The data item represented by 'a' (i.e the data item stored in a's memory cells) can be accessed by the expression *pa, where * is a unary operator, called the **indirection operator**, that operates only on a pointer variable. Therefore, *pa and 'a' both represent the same data item (ie the contents of the same memory cells). Further more if we write pa= &a and b = *pa, then 'a' and 'b' will both represents the same value i.e the value of a will indirectly be assigned to 'b'

Let us consider an example :

```
# include <stdio.h>
main ( )
{
int a=3;
int b;
int *pa; /* to an integer */
int *pb;/*pointer to an integer */
pa=&a;/* assign address of a to pa */
b=*pa;/* assign value of a to b*/
pb=&b;/* assign address of b to pb */
printf ("\n% d %d % d %d", a, &a,pa, *pa);
printf ("\n% d %d % d %d", b, &b, pb, *pb);
}
```

```
The output is :    3      F8E   F8E   3
                   3      F8C   F8C   3
```

The unary operators & and * are members of the same precedence group as the other unary operators i.e. -, ++, —, ! , size of and(type). The address operator (&) must act upon operands associated with unique addresses, such as ordinary variables or single array elements. Thus, the address operator cannot act upon arithmetic expression, such as 2*(a+b). The indirection operator(*) can only act upon operands that are pointers.

Let us consider an example

```
# include <stdio.h>
main()
{
int i=3,* j;k;    /*j is a pointer to integer */
j=&i;      /*j points to the location of i*/
k=*j ;          /*assign to k the value pointed to by j*/
*j=4            /assign 4 to the location pointed to by j*/
printf("i=%d, *j=%d, k=%d\n," i, *j, k);
}
```

The output is as follows:

```
i=4,*j=4, k=3
```

17.4 POINTER DECLARATION

Pointer variables, like all other variables, must be declared before they may be used in a C program. The interpretation of a pointer declaration is somewhat different than the interpretation of other variable declarations. When a pointer variable is declared, the variable name must be preceded by an asterisk(*). This identifies the fact that the variable is a pointer. The data type that appears in the declaration refers to the object of the pointer, i.e the data item that is stored in the address represented by the pointer, rather than the pointer itself.

Thus, a pointer declaration may be written in general terms as datatype *ptr;

Where ptr is the name of the pointer variable and data-type refers to the data type of the pointer's object.

For example,

```
float a,b;
float *pb;
```

The first line declares a and b to be floating-point variables. The second line declares pb to be a pointer variable whose object is a floating-point quantity i.e. 'pb' point to a floating point quantity. 'pb' represents an address, not a floating-point quantity.

Within a variable declaration, a pointer variable can be initialized by assigning it the address of another variable.

17.5 POINTERS AS ARGUMENTS

Pointers are often passed to a function as arguments. This allows data items within the calling portion of the program to be accessed by the function, altered within the function, and then returned to the calling portion of the program in altered form.

When an argument is passed by value, the data item is copied to the function. Thus any alteration made to the data item within the function is not carried over into the calling routine when an argument is passed by reference, however the address of a data item is passed to the function. The contents of that address can be accessed freely, either within the function or within the calling routine. Moreover, any change that is made to the data item will be recognized in both the function and the calling routine. Thus, the use of a pointer as a function argument permits the corresponding data item to be altered globally from within the function.

If formal arguments are pointers, then each must be preceded by an asterisk. Also, if a function declaration is included in the calling portion of the program, the data type of each argument that corresponds to a pointer must be followed by an asterisk.

Let us consider an example of passing by reference

```
#include <stdio.h>
main ()
{
int i,j;
i=2;
j=5;
printf("i=%d and j=%d\n", i,j);
/* Now exchange them */
swap(&i,&j);
printf("\nnow i=%d and j=%d\n", i,j);
}
swap(i,j)
int *i,*j;
{
int temp=*i; /* create temp and store into it the value  pointed
to by "i"*/
*i=*j; /*The value pointed to by j is stored in the location pointed
to by i*/
*j=temp;/* Assign temp to the location pointed to by j*/
}
```

The output is as follows:

```
i=2 and j=5
Now i=5 and j=2
```

If the formal parameters i and j of the swap function were declared merely as integers, and the main function passed only i and j rather than their addresses, the exchange made in swap would have no effect on the variables i and j in main. The variable temp in swap function is of type int, not int*. The values being exchanged are not the pointers, but the integers being pointed to by them. Also temp is initialized immediately upon declaration to the value pointed to by i.

Let us consider it again with the help of another example:-

The following program counts the no. of vowels and consonants in a line of text with the help of pointers.

```
# include <stdio.h>
main()
{
char str[80];
int vowels=0;
int consonants=0;
void scan_line(char str[ ], int *v, int *c);
printf("Enter a line of text\n");
scanf("%[ "\n]" , str);
scan_line(str, &vowels, &consonants);
printf("\No. of vowels %d",vowels);
printf("\No. of consonants %d", consonants);
}
void scan_line(char str[ ], int * v, int *c)
{
char c;
int count=0;
while  ((c=toupper(str[count]))!='\0')
{
if(c = = A !!     c= =E !!   c= =I !!     c= =O !!     c= =U)
++*v;
else  if(c>='A'&&  c<='Z')
++*c;
++ count;
}
return;
}
```

It is possible to pass a portion of an array, rather than an entire array, to a function. To do so, the address of the first array element to be passed must be specified as an argument. The remainder of the array, starting with the specified array element, will then be passed to the function.

INTEXT QUESTIONS

1. What numeric value is associated with every memory location ?
2. How are variable names translated to their corresponding addresses?
3. What must be done to a pointer variable before it can be put to use?
4. What is the role played by the & operator in a call to the scanf function ?
5. Can addresses be stored ? If so, where ?

17.6 POINTERS AND ONE MIDEMNSIONAL ARRAYS

Array name is really a pointer to the first element in that array. Therefore, if x is a one-dimensional array, then the address of the first array element can be expressed as either &x[0] or x. Similarly, the address of the second array element can be written as either & x [1] or as (x+1) and so on. Here we should keep it in mind that the expression (x+1) is a symbolic representation for an address specification rather than an arithmetic expression. When writing the address of an array element in the form(x+i), there is no need to concern with the number of memory cells associated with each type of array element, the C compiler adjusts for this automatically. The programmer must specify only the address of the first array element and the number of array elements beyond the first.

Since &x[i] and (x+i) both represent the address of the ith element of x then x [i] and *(x+i) both represent the contents of that address. The two terms are interchangeble.

When assigning a value to an array element such as x[i] the left side of the assignment statement may be written as either x[i]or *(x+i).

Thus a value may be assigned directly to an array element, or it may be assigned to the memory area whose address is that of the array element . Thus expression such as x,(x+i) and &x [i] cannot appear on the left side of an assignment statement.

Numerical array elements cannot be assigned initial values if the array is defined as a pointer variable. Therefore a conventional array definition is required if initial values will be assigned to the elements of a numerical array.

Let us assume that x is to be defined as a one-adimentional, 10 element array of integers. It is possible to define x as a pointer variable rather than as an array. Thus we can write int *x; instead of int x[10];

However x is not automatically assigned a memory block when it is defined as a pointer variable, To assign sufficient memory for x ,we can make use of the library function malloc as follows :

```
x= malloc (10*size of (int));
```

This function reserves a block of memory whose size is equivalent to the size of an integer quantity. The function return a pointer to a character. If the declaration is to include the assignment of initial values, then x must be defined as an array rather then as a pointer variable for example

```
          int x[10]= {1,2,3,4,5,6,7,8,9,10};
            or int x[]= {1,2,3,4,5,6,7,8,9,10};
```

For character arrays you can write them in the form of pointer as char*x="this is string";

17.7 ARITHMETICS ON POINTERS

An integer value can be added to or subtracted from a pointer variable. For example, px is a pointer variable representing the address of some variable x. We can write expressions such as a ++ px ,- - px (px +3),(px+i) and (px – i) where i is an integer variable. Each expression will represent an address located some distance from the original address represented by px. The exact distance will be the product of the integer quantity and the number. of bytes or words associated with the data item to which px points. One pointer variable can be subtracted from another provided both variables points to elements of the same array. The resulting value indicates the number of words or bytes separating the corresponding array elements.

Pointer variables can be compared provided both variables point to objects of the same data type. Such comparisons can be useful when both pointer variables point to elements of the same array. The following points must be kept in mind about operations on pointers:-

– A pointer variable can be assigned the address of an ordinary variable.

– A pointer variable can be assigned the value of another pointer variable provided both pointers point to objects of the same data type.

– An integer quantity can be added to or subtracted from a pointer variable.

– A pointer variable can be assigned a null (zero) value.

– One pointer variable can be subtracted from another provided both pointers point to elements of the same array.

– Two pointer variables can be compared provided both pointers point to objects of the same data type.

17.8 POINTERS AND MULTIDIMENSIONAL ARRAYS

A two dimensional array, is actually a collection of one dimensional arrays. Therefore, we can define a two dimensional array as a pointer to a group of contiguous one-dimensional arrays. A two dimensional array declaration can be written as

```
data type (*ptr) [expression2];
```

Notice the parentheses that surround the array name and the preceding asterisk in the pointer version of each declaration. These parentheses must be present. Without them we would be defining an array of pointers rather than a pointer to a group of arrays, since these particulars symbols (i.e. The square brackets and asterisk) would normally be evaluated right-to-by.

For example, if x is a two dimensional integer array having 10 rows and 20 columns, then we can declare x as

```
int(*x)[20];
```

rather than int x(10) (20);

Similarly, three dimmensional integer array can be written as int (*x) (20) (30)

An individual array element within a multi-dimensional array can be accessed by repeatedly using the indirection operator. If n is a 2D array having 10 rows and 20 columns, then item in row 2, column 5 can be accessed by writing either

```
x[2][5]
or
*(*(x+2)+5)
```

Here, (x+2) is a pointer to row 2. Therefore, the object of this pointer *(x+2), refers to the entire row. Since row 2 is a one-dimensional array, *(x+2) is actually a pointer to the first element in row 2. We now add 5 to this pointer. Hence,(*(x+2)x5) is a pointer to element 5 in row 2. The object of this pointer, (*(x+2)+5) therefore refers to the item in column 5 of row2 which is x[2][5].

INTEXT QUESTIONS

6. What is meant by passing by reference ?
7. Why is only the address of an array's first element necessary in order to access the entire array ?
8. If an array is declared as

   char x[10][12];

   What is referred to by x[5] ?

17.9 ARRAYS OF POINTERS

A multi-dimensional array can be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays. In such cases the newly defined array will have one less dimension than the original multi-dimensional array. Each pointer will indicate the beginning of a separate (n-1) dimensional array.

In general terms, a 2D array can be defined as a one dimensional array of pointers by writing.

data type *array [expression 1];

In this type of declaration, array name and its preceding asterisk are not enclosed in parentheses. Thus, a right-to-left rule first associates the pairs of square brackets with array, defining the named object as an array. The preceding asterisk then establishes that the array will contain pointers.

For example x is a 2D integer array having 10 rows and 20 columns. We can define x as a one-dimensional array of pointers by writing

int *x[10];

An individual array element, such as x[2][5], can be accessed by writing *(x[2]+5)

In this expression, x[2] is a pointer to the first element in row 2, so that (x[2]   +5) points to element 5 within row 2. The object of this pointer, *(x[2]+5), therefore refers to x[2][5].

Pointer arrays offer a particularly convenient method for storing strings. In this situation, each array element is a character-type pointer that indicates the beginning of a separate string. Each individual string can be accessed by referring to its corresponding pointer.

Let us consider an example of array of pointers: (To sort list of strings)

```
#include<stdio.h>
#include<stdlib.h>
main()
{
int i, n=0;
char *x[10];
int reorder(int n, char *x[]);
printf("Enter each string on a separate line,)
"Type\'END\' to finish");
do
{
x[n]=malloc(12*size of (char));
printf("\nstring %d", n+1);
scanf("%s", x[n]);
}
while ( stringcmp(x[n++], “END”));
reorder(- - n,x);
printf (or \n\n Recordered List:\n");
for(i=0; i<n; ++i)
printf(“\n string %d:%s”, i+1, x[i]);
}
reorder(int n, char * x[])
{
char *temp;
int i, item;
for(item=0; item <n-1; ++ item)
for(i=item +1; i<n; ++i)
if (stringcmp(x[item], x[i]>0)
{
temp=x[item];
x[item]=x[i];
x[i]=temp;
}
return;
}
```

An advantage to use arrays of pointers is that a fixed block of memory need not be reserved in advance, as is done when initializing a conventional array.

If some of the strings are particularly long, there is no need to worry about the possibility of exceeding some maximum specified string length. Arrays of this type are oftern referred to as **ragged arrays.**

17.10 PASSING FUNCTIONS TO OTHER FUNCTIONS

When a function declartion appears within another function, the name of the function being declared becomes a pointer to that function. Such pointer can be passed to other functions as arguments. This has the effect of passing one function to another , as though the first function was a variable. The first functiion can then be accessed within the second function. Successive calls to the second function can pas different pointers to the second funtion.

When a function accepts another function's name as an argument a formal argument declaration must identify that argument as a pointer to another function. A formal argument that is a pointer to a function can be declared as

data-type (*function name)();

Where data-type refers to the data type of the quantity returned by the function.

This function can then be accessed by means of the indirection operator. To do so the indirection operator must precede the function name. Both the indirection operator and the function name must be enclosed in parentheses that is

(*function-name)(argument 1, argument 2..., argument n);

In the last, we mention that pointer declaration can become complicated and some care is required for its interpretation. This is particularly take in case of declarations that involve functions or arrays.

One difficulty is the dusl use of parentheses. In particular, parentheses are used to indicate functions and they are used for nesting purposes within more complicated declarations. Thus the declaration

int *p(int a);

Indicates a function that accepts an integer argument, and returns a pointer to an integer. On the other hand, the declaration.

int(*p)(int a);

indicates a pointer to a function that accepts an integer argument and returns an integer. In this last declaration, the first pair of parentheses is used for nesting and the second pair is used to indicate a function.

17.11 WHAT YOU HAVE LEARNT

In this lesson you have learnt about pointers. You can now declare pointers very effciently. Now you can pass pointers to a function. You can do operations on pointer. Now you are familiar with arrays of pointers. You can use pointers with one as well as multi dimensional arrays.

17.12 TERMINAL QUESTIONS

1. What is meant by the address of a memory cell?
2. What kind of information does a pointer variable represent ?
3. How is a pointer variable declared ?
4. How can a function return a pointer to its calling routine ?
5. How is a multidimensional array defined in terms of a pointer to a collection of contiguous arrays of lower dimensioanlity ?
6. A 'C' program contains the following declaration :

   Static int x[8]= {10,20,30,40,50,60,70,80};

   (a) What is the meaning of x?

   (b) What is the meaning of (x+2) ?

   (c) What is the value of *x?

   (d) What is the value of (*x+2) ?

   (e) What is the value of *(x+2) ?
7. Write a program to find the length of a string using the concept of pointers
8. Write a program to copy a string using pointers.
9. Write a program to concatenate two strings using pointers.
10. Write a program to compare two strings using pointers.

17.13 KEY TO IN-TEXT QUESTIONS

1. its address
2. with the & operator
3. After it has been declared, it must be initialized
4. It passes the address of the variable being input, making it possible for the contents of that location to be changed.

5. Addresses can be stored in pointer variables.

6. Passing by reference is the term used to refer to the passing of a variable’s address to a function.

7. Since all the elements of an array are of the same type, and all the elements are stored continguously in memory, the location of the first element can be used to find the second element, the third element, and so forth.

8. x[5] is a pointer to the sixth row (row #5) of the array.