UNIT 7 POINTERS

Structure

7.0 INTRODUCTION

If you want to be proficient in the writing of code in the C programming language, you must have a thorough working knowledge of how to use pointers. One of those things, beginners in C find difficult is the concept of pointers. The purpose of this unit is to provide an introduction to pointers and their efficient use in the C programming. Actually, the main difficulty lies with the C's pointer terminology than the actual concept.

C uses pointers in three main ways. First, they are used to create *dynamic data structures*: data structures built up from blocks of memory allocated from the heap at run-time. Second, C uses pointers to handle *variable parameters* passed to functions. And third, pointers in C provide an alternative means of accessing information stored in arrays, which is especially valuable when you work with strings.

A normal variable is a location in memory that can hold a value. For example, when you declare a variable *i* as an integer, four bytes of memory is set aside for it. In your program, you refer to that location in memory by the name *i*. At the machine level, that location has a memory address, at which the four bytes can hold one integer value. A *pointer* is a variable that points to another variable. This means that it holds the memory address of another variable. Put another way, the pointer does not hold a value in the traditional sense; instead, it holds the address of another variable. It points to that other variable by holding its address.

Because a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That addresses points to a value. There is the pointer and the value pointed to. As long as you're careful to ensure that the pointers in your programs always point to valid memory locations, pointers can be useful, powerful, and relatively trouble-free tools.

We will start this unit with a basic introduction to pointers and the concepts surrounding pointers, and then move on to the three techniques described above. Thorough knowledge of the *pointers* is very much essential for your future courses like the *data structures etc.*.

7.1 OBJECTIVES

After going through this unit you should be able to:

- understand the concept and use pointers;
- address and use of indirection operators;
- make pointer type declaration, assignment and initialization;
- use null pointer assignment;
- use the pointer arithmetic;
- handle pointers to functions;
- see the underlying unit of arrays and pointers; and
- understand the concept of dynamic memory allocation.

7.2 POINTERS AND THEIR CHARACTERISTICS

Computer's memory is made up of a sequential collection of storage cells called bytes. Each byte has a number called an address associated with it. When we declare a variable in our program, the compiler immediately assigns a specific block of memory to hold the value of that variable. Since every cell has a unique address, this block of memory will have a unique starting address. The size of this block depends on the range over which the variable is allowed to vary. For example, on 32 bit PC's the size of an integer variable is 4 bytes. On older 16 bit PC's integers were 2 bytes. In C the size of a variable type such as an integer need not be the same on all types of machines. If you want to know the size of the various data types on your system, running the following code given in the Example 7.1 will give you the information.

Example 7.1

Write a program to know the size of the various data types on your system.

```
# include <stdio.h>
main()
{
  printf("n Size of a int = %d bytes", sizeof(int));
  printf("\n Size of a float = %d bytes", sizeof(float));
  printf("\n Size of a char = %d bytes", sizeof(char));
}
```

OUTPUT

```
Size of int = 2 bytes
Size of float = 4 bytes
Size of char = 1 byte
```

An *ordinary variable* is a location in memory that can hold a value. For example, when you declare a variable *num* as an integer, the compiler sets aside 2 bytes of memory (depends up the PC) to hold the value of the integer. In your program, you refer to that location in memory by the name *num*. At the machine level that location has a memory address.

int num = 100;

We can access the value 100 either by the name num or by its memory address. Since addresses are simply digits, they can be stored in any other variable. Such variables that hold addresses of other variables are called *Pointers*. In other words, a *pointer* is simply a variable that contains an address, which is a location of another variable in memory. A pointer variable "points to" another variable by holding its address. Since a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That addresses points to a value. There is a pointer and the value pointed to. This fact can be a little confusing until you get comfortable with it, but once you get familiar with it, then it is extremely easy and very powerful. One good way to visualize this concept is to examine the figure 7.1 given below:

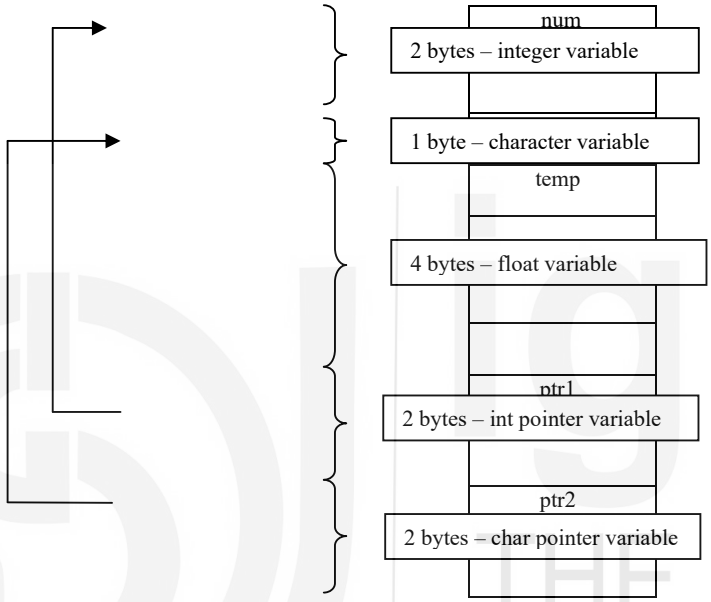


Figure 7.1: Concept of pointer variables

Let us see the important features of the pointers as follows:

Characteristic features of Pointers:

With the use of pointers in programming,

i. The program execution time will be faster as the data is manipulated with the help of addresses directly.
ii. Will save the memory space.
iii. The memory access will be very efficient.
iv. Dynamic memory is allocated.

7.3 THE ADDRESS AND INDIRECTION OPERATORS

Now we will consider how to determine the address of a variable. The operator that is available in C for this purpose is "&" (*address of*) operator. The operator & and the immediately preceding variable returns the address of the variable associated with it. C's other unary pointer operator is the "*", also called as *value at address* or indirection operator. It returns a value stored at that address. Let us look into the illustrative example given below to understand how they are useful.

Example 7.2

Write a program to print the address associated with a variable and value stored at that address.

```
/* Program to print the address associated with a variable and value stored at that address*/

# include <stdio.h>
main()
{
  int qty = 5;
  printf("Address of qty = %u\n",&qty);
  printf("Value of qty = %d \n",qty);
  printf("Value of qty = %d",*(&qty));
 }
```

OUTPUT

```
Address of qty = 65524
Value of qty = 5
Value of qty = 5
```

Look at the *printf* statement carefully. The format specifier *%u* is taken to increase the range of values the address can possibly cover. The system-generated address of the variable is not fixed, as this can be different the next time you execute the same program. Remember unary operator operates on single operands. When *&* is preceded by the variable qty, has returned its address. Note that the *&* operator can be used only with simple variables or array elements. It cannot be applied to expressions, constants, or register variables.

Observe the third line of the above program. **(&qty)* returns the value stored at address 65524 i.e. 5 in this case. Therefore, *qty* and **(&qty)* will both evaluate to 5.

7.4 POINTER TYPE DECLARATION AND ASSIGNMENT

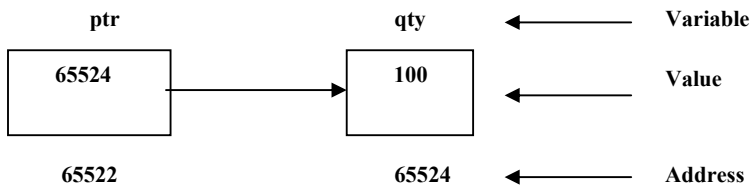
We have seen in the previous section that *&qty* returns the address of *qty* and this address can be stored in a variable as shown below:

ptr = &qty;

In C, every variable must be declared for its data type before it is used. Even this holds good for the pointers too. We know that *ptr* is not an ordinary variable like any integer variable. We declare the data type of the pointer variable as that of the type of the data that will be stored at the address to which it is pointing to. Since *ptr* is a variable, which contains the address of an integer variable *qty*, it can be declared as:

*int *ptr;*

where *ptr* is called a *pointer variable*. In C, we define a pointer variable by preceding its name with an asterisk(*). The "*" informs the compiler that we want a pointer variable, i.e. to set aside the bytes that are required to store the address in memory. The *int* says that we intend to use our pointer variable to store the address of an integer. Consider the following memory map:



Let us look into an example given below:

Example 7.3

```
/* Program below demonstrates the relationships we have discussed so far */

# include <stdio.h>
main()
{
  int qty = 5;
  int *ptr;          /* declares ptr as a pointer variable that points to an integer variable
*/
  ptr = &qty;   /* assigning qty's address to ptr -> Pointer Assignment */

  printf("Address of qty = %u \n", &qty);
  printf("Address of qty = %u \n", ptr);
  printf("Address of ptr = %u \n", &ptr);
  printf("Value of ptr = %d \n", ptr);
  printf("Value of qty = %d \n", qty);
  printf("Value of qty = %d \n", *(&qty));
  printf("Value of qty = %d", *ptr);
}
```

OUTPUT

```
Address of qty = 65524
Address of ptr = 65522
Value of ptr = 65524
Value of qty = 5
Value of qty = 5
Value of qty = 5
```

Try this as well:

Example 7.4

```
/* Program that tries to reference the value of a pointer even though the pointer is
uninitialized */

# include <stdio.h>
main()
{
   int *p;   /* a pointer to an integer */
    *p = 10;
   printf("the value is %d", *p);
   printf("the value is %u",p);
}
```

This gives you an error. The pointer *p* is uninitialized and points to a random location in memory when you declare it. It could be pointing into the system stack, or the global variables, or into the program's code space, or into the operating system.

When you say **p=10;* the program will simply try to write a 10 to whatever random location ***p*** points to. The program may explode immediately. It may subtly corrupt data in another part of your program and you may never realize it. Almost always, an uninitialized pointer or a bad pointer address causes the fault.

This can make it difficult to track down the error. Make sure you initialize all pointers to a valid address before dereferencing them.

Within a variable declaration, a pointer variable can be initialized by assigning it the address of another variable. Remember the variable whose address is assigned to the pointer variable must be declared earlier in the program. In the example given below, let us assign the pointer *p* with an address and also a value 10 through the **p*.

Example 7.5

Let us say,

```
int x; /* x is initialized to a value 10*/
p = &x;   /* Pointer declaration & Assignment */
*p=10;
```

Let us write the complete program as shown below:

```
# include <stdio.h>
main()
{
   int *p;   /* a pointer to an integer */
   int x;
   p = &x;
   *p=10;
   printf("The value of x is %d",*p);
   printf("\nThe address in which the x is stored is %d",p);
}
```

OUTPUT

```
The value of x is 10
The address in which the x is stored is 52004
```

This statement puts the value of 20 at the memory location whose address is the value of *px*. As we know that the value of *px* is the address of *x* and so the old value of *x* is replaced by 20. This is equivalent to assigning 20 to *x*. Thus we can change the value of a variable *indirectly* using a pointer and the *indirection operator.*

7.4.1 Pointer to a Pointer

The concept of pointer can be extended further. As we have seen earlier, a pointer variable can be assigned the address of an ordinary variable. Now, this variable itself could be another pointer. This means that a pointer can contain address of another pointer. The following program will makes you the concept clear.

Example 7.6

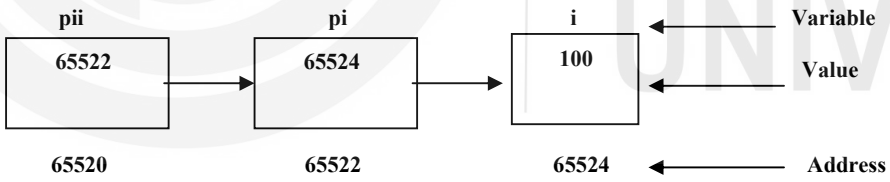
```
/* Program that declares a pointer to a pointer */

# include<stdio.h>
main()
{
```

```
 int i = 100;
 int *pi;
 int **pii;
 pi = &i;
 pii = &pi;

 printf ("Address of i = %u \n", &i);
 printf("Address of i = %u \n", pi);
 printf("Address of i = %u \n", *pii);
 printf("Address of pi = %u \n", &pi);
 printf("Address of pi = %u \n", pii);
 printf("Address of pii = %u \n", &pii);
 printf("Value of i = %d \n", i);
 printf("Value of i = %d \n", *(&i));
 printf("Value of i = %d \n", *pi);
 printf("Value of i = %d", **pii);
}
```

OUTPUT

```
Address of i = 65524
Address of i = 65524
Address of i = 65524
Address of pi = 65522
Address of pi = 65522
Address of pii = 65520

Value of i = 100
Value of i = 100
Value of i = 100
Value of i = 100
```

Consider the following memory map for the above shown example:



7.4.2 Null Pointer Assignment

It does make sense to assign an integer value to a pointer variable. An exception is an assignment of 0, which is sometimes used to indicate some special condition. A macro is used to represent a null pointer. That macro goes under the name *NULL*. Thus, setting the value of a pointer using the *NULL*, as with an assignment statement such as *ptr = NULL*, tells that the pointer has become a *null* pointer. Similarly, as one can test the condition for an integer value as zero or not, like *if (i == 0)*, as well we can test the condition for a null pointer using *if(ptr == NULL)* or you can even set a pointer to *NULL* to indicate that it's no longer in use. Let us see an example given below.

Example 7.7

```
# include<stdio.h>
# define NULL 0
```

```
main()
{
  int *pi = NULL;
  printf("The value of pi is %u", pi);
}
```

OUTPUT

The value of pi is 0

Check Your Progress 1

1. How is a pointer variable being declared? What is the purpose of data type included in the pointer declaration?

   ……

   ……

   ……

   ……

2. What would be the output of following programs?

   (i)
   ```
   void main()
   {
      int i = 5;
      printf("Value of i = %d   Address of i = %u", i, &i);
      &i = 65534;
      printf("\n New value of i = %d  New Address of i = %u", i, &i);
      }
   ```

   (ii)
   ```
   void main()
   {
      int *i, *j;
      j = i * 2;
      printf("j = %u", j);
   }
   ```

   ……

   ……

   ……

3. Explain the effect of the following statements:

   (i) `int x = 10, *px = &x;`

   (ii) `char *pc;`

   (iii)
   ```
   int x;
   void *ptr = &x;
   *(int *) ptr = 10;
   ```

……

……

……

7.5 POINTER ARITHMETIC

Pointer variables can also be used in arithmetic expressions. The following operations can be carried out on pointers:

1. Pointers can be incremented or decremented to point to different locations like

```
ptr1 = ptr2 + 3;
ptr ++;
-- ptr;
```

However, *ptr++* will cause the pointer *ptr* to point the next address value of its type. For example, if *ptr* is a pointer to float with an initial value of 65526, then after the operation *ptr* **++** or *ptr* **=** *ptr+1*, the value of *ptr* would be 65530. Therefore, if we increment or decrement a pointer, its value is increased or decreased by the length of the data type that it points to.

2. If *ptr1* and *ptr2* are properly declared and initialized pointers, the following operations are valid:

```
res = res + *ptr1;
*ptr1 = *ptr2 + 5;
prod = *ptr1 * *ptr2;
quo = *ptr1 /  *ptr2;
```

Note that there is a blank space between / and * in the last statement because if you write /* together, then it will be considered as the beginning of a comment and the statement will fail.

3. Expressions like *ptr1 == ptr2*, *ptr1 < ptr2*, and *ptr2 != ptr1* are permissible provided the pointers *ptr1* and *ptr2* refer to same and related variables. These comparisons are common in handling arrays.

Suppose *p1* and *p2* are pointers to related variables. The following operations cannot work with respect to pointers:

1. Pointer variables cannot be added. For example, *p1 = p1 + p2* is not valid.

2. Multiplication or division of a pointer with a constant is not allowed. For example, *p1 * p2* or *p2 / 5* are invalid.

3. An invalid pointer reference occurs when a pointer's value is referenced even though the pointer doesn't point to a valid block. Suppose *p* and *q* are two pointers. If we say, *p = q;* when *q* is uninitialized. The pointer *p* will then become uninitialized as well, and any reference to ***p* is an invalid pointer reference.

7.6 PASSING POINTERS TO FUNCTIONS

As we have studied in the FUNCITONS that arguments can generally be passed to functions in one of the two following ways:

1. Pass by value method
2. Pass by reference method

In the first method, when arguments are passed by value, a copy of the *values* of actual arguments is passed to the calling function. Thus, any changes made to the variables inside the function will have no effect on variables used in the actual argument list.

However, when arguments are passed by reference (i.e. when a pointer is passed as an argument to a function), the *address* of a variable is passed. The contents of that address can be accessed freely, either in the called or calling function. Therefore, the function called by reference can change the value of the variable used in the call.

Here is a simple program that illustrates the difference.

Example 7.8

Write a program to swap the values using the pass by value and pass by reference methods.

```
/* Program that illustrates the difference between ordinary arguments, which are
passed by value, and pointer arguments, which are passed by reference */

# include <stdio.h>
main()
{
        int x = 10;
        int y = 20;
       void swapVal( int, int );          /* function prototype */
       void swapRef( int *, int * );      /*function prototype*/
        printf("PASS BY VALUE METHOD\n");
        printf("Before calling function swapVal    x=%d   y=%d",x,y);
        swapVal(x, y);             /* copy of the arguments are passed */
        printf("\nAfter calling function swapVal   x=%d   y=%d",x,y);
              printf("\n\nPASS BY REFERENCE METHOD");
        printf("\nBefore calling function swapRef   x=%d   y=%d",x,y);
        swapRef(&x,&y);      /*address of arguments are passed */
        printf("\nAfter calling function swapRef     x=%d   y=%d",x,y);
}

/* Function using the pass by value method*/
void swapVal(int x, int y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
        printf("\nWithin function swapVal          x=%d   y=%d",x,y);
        return;
}

/*Function using the pass by reference method*/
void swapRef(int *px, int *py)
{
         int temp;
         temp = *px;
         *px = *py;
         *py = temp;
          printf("\nWithin function swapRef       *px=%d   *py=%d",*px,*py);
          return;
}
```

OUTPUT

```
PASS BY VALUE METHOD
Before calling function swapVal   x=10       y=20
Within function swapVal           x=20       y=10
After calling function swapVal    x=10       y=20

PASS BY REFERENCE METHOD
Before calling function swapRef   x=10       y=20
Within function swapRef           *px=20     *py=10
After calling function swapRef    x=20       y=10
```

This program contains two functions, *swapVal* and *swapRef*.

In the function *swapVal*, arguments *x* and *y* are passed by *value*. So, any changes to the arguments are local to the function in which the changes occur. Note the values of *x* and *y* remain unchanged even after exchanging the values of *x* and *y* inside the function ***swapVal***.

Now consider the function *swapRef*. This function receives two *pointers* to integer variables as arguments identified as pointers by the indirection operators that appear in argument declaration. This means that in the function *swapRef*, arguments *x* and *y* are passed by *reference.* So, any changes made to the arguments inside the function *swapRef* are reflected in the function *main().* Note the values of *x* and *y* is interchanged after the function call *swapRef*.

7.6.1 A Function returning more than one value

Using *call by reference* method we can make a function return more than one value at a time, which is not possible in the *call by value* method. The following program will makes you the concept very clear.

Example 7.9

Write a program to find the perimeter and area of a rectangle, if length and breadth are given by the user.

```
/* Program to find the perimeter and area of a rectangle*/

#include <stdio.h>
void main()
{
float len,br;
float peri, ar;
void periarea(float length, float breadth, float *, float *);
printf("\nEnter the length and breadth of  a rectangle in metres: \n");
scanf("%f %f",&len,&br);
periarea(len,br,&peri,&ar);
printf("\nPerimeter of the rectangle is %f metres", peri);
printf("\nArea of the rectangle is %f sq. metres", ar);
}

void periarea(float length, float breadth, float *perimeter, float *area)
{
*perimeter = 2 *(length +breadth);
*area = length * breadth;
}
```

OUTPUT

```
Enter the length and breadth of  a rectangle in metres:
23.0 3.0
Perimeter of the rectangle is 52.000000 metres
Area of the rectangle is 69.000000 sq. metres
```

Here in the above program, we have seen that the function *periarea* is returning two values. We are passing the values of *len* and *br* but, addresses of peri and *ar*. As we are passing the addresses of *peri* and *ar*, any change that we make in values stored at addresses contained in the variables ***perimeter* and ***area*, would make the change effective even in *main()* also.

7.6.2 Function returning a pointer

A function can also return a pointer to the calling program, the way it returns an int, a float or any other data type. To return a pointer, a function must explicitly mention in the calling program as well as in the function prototype. Let's illustrate this with an example:

Example: 7.10

Write a program to illustrate a function returning a pointer.

```
/*Program that shows how a function returns a pointer */

# include<stdio.h>

void main()
{
  float *a;
  float *func();        /* function prototype */
  a = func();
  printf("Address = %u", a);
}
float *func()
{
   float r = 5.2;
   return(&r);
}
```

OUTPUT

```
Address = 65516
```

This program only shows how a function can return a pointer. This concept will be used later while handling arrays.

Check Your Progress 2

1. Tick mark(√)whether each of the following statements are true or false.

(i) An integer is subtracted from a pointer variable. ☐True ☐False

(ii) Pointer variables can be compared. ☐True ☐False

(iii) Pointer arguments are passed by value. ☐True ☐False

(iv) Value of a local variable in a function can be changed by another function. ☐ True ☐ False

(v) A function can return more than one value. ☐ True ☐ False

(vi) A function can return a pointer. ☐ True ☐ False

7.7 ARRAYS AND POINTERS

Pointers and arrays are so closely related. An array declaration such as *int arr[5]* will lead the compiler to pick an address to store a sequence of 5 integers, and *arr* is a name for that address. The array name in this case is the *address* where the sequence of integers starts. Note that the value is not the first integer in the sequence, nor is it the sequence in its entirety. The value is just an address.

Now, if *arr* is a one-dimensional array, then the address of the first array element can be written as *&arr[0]* or simply *arr*. Moreover, the address of the second array element can be written as *&arr[1]* or simply *(arr+1).* In general, address of array element *(i+1)* can be expressed as either *&arr[i]* or as *(arr+ i).* Thus, we have two different ways for writing the address of an array element. In the latter case i.e, expression *(arr+ i)* is a symbolic representation for an address rather than an arithmetic expression. Since *&arr[i]* and *(ar+ i)* both represent the address of the i^{th} element of *arr,* so *arr[i]* and **(ar + i)* both represent the contents of that address i.e., the value of i^{th} element of *arr*.

Note that it is not possible to assign an arbitrary address to an array name or to an array element. Thus, expressions such as *arr*, *(arr+ i)* and *arr[i]* cannot appear on the left side of an assignment statement. Thus we cannot write a statement such as:

```
&arr[0] = &arr[1];      /* Invalid */
```

However, we can assign the value of one array element to another through a pointer, for example,

```
ptr = &arr[0];      /*  ptr is a pointer to arr[ 0]  */
arr[1] = *ptr;      /* Assigning the value stored at address to arr[1]  */
```

Here is a simple program that will illustrate the above-explained concepts:

Example 7.11

```
/* Program that accesses array elements of a one-dimensional array using pointers */

# include<stdio.h>
main()
{

  int arr[ 5 ] = {10, 20, 30, 40, 50};
  int i;

  for(i = 0;  i < 5;  i++)
    {
        printf("i=%d\t arr[i]=%d\t *(arr+i)=%d\t", i, arr[i], *(arr+i));
        printf("&arr[i]=%u\t arr+i=%u\n", &arr[i],(arr+i));     }
}
```

OUTPUT:

```
i=0    arr[i]=10    *(arr+i)=10    &arr[i]=65516    arr+i=65516
i=1    arr[i]=20    *(arr+i)=20    &arr[i]=65518    arr+i=65518
i=2    arr[i]=30    *(arr+i)=30    &arr[i]=65520    arr+i=65520
i=3    arr[i]=40    *(arr+i)=40    &arr[i]=65522    arr+i=65522
i=4    arr[i]=50    *(arr+i)=50    &arr[i]=65524    arr+i=65524
```

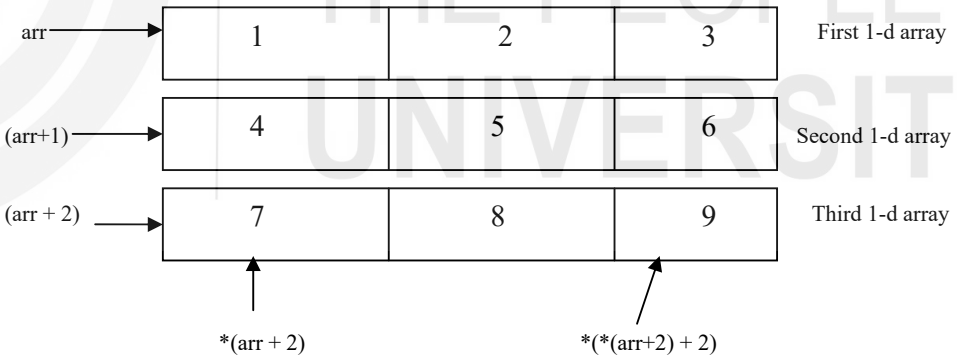
Note that *i* is added to a pointer value (address) pointing to integer data type (i.e., the array name) the result is the pointer is increased by i times the size (in bytes) of integer data type. Observe the addresses 65516, 65518 and so on. So if *ptr* is a char pointer, containing addresses *a*, then *ptr+1* is *a+1*. If *ptr* is a float pointer, then *ptr+ 1* is *a+ 4*.

Pointers and Multidimensional Arrays

C allows multidimensional arrays, lays them out in memory as contiguous locations, and does more behind the scenes address arithmetic. Consider a 2-dimensional array.

int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

The compiler treats a 2 dimensional array as an array of arrays. As you know, an array name is a pointer to the first element within the array. So, **arr** points to the first 3-element array, which is actually the first row (i.e., row 0) of the two-dimensional array. Similarly, *(arr + 1)* points to the second 3-element array (i.e., row 1) and so on. The value of this pointer, *(arr + 1), refers to the entire row. Since row 1 is a one-dimensional array, *(arr + 1)* is actually a pointer to the first element in row 1. Now add 2 to this pointer. Hence, *(*(arr + 1) + 2)* is a pointer to element 2 (i.e., the third element) in row 1. The value of this pointer, **(*(arr + 1) + 2),* refers to the element in column 2 of row 1. These relationships are illustrated below:



7.8 ARRAY OF POINTERS

The way there can be an array of integers, or an array of float numbers, similarly, there can be array of pointers too. Since a pointer contains an address, an array of pointers would be a collection of addresses. For example, a multidimensional array can be expressed in terms of an array of pointers rather than a pointer to a group of contiguous arrays.

Two-dimensional array can be defined as a one-dimensional array of integer pointers by writing:

*int *arr[3];*

rather than the conventional array definition,

int arr[3][5];

Similarly, an n-dimensional array can be defined as (n-1)-dimensional array of pointers by writing

*data-type *arr[subscript 1] [subscript 2]…. [subscript n-1];*

The subscript1, subscript2 indicate the maximum number of elements associated with each subscript.

Example 7.12

Write a program in which a two-dimensional array is represented as an array of integer pointers to a set of single-dimensional integer arrays.

/* Program calculates the difference of the corresponding elements of two table of integers */

```
# include <stdio.h>
# include <stdlib.h>
# define MAXROWS 3
void main()
{
     int *ptr1[MAXROWS], *ptr2 [MAXROWS], *ptr3 [MAXROWS];
     int rows, cols, i, j;
     void inputmat(int *[ ], int, int);
     void dispmat(int *[ ], int, int);
     void calcdiff(int *[ ], int *[ ], int *[ ], int, int);

     printf("Enter no. of rows & columns \n");
     scanf("%d%d", &rows, &cols);

     for(i = 0; i < rows; i++)
     {
          ptr1[ i ] =(int *) malloc(cols * sizeof(int));
          ptr2[ i ] =(int *) malloc(cols * sizeof(int));
          ptr3[ i ] =(int *) malloc(cols * sizeof(int));
     }

     printf("Enter values in first matrix \n");
     inputmat(ptr1, rows, cols);
     printf("Enter values in second matrix \n");
     inputmat(ptr2, rows, cols);
     calcdiff(ptr1, ptr2, ptr3, rows, cols);
     printf("Display difference of the two matrices \n");
     dispmat(ptr3, rows, cols);
}

 void inputmat(int *ptr1[MAXROWS], int m, int n)
 {
      int i, j;
      for(i = 0; i < m; i++)
      {
           for(j = 0; j < n; j++)
           {
                scanf("%d",(*(ptr1 + i) + j));
```

```
            }
        }
        return;
}

void dispmat(int *ptr3[  MAXROWS ], int m, int n)
{
    int i, j;
    for(i = 0; i < m; i++)
    {
        for(j = 0; j < n; j++)
        {
            printf("%d ", *(*(ptr3 + i) + j));
        }
        printf("\n");
    }
    return;
}

void calcdiff(int *ptr1[ MAXROWS ], int *ptr2 [ MAXROWS ],
                        int *ptr3[MAXROWS], int m, int n)
{
    int i, j;
    for(i = 0; i < m; i++)
    {
        for(j = 0; j < n; j++)
        {
            *(*(ptr3 + i) + j) = *(*(ptr1 + i) + j) - *(*(ptr2 + i) + j);
        }
    }
    return;
}
```

OUTPUT

```
Enter no. of rows & columns
3  3
Enter values in first matrix
2    6   3
5    9   3
1    0   2
Enter values in second matrix
3    5   7
2    8   2
1    0   1
Display difference of the two matrices
-1   1  -4
 3   1   1
 0   0   1
```

In this program, *ptr1*, *ptr2*, *ptr3* are each defined as an array of pointers to integers. Each array has a maximum of MAXROWS elements. Since each element of ptr1, ptr2, ptr3 is a pointer, we must provide each pointer with enough memory for each row of integers. This can be done using the library function *malloc* included in *stdlib.h* header file as follows:

*ptr1[i] =(int *) malloc(cols * sizeof(int));*

This function reserves a block of memory whose size(in bytes) is equivalent to *cols * sizeof(int).* Since cols = 3, so 3 * 2(size of int data type) i.e., 6 is allocated to each *ptr1[1], ptr1[2]* and *ptr1[3].* This *malloc* function returns a pointer of type *void*. This means that we can assign it to any type of pointer. In this case, the pointer is type-casted to an integer type and assigned to the pointer *ptr1[1], ptr1[2]* and *ptr1[3].* Now, each of *ptr1[1], ptr1[2]* and *ptr1[3]* points to the first byte of the memory allocated to the corresponding set of one-dimensional integer arrays of the original two-dimensional array.

The process of calculating and allocating memory at run time is known as *dynamic memory allocation.* The library routine *malloc* can be used for this purpose.

Instead of using conventional array notation, pointer notation has been used for accessing the address and value of corresponding array elements which has been explained to you in the previous section. The difference of the array elements within the function *calcdiff* is written as

**(*(ptr3 + i) + j) = *(*(ptr1 + i) + j) - *(*(ptr2 + i) + j);*

7.9 POINTERS AND STRINGS

As we have seen in strings, a string in C is an array of characters ending in the null character(written as '\0'), which specifies where the string terminates in memory. Like in one-dimensional arrays, a string can be accessed via a pointer to the first character in the string. The value of a string is the(constant) address of its first character. Thus, it is appropriate to say that a string is a constant pointer.
A string can be declared as a character array or a variable of type *char **. The declarations can be done as shown below:

```
char country[ ] = "INDIA";
char *country = "INDIA";
```

Each initialize a variable to the string “INDIA”. The second declaration creates a pointer variable *country* that points to the letter I in the string "INDIA" somewhere in memory.

Once the base address is obtained in the pointer variable *country*, **country* would yield the value at this address, which gets printed through,

```
printf("%s", *country);
```

Here is a program that dynamically allocates memory to a character pointer using *the* library function *malloc* at run-time. An advantage of doing this way is that a fixed block of memory need not be reserved in advance, as is done when initializing a conventional character array.

Example 7.13

Write a program to test whether the given string is a palindrome or not.

/* Program tests a string for a palindrome using pointer notation */

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

main()
```

```
{
    char *palin, c;
    int i, count;

    short int palindrome(char,int);          /*Function Prototype */
    palin =(char *) malloc(20 * sizeof(char));
    printf("\nEnter a word: ");
    do
    {
        c = getchar();
        palin[i] = c;
        i++;
    }while(c != '\n');

    i = i-1;
    palin[i] = '\0';
    count = i;

    if(palindrome(palin,count) == 1)
        printf("\nEntered word is not a palindrome.");
    else
        printf("\nEntered word is a palindrome");
    }

short int palindrome(char *palin, int len)
{
    short int i = 0, j = 0;
    for(i=0 , j=len-1; i < len/2;i++,j--)
    {
        if(palin[i] == palin[j])
            continue;
        else
            return(1);
    }
    return(0);
}
```

OUTPUT

Enter a word: malayalam

Entered word is a palindrome.

Enter a word: abcdba

Entered word is not a palindrome.

Array of pointers to strings

Arrays may contain pointers. We can form an array of strings, referred to as a string array. Each entry in the array is a string, but in C a string is essentially a pointer to its first character, so each entry in an array of strings is actually a pointer to the first character of a string. Consider the following declaration of a string array:

```
char *country[ ] = {
                    "INDIA", "CHINA", "BANGLADESH", "PAKISTAN", "U.S"
                      };
```

The **country[ ]* of the declaration indicates an array of five elements. The *char** of the declaration indicates that each element of array country is of type "pointer to char". Thus, *country [0]* will point to INDIA, *country[1]* will point to CHINA, and so on.

Thus, even though the array *country* is fixed in size, it provides access to character strings of any length. However, a specified amount of memory will have to be allocated for each string later in the program, for example,

country[i] = (char *) malloc(15 * sizeof(char));

The *country* character strings could have been placed into a two-dimensional array but such a data structure must have a fixed number of columns per row, and that number must be as large as the largest string. Therefore, considerable memory is wasted when a large number of strings are stored with most strings shorter than the longest string.

As individual strings can be accessed by referring to the corresponding array element, individual string elements be accessed through the use of the indirection operator. For example, **(* country + 3) + 2)* refers to the third character in the fourth string of the array *country*. Let us see an example below.

Example 7.14

Write a program to enter a list of strings and rearrange them in alphabetical order, using a one-dimensional array of pointers, where each pointer indicates the beginning of a string:

```
/* Program to sort a list of strings in alphabetical order using an array of pointers */

# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
# include <string.h>

void readinput(char *[ ], int);
void writeoutput(char *[ ], int);
void reorder(char *[ ], int);

main()
{
       char *country[ 5 ];
       int i;
       for(i = 0; i < 5; i++)
          {
            country[ i ] =(char *) malloc(15 * sizeof(char));
          }
      printf("Enter five countries on a separate line\n");
      readinput(country, 5);
      reorder(country, 5);
      printf("\nReordered list\n");
      writeoutput(country, 5);
      getch();
}

void readinput(char *country[ ], int n)
{
       int i;
```

```
        for(i = 0; i < n; i++)
                {   scanf("%s", country[ i ]);   }
         return;
}

void writeoutput(char *country[ ], int n)
{
        int i;
        for(i = 0; i < n; i++)
        {   printf("%s", country[ i ]);
            printf("\n");   }
         return;
}

void reorder(char *country[ ], int n)
{
         int i, j;
         char *temp;
         for(i = 0; i < n-1; i++)
         {
             for(j = i+1; j < n; j++)
             {
                 if(strcmp(country[ i ], country[ j ]) > 0)
                 {
                     temp = country[ i ];
                     country[ i ] = country[ j ];
                     country[ j ] = temp;
                 }
             }
         }
         return;
}
```

OUTPUT

```
Enter five countries on a seperate line
INDIA
BANGLADESH
PAKISTAN
CHINA
SRILANKA

Reordered list
BANGLADESH
CHINA
INDIA
PAKISTAN
SRILANKA
```

The limitation of the string array concept is that when we are using an array of pointers to strings we can initialize the strings at the place where we are declaring the array, but we cannot receive the strings from keyboard using *scanf()*.

Check Your Progress 3

1. What is meant by array of pointers?

……

……

2. How the indirection operator can be used to access a multidimensional array element.

……

……

3. A C program contains the following declaration.
float temp[3][2] = {{13.4, 45.5}, {16.6, 47.8}, {20.2, 40.8}};

(i) What is the meaning of temp?
(ii) What is the meaning of (temp + 2)?
(iii) What is the meaning of *(temp + 1)?
(iv) What is the meaning of (*(temp + 2) + 1)?
(v) What is the meaning of *(*(temp) + 1) + 1)?
(vi) What is the meaning of *(*(temp + 2))?

……

……

……

7.10 SUMMARY

In this unit we have studied about pointers, pointer arithmetic, passing pointers to functions, relation to arrays and the concept of dynamic memory allocation. A pointer is simply a variable that contains an address which is a location of another variable in memory. The unary operator &, when preceded by any variable returns its address. C's other unary pointer operator is *, when preceded by a pointer variable returns a value stored at that address.

Pointers are often passed to a function as arguments by reference. This allows data items within the calling function to be accessed, altered by the called function, and then returned to the calling function in the altered form. There is an intimate relationship between pointers and arrays as an array name is really a pointer to the first element in the array. Access to the elements of array using pointers is enabled by adding the respective subscript to the pointer value (i.e. address of zeroth element) and the expression preceded with an indirection operator.

As pointer declaration does not allocate memory to store the objects it points at, therefore, memory is allocated at run time known as *dynamic memory allocation.* The library routine *malloc* can be used for this purpose.

7.11 SOLUTIONS / ANSWERS

Check Your Progress 1

1. Refer to section 7.4. The data type included in the pointer declaration, refers to the type of data stored at the address which we will be storing in our pointer.

2. (i) Compile-time Error : Lvalue Required. Means that the left side of an assignment operator must be an addressable expression that include a variable or an indirection through a pointer.

(ii) Multiplication of a pointer variable with a constant is invalid.

3. (i) Refer section 7.4
(ii) Refer section 7.4
(iii) This means pointers can be of type void but can't be de-referenced without explicit casting. This is because the compiler can't determine the size of the object the pointer points to.

Check Your Progress 2

1 (i) True.
(ii) True.
(iii) False.
(iv) True.
(v) True.
(vi) True.

Check Your Progress 3

1. Refer section 7.4.

2. Refer section 7.4 to comprehend the convention followed.

3. (i) Refers to the base address of the array temp.

(ii) Address of the first element of the last row of array temp i.e. address of element 20.2.

(iii) Will give you 0. To get the value of the last element of the first array i.e. the correct syntax would be *(*(temp+0)+1).

(iv) Address of the last element of last row of array temp i.e. of 40.8.

(v) Displays the value 47.8 i.e., second element of last row of array temp.

(vi) Displays the value 20.2 i.e., first element of last row of array temp.

7.12 FURTHER READINGS

1. Programming with C, Second Edition, *Gottfried Byron S*, Tata McGraw Hill, India.
2. The C Programming Language, Second Edition, *Brian Kernighan and Dennis Richie*, PHI, 2002.
3. Programming in ANSI C, Second Edition, *Balaguruswamy E*, Tata McGraw Hill, India, 2002.
4. How to Solve it by Computer, *R.G.Dromey*, PHI, 2002.
5. C Programming in 12 easy lessons, *Greg Perry*, SAMS, 2002.
6. Teach Yourself C in 21 days, Fifth Edition, *Peter G*, Fifth edition,SAMS, 2002.