Pointers

8.1 INTRODUCTION Pointers are another important feature of C language. They are a powerful tool and handy to use once they are mastered. There are a number of reasons for using pointers.

1. A pointer enables us to access a variable that is defined outside the function.
2. Pointers are more efficient in handling the data tables.
3. Pointers reduce the length and complexity of a program.
4. They increase the execution speed.
5. The use of a pointer array to character strings results in saving of data storage space in memory.

8.2 ACCESSING THE ADDRESS OF A VARIABLE

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. We can determine the address of the variable with the help of the operator **&** in **C**. We have already seen the use of this *address operator* in the **scanf** function. The operator **&** immediately preceding a variable returns the address of the variable associated with it. For example,

p = &quantity;

would assign the address to the variable **p**. The **&** operator can be remembered as 'address of'.

Example: Accessing Addresses of variables

```
#include<stdio.h>
#include<conio.h>
void main()
{
  char a;
  int x;
  float p, q;
  clrscr();
  a = 'A';
  x = 125;
  p = 10.25, q = 18.76;
  printf("%c is stored at address %u", a, &a);
  printf("\n%d is stored at address %u", x, &x);
  printf("\n%f is stored at address %u", p, &p);
  printf("\n%f is stored at address %u", q, &q);
  getch();
}
```

8.3 DECLARING AND INITIALIZING POINTERS

In C, every variable must be declared for its type. Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them. The declaration of a pointer variable takes the following form:

datatype *pt_name;

This tells the compiler three things about the variable **pt_name**.

1. The asterisk (*) tells the variable **pt_name** is a pointer variable.
2. **pt_name** needs a memory location.
3. **pt_name** points to a variable of type *datatype*.

For example,

int *p;

declares the variable **p** as a pointer variable that points to an integer data type. Remember that the type **int** refers to the data type of the variable being pointed to by **p** and not the type of the value of the pointer. Similarly, the statement

float *x;

declares **x** as a pointer to a floating point variable.
Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement such as

p = &quantity;

which causes **p** to point to quantity. That is, **p** now contains the address of quantity. This is known as *pointer initialization*. Before a pointer is initialized, it should not be used.

A pointer variable can be initialized in its declaration itself. For example,

int x, *p=&x;

is perfectly valid. It declares x as an integer variable and p as a pointer variable and then initializes **p** to the address of **x**. Note carefully that this is an initialization of **p**, not ***p**. And also remember that the target variable **x** is declared first. The statement

int *p=&x, x;

is not valid.

8.4 ACCESSING A VARIABLE THROUGH ITS POINTER

Once a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer. This is done by using another unary operator * (asterisk), usually known as the indirection operator. Consider the following statements:

```
int quantity, *p, n;
quantity = 179;
```

p = &quantity;
n = *p;

The first line declares **quantity** and **n** as integer variables and **p** as a pointer variable pointing to an integer. The second line assigns the value 179 to **quantity** and the third line assigns the address of **quantity** to the pointer variable **p**. The fourth line contains the indirection operator *. When the operator * is placed before a pointer variable in an expression, the pointer returns the value of the variable of which the pointer value is the address. In this case, ***p** returns the value of the variable **quantity**, because **p** is the address of **quantity**. The * can be remembered as ***'value of address'***. Thus the value of **n** would be 179.

Example: Program to illustrate the use of indirection operator '*'

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int x, y;
  int *ptr;
  clrscr();
  x = 10;
  ptr = &x;
  y = *ptr;
  printf("Value of x is %d",x);
  printf("\n%d is stored at address %u", x, &x);
  printf("\n%d is stored at address %u", *&x , &x);
  printf("\n%d is stored at address %u", *ptr, ptr);
  printf("\n%d is stored at address %u", y, &*ptr);
  printf("\n%d is stored at address %u", ptr, &ptr);
  printf("\n%d is stored at address %u", y, &y);
  getch();
}
```

Example: Arithmetic operations on Pointers

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a, b, *p1, *p2;
   clrscr();
   a = 4;
   b = 2;
  p1 = &a;
  p2 = &b;
  printf("\n Sum = %d",*p1 + *p2);
  printf("\n Sub = %d",*p1 - *p2);
  printf("\n Mul = %d",*p1 * *p2);
  printf("\n Div = %d",*p1 / *p2);
  printf("\n Rem = %d",*p1 % *p2);
  getch();
}
```

8.5 POINTERS AND ARRAYS

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element.

If we declare **p** as an integer pointer, then we can make the pointer **p** to point to the array **x** by the following assignment:

p = x;

This is equivalent to

p = &x[0];

When handling arrays, instead of using array indexing, we can use pointers to access array elements. Note that ***(p+3)** gives the value of **x[3]**. The pointer accessing method is much faster than array indexing.

Example: Pointers in one-dimensional array

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int *p, sum=0,i;
  clrscr();
  printf("Enter 5 elements:");
  for(i=0;i<=4;i++)
  scanf("%d",*(p+i));
  for(i=0;i<=4;i++)
  sum = sum + *p;
  printf("\nThe sum is:%d",sum);
  getch();
}
```

Pointers can be used to manipulate two-dimensional arrays as well. We know that in a one-dimensional array x, the expression

***(x + i)** or ***(p + i)**

represents the element **x[i]**. Similarly, an element in a two-dimensional array can be represented by the pointer expression as follows:

***(*(a+i)+j)** or ***(*(p+i)+j)**

8.6 DYNAMIC MEMORY ALLOCATION

C language requires the number of elements in an array to be specified at compile time. But we may not be able to do so always. Our initial judgment of size, if it is wrong, may cause failure of the program or wastage of memory space.

Many languages permit a programmer to specify an array's size at run time. The process of allocating memory at run time is known as ***dynamic memory allocation***. In **C** language there are four library routines known as ***"memory management functions"*** that can be used for allocating and freeing memory during program execution.

Memory Allocation Functions

Function	Task
malloc	Allocates requested size of bytes and returns a pointer to the first byte of the allocated space.
calloc	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
Free	Frees previously allocated space.
realloc	Modifies the size of previously allocated space.

Allocating a Block of Memory

A block of memory may be allocated using the function **malloc**. The **malloc** function reserves a block of memory of specified size and returns a pointer of type **void**. This means that we can assign it to any type of pointer. It takes the following form:

ptr = (datatype *)**malloc**(byte-size);

ptr is a pointer of type *datatype*. The **malloc** returns a pointer (of *datatype*) to an area of memory with size *byte-size*.

Example:

x = (int *) malloc (sizeof(int) * n);

Note: In above example **n** is always a *numeric*.

Example: Use of malloc Function

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int *p, n, i;
  clrscr();
  printf("Enter n value:");
  scanf("%d",&n);
  p = (int) malloc(sizeof(int)*n);
  printf("\nEnter %d numbers:",n);
  for(i=0;i<n;i++)
  scanf("%d",*(p+i));
  printf("\nThe numbers are:");
  for(i=0;i<n;i++)
  printf("\n%d",*(p+i));
  getch();
}
```

Allocating Multiple Blocks of Memory

calloc is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. While **malloc** allocates a single block of storage space, **calloc** allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero. The general form of **calloc** is:

ptr = (datatype *) **calloc** (n,elem-size);

The above statement allocates contiguous space for *n* blocks, each of size *elem-size* bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space, a NULL pointer is returned.

Releasing the Used Space

Compile-time storage of a variable is allocated and released by the system in accordance with its storage class. With the dynamic run-time allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited. We may release that block of memory for future use, using the **free** function:

free(ptr);

ptr is a pointer to a memory block which has already been created by **malloc** or **calloc**.

Altering the Size of a Block

It is likely that we discover later, the previously allocated memory is not sufficient and we need additional space for more elements. It is also possible that the memory allocated is much larger than necessary and we want to reduce it. In both the cases, 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 by the statement

ptr = **malloc**(size);

then reallocation of space may be done by the statement

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

8.7 POINTERS AND CHARACTER STRINGS

We know that a string is an array of characters, terminated with a null character. Like in one-dimensional arrays, we can use a pointer to access the individual characters in a string.

Example: Pointers and Character Strings

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

```
  char *str;
  int i=0;
  clrscr();
  printf("Enter a string:");
  gets(str);
  while(*str!='\0')
  {
    i++;
   str++;
  }
  printf("\nThe length is:%d",i);
  getch();
}
```

8.8 POINTERS AND FUNCTIONS

In functions we can pass the duplicate values for the actual parameters, but we can pass the address of a variable as an argument to a function in the normal fashion. When we pass addresses to a function, the parameters receiving the addresses should be pointers. The process of calling a function using pointers to pass the addresses of variable is known as ***call by address***.

Example: Pointers as function Parameters

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a,b;
   clrscr();
   printf("Enter 2 numbers:");
   scanf("%d %d",&a,&b);
   printf("\nBefore exchange: a=%d, b=%d",a, b);
   swap(&a,&b);
   printf("\nAfter exchange: a=%d, b=%d",a, b);
   getch();
}
void swap(int *x, int *y)
{
   int z;
   z = *x;
  *x = *y;
  *y = z;
}
```

8.9 POINTERS AND STRUCTURES

We know that the name of an array stands for the address of its zeroth element. The same thing is true of the names of arrays of structure variables. Suppose **product** is an array variable of **struct** type. The name **product** represents the address of its zeroth element. Consider the following declaration:

```
struct inventory
{
  char name[30];
```

int number;
}**product[3], *ptr;**

Example: Pointers to Structure variables

```
#include<stdio.h>
#include<conio.h>
struct invent
{
   char name[20] ;
   int number;
}product[3], *ptr;
void main()
{
  clrscr();
  printf("INPUT\n\n");
  printf("Enter name and number(3 records):");
  for(ptr = product;ptr<product+3;ptr++)
  scanf("%s %d",ptr->name,&ptr->number);
  printf("\n\nOUTPUT");
  for(ptr = product;ptr<product+3;ptr++)
  printf("\n%s\t%d",ptr->name,ptr->number);
  getch();
}
```

8.10 DYNAMIC MEMORY ALLOCATION

We may also use **malloc** to allocate space for complex data types such as structures. Example:

st_var = (struct store *)malloc(sizeof(struct store));

where **st_var** is a pointer of type **struct store**.