Chapter-12
POINTER

Definition:
The pointer in C language is a variable which stores the address of another variable. Since Pointer is also a kind of variable , thus pointer itself will be stored at some different memory location. This variable can be of type int, char, float etc.

Address in C:
Whenever a variable is defined in C language, a memory location is assigned for it, in which it's value will be stored. We can easily check this memory address, using the & operator.

If var is the name of the variable, then &var will give it's address.

Example:
```
#include<stdio.h>
int main()
{
    int var = 7;
    printf("Value of the variable var is: %d\n", var);
   printf("Memory address of the variable var in Decimal is: %d\n", &var);         // d for decimal value
   printf("Memory address of the variable var in Hexadecimal is: %x\n", &var);  // x for hexadecimal
   return 0;
}
```

Declaring a pointer:
The pointer in c language is declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

Syntax:
```
 Data_type *var_name;
```

E.g.:
```
int *a;
```

Here, a is a pointer variable which will holds the address of another variable.

Program:
```
#include<stdio.h>
int main()
{
int number=50;
int *p;  //pointer declaration
p=&number; // pointer initialization stores the address of number variable
printf("Address of p variable is %x \n",p);
return 0;
}
```

Accessing variables through pointers:

- ‘&’ is the reference operator
- ‘*’ is the dereference operator

Accessing variables through pointers:
We can access the value of the variable by using asterisk (*****) - it is known as **dereference operator.**

Program:
```
#include <stdio.h>
int main()
{
        //normal variable
        int num = 100;

        //pointer variable
        int *ptr;

        //pointer initialization
        ptr = &num;

        //printing the value
        printf("value of num = %d\n", *ptr);

        return 0;
}
```

Output:
```
value of num = 100
```

Pointer to Pointer:
Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value.

The following declaration declares a pointer to a pointer of type int :

int **var, ***var;

program 1:
```
#include <stdio.h>
int main ()
 {
   int  var;
   int  *ptr;
   int  **pptr;

   var = 3000;

   /* take the address of var */
   ptr = &var;
   /* take the address of ptr using address of operator & */
   pptr = &ptr;

   /* take the value using pptr */
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);

   return 0;
}
```

Output:

```
Value of var = 3000
Value available at *ptr = 3000
Value available at  **pptr = 3000
```

program 2:

```
#include <stdio.h>
 int main ()
{
   int  var,*ptr1,**ptr2,***ptr3;
   var = 3000;
   ptr1 = &var;
   ptr2 = &ptr1;
   ptr3 = &ptr2;

   printf("Value of var = %d\n", var );
   printf("Value available at *ptr1 = %d\n", *ptr1 );
   printf("Value available at **ptr2 = %d\n",**ptr2);
   printf("Value available at ***ptr3 = %d\n", ***ptr3);
   return 0;
}
```

Output:

```
Value of var = 3000
Value available at *ptr1 = 3000
Value available at **ptr2 = 3000
Value available at ***ptr3 = 3000
```

C program to add 2 numbers using pointer:

```
#include <stdio.h>
int main()
{
   int first, second, *p, *q, sum;
   printf("Enter two integers to add\n");
   scanf("%d%d", &first, &second);
   p = &first;
   q = &second;
   sum = *p + *q;
   printf("Sum of the numbers = %d\n", sum);
   return 0;
}
```

Call by value and Call by reference in C:
There are two methods to pass the data into the function in C language, i.e., *call by value* and *call by reference*.

Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we cannot modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

PROGRAM FOR CALL BY VALUE

```
#include<stdio.h>
void change(int num) //formal parameter
 {
    printf("Before adding value inside function num=%d \n",num);
    num=num+100;
    printf("After adding value inside function num=%d \n", num);
 }
int main()
 {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(x); //passing value in function    actual parameter
    printf("After function call x=%d \n", x);
    return 0;
}
```

OUTPUT:

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.

PROGRAM:

```
#include<stdio.h>
void change(int *num)
{
    printf("Before adding value inside function num=%d \n",*num);
   *num=*num+10;
    printf("After adding value inside function num=%d \n", *num);
}
int main()
 {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(&x);//passing reference in function
    printf("After function call x=%d \n", x);
    return 0;
}
```

OUTPUT:

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=110
After function call x=110
```

Difference between call by value and call by reference in c:

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

Structure and pointer:
Arrow (->) is used to access structure member using pointer.

Program:

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

```
#include<stdio.h>
struct student
{
            int roll;
            char name[20];
}s1={10,"neha"};
void data(struct student*);
int main()
{
     data(&s1);
    return 0;
}
void data(struct student *p)
{
  printf("roll no is %d\n",p->roll);
  printf("roll no is %d\n",(*p).roll);
    printf("name is %s\n",p->name);
  printf("roll no is %s\n",(*p).name);
}
```

Output:

```
roll no is 10
roll no is 10
name is neha
roll no is neha
```

NULL Pointer:

A NULL pointer is defined as a special pointer value that points to nowhere in the memory. **Null** is a built-in constant that has a value of zero and NULL is defined under stdio.h header file.

E.g.:

```
 #include<stdio.h>
  int *p=NULL;
```

Here, p is a NULL pointer, this indicates that the pointer variable p does not point to any part of the memory.

Compatibility:

We should not store the address of a data variable of one type into a pointer variable of another type. During assigning we should see that the type of variable and type of pointer variable should be of same type otherwise it results in syntax error.

E.g. 1:

```
   int a=10;
   int  *p;
```

p=&a; // **valid:**here pointer variable 'p' is of integer type and variable a is also of type integer so type of both variables are compatible.

Eg 2:

```
  int a=10;
  float  *p;
```

p=&a; // **invalid:** here pointer variable 'p' is of float type and variable a is of integer so type of both variables are in-compatible.

void pointer:-

- A void pointer is a special type of pointer. It can point to any data type, Also any pointer can be assigned as a void pointer. Hence, void pointer is also called universal pointer.
- The only limitation in void pointer is that the data cannot be de-referenced directly using indirection operator (*).

Eg: int a=10;
char b='a';
void *ptr,*ptr1; // here ptr and ptr1 is of void type pointer so it can store address of any type
ptr=&a;
ptr1=&b;

program

```
#include<stdlib.h>
int main() {
   int a = 7;
   float b = 7.6;
   void *p;
   p = &a;
   printf("Integer variable is = %d", *( (int*) p) );
   p = &b;
   printf("\nFloat variable is = %f", *( (float*) p) );
   return 0;
}
```

Here, we are converting void pointer into integer and float pointer. But in the case of a void pointer we need to typecast the pointer variable to dereference it.

Dangling/Wild Pointer:
A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.

Example:

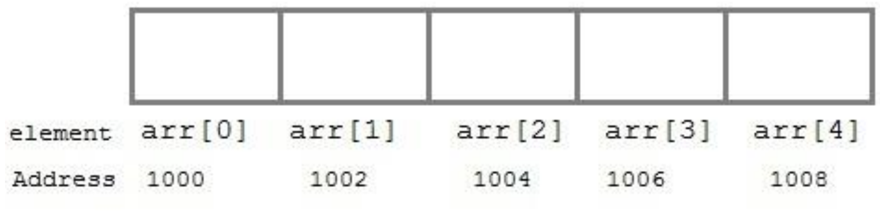
```
int main()
{
   float *p = (float *)malloc(sizeof(float));
   //dynamic memory allocation.
   free(p);
   //after calling free() p becomes a dangling pointer

   p = NULL;
   //now p no more a dangling pointer.
}
```

Pointer and Array:-

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.

Pointer to Array:
We can use a pointer to point to an array, and then we can use that pointer to access the array elements.

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++;
   }
   return 0;
}#
include
```

OR

```
#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+i);

   }
   return 0;
}
```

Output:
1 2 3 4 5

Program to take input from user using pointer in array element:

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

    for (i = 0; i < 5; i++)
   {
      printf("%d ",a[i]);
      p++;
   }

   return 0;
}
```

Output:

```
enter elements:
12 43 54 24 54
Elements are:
12 43 54 24 54
```

Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc():

1. **malloc():**

- “malloc” or “memory allocation” method in C is used to dynamically allocate block of memory with the specified size. It is defined in stdlib.h header file.
- It returns a pointer of type void which can be cast into a pointer of any form.
- It doesn't initialize memory at execution time, so it has garbage value initially.

Syntax:
ptr = (cast-type*) malloc (byte-size)

Example:

```
int *p;
p= (int*) malloc (5 * sizeof(int));
```

Since the size of int is 4 bytes, this statement will allocate 20 bytes of memory. And, the pointer p holds the address of the first byte in the allocated memory.

Program 1:

```
#include <stdlib.h>
#include<stdio.h>
int main()
{
   int n,i,*p;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   p=(int*)malloc(n * sizeof(int));  //memory allocated using malloc

   *(p+0)=20; // *p is equal to *(p+0)
   *(p+1)=21;
   *(p+2)=16;
   *(p+3)=17;
    printf("%d ",*(p+0));
    printf("%d ",*(p+1));
    printf("%d ",*(p+2));
   return 0;
}
```

Program 2:

```
#include <stdlib.h>
#include<stdio.h>
int main()
{
   int n,i,*p;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   p=(int*)malloc(n * sizeof(int));  //memory allocated using malloc

     printf("Enter elements of array:\n");
     for(i=0;i<n;++i)
     {
       scanf("%d",&*(p+i));
     }
     printf("Elements of array are\n");
     for(i=0;i<n;i++)
     {
       printf("%d\n",*(p+i));
     }
   return 0;
}
```

Output:

```
Enter number of elements: 3
Enter elements of array:
12
32
43
Elements of array are
12
32
43
```

Program 3:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int *p,i,n;
scanf("%d",&n);
p=(int*)malloc(n*sizeof(int));
for(i=0;i<n;i++)
{
scanf("%d",&p[i]);
}
for(i=0;i<n;i++)
{
printf("%d ",p[i]);
}
return 0;
}
```

2. calloc():

- The name "calloc" stands for contiguous allocation.
- The malloc() function allocates memory and leaves the memory uninitialized. Whereas, the calloc() function allocates memory and initializes to zero and it takes 2 arguments.

Syntax:
ptr = (cast-type*)calloc(n, element-size);

here,

- n indicates number of blocks
- element-size indicates size of each block

program 1:

```
#include <stdlib.h>
#include<stdio.h>
int main()
{
    int n,i,*p;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    p=(int*)calloc(n , sizeof(int));  //memory allocated using malloc
      printf("Enter elements of array:\n");
      for(i=0;i<n;i++)
      {
        scanf("%d",&*(p+i));
      }
      printf("Elements of array are\n");
      for(i=0;i<n;i++)
      {
        printf("%d\n",*(p+i));
      }
    return 0;
}
```

Output:

```
Enter number of elements: 4
Enter elements of array:
1 2 43 54
Elements of array are
1 2 43 54
```

Program 2:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int *p,i,n;
printf("enter numbers of required blocks:\n");
scanf("%d",&n);
p=(int*)calloc(n,sizeof(int));
printf("enter values:\n");
for(i=0;i<n;i++)
{
scanf("%d",&p[i]);
}
printf("values are:\n");
for(i=0;i<n;i++)
{
printf("%d ",p[i]);

 }
return 0;
}
```

Output:

```
enter numbers of required blocks:
3
enter values:
12 32 45
values are:
12 32 45
```

3. realloc:

- If memory is not sufficient for malloc( ) or calloc( ), you can reallocate the memory by realloc() function.
- In short, it changes the memory size.

Syntax:

ptr=realloc(ptr1, **new**-size);

program:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr, i , n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);//2
    ptr = (int*) malloc(n1 * sizeof(int));
    printf("Addresses of previously allocated memory:\n ");
    for(i = 0; i < n1; i++)
         printf("%d\n",ptr + i);
    printf("\nEnter the new size: ");
    scanf("%d", &n2);//3
    // rellocating the memory
    ptr =(int*)realloc(ptr, n2 * sizeof(int));
    printf("Addresses of newly allocated memory:\n ");
    for(i = 0; i < n2;i++)
   printf("%d\n", ptr + i);

    return 0;
}
```

Output:

```
Enter size: 2
Addresses of previously allocated memory:
12610464
```

```
12610468

Enter the new size: 4
Addresses of newly allocated memory:
12610464
12610468
12610472
12610476
```

Program 2:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr,*ptr1 ,i , n1, n2;
    printf("Enter size:\n");
    scanf("%d", &n1);
    ptr = (int*) malloc(n1 * sizeof(int));

    printf("enter elements:\n");
    for(i=0;i<n1;i++)
    {
     scanf("%d",&*(ptr+i));
            }

    printf(" elements are:\n");
    for(i=0;i<n1;i++)
    {
            printf("%d ",*(ptr+i));
            }
    printf("\nAllocate new size:\n");
    scanf("%d", &n2);//6
            ptr = realloc(ptr, n2 * sizeof(int));
    printf("enter new elements:\n");
    for(i=n1;i<n2;i++)
    {
            scanf("%d",&*(ptr+i));

   }
    printf("\nnew elements are:\n");
    for(i=0;i<n2;i++)
    {
            printf("%d ",*(ptr+i));
            }
    return 0;
}
```

Output:

```
Enter size:
2
```

```
enter elements:
12
32
elements are:
12 32
Allocate new size:
3
enter new elements:
32
new elements are:
     12  2 32
```

4. free():

- It deallocates the memory previously allocated by a call to calloc, malloc, or realloc.

Syntax:

```
free(ptr);
```

program:

```
#include <stdlib.h>
#include<stdio.h>
  int main()
  {
    int n,i,*p;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    p=(int*)calloc(n , sizeof(int));  //memory allocated using malloc
      printf("Enter elements of array:\n");
      for(i=0;i<n;i++)
      {
        scanf("%d",&*(p+i));
      }
      printf("Elements of array are\n");
      for(i=0;i<n;i++)
      {
        printf("%d\n",*(p+i));
      }
       printf("before deallocation value is %d\n",*p);
      free(p);
      printf("memory is deallocated successfully:\n");
      printf("after deallocation value is  %d\n",*p);
    return 0;
}
```

Output:

```
Enter number of elements: 2
Enter elements of array:
```

```
11
14
Elements of array are
11
14
before deallocation value is 11
memory is deallocated successfully:
after deallocation value is  1971168
```

***