Chapter-11

An Introduction To Pointer

11.1 INTRODUCTION

As we know, computers use their memory for storing the instruction of a program, as well as the values of the variables that are associated with it. The computer's memory is a sequential collection of storage cells as shown in figure 11.1 Each cell has a unique address associated with it.

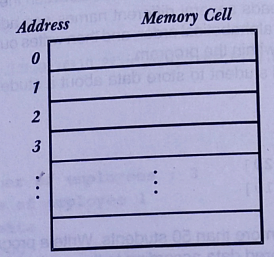


Fig. 11.1
(Memory Organisation)

Whenever we declare a variable, the system allocates an appropriate location or cell to hold the value of the variable. For example, Consider a integer variable **count** and consider the following assignment statement.

```
count  = 10;
```

This statement instructs the system to find a location or cell for the integer variable **count** and puts the value 10 in that location. Let us assume that the system has selected the memory location 50 for **count** as shown in fig. 11.2

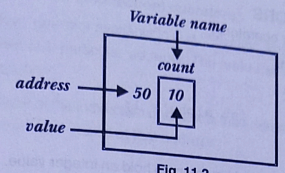


Fig. 11.2

Now, we may have access to the value 10 by using either the name **count** or the address 50. We can define a variable which stores the memory address. Such variables that hold memory addresses are called **pointers.**

Therefore, a **pointer** is nothing but a variable that contains an address which is a location of another variable in memory.

11.2 POINTER VARIABLES

A pointer is a variable that stores the address of some other variable. In some situations, it is easy to access the variable through its sotrage address in the main memory. The pointer variables can also access the value stored in the variable whose address they contain. In this way, the pointer is much more powerful than an ordinary variable.

11.2.1 ADVANTAGES OF THE POINTER VARIABLES

The advantages of the pointer variables are :

(i) The storage size can be adjusted dynamically as desired by the program.

(ii) Storage may be shared among several variables.

(iii) Complex data structures, like linked list, stack, queue etc, may be designed and manipulated.

(iv) Pointers increase the execution speed.

(v) Pointers reduce the length and complexity of a program.

(vi) With the use of pointers, one can return multiple values from a function.

(vii) Pointers are used to efficiently handle the arrays or data tables.

(viii) By using pointers, one can pass an array or a string as a parameter to a function.

(ix) A pointer enables us to access a variable that is defined outside the function.

(x) The use of pointer array to character string results in saving of data storage space in memory.

11.3 POINTER OPERATORS

Let us consider two main pointer operators :

(A) the 'address of' operators

The declaration statement

```
int alpha = 1;
```

tells the compiler to

- Reserve just enough space in memory to hold an integer value.
- Associate the name alpha with this space.
- Store the value 1 there.

We may represent alpha's location in memory by the following diagram.

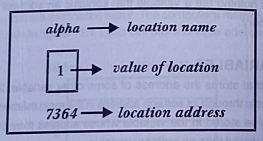


fig. 11.3

Example 11.1

Assuming that the compiler has selected memory location 7364 to store the value 1. We can print this location address through the following programs.

```
#include <stdio.h>
main( )
{
int alpha = 1;
printf ("The value %d is stored at address %d\n", alpha, &alpha);
}
Run
The value 1 is stored at address 7364.
```

The & symbol is known as **address of** operator. The expression **&alpha** returns the address associated with the variable alpha; in this case it happens to be 7364.

(B) The 'Value of address ' (Indirection) operators

The other pointer operator available in C is ' * ' called '**value at address**' operator. It gives the value stored at a particular address. The '**value at address**' operator is also called '**indirection**' operator.

Since the operand of the indirection operator is the address of a variable stored in memory, and it returns the value stored at that address.

Therefore,

***(&alpha) is the value stored at alpha's address, which is alpha itself**

Example 11.2

Study the following program

```
#include < stdio.h >
main( )
{
int alpha = 1;
printf ("The value %d is stored at address%d\n", alpha, &alpha);
printf ("The value %d is stored at address %d\n", * (&alpha), &alpha);
}
Run :
The value 1 is stored at address 7364
The value 1 is stored at address 7364
```

11.4 DECLARATION OF A POINTER DATA TYPE

Since addresses themselves are values they can be given names and stored in memory just like any other value. To do this, we have to make a declaration of the form:

```
int * beta ;
```

This is a pointer declaration. This tells the compiler that beta is a variable which will be used to store the address of an integer. We can assign to beta the address of alpha using the assignment statement as follows:

```
beta = &alpha;
```

Now, the pointer variable beta points to the integer variable alpha.

Since beta is also a variable, the compiler allocates space for it in the memory like for any other variable. The following memory diagram would illustrate the contents of alpha and beta.

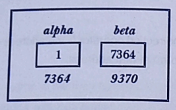


fig. 11.4

It is clear from the diagram that beta's value is alpha's address.

Example 11.3

To illustrate the above, consider the following program:

```
#include < stdio.h >
main( )
{
int alpha;
int * beta;
alpha = 1;
beta = &alpha;
printf("The value %d is stored at address %d\n", alpha, &alpha);
printf("The value %d is stored at address %d\n", beta, &beta);
printf("The value %d is stored at address %d.", *beta, beta);
}
```

```
Run :
The value 1 is stored at address 7364.
The value 7364 is stored at address 9370.
The value 1 is stored at address 7364.
```

Hence, to declare a pointer of any type, the declaration statement is :

```
data_type    *variable_name;
```

Example 11.4

```
int      *x;
float    *y;
char     *z;
```


In this example, x, y and z are declared as pointer variables, i.e. variables capable of holding addresses. Note that memory addresses are always going to be whole numbers, therefore, pointers always contain whole number. The declaration **float *y** means **y** is going to contain the address of a floating point value. Similarly **char *z** means that **z** is going to contain the address of a char value.

We know, pointer is a variable which contains address of another variable. Now this variable itself might be another pointer. Hence we now have pointer which contains another pointer's address. Consider the following example

Example 11.5

```
#include < stdio.h >
main( )
{
        int a = 5, *b,**c;
        b = &a;
        c = &b;
        printf ("address of a = %d\n", &a);
        printf ("address of a = %d\n", b);
        printf ("address of a = %d\n", *c);
        printf ("address of b = %d\n", &b);
        printf ("address of b = %d\n", c);
        printf ("address of c = %d\n", &c);
}
```

```
Run
address of a = 1234
address of a = 1234
address of a = 1234
address of b = 2345
address of b = 2345
address of c = 4567
```

Observe how the variables **b** and **c** have been declared,

```
int a = 5, *b,**c;
```

In this example, **a** is an integer variable, **b** is an integer pointer, whereas **c** is a pointer to a integer pointer as shown in the figure. 11.5.

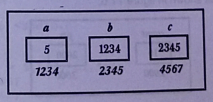


fig. 11.5

We can extend this concept still further by creating a pointer to a pointer to an integer pointer. Therefore, there is no limit on how far can we go on extending this definition.

11.5 ASSIGNMENT THROUGH POINTER

The statement **count=10** assigns the value of **10** to variable **count**. The same assignment is possible by putting the value of **10** at the address assigned to store variable **count**. The steps for the assignment in C are given below :

```
(1)   int count;
(2)   int *ptr;
(3)   ptr = & count;
(4)   *count=10;
```

The last statement

```
*Count=10;
```

stores the number 10, at the address ptr. Since, it is the address of variable **count**.

11.6 MEANING OF LVALUE AND RVALUE OF A VARIABLE

(i) An "lvalue" of a variable is the value of its address i.e. where it is stored in memory.
(ii) The "rvalue" of a variable is the value stored in that variable.

Example 11.6

Consider the following statement

```
int count = 10;
```

In this example, the rvalue for count is 10.

Example 11.7

Consider the following statement

```
int count = 10, * ptr = &count;
```

Conside the memory map as shown in figure 11.6.

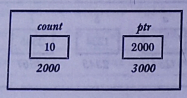


fig. 11.6

In this example, **rvalue** for **count** is 10 and **lvalue** is 2000. Also, rvalue for **ptr** is 2000 and lvalue is 3000.

11.7 OPERATIONS ON POINTERS

The following operations can be performed on a pointer.

11.7.1 ASSIGNMENT OPERATION

Only pointers of the same type can be assigned to each other.Let P1 and P2 are pointer variables of the same type, and suppose the values stored in the locations, to which they point, are A and B as shown in figure. 11.7.

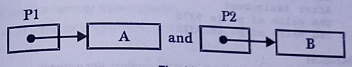


Fig. 11.7

Consider the following assignments statement

```
P1 = P2;
```

this statement implies that P1 and P2 point to the same location as shown in fig 11.8.

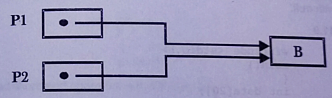


Fig. 11.8

A pointer variable can also be made to point to nothing, by assigning the special value **null** to the variable. For example

```
P =  null;
```

Note that **null** is a reserved word in **C**.

Example 11.8

Study the following program

```
#include <stdio.h>
main()
{
  int a = 5,b=3,*p1,*p2;
  p1 = &a;
  p2 = &b;
  printf("Before assignment\n");
  printf("The value of p1 is %d\n",p1);
  printf("The value of p2 is %d\n",p2);
  p1 = p2;  /* assignment */
```

```
    printf("After assignment\n");
    printf("The value of p1 is %d\n",p1);
    printf("The value of p2 is %d\n",p2);
}
Run
Before Assignment
The value of p1 is 7364
The value of p2 is 9370
After Assignment
The value of p1 is 9370
The value of p2 is 9370
```

11.7.2 COMPARISON

Pointers can be compared using the relational operators. For example

```
if (p ==q)
    break;
```

where p and q are the pointer variables of the same type.
The comparison can test for either equality or inequality. Moreover a pointer variable can be compare with **null.**

Example 11.9

```
#include <stdio.h>
main()
{
int data[20];
int* p1;
int *p2;
for (int i = 0; i <20;i++)
{
data[i] = i;
}
p1 = &data [1];
p2 = &data [2];
if (p1 > p2)
{
printf ("\n\n p1 is greater than p2");
}
else
{
printf ("\n\n p2 is greater than p1");
}
}
```

11.7.3 POINTER ARITHMETIC

C language allows you to operform arithmetic operations on pointers. The arithmetic operations on pointer can effect the memory location pointed by pointers or the data stored at the pointed location. The commonly used arithmetic operations on pointers are :

(a) **Addition of a number to a pointer**
C allows us to add integer to pointers. For example

```
int i = 3, *p, *q;
p = &i;
p = p + 1;
q = p + 3;
```

(b) **Substraction of a number from a pointer**
C allows us to substract integer from pointers. For example

```
int i = 3, *p, *q;
p = &i;
p = p – 1;
q = p – 3;
```

(c) **Substraction of one pointer from another**
One pointer variable can be substracted from another variable provided both variables points to elements of the same data type. For example

```
int i = 3, j = 12, *p, *q;
p = &i;
q = &j;
printf (" %d " , q – p);
```

We may also used shorthand operators with the pointers

```
p++;
– –q;
x += *p;
```

Don't attempt the following operations on pointers, they would never work out

1. Addition of two pointers
2. Division of a pointer with a constant.
3. Multiplication of a pointer with a constant.

11.7.4 POINTER CONVERSION

Pointer Conversion is a very powerful yet very dangerous feature. Pointer Conversion means type casting of one pointer into another Before concept of pointer conversion you must understand the concept of a void pointer. Void pointer technically is a pointer which is pointing to the unknown. Void pointer has special property that it can be type casted into any other pointer without any type casting though every other conversion needs an type casting. Also in dynamic memory allocation function such as malloc( ) and alloc( ) returns void pointer which can be easily converted to other types.

```
    printf("After assignment\n");
    printf("The value of p1 is %d\n",p1);
    printf("The value of p2 is %d\n",p2);
}
Run
Before Assignment
The value of p1 is 7364
The value of p2 is 9370
After Assignment
The value of p1 is 9370
The value of p2 is 9370
```

11.7.2 COMPARISON

Pointers can be compared using the relational operators. For example

```
if (p ==q)
    break;
```

where p and q are the pointer variables of the same type.
The comparison can test for either equality or inequality. Moreover a pointer variable can be compare with **null.**

Example 11.9

```
#include <stdio.h>
main()
{
int data[20];
int* p1;
int *p2;
for (int i = 0; i <20;i++)
{
data[i] = i;
}
p1 = &data [1];
p2 = &data [2];
if (p1 > p2)
{
printf ("\n\n p1 is greater than p2");
}
else
{
printf ("\n\n p2 is greater than p1");
}
}
```

11.7.3 POINTER ARITHMETIC

C language allows you to operform arithmetic operations on pointers. The arithmetic operations on pointer can effect the memory location pointed by pointers or the data stored at the pointed location. The commonly used arithmetic operations on pointers are :

(a) **Addition of a number to a pointer**
C allows us to add integer to pointers. For example

```
int i = 3, *p, *q;
p = &i;
p = p + 1;
q = p + 3;
```

(b) **Substraction of a number from a pointer**
C allows us to substract integer from pointers. For example

```
int i = 3, *p, *q;
p = &i;
p = p – 1;
q = p – 3;
```

(c) **Substraction of one pointer from another**
One pointer variable can be substracted from another variable provided both variables points to elements of the same data type. For example

```
int i = 3, j = 12, *p, *q;
p = &i;
q = &j;
printf (" %d " , q – p);
```

We may also used shorthand operators with the pointers

```
p++;
– –q;
x += *p;
```

Don't attempt the following operations on pointers, they would never work out

1. Addition of two pointers
2. Division of a pointer with a constant.
3. Multiplication of a pointer with a constant.

11.7.4 POINTER CONVERSION

Pointer Conversion is a very powerful yet very dangerous feature. Pointer Conversion means type casting of one pointer into another Before concept of pointer conversion you must understand the concept of a void pointer. Void pointer technically is a pointer which is pointing to the unknown. Void pointer has special property that it can be type casted into any other pointer without any type casting though every other conversion needs an type casting. Also in dynamic memory allocation function such as malloc( ) and alloc( ) returns void pointer which can be easily converted to other types.

Also there is a pointer called null pointer which seems like void pointer but is entirely different. Null pointer is a pointer which points to nothing. In fact it points to the base address of your CPU register and since register is not addressable usage of a null pointer will lead to chaos or at minimum a segmentation fault.

Also be careful while typecasting one pointer to another because even after type casting your pointer can point to anything but it will still think it is pointing to something of it declared type and have properties of the original type.

Example 11.10

Program below shows a type casting of one pointer into another -

```
#include <stdio.h>
main()
{
int i = 10;
char* p1   int *p2;
p2 = &i;
p1 = (char *) p2;   // Type Casting and Pointer Conversion
printf (" *p1 = %c /n *p2 = %d", *p1,*p2);
}
```

11.8 POINTER INCREMENTS AND SCALE FACTOR

Pointers can be incremented like

```
p = p + 1;
```

A pointer when incremented always points to an immediately next location of its type. For example, if p is an integer pointer with an initial value say 1400, then after the operation p = p + 1, the value of p will be 1402 and not 1401. That is, when we increment a pointer its value is incremented by the length of the data type that it points to. This length is called the **Scale Factor**. Scale Factor of various data types are as follows

Data Type	Scalar value for 16 bit PCs	Scalar value for 32 bit PC's
characters	1 byte	1 byte
integers	2 bytes	4 bytes
long integers	4 bytes	4 bytes
floats	4 bytes	4 bytes
doubles	8 bytes	8 bytes

sizeof operator is used to find the length of various data types. For example, if **x** is a variable then **sizeof(x)** returns the number of bytes needed for the variable.

Example 11.11

```
#include <stdio.h>
main()
{
  printf("Size of int is %d\n",sizeof(int));
  printf("Size of float is %d\n",sizeof(float));
  printf("Size of char is %d\n",sizeof(char));
  printf("Size of long is %d\n",sizeof(long));
  printf("Size of double is %d\n",sizeof(double));
}
Run
SIZE ofint  is 2
SIZE offloat is 4
SIZE ofchar is 1
SIZE oflong is 4
SIZE ofdoubleis 8
```

11.9 POINTERS AND ARRAYS

Arrays and Pointers are very closely linked. An array name is 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 express as either &x[0] or simply as x. Moreover, the address of the second array element can be written as either &x[1] or as (x + 1) and so on. Since &x[i] and (x + i) both represent the address of the ith element of x, it would seem resonable that x[i] and * (x + i) both represent the contents of that address i.e., the value of the ith element of x.

We know that array elements are always stored in contiguous memory locations. A pointer when incremented always points to an immediately next location of its type.

Study the following program that prints out the memory locations in which the elements of the array we stored.

Example 11.12

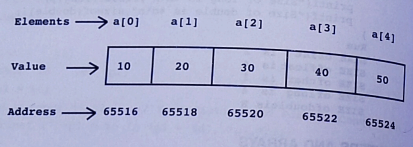
```
#include <stdio.h>
main()
{
  int a[ ]={10,20,30,40,50};
  int i;
  for(i=0;i<5;++i)
    printf("\nElement no. is %d and address is
%u",i,&a[i]);
  }
Run :
```

```
Element no. is 0 and address is 65516
Element no. is 1 and address is 65518
Element no. is 2 and address is 65520
Element no. is 3 and address is 65522
Element no. is 4 and address is 65524
```

The figure 11.10 shows how the array

```
int a[ ]={10,20,30,40,50};
```

is located in memory



If we declared **p** as an integer pointer, then we can make the pointer **p** to point to the array a by the following assignment

```
p = a;
```

This is equivalent to

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

Next program shows one way in which we can access the elements of this array

Example 11.13

```
#include <stdio.h>
main()
{
  int a[ ]={10,20,30,40,50};
  int i;
  for(i=0;i<5;++i)
    printf("\nElement no. is %d and address is
%u",i,&a[i]);
  }
Run :
```



```
Element no. is 0 and address is 65516
Element no. is 1 and address is 65518
Element no. is 2 and address is 65520
Element no. is 3 and address is 65522
Element no. is 4 and address is 6552
```

Another method of accessing each value of a using pointer is given below

Example 11.14

```
#include <stdio.h>
main()
{
  int a[ ]={10,20,30,40,50};
  int i,* j;
  j=a;
  for(i=0;i<5;++i)
  {
   printf("\nElement no. is %d and address is %u",*J,
J);
   j++;
  }
 }
Run :
Element no. is 0 and address is 65516
Element no. is 1 and address is 65518
Element no. is 2 and address is 65520
Element no. is 3 and address is 65522
Element no. is 4 and address is 6552
```

In this program, we have collected the base address of the array (address of the 0th element) in the variable **j** in the beginning using the statement

```
j=a;
```

j contains the address 65516 in the first attempt. On incrementing **j** it points to the next memory location of its type (that is location no. 65518). But location number 65518 contains the second element of the array and so on till the last element of the array has been printed.

A question arises as to which of the above two methods should be used ? Accessing array elements by pointer is always faster than accessing them by subscripts.

We now know that on mentioning the name of array we get its base address (address of the 0th element). One can easily see that ***a** and ***(a + 0)** both referred to first element of the array (i.e. 10)

```
int a[ ]={10,20,30,40,50};
```

Similarly, ***(a + 1)** refers to the second element of the array that is 20. This means that all the following notations are same.

```
*(a + i)
*(i + a)
a[i]
i[a]
```

Following program prove this fact.

Example 11.15

```
#include <stdio.h>
main()
{
  int a[ ]={10,20,30,40,50};
  int i;
  for(i=0;i<5;++i)
  {
    printf("\nElement
%d%3d%3d%3d",*(i+a),*(a+i),i[a],a[i]);
  }
}
```

```
Run :
Element is 10 10 10 10
Element is 20 20 20 20
Element is 30 30 30 30
Element is 40 40 40 40
Element is 50 50 50 50
```

Example 11.16

Write a program to read an array of numbers and find the largest number in it using pointer.

```
#include <stdio.h>
main()
{
  int n,i;
  float a[20],big;
  printf("Enter total numbers : ");
  scanf("%d",&n);
  printf("Enter %d real numbers\n",n);
  for(i=0;i<n;i++)
     scanf("%f",&a[i]);
  big = *a;
  for(i=1;i<n;i++)
    if(*(a+i) > big) big = *(a+i);
  printf("Largest number is : %f",big);
}
```

```
Run :
Enter total numbers : 5
Enter 5 real numbers
10
50
11
15
30
Largest number is : 50.00000
```

11.10 POINTERS AND TWO-DIMENSIONAL ARRAYS

Matrix X with maximum of M rows and N columns is declared as **X[M][N]**. Each row of the matrix will be associated with a pointer. Therefore there will be M pointers, each pointing to a row of the matrix. The array of pointers pointing to the row of the matrix will be represented by ***X[M]**. There will be a pointer to the array of pointers ***X[M]**. This will be represented as ****X**. Thus the pointer to a two dimensional array is a pointer to an array of pointers. If X is a two dimensional array, you can access element **x[i][] by *(*(X + i)+j)**.

Example 11.17

The following program will display the following matrix

$$x=\begin{pmatrix}6 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 7 & 1 & 5\end{pmatrix}$$

```
#include <stdio.h>
main()
{
int x[3][4] = {{6, 2, 3, 4},
               {5, 6, 7, 8},
               {9, 7, 1, 5}};
int i, j;
for (i=0; i<3; i++)
   {
     for (j=0; j<4; j++)
     printf("%4d", *(*(x + i) + j));
     printf("\n");
   }
 }
```

```
Run :
   6   2   3   4
   5   6   7   8
   9   7   1   5
```

Example 11.18

Write a program to reverse array elements through use of pointer.

Program

```
#include <stdio.h>
main()
{
  int a[] = { 18,20,22,24,26,28,30 };
  int *ptr,i,size;
  ptr = a;
  /* Find the number of elements in the array a */
  size = sizeof(a)/sizeof(int);
  /* print the array elements */
  printf("\nArray elements are\n");
  for(i=0;i<size;++i)
    printf("%3d",*(ptr + i));
  /* print the array elements in reverse */
  printf("\nReverse array elements are\n");
  for(i = size - 1;i >= 0;--i)
    printf("%3d",*(ptr + i));
}
```

Run

```
Array elements are
18   20   22   24   26   28   30
Reverse array elements are
30   28   26   24   22   20   18
```


11.11 POINTERS AND STRINGS

A string is an array of characters terminated by a NULL ('/0') character. An array of **char** pointers is generally better than two dimensional array of characters due to the following reasons :

(i) Better utilization of memory.

(ii) Manipulation of strings is easy using an array of pointers.

For example, An array of pointers will contain list of addresses. The list of students can be represented using the following declaration :

```
char*name[35];
```

In this declaration, each element of the list will point to one string. The following statement shows the way list of strings can be initialized.

```
char *name [35] = {"Amit",
                   "Savita",
                   "Shivank",
                   .
                   .
                   .
                   "Aman"};
```

Example 11.19

Program : (+STR 16.C)

```
/* Program to illustrates the input/output
   of array of pointers to strings */
#include <stdio.h>
#include <string.h>
main()
{
  char *name[35] = { "Amit",
                     "Savita",
                     "Aman",
                     "Suman" } ;
  int i;
  printf("\nList of students is\n");
  for(i = 0;i <= 34;++i)
  {
      puts(name[i]);
  }
}
```

Run

```
List of student is
Amit
Savita
Aman
Suman
```

11.11.1 LIMITATIONS OF ARRAY OF POINTER TO STRINGS

This approach also suffers from one limitation. Limitation is that we can not input a string using ***scanf*** *function, because we have to pass the address of the variable in which input value is to be stored. As we know that when an array is declared, its elements contain garbage value. Therefore, in case of array of pointers to strings, the garbage values do not represent a valid addresses to be sent to the* ***scanf*** *function.*

Example 11.20

Write a program that would sort a list of names in alphabetical order by using bubble sort.

Program : (+STR 18.C)

```
#include <stdio.h>
#include <string.h>
main()
{
  char name[35][21],temp[21];
  int n,i,j;
  printf("Enter number of elements : ");
  scanf("%d",&n);
  printf("Enter %d name\n",n);
  for(i = 1;i <= n;++i)
      scanf("%s",name[i]);
    /* interchange array elements */
  for(i=1;i<=n-1;i++)
  {
    for(j=1;j<=n-i;++j)
    {
      if( strcmp(name[j],name[j+1]) > 0)
      {
    strcpy(temp,name[j]);
    strcpy(name[j],name[j+1]);
    strcpy(name[j+1],temp);
      }
    }
  }
  /* write output */
  printf("The sorted list is \n");
  for(i = 1;i <= n;++i)
      printf("%s\n", name[i]);
}
```



```
Run :
Enter number of elements : 4
Enter 4 name
amit
suman
aman
rakesh
The sorted list is
aman
amit
rakesh
suman
```

11.12 POINTER AND FUNCTIONS

Main usages of pointers in relation to functions is **pointers as function arguments**. Call-by-reference method passes the addresses or references of the actual parameters to the called function. Thus the actual and formal parameters share the same memory locations. In this method, the changes done to the values in the formal parameters are reflected into the corresponding actual parameters into the calling program. The formal parameters are declared as pointers to types that match the data types of the actual parameters.

Example 11.21

```
/* Call by reference */
#include <stdio.h>
main()
{
  int a=10,b=20;
  printf("\nValue of actual parameters\n");
  printf("1. Before function call is %d%5d\n",a,b);
  swap(&a,&b);
  printf("2. After function call is %d%5d",a,b);
}

swap(int *x,int *y)
{
   int t;
   t = *x;
   *x = *y;
   *y = t;
}
```

```
Run
Value of actual parameters
1. Before function call in 10    20
2. After function call is  20    10
```

11.13 A FUNCTION RETURN MORE THAN ONE VALUE

Using a call by reference, you can make a function return more than one value at a time, which is not possible ordinarily.

Example 11.22

```
/* Return more than one value by Call by reference */
#include <stdio.h>
main()
{
  int x,square,cube;
  printf("\nEnter a number ");
  scanf("%d",&x);
  cal(x,&square,&cube);
  printf("Square of the number is %d\n",square);
  printf("Cube of the number is %d\n",cube);
}
cal( int a, int *b,int *c)
{
  *b = a * a;
  *c = a * a * a;
}
```

```
Run
Enter a number 5
Square of the number is 25
Cube of the number is 125
```

In this example, you are passing the value of **x** but, addresses of **square** and **cube**. Therefore the changes made to the values in the formal parameters (b and c) are also reflected to the corresponding actual parameters (square and cube). Thus, you can return more than one value from a called function and hence, overcome the limitation of the return statement.

11.14 PASSING AN ENTIRE ARRAY TO A FUNCTION

You can also pass an entire array to a function rather than its individual elements.

Example 11.23

```
/* Passing an entire array to a function */
#include <stdio.h>
main()
{
  int a[] = {10,20,30,40,50};
  sample(a,5);
}
sample( int *x, int n)
{
  int i;
  printf("Array elements are\n");
  for(i=0;i<n;i++)
  {
    printf("%4d",*x);
    x++;  /* point to next location */
  }
}
```

```
Run
Array elements are
10  20  30  40  50
```

In this example, you are passing the address of zeroth element to the sample ( ) function by just passing the name of the array. The following two function call are same :

```
sample (a, 5);
sample (&a [0], 5];
```

The for loop is used to access the array elements using pointers. It is also necessary to pass the total number of elements in the array, otherwise the **sample ( )** functions would not know when to terminate the **for** loop.

Example 11.24

Write a program to find the largest number from the given n numbers by passing entire array to a function

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

```
{
  int n,i;
  int a[20];
  printf("Enter total numbers : ");
  scanf("%d",&n);
  printf("Enter %d integer numbers\n",n);
  for(i=0;i<n;i++)
      scanf("%d",&a[i]);
  largest(a,n);
}

largest(int *x,int size)
{
  int big,i;
  big = *x;
  for(i=1;i< size;i++)
  {
     x++;
     if(*x > big) big = *x;
  }
  printf("Largest number is : %d\n",big);
}
```

```
Run
Enter total numbers : 5
Enter 5 integer numbers
23 15    89    45    34
Largest number is : 89
```

11.15 POINTER TO A FUNCTION

Using pointers, you can access a function. Every function has an address location in the memory and if you know the address of the function, you can access the function through its address. A pointer to a funciton is declared as follows

```
rtype (*ptr) (formal arguments);
```

This statement tells the compiler that **ptr** is a pointer to a function which returns **rtype** value. The parentheses around ***ptr** are necessary. Note that a statement like

```
rtype *ptr (format argument);
```

would declare **ptr** as a function returning a pointer to **rtype**.

The function will be accessed as

```
*ptr (actual arguments);
```

Before using this function, it is assumed that the address of the function is assigned to variable ptr

Example 11.25

```
/* Use of pointer to a function */
#include <stdio.h>
main()
{
  void mess();
  void (*ptr)();  /* pointer to function */
  ptr = mess; /* assign address of function to ptr */
  printf("Address of the function is %d\n",ptr);
  (*ptr)();  /* call function through pointer */
}
void mess()
{
  printf("I am in the subprogram\n");
}
```

```
Run
Address of the function is 679
I am in the subprogram
```

11.16 POINTERS AND STRUCTURES

You can also use pointers for structure variables. Consider the following declaration

```
struct student
{
int roll;
char name [20];
int age;
char class [8];
} ;

struct student s, *ptr ;
```

It declares a structures variables **s** and a pointer variable **ptr** to structure of type student. The beginning address of **s** can be assigned to **ptr** by writing

```
ptr = &s ;
```

The elements of structure can be accessed using arrow operater (→). Syntax is

```
ptr → member
```

Example 11.26

The element of structure of type **student** pointed to by pointer **ptr** can be accessed as

```
ptr → roll
ptr → name
ptr → age
ptr → class
```

We could also use the notation

```
(*ptr). member
```

to access the member of structure. The parentheses around ***ptr** are necessary because the dot operator (.) has a higher precedence than the operator(*).

Example 11.27

```
/* Program to illustrate pointers and structures */
#include <stdio.h>
main()
{
  struct student
  {
      int roll;
      char name[20];
      int age;
      char class[8];
  };
  struct student s, *ptr;
  ptr = &s;

  printf("Enter the student record\n");
  printf("Roll no. : ");
  scanf("%d",&ptr->roll);
  printf("Name : ");
  scanf("%s",&ptr->name);
  printf("Age : ");
  scanf("%d",&ptr->age);
  printf("Class :");
  scanf("%s",&ptr->class);
  printf("\nRecord details are\n");
```



```
  printf("Roll no. : %d\n",ptr->roll);
  printf("Name : %s\n",ptr->name);
  printf("Age : %d\n",ptr->age);
  printf("Class : %s\n",ptr->class);
}
Run
Enter the student record
Roll no. : 1001
Name : Shivank
Age : 7
Class : 2nd
Record details are
Roll No. : 1001
Name : Shivank
Age : 7
Class : 2nd
```

16.1 PASSING STRUCTURE TO A FUNCTION USING CALL BY REFERENCE METHOD

In this method, the address of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. Thus, the changes done to the contents of such a varible inside the function, are reflected back to the calling function.

Example 11.28

```
/* Program to illustrate passing of structure
 variables,by reference, to a function */
#include <stdio.h>
struct student
{
     int roll;
     char name[20];
     int age;
};
main()
{
  struct student st;
  void read_data(struct  student *s); /*function prototype
*/
  read_data(&st);  /*structure being passed by reference*/
  printf("\nRecord details are\n");
  printf("Roll no. : %d\n",st.roll);
  printf("Name : %s\n",st.name);
  printf("Age : %d\n",st.age);
}
```

```
void read_data(struct student *s)
{
  printf("Enter the student record\n");
  printf("Roll no. : ");
  scanf("%d",&s->roll);
  printf("Name : ");
  scanf("%s",&s->name);
 printf("Age : ");
 scanf("%d",&s->age);
}
```

```
Run
Enter the student record
Roll No. : 1
Name : Amit
Age : 16
Record details are
Roll No. : 1
Name : Amit
Age : 16
```

11.17 SELF-REFERENTIAL STRUCTURES

The self-refrential structures are structures that include an member which is a pointer to the another structure of the same type. **Syntax is**

```
struct sname
{
datatype member 1 ;
datatype member 2 ;
...
struct sname *ptr ;
} ;
```

Where ***ptr*** *refers to the name of a pointer variable. Thus, the structure of type* **sname** *will contain a member that points to another structure of type* **sname**. *Such structures are known as self-referential structures.*

Example 11.29

Consider the following declaration ;

```
struct list_element
{
char info [25];
struct list - element *next ;
} ;
```

This is a structure of type list-element.

The structure contains two members, a 25-element character array, called **info**, and a pointer to another structure of the same type called next. Therefore, this is a self-referential structure. These structures find their application in building complex data structure such as **linked lists**, **trees** and **graphs**.

11.18 POINTER TO POINTERS

We know, pointer is a variable which contains address of another variable. Now this variable is itself might be another pointer. Hence, we have a pointer which contains another pointer address. The pointer to pointers are declared by using double asterisk preceding their names. Syntax of declaring pointer to pointer is :

```
data type **variable_name
```

Example 11.30

```
int **ptr;
```

The above statement declares a pointer to pointer variable "ptr".

Example 11.31

```
main( )
{
int a=5, *b, **c;
b=&a;
c=&b;
. . .
}
```

In this example, c is the pointer to pointer.

11.19 DYNAMIC ALLOCATION USING POINTERS

An array is a static data structure and, therefore, its size should be known in advance (before its usage. For example, if a list of data is to be sorted then an array of approximate size is used. This is totally a guess work and can result in overflow or wastage of computer memory.

Most often you face situations in programming where the data is dynamic in nature. That is, the number of data items keep changing during execution of the program. For example, consider a program for processing the list of customers of a corporation. The list grows when names are added and shrinks when names are deleted. When list grows you need to allocate more memory space to the list to accomodate additional data items. Such situations can be handled more easily and effectively by using what is known as **dynamic data structures** in conjunction with **dynamic memory management techniques**.

Therefore, the process of allocating memory at run time is known as **dynamic memory allocation**. Although C does not inherently have this facility, there are four library routines known as "memory management functions" that can be used for allocating and freeing memory during program execution.

These functions are :

(i) malloc( )
(ii) calloc( )
(iii) realloc( )
(iv) free( )

These functions are defined either in the header file "alloc.h" or in "stdlib.h" or in both.

11.19.1 MEMORY ALLOCATION PROCESS

Before we discuss these functions. Let us look at the memory allocation process associated with a C program. The RAM can be divided into four areas as shown in fig. 14.1

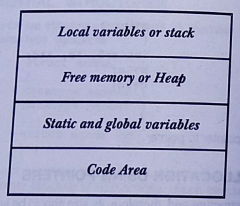


Fig. 11.9

(i) Local variables are stored in area called stack.

(ii) The code area is the region in the RAM where your C program instructions, after they are translated into machine language are stored .

(iii) There is a seperate area for storing the global and static variables.

(iv) The free memory area is called the heap. The size of heap keeps changing when program is executed due to creations and death of variables.

11.20 malloc( ) FUNCTION

The function most commonly used for dynamic memory allocation is **malloc( )**. The syntax of the function is

```
malloc(x);
```

where x is an unsigned integer which stands for the number of bytes you want to draw from the heap. The function returns a pointer, if your request is successful. It returns NULL, otherwise. malloc returns a **void** pointer. It does not point to anything. If you want it to point to any type of data you should write

```
(data_type *) malloc(x);
```

where **data_type** is the type of data to which the pointer returned by function **malloc( )** will point. It can be **char, int, float, double** or any other structure or union which has been defined by the user.

Example 11.32

Consider the following statement

```
ptr = (int *) malloc(sizeof(int));
```

On successful execution of this statement, a memory space equivalent to size of an **int** bytes is reserved and the address of the first byte of memory allocated is assigned to the pointer **ptr** of type of int.

Example 11.33

The statement

```
ptr = (char*) malloc (5);
```

allocates 5 bytes of space for the pointer **ptr** of type **char** as shown in fig. 14.2

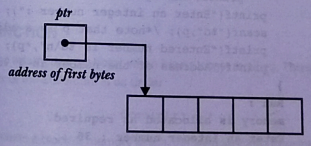


fig. 11.10

You may also use malloc to allocate space for complex data types such as structures.

Example 11.34

```
#include    <stdio.h>
 struct rec
{
      int i ;
      float f ;
      char c ;
};
main( )
{
     struct rec  *p;
     p = (struct rec*) malloc (sizeof (struct rec));
           . . .
           . . .
}
```

Example 11.35

Write a program to allocate a number of bytes required to store an integer number, tests for success, and then reads an integer number and stores it at the address returned by malloc( ).

```
#include <stdio.h>
#include <alloc.h>
main()
{
  int *p;
  p = (int *)malloc(sizeof(int));
  /* test that malloc is able to allocate memory */
  if( p == NULL )
     printf("malloc is unable to allocate memory\n");
  else
     printf("memory is allocated as required\n");
  printf("Enter an integer number :");
  scanf("%d",p); /*note that p is a address */
  printf("Entered number is %d\n",*p);
  printf("Address of the number is : %d",p);
}
```

```
Run :
memory is allocated as required.
Enter an integer number : 36
Entered number is 36
Address of the number is : 1942
```


11.21 calloc( ) FUNCTION

The function **calloc( )**, like **malloc( )** allocates memory in a dynamic manner. It takes, as arguments, two values as given below :

```
p = (t *)calloc(n, b);
```

It does the type casting to type **t** as in case of **malloc( )**. The above statement, allocates **n** blocks of memory, each block with **b** bytes. If there is not enough space, a NULL pointer is returned. This function is useful for storing arrays, etc. If each element of an array requires **b** bytes and the array is required to store **k** elements we can allocate memory by the following statement:

```
p = (t  *) calloc(k, b);
```

The same allocation will be done by **malloc( )**, if we call the function as :

```
p = (t  *) malloc(k * b);
```

11.22 realloc( ) FUNCTION

We can change the memory size already allocated with the help of the function **realloc**. This process is called the **reallocation of memory**. For example, if the original allocation is done on the statement

```
ptr = malloc(size);
```

then reallocation of space may be done by the statement

```
ptr = realloc(ptr, newsize);
```

The function allocates a new memory space of size **newsize** to the pointer variable **ptr** and returns a pointer to the first byte of the new memory block.

11.23 free( ) FUNCTION

The memory must be returned to the heap, when it is no longer required. This is done with free( ) function. The syntax of the statement is as under

```
free(ptr)
```

Here ptr is the pointer to a block of memory which has been allocated from the heap on request. The function is **void** in nature and does not return anything.

EXERCISE

1. What is a Pointer Variable ?
2. How can we get the address of a variable ?
3. How is a pointer variable declared ?
4. How can the value pointed to by pointer variable be accessed ?
5. What do you mean by a Pointer ? What are the advantages of the pointer variables in the program ?
6. Explain various operations on Pointers by using suitable example.
7. What do you mean by **lvalue** and **rvalue** of a variable ?
8. What do you mean by Scale Factor ?
9. What is the purpose of **sizeof** operator ?
10. A C program contains the following statements:

```
int i, j = 30;
int *pi, *pj = &j;
*pj = j + 10;
i = *p + 5;
pi = pj;
*pi = i + j;
```

Suppose each integer quantity occupies 2 bytes of memory. If the value assigned to i begins at address 1030 and the value assigned to j begins at address 1032, then

(i) What value is represented by &j ?
(ii) What value is represented by &i ?
(iii) What value is represented by pi ?
(iv) What value is represented by pi + 2 ?
(v) What value is assigned to pj ?
(vi) What value is assigned to *pj ?
(vii) What value is assigned to i ?
(viii) What value is assigned to *pi ?

11. Describe two different ways to specify the address of an array element.
12. How can the indirection operator be used to access a multidimensional array element ?
13. What is the relationship between array name and a pointer ?
14. How can a portion of an array be passed to a function ?
15. How can a function return a pointer to its calling routine ?

16. What is the relationship between a function name and a pointer ?
17. Suppose a formal argument within the definition of function p is a pointer to function q. How is the formal argument declared within p ? In this declaration to what does the data type refer ? How is function q accessed within function p ?
18. How is a structure type pointer variable declared ?
19. How can an individual structure member be accessed in term of its corresponding pointer variable ?
20. What is a self-referential structure ? For what kind of applications are self-referential structures useful ?
21. What is Dynamic Memory Location ? How does it help in building complex programs ?
22. Explain the following functions with the help of examples:

(i) `malloc( )`
(ii) `calloc( )`
(iii) `free( )`
(iv) `realloc( )`

23. What is the principle difference between the functions malloc and calloc.