Module-III

BCSE102L_STRUCTURED AND OBJECT-ORIENTED-PROGRAMMING

Prepared by
Dr.M.Suguna

POINTERS

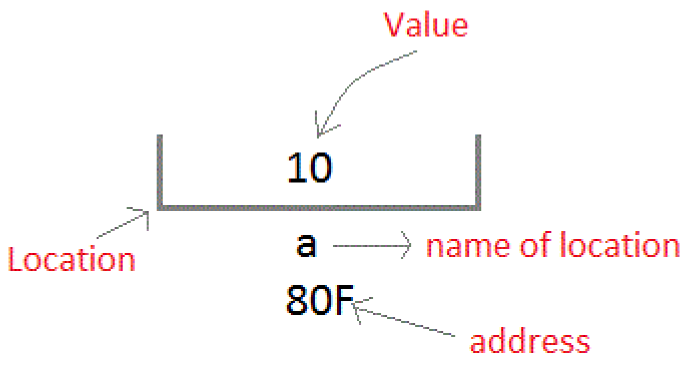
Pointer Fundamentals – Pointer Declaration – Passing Pointers to a Function – Pointers and one dimensional arrays – operations on pointers– Dynamic memory allocation

Pointer Fundamentals:

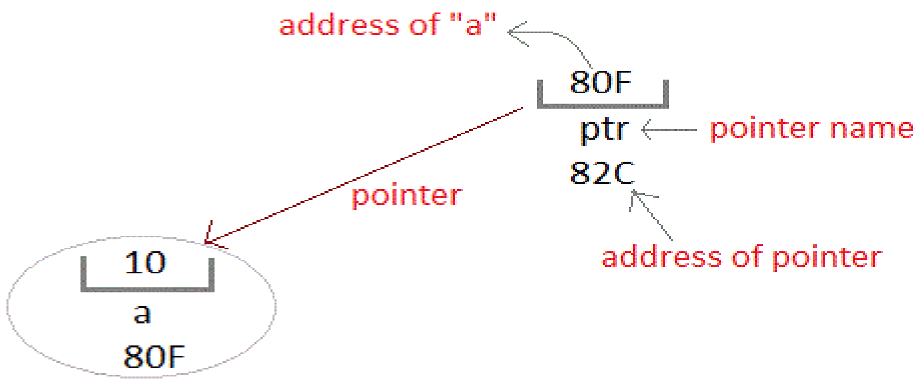
Whenever a **variable** is declared in a program, system allocates a location i.e an address to that variable in the memory, to hold the assigned value. This location has its own address number, which we just saw above.

Let us assume that system has allocated memory location 80F for a variable a.

int a = 10;



We can access the value 10 either by using the variable name a or by using its address 80F. The question is how we can access a variable using it's address? Since the memory addresses are also just numbers, they can also be assigned to some other variable. The variables which are used to hold memory addresses are called **Pointer variables**. A **pointer** variable is therefore nothing but a variable which holds an address of some other variable. And the value a **pointer variable** gets stored in another memory location.



Benefits of using pointers:

Below we have listed a few benefits of using pointers:

1. Pointers are more efficient in handling Arrays and Structures.
2. Pointers allow references to function and thereby helps in passing of function as arguments to other functions.
3. It reduces length of the program and its execution time as well.
4. It allows C language to support Dynamic Memory management.

Declaring, Initializing and using a pointer variable

In this tutorial, we will learn how to declare, initialize and use a pointer. We will also learn what NULL pointer are and where to use them. Let's start!

Declaration of Pointer variable:

General syntax of pointer declaration is,

datatype *pointer_name;

Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing. void type pointer works with all data types, but is not often used.

Here are a few examples:

```
int *ip// pointer to integer variable
float *fp;// pointer to float variable
double *dp;// pointer to double variable
char *cp;// pointer to char variable
```

Initialization of Pointer variable:

Pointer Initialization is the process of assigning address of a variable to a **pointer** variable. Pointer variable can only contain address of a variable of the same data type. In C language **address operator** & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
#include<stdio.h>
void main()
{
   int a =10;
   int *ptr;//pointer declaration
ptr=&a;//pointer initialization
}
```

Pointer variable always point to variables of same datatype. Let's have an example to showcase this:

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

```
{
   float a;
   int *ptr;
ptr=&a;// ERROR, type mismatch
}
```

If you are not sure about which variable's address to assign to a pointer variable while declaration, it is recommended to assign a NULL value to your pointer variable. A pointer which is assigned a NULL value is called a **NULL pointer**.

```
#include <stdio.h>
int main()
{
   int *ptr= NULL;
return0;
}
```

Using the pointer or Dereferencing of Pointer:

Once a pointer has been assigned the address of a variable, to access the value of the variable, pointer is **dereferenced**, using the **indirection operator** or **dereferencing operator** *.

```
#

include <stdio.h>
int main()
{
   int a,*p;// declaring the variable and pointer
   a =10;
   p =&a;// initializing the pointer

printf("%d",*p);//this will print the value of 'a'

printf("%d",*&a);//this will also print the value of 'a'

printf("%u",&a);//this will print the address of 'a'

printf("%u", p);//this will also print the address of 'a'

printf("%u",&p);//this will print the address of 'p'

return0;
}
```

Points to remember while using pointers:

1. While declaring/initializing the pointer variable, * indicates that the variable is a pointer.
2. The address of any variable is given by preceding the variable name with Ampersand &.
3. The pointer variable stores the address of a variable. The declaration int *a doesn't mean that a is going to contain an integer value. It means that a is going to contain the address of a variable storing integer value.
4. To access the value of a certain address stored by a pointer variable, * is used. Here, the * can be read as **'value at'**.

Pointer to a Pointer (Double Pointer):

Pointers are used to store the address of other variables of similar datatype. But if you want to store the address of a pointer variable, then you again need a pointer to store it. Thus, when one pointer variable stores the address of another pointer variable, it is known as **Pointer to Pointer** variable or **Double Pointer**.

Syntax:

```
int **p1;
```

Here, we have used two indirection operator (*) which stores and points to the address of a pointer variable i.e, int *. If we want to store the address of this (double pointer) variable p1, then the syntax would become:

```
int ***p2
```

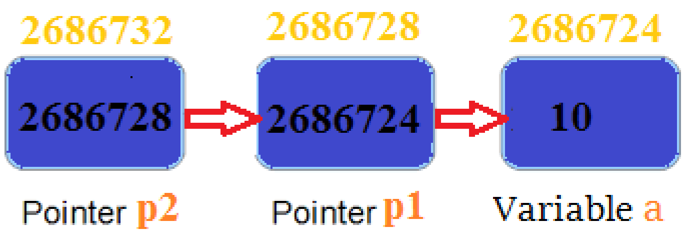
Simple program to represent Pointer to a Pointer

```
#include <stdio.h>
int main(){
int  a=10;
int  *p1;//this can store the address of variable a
int  **p2;

/*
        this can store the address of pointer variable p1 only.
        It cannot store the address of variable 'a'
    */

    p1 =&a;
    p2 =&p1;

printf("Address of a = %u\n",&a);
printf("Address of p1 = %u\n",&p1);
printf("Address of p2 = %u\n\n",&p2);

// below print statement will give the address of 'a'
printf("Value at the address stored by p2 = %u\n",*p2);
printf("Value at the address stored by p1 = %d\n\n",*p1);

printf("Value of **p2 = %d\n",**p2);//read this *(*p2)

/*
        This is not allowed, it will give a compile time error-
        p2 = &a;
printf("%u", p2);
    */
return0;
}
```

Output

```
Address of a = 2686724
Address of p1 = 2686728
Address of p2 = 2686732
Value at the address stored by p2 = 2686724
Value at the address stored by p1 = 10
Value of **p2 = 10
```

Explanation of the above program:



- p1 pointer variable can only hold the address of the variable a (i.e Number of indirection operator(*)-1 variable). Similarly, p2 variable can only hold the address of variable p1. It cannot hold the address of variable a.
- *p2 gives us the value at an address stored by the p2 pointer. p2 stores the address of p1pointer and value at the address of p1 is the address of variable a. Thus, *p2 prints address of a.
- **p2 can be read as *(*p2). Hence, it gives us the value stored at the address *p2. From above statement, you know *p2 means the address of variable a. Hence, the value at the address *p2 is 10. Thus, **p2 prints 10.

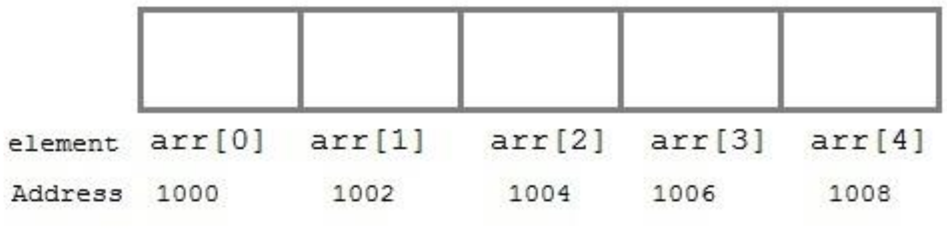
Pointer and Arrays:

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e address of the first element of the array is also allocated by the compiler.

Suppose we declare an array arr,

int arr[5]={1,2,3,4,5};

Assuming that the base address of arr is 1000 and each integer requires two bytes, the five elements will be stored as follows:



Here variable arr will give the base address, which is a constant pointer pointing to the first element of the array, arr[0]. Hence arr contains the address of arr[0] i.e 1000. In short, arr has two purpose - it is the name of the array and it acts as a pointer pointing towards the first element in the array.

arr is equal to &arr[0] by default

We can also declare a pointer of type int to point to the array arr.

```
int *p;
p =arr;
// or,
p =&arr[0];//both the statements are equivalent.
```

Now we can access every element of the array arr using p++ to move from one element to another.

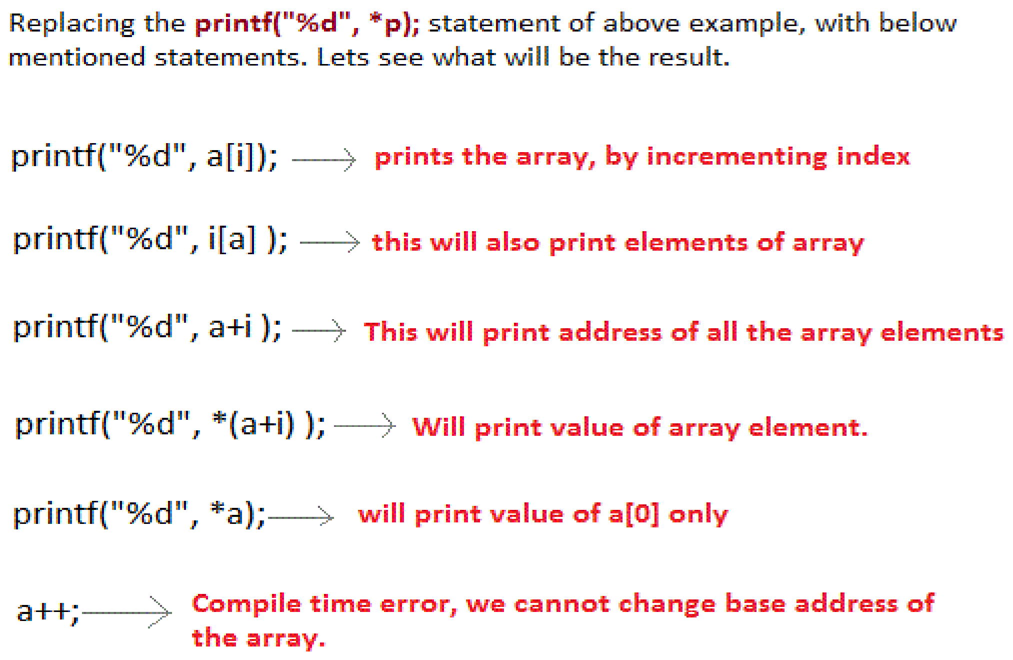
NOTE: You cannot decrement a pointer once incremented. p-- won't work.

Pointers and one dimensional arrays:

As studied above, we can use a pointer to point to an array, and then we can use that pointer to access the array elements. Let’s have an example,

```
#include <stdio.h>
int main()
{
   int i;
   int a[5]={1,2,3,4,5};
   int *p =a;// same as int*p = &a[0]
for(i=0;i<5;i++)
{
printf("%d",*p);
      p++;
}

return0;
}
```

In the above program, the pointer *p will print all the values stored in the array one by one. We can also use the Base address (a in above case) to act as a pointer and print all the values.



Replacing the **printf("%d", *p);** statement of above example, with below mentioned statements. Lets see what will be the result.

```
printf("%d", a[i]);  ⟶  prints the array, by incrementing index

printf("%d", i[a] ); ⟶  this will also print elements of array

printf("%d", a+i );  ⟶  This will print address of all the array elements

printf("%d", *(a+i) ); ⟶  Will print value of array element.

printf("%d", *a); ⟶  will print value of a[0] only

a++; ⟶  Compile time error, we cannot change base address of
        the array.
```

The generalized form for using pointer with an array,
*(a+i)
is same as:
a[i]

Pointer to Multidimensional Array:

A multidimensional array is of form, a[i][j]. Let’s see how we can make a pointer point to such an array. As we know now, name of the array gives its base address. In a[i][j], a will give the base address of this array, even a + 0 + 0 will also give the base address, that is the address of a[0][0]element.

Here is the generalized form for using pointer with multidimensional arrays.

    ((a +i)+ j)

which is same as,

      a[i][j]

Pointer and Character strings:

Pointer can also be used to create strings. Pointer variables of char type are treated as string.

```
char *str="Hello";
```

The above code creates a string and stores its address in the pointer variable str. The pointer strnow points to the first character of the string "Hello". Another important thing to note here is that the string created using char pointer can be assigned a value at **runtime**.

```
char *str;
str="hello";//this is Legal
```

The content of the string can be printed using printf() and puts().

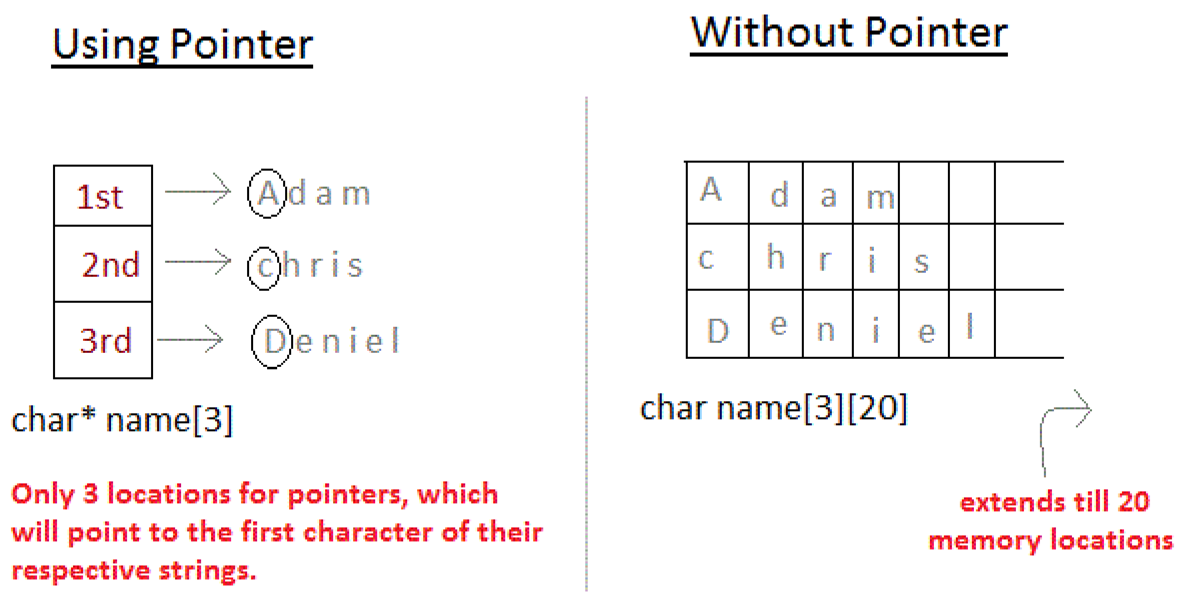
```
printf("%s",str);
puts(str);
```

Notice that str is pointer to the string, it is also name of the string. Therefore, It cant need to use indirection operator *.

Array of Pointers-Example:

Pointers are very helpful in handling character array with rows of varying length.

```
char *name[3]={
"Adam",
"chris",
"Deniel"
};
//Now lets see same array without using pointer
char name[3][20]={
"Adam",
"chris",
"Deniel"
};
```


Using Pointer
1st
2nd
3rd
Adam
Chris
Deniel
char* name[3]
Only 3 locations for pointers, which will point to the first character of their respective strings.
Without Pointer
A d a m
c h r i s
D e n i e l
char name[3][20]
extends till 20 memory locations

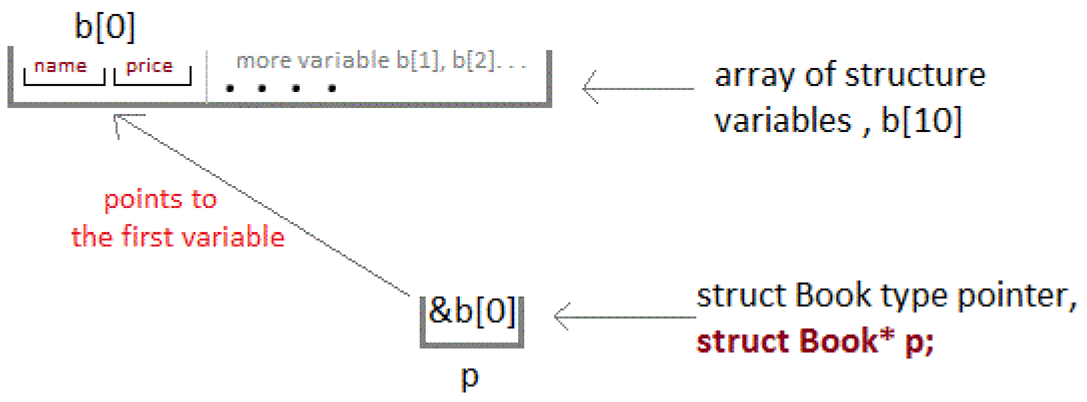
In the second approach memory wastage is more, hence it is preferred to use pointer in such cases.
When it say memory wastage, it doesn't means that the strings will start occupying less space, no, characters will take the same space, but when we define array of characters, a contiguos memory space is located equal to the maximum size of the array, which is a wastage, which can be avoided if we use pointers instead.

Pointer to Structure Array:

Like array of integers, array of pointers etc, we can also have array of structure variables. And to use the array of structure variables efficiently, we use **pointers of structure type**. We can also have pointer to a single structure variable, but it is mostly used when we are dealing with array of structure variables.

```
#include <stdio.h>
struct Book
{
   char name[10];
   int price;
}
```

```
int main()
{
   struct Book a;//Single structure variable
   struct Book*ptr;//Pointer of Structure type
ptr=&a;

   struct Book b[10];//Array of structure variables
   struct Book*p;//Pointer of Structure type
   p =&b;

return0;
}
```


b[0]
name
price
more variable b[1], b[2]. . .
array of structure variables , b[10]
points to the first variable
&b[0]
p
struct Book type pointer,
struct Book* p;

Accessing Structure Members with Pointer:

To access members of structure using the structure variable, we used the dot**.** operator. But when it have a pointer of structure type, we use arrow -> to access structure members.

```
#include <stdio.h>
struct my_structure{
   char name[20];
   int number;
   int rank;
};

int main()
{
   struct my_structure variable ={"StudyTonight",35,1};

   struct my_structure*ptr;
    ptr=&variable;
printf("NAME: %s\n",ptr->name);
printf("NUMBER: %d\n",ptr->number);
printf("RANK: %d",ptr->rank);

return0;
}
```

output

```
NAME: StudyTonight
NUMBER: 35
RANK: 1
```

Passing Pointers to a Function:

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. A pointer to a function is declared as follows,

type (*pointer-name)(parameter);

Here is an example :

int (*sum)();//legal declaration of pointer to function

int *sum();//This is not a declaration of pointer to function

A function pointer can point to a specific function when it is assigned the name of that function.

```
int sum(int, int);
int (*s)(int, int);
s = sum;
```

Here s is a pointer to a function sum. Now sum can be called using function pointer s along with providing the required argument values.

s (10,20);

Example Time: Swapping two numbers using Pointer:

```
#include <stdio.h>
void swap(int *a, int *b);
int main()
{
   int m =10, n =20;
printf("m = %d\n", m);
printf("n = %d\n\n", n);

swap(&m,&n);//passing address of m and n to the swap function
printf("After Swapping:\n\n");
printf("m = %d\n", m);
printf("n = %d", n);
return0;
}
```

```
/*
   pointer 'a' and 'b' holds and
   points to the address of 'm' and 'n'
*/
void swap(int *a, int *b)
{
   int temp;
   temp =*a;
*a =*b;
*b = temp;
}
```

output:

```
m = 10
n = 20
After Swapping:
m = 20
n = 10
```

Functions returning Pointer variables:

A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only inside the function. Hence if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends.

```
#include <stdio.h>
int*larger(int*, int*);
void main()
{
   int a =15;
   int b =92;
   int *p;
   p =larger(&a,&b);
printf("%d is larger",*p);
}
int*larger(int *x, int *y)
{
if(*x >*y)
return x;
else
return y;}
```

output

```
92 is larger
```

Safe ways to return a valid Pointer.

1. Either use **argument with functions**. Because argument passed to the functions are declared inside the calling function, hence they will live outside the function as well.

2. Or, use static **local variables** inside the function and return them. As static variables have a lifetime until the main() function exits, therefore they will be available througout the program.

Example of Pointer to Function

```
#include <stdio.h>

int sum(int x, int y)
{
return x+y;
}

int main()
{
   int (*fp)(int, int);
fp= sum;
   int s =fp(10,15);
printf("Sum is %d", s);

return0;
}
```

output

```
25
```

Operations on pointers-c programming language

Pointers are variables that contain the memory address of another variable. Since an address in a memory is a numeric value we can perform arithmetic operations on the pointer values. The different operations that can be possibly performed on pointers are:

- Incrementing/Decrementing a pointer
- Addition/Subtraction of a constant number to a pointer
- Subtraction of one pointer from another
- Comparison of two pointers
- Table of Contents
- Incrementing/Decrementing a Pointer
- Addition/Subtraction of a constant number to a pointer
- Subtraction of one pointer from another
- Comparison of two pointers
- Operations not possible with pointers

Operations on Pointers:

16 bit Machine

In a 16 bit machine, size of all types of pointer, be it int*, float*, char* or double* is always **2 bytes**. But when we perform any arithmetic function like increment on a pointer, changes occur as per the size of their primitive data type.

Size of datatypes on 16-bit Machine:

Type	**Size (in bytes)**
int or signed int	2
char	1
long	4
float	4
double	8
long double	10

Examples for Pointer Arithmetic

Now lets take a few examples and understand this more clearly.

int*i;
i++;

In the above case, pointer will be of 2 bytes. And when we increment it, it will increment by 2 bytes because int is also of 2 bytes.

float*i;
i++;

In this case, size of pointer is still 2 bytes initially. But now, when we increment it, it will increment by 4 bytes because float datatype is of 4 bytes.

double*i;
i++;

Similarly, in this case, size of pointer is still 2 bytes. But now, when we increment it, it will increment by 8 bytes because its data type is double.

---

32 bit Machine

The concept of pointer arithmetic remains exact same, but the size of pointer and various datatypes is different in a 32 bit machine. Pointer in 32 bit machine is of **4 bytes**.

And, following is a table for **Size of datatypes on 32-bit Machine :**

Type	Size (in bytes)
int or signed int	4
char	2
long	8
float	8
double	16

Program for pointer arithmetic (32-bit machine)

```
#include <stdio.h>
int main()
{
   int m =5, n =10, o =0;

   int *p1;
   int *p2;
   int *p3;

   p1 =&m;//printing the address of m
   p2 =&n;//printing the address of n

printf("p1 = %d\n", p1);
printf("p2 = %d\n", p2);

   o =*p1+*p2;
printf("*p1+*p2 = %d\n", o);//point 1

   p3 = p1-p2;
printf("p1 - p2 = %d\n", p3);//point 2

   p1++;
printf("p1++ = %d\n", p1);//point 3

   p2--;
printf("p2-- = %d\n", p2);//point 4

return0;
}
```

output:

```
p1 = 2680016
p2 = 2680012
*p1+*p2 = 15
p1-p2 = 1
p1++ = 2680020
p2-- = 2680008
```

Explanation of the above program:

1. **Point 1:** Here, * means 'value at the given address'. Thus, it adds the value of m and n which is 15.
2. **Point 2:** It subtracts the addresses of the two variables and then divides it by the size of the pointer datatype (here integer, which has size of 4 bytes) which gives us the number of elements of integer data type that can be stored within it.
3. **Point 3:** It increments the address stored by the pointer by the size of its datatype(here 4).
4. **Point 4:** It decrements the address stored by the pointer by the size of its datatype(here 4).
5. **Point 5:** Addition of two pointers is not allowed.

Pointers as Function Argument

Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as **call by reference**. When a function is called by reference any change made to the reference variable will effect the original variable

---

INCREMENTING / DECREMENTING A POINTER

Any pointer variable when incremented points to the next memory location of its type. For **example:**

```
int a=5,*x;
x=&amp;a;
x++;
```

Here is a program which shows both Increment and decrement values of a pointer variable:

```
#include <stdio.h>
int main()
{
   int a=5,*x;//declaring the pointer for integer variable
   char b='z',*y;//declaring the pointer for char variable
   x=&a;//storing the memory location of variable a in pointer variable x
   /*The corresponding values of the increment and decrement operations on pointer variable
x are given below*/
printf("x= %d\n",x);//printing the actual value of x
   x++;
printf("x++= %d\n",x);//the value gets incremented by 4 bytes because the size of one int
variable is 4 bytes
```

```
    x--;
printf("x--= %d\n",x);//the value gets decremented by 4 bytes and changes back to the original
value
    y=&b;//storing the memory location of variable b in pointer variable y
    /*The corresponding values of the increment and decrement operations on pointer variable
y are given below*/
printf("y= %d\n",y);//printing the actual value of y
    y++;
printf("y++= %d\n",y);//the value gets incremented by 1 byte because the size of one char
variable is 1 bytes
    y--;
printf("y--= %d\n",y);//the value gets decremented by 1 byte and changes back to the original
value
    return 0;
}
```

Output:-

```
x= 2012376492
x++= 2012376496
x--= 2012376492
y= 2012376491
y++= 2012376492
y--= 2012376491
```

ADDITION/SUBTRACTION OF A CONSTANT NUMBER TO A POINTER

Addition or subtraction of a constant number to a pointer is allowed. The result is similar to the increment or decrement operator with the only difference being the increase or decrease in the memory location by the constant number given. Also, not to forget the values get incremented or decremented according to the type of variable it stores. The following program shows an example of addition and subtraction of a constant number to a pointer:

Note: – Output may vary every time the program is run because memory locations may differ with each execution.

```
#include <stdio.h>
int main()
{
    int a=5,*x;//declaring the pointer for integer variable
    char b='z',*y;//declaring the pointer for char variable
    x=&a;//storing the memory location of variable a in pointer variable x
    /*The corresponding values of the addition and subtraction operations on pointer variable
x are given below*/
printf("x= %d\n",x);//printing the actual value of x
printf("x+3= %d\n",x+3);//the value incremented by 3
printf("x-2= %d\n",x-2);//the value decremented by 2
```

```
    y=&b;//storing the memory location of variable b in pointer variable y
    /*The corresponding values of the addition and subtraction operations on pointer variable
y are given below*/
printf("y= %d\n",y);//printing the actual value of y
printf("y+3= %d\n",y+3);//the value incremented by 3
printf("y-2= %d\n",y-2);//the value decremented by 2
    return 0;
}
```

Output:-

```
x= 593874396
x+3= 593874408
x-2= 593874388
y= 593874395
y+3= 593874398
y-2= 593874393
```

SUBTRACTION OF ONE POINTER FROM ANOTHER

A pointer variable can be subtracted from another pointer variable only if they point to the elements of the same array. Also, subtraction of one pointer from another pointer that points to the elements of the same array gives the number of elements between the array elements that are indicated by the pointer. The following example shows this:

```
#include <stdio.h>
int main()
{
    int num[10]={1,5,9,4,8,3,0,2,6,7},*a,*b;
    a=&num[2];//storing the address of num[2] in variable a
    b=&num[6];//storing the address of num[6] in variable b
printf("a = %d\n",a);
printf("b = %d\n",b);
printf("a-b = %d\n",b-a);//prints the number of elements between the two elements indicated
by the pointers
printf("*a-*b = %d\n",*a-*b);//prints the difference in value of the two elements
    return 0;
}
```

Output:-

```
a= 2686680
b= 2686696
a-b = 4
*a-*b = 9
```

We get the result as 4 which is not the arithmetic difference of the address values of the two variables. We rather get the number of elements separating the corresponding array elements. *a-*b gives the difference of the values stored in the respective positions in the array.

```
#include <stdio.h>
int main()
{
   int num[10]={1,5,9,4,8,3,0,2,6,7},*a,*b,*c;
   a=&num[2];//storing the address of num[2] in variable a
   b=(num+2);//base address plus 2 stores the address of num[2] in the variable b
   c=&num[6];//storing the address of num[6] in variable b
   /*Print values of all the pointers*/
printf("a= %d\n",a);
printf("b= %d\n",b);
printf("c= %d\n",c);
   if(a==b)//comparing for equality
printf("a and b point to the same location and the value is: %d\n",*a);
   if(a!=c)//comparing for inequality
printf("a and c do not point to the same location in the memory");
   return 0;
}
```

Output:-

```
a= 2686676
b= 2686676
c= 2686692
a and b point to the same location and the value is: 9
a and c do not point to the same location in the memory
```

OPERATIONS NOT POSSIBLE WITH POINTERS

- There are a few operations that are not possible with pointers. These are:
- Addition of two pointer variables
- Multiplication of a pointer with a constant value
- Division of a pointer with a constant value

DYNAMIC MEMORY ALLOCATION IN C:

The process of allocating memory during program execution is called dynamic memory allocation.

C language offers 4 dynamic memory allocation functions. They are,

1. malloc()
2. calloc()
3. realloc()
4. free()

Function	**Syntax**
malloc ()	malloc (number *sizeof(int));
calloc ()	calloc (number, sizeof(int));
realloc ()	realloc (pointer_name, number * sizeof(int));
free ()	free (pointer_name);

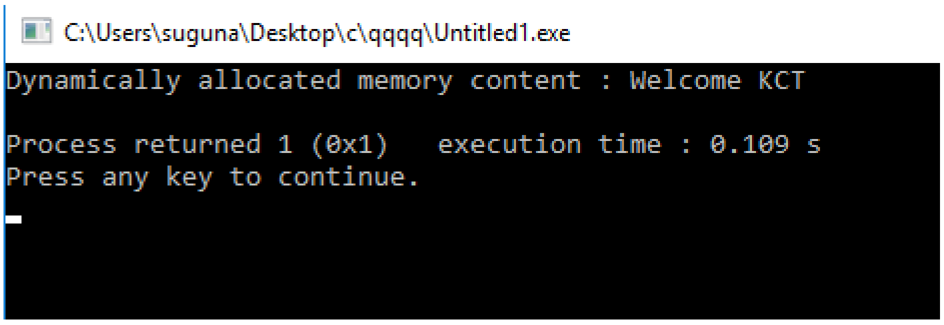
1. MALLOC () FUNCTION IN C:

- malloc () function is used to allocate space in memory during the execution of the program.
- malloc () does not initialize the memory allocated during execution. It carries garbage value.
- malloc () function returns null pointer if it couldn’t able to allocate requested amount of memory.

A). EXAMPLE PROGRAM FOR MALLOC () FUNCTION IN C:

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

int main()
{
    char *mem_allocation;
    /* memory is allocated dynamically */
mem_allocation = malloc( 20 * sizeof(char) );
if( mem_allocation== NULL )
    {
printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
strcpy( mem_allocation,"Welcome KCT");
    }
printf("Dynamically allocated memory content : " \
         "%s\n", mem_allocation );
    free(mem_allocation);
}
```

Output:

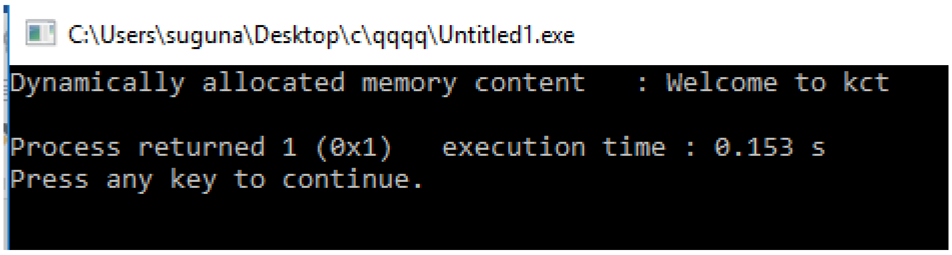

C:\Users\suguna\Desktop\c\qqqq\Untitled1.exe

```
Dynamically allocated memory content : Welcome KCT

Process returned 1 (0x1)   execution time : 0.109 s
Press any key to continue.
```

B) CALLOC () FUNCTION IN C:

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

int main()
{
    char *mem_allocation;
    /* memory is allocated dynamically */
mem_allocation = calloc( 20, sizeof(char) );
if( mem_allocation== NULL )
    {
printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
strcpy( mem_allocation,"fresh2refresh.com");
    }
printf("Dynamically allocated memory content   : " \
            "%s\n", mem_allocation );
       free(mem_allocation);
}
```

Output

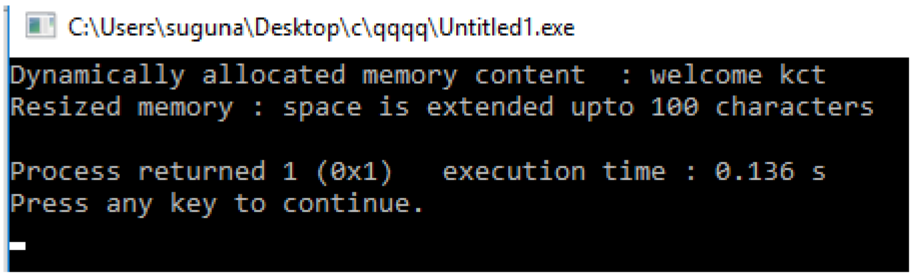

C:\Users\suguna\Desktop\c\qqqq\Untitled1.exe

```
Dynamically allocated memory content    : Welcome to kct

Process returned 1 (0x1)   execution time : 0.153 s
Press any key to continue.
```

C) REALLOC () FUNCTION IN C:

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

```
int main()
{
   char *mem_allocation;
   /* memory is allocated dynamically */
mem_allocation = malloc( 20 * sizeof(char) );
if( mem_allocation == NULL )
   {
printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
strcpy( mem_allocation,"welcomekct");
   }
printf("Dynamically allocated memory content  : " \
         "%s\n", mem_allocation );
mem_allocation=realloc(mem_allocation,100*sizeof(char));
if( mem_allocation == NULL )
   {
printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
strcpy( mem_allocation,"space is extended upto " \
                       "100 characters");
   }
printf("Resized memory : %s\n", mem_allocation );
   free(mem_allocation);}
```



C:\Users\suguna\Desktop\c\qqqq\Untitled1.exe

```
Dynamically allocated memory content  : welcome kct
Resized memory : space is extended upto 100 characters

Process returned 1 (0x1)   execution time : 0.136 s
Press any key to continue.
```

DIFFERENCE BETWEEN STATIC MEMORY ALLOCATION AND DYNAMIC MEMORY ALLOCATION IN C:

DIFFERENCE BETWEEN MALLOC () AND CALLOC () FUNCTIONS IN C:

malloc()	**calloc()**
It allocates only single block of requested memory	It allocates multiple blocks of requested memory

int *ptr;ptr = malloc(20 * sizeof(int));For the above, 20*4 bytes of memory only allocated in one block.
Total = 80 bytes
	int *ptr;Ptr = calloc(20, 20 * sizeof(int));
	For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory.
	Total = 1600 bytes

malloc () doesn't initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
type cast must be done since this function returns void pointer int *ptr;ptr = (int*)malloc(sizeof(int)*20);	Same as malloc () function int *ptr;ptr = (int*)calloc(20, 20 * sizeof(int));