UNIT – 5:

Pointers

Introduction

Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a good C programmer.

As you know, every variable needs memory location and every memory location has its address defined which can be accessed using **ampersand (&) operator**, which denotes an address in memory.

- Consider the following example, which prints the address of the variables defined –

```
#include <stdio.h>

void main () {

  int  var1;
  char var2[10];

  printf("Address of var1 variable: %x\n", &var1  );
  printf("Address of var2 variable: %x\n", &var2  );

  getch();
}
```

Output :

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

5.1. Pointer Variable Declaration & Memory Storage

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location.

- Like any variable or constant, you must declare a pointer before using it to store any variable address.
- The general form of a pointer variable declaration is –

```
type *var-name;
```

- Here, **type** is the pointer's base type; it must be a valid C data type
- **var-name** is the name of the pointer variable.
- The asterisk * used to declare a pointer is the same asterisk used for multiplication.
- However, in this statement the asterisk is being used to declare a variable as a pointer.

Example

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

5.2. Address and value operators

Address Operator

Ampersand (&) operator, which denotes an address in memory, It is also considered as **Address of** operator.

Value Operator

Asterisk (*) unary operator, which is denoted as **Value At** operator. while using this operator as prefix to pointer variable the variable will return Value stored at that particular location.

Let us understand Above all using a program.

(a) We define a pointer variable,
(b) assign the address of a variable to a pointer and
(c) finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand.

example

```
#include <stdio.h>

void main () {

  int  var = 20;            /* actual variable declaration */
  int  *ip;                 /* pointer variable declaration */

  ip = &var;                /* store address of var in pointer variable*/

  printf("Address of var variable: %x\n", &var  );

  /* address stored in pointer variable */
  printf("Address stored in ip variable: %x\n", ip );

  /* access the value using the pointer */
  printf("Value of *ip variable: %d\n", *ip );

   getch();
}
```

Output

```
Address of var variable: 22001
Address stored in ip variable: 22001
Value of *ip variable: 20
```

5.3. Pointer Arithmetic

Operations Allowed

- C allows us to add integers to pointer variable, to subtract integers from pointer variable or to subtract two pointer variables.

  That means if P1 and P2 are pointer variables then following operations are possible.

```
P1+2    //add integer
P2-3    //subtract integer
P2-P1   //subtract two pointers
```

- Apart from above operations following short hand operations are also possible on pointer variables.

```
P1++    //increment by one
P2--    //decrement by one
```

- In addition to arithmetic operations described above comparing two pointer variables using relational operators is also possible.

```
P1>P2    or    P1 == P2    or    P1!= P2
```

Operations Not Allowed

- We may not use pointer for few arithmetic operations like addition, Multiplication and Division which means we cannot perform following tasks on an pointer variable.

```
P1 + P2    //addition of two pointer variable NOT ALLOWED
P1 * P2    //multiplication of two pointer variable NOT ALLOWED
P1 / P2    //division of two pointer variable NOT ALLOWED
```

Pointer Increment and Scale Factor

- Incrementing a pointer using integer number will not increment address value by that number, it will move the pointer to location of variable which stored at given number.
- Let us understand using an example, if an integer pointer P1 is pointing to an integer array of size 5. here A is our array name.

A[0] -> 5000	A[1]->5002	A[2]->5004	A[3]->5006	A[4]->5008

  - P1 = &a[0]; //Which means address stored in pointer will be 5000
  - If we write P1 = P1 + 1;
    - It will not add 1 to the pointer value 5000
    - It will add (1*2) to the pointer value 5000 and value will become 5002 which is the location of next element.
    - Here, 1 is the integer value we provided and 2 is the scale factor of data type integer because each integer variable will require 2 bytes memory for storing value.
- Similar logic of scale factor is applicable for all increment and decrement related operations.

- Scale factor for various data types is given below
  - Character - 1 (byte)
  - Integer - 2 (bytes)
  - Float - 4 (bytes)
  - Long integer - 4 (bytes)
  - Doubles - 8 (bytes)

5.4. Passing pointers to functions

While passing pointer to function, we are **passing the address(memory location) of the variable** to the called function.

Any updates made inside the called function **will modify the original copy** since we are directly modifying the content of the exact memory location.

Example

```
#include<stdio.h>
void interchange(int *num1,int *num2);
int main() {
   int num1=50,num2=70;
   interchange(&num1,&num2);
   printf("\nNumber 1 : %d",num1);
   printf("\nNumber 2 : %d",num2);
   return(0);
}
void interchange(int *num1,int *num2)
{
   int temp;
   temp  = *num1;
   *num1 = *num2;
   *num2 = temp;
}
```

Output :

```
Number 1 : 70
Number 2 : 50
```

5.5. Pointer to Array

When an array is declared compiler allocates a base address and required storage for all the elements of the array in contiguous manner. Where base address is the memory location of first element of the array.

- **The name of array itself is a pointer having value of base address. That means if we write**
  int a[5];
- **a will be having address of first element of the array that is a[0].**

5.5.1. Pointer to One-Dimensional Array

If we declare array of 5 elements it will be represented in following way in memory.

Int a[5] = {1,2,3,4,5};

element→	a[0]	a[1]	a[2]	a[3]	a[4]
values→	1	2	3	4	5
address→	5000	5002	5004	5006	5008

- name array name "a" is a constant pointer pointing to base address that is &a[0] (address of first element in array).
- So we can say that **a = &a[0] = 5000**
- If we declare ptr as a pointer to this array we can initialize this pointer in following ways.
  - Ptr = a;
    Or
  - Ptr = &a[0];
- For accessing elements using pointer to array we can simply increment pointer to move pointer to desired element. which is abstracted below.
  - Ptr = &a[0] = 5000
  - Ptr + 1 = &a[0] = 5002 //here is 1 is an integer
  - Ptr + 2 = &a[0] = 5004 which will be multiplied by scale
  - Ptr + 3 = &a[0] = 5006 of an integer data type when used with
  - Ptr + 4 = &a[0] = 5008 pointer variable of type integer.

5.5.2. Pointer to Multi-Dimensional Array

Similar to one dimensional array pointer is very useful with multi dimensional array.we know that in one dimensional array ***(ptr + i)** will give the value stored at i^{th} location in two dimensional array this will be achived using following statement.

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

- Where **"i" is** row number and **"j"** is column number.

- Following different expressions will be used for accessing different elements from two dimensional array
  - Ptr → pointer to first row
  - Ptr + i → pointer to i^{th} row
  - *(ptr + i) → pointer to first element of i^{th} row
  - *(ptr +i)+j → pointer to j^{th} element in i^{th} row
  - *(*(ptr + i) + j) → value at pointer to j^{th} element in i^{th} row

5.6. Array of Pointers

Before we understand the concept of arrays of pointers, let us consider the following example, which uses an array of 3 integers −

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
  int  var[] = {10, 100, 200};
  int i;

  for (i = 0; i < MAX; i++) {
    printf("Value of var[%d] = %d\n", i, var[i] );
  }

  return 0;
}
```

Output

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

Declaring and using Array of Pointer

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer –

```
int *ptr[MAX];
```

- It declares **ptr** as an array of MAX integer pointers.
- Thus, each element in ptr, holds a pointer to an int value.

Example

```
#include <stdio.h>

const int MAX = 3;

void main () {
  int  var[] = {10, 100, 200};
  int i, *ptr[MAX];
  for ( i = 0; i < MAX; i++) {
    ptr[i] = &var[i]; /* assign the address of integer. */
  }
  for ( i = 0; i < MAX; i++) {
    printf("Value of var[%d] = %d\n", i, *ptr[i] );
  }
  Getch();
}
```

Output

```
Value of var[0] = 10
```

```
Value of var[1] = 100
Value of var[2] = 200
```

String pointer (Array of char array)Example

```
#include <stdio.h>

const int MAX = 4;

void main () {

  char *names[] = {
    "Zara Ali",
    "Hina Ali",
    "Nuha Ali",
    "Sara Ali",
  };

  int i = 0;

  for ( i = 0; i < MAX; i++) {
    printf("Value of names[%d] = %s\n", i, names[i] );
  }

  Getch();
}
```

Output

```
Value of names[0] = Zara Ali
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali
```