C dereference pointer

As we already know that **"what is a pointer"**, a pointer is a variable that stores the address of another variable. The dereference operator is also known as an indirection operator, which is represented by (*). When indirection operator (*) is used with the pointer variable, then it is known as **dereferencing a pointer.** When we dereference a pointer, then the value of the variable pointed by this pointer will be returned.

Why we use dereferencing pointer?

Dereference a pointer is used because of the following reasons:

- It can be used to access or manipulate the data stored at the memory location, which is pointed by the pointer.
- Any operation applied to the dereferenced pointer will directly affect the value of the variable that it points to.

Let's observe the following steps to dereference a pointer.

First, we declare the integer variable to which the pointer points.

```
int x =9;
```

Now, we declare the integer pointer variable.

```
int *ptr;
```

After the declaration of an integer pointer variable, we store the address of 'x' variable to the pointer variable 'ptr'.

```
ptr=&x;
```

We can change the value of 'x' variable by dereferencing a pointer 'ptr' as given below:

```
*ptr =8;
```

The above line changes the value of 'x' variable from 9 to 8 because 'ptr' points to the 'x' location and dereferencing of 'ptr', i.e., *ptr=8 will update the value of x.

Let's combine all the above steps:

```
#include <stdio.h>
int main()
{
   int x=9;
   int *ptr;
   ptr=&x;
```

```
    *ptr=8;
    printf("value of x is : %d", x);
    return 0;}
```

OUTPUT

value of x is : 8

Let's consider another example.

```
#include <stdio.h>
int main()
{
   int x=4;
   int y;
   int *ptr;
   ptr=&x;
   y=*ptr;
   *ptr=5;
   printf("The value of x is : %d",x);
   printf("\n The value of y is : %d",y);
   return 0;
}
```

OUTPUT

The value of x is : 5

The value of y is : 4

In the above code:

- We declare two variables 'x' and 'y' where 'x' is holding a '4' value.
- We declare a pointer variable 'ptr'.
- After the declaration of a pointer variable, we assign the address of the 'x' variable to the pointer 'ptr'.
- As we know that the 'ptr' contains the address of 'x' variable, so '*ptr' is the same as 'x'.
- We assign the value of 'x' to 'y' with the help of 'ptr' variable, i.e., y=***ptr** instead of using the 'x' variable.

Let's consider another scenario.

```
#include <stdio.h>
int main()
{
  int a=90;
```

```
  int *ptr1,*ptr2;
  ptr1=&a;
  ptr2=&a;
  *ptr1=7;
  *ptr2=6;
   printf("The value of a is : %d",a);
   return 0;
}
```

In the above code:

- First, we declare an 'a' variable.
- Then we declare two pointers, i.e., ptr1 and ptr2.
- Both the pointers contain the address of 'a' variable.
- We assign the '7' value to the *ptr1 and '6' to the *ptr2. The final value of 'a' would be '6'.

OUTPUT

The value of a is : 6

C Function Pointer

As we know that we can create a pointer of any data type such as int, char, float, we can also create a pointer pointing to a function. The code of a function always resides in memory, which means that the function has some address. We can get the address of memory by using the function pointer.

Let's see a simple example.

```
#include <stdio.h>
int main()
{
   printf("Address of main() function is %p",main);
   return 0;
}
```

Output

Address of main() function is 0x400536

In the above output, we observe that the main() function has some address. Therefore, we conclude that every function has some address.

Declaration of a function pointer

Till now, we have seen that the functions have addresses, so we can create pointers that can contain these addresses, and hence can point them.

Syntax of function pointer

```
return type (*ptr_name)(type1, type2…);
```

For example:

```
int (*ip) (int);
```

In the above declaration, *ip is a pointer that points to a function which returns an int value and accepts an integer value as an argument.

```
float (*fp) (float);
```

In the above declaration, ***fp** is a pointer that points to a function that returns a float value and accepts a float value as an argument.

We can observe that the declaration of a function is similar to the declaration of a function pointer except that the pointer is preceded by a '*****'. So, in the above declaration, fp is declared as a function rather than a pointer.

Till now, we have learnt how to declare the function pointer. Our next step is to assign the address of a function to the function pointer.

```
1. float (*fp) (int , int);    // Declaration of a function pointer.
2. float func( int , int );    // Declaration of  function.
3. fp = func;                  // Assigning address of func to the fp pointer.
```

In the above declaration, '**fp**' pointer contains the address of the '**func**' function.

Note: Declaration of a function is necessary before assigning the address of a function to the function pointer.

Calling a function through a function pointer

We already know how to call a function in the usual way. Now, we will see how to call a function using a function pointer.

Suppose we declare a function as given below:

```
float func(int , int);     // Declaration of a function.
```

Calling an above function using a usual way is given below:

```
result = func(a , b);    // Calling a function using usual ways.
```

Calling a function using a function pointer is given below:

```
result = (*fp)( a , b);    // Calling a function using function pointer.
```

Or

```
result = fp(a , b);        // Calling a function using function pointer, and indirection operator can be removed.
```

The effect of calling a function by its name or function pointer is the same. If we are using the function pointer, we can omit the indirection operator as we did in the second case. Still, we use the indirection operator as it makes it clear to the user that we are using a function pointer.

Let's understand the function pointer through an example.

```
#include <stdio.h>
int add(int,int);
int main()
{
  int a,b;
  int (*ip)(int,int);
  int result;
  printf("Enter the values of a and b : ");
  scanf("%d %d",&a,&b);
  ip=add;
  result=(*ip)(a,b);
  printf("Value after addition is : %d",result);
   return 0;
}
int add(int a,int b)
{
   int c=a+b;
   return c;
}
OUTPUT
Enter the values of a and b : 22 23
Value after addition is : 45
```

Passing a function's address as an argument to other function

We can pass the function's address as an argument to other functions in the same way we send other arguments to the function.

Let's understand through an example.

```
include <stdio.h>
void func1(void (*ptr)());
void func2();
```

```
int main()
{
   func1(func2);
    return 0;
}
void func1(void (*ptr)())
{
   printf("Function1 is called");
   (*ptr)();
}
void func2()
{
   printf("\nFunction2 is called");
}
```

In the above code, we have created two functions, i.e., func1() and func2(). The func1() function contains the function pointer as an argument. In the main() method, the func1() method is called in which we pass the address of func2. When func1() function is called, 'ptr' contains the address of 'func2'. Inside the func1() function, we call the func2() function by dereferencing the pointer 'ptr' as it contains the address of func2.

OUTPUT

Function1 is called

Function2 is called

Array of Function Pointers

Function pointers are used in those applications where we do not know in advance which function will be called. In an array of function pointers, array takes the addresses of different functions, and the appropriate function will be called based on the index number.

Let's understand through an example.

```
#include <stdio.h>
   float add(int,int);
   float sub(int,int);
   float mul(int,int);
   float div(int,int);
   int main()
   {
      int x;            // variable declaration.
      int y;
      float (*fp[4]) (int,int);       // function pointer declaration.
      fp[0]=add;            // assigning addresses to the elements of an array of a function
pointer.
      fp[1]=sub;
      fp[2]=mul;
```

```
    fp[3]=div;
    printf("Enter the values of x and y :");
    scanf("%d %d",&x,&y);
    float r=(*fp[0]) (x,y);        // Calling add() function.
    printf("\nSum of two values is : %f",r);
     r=(*fp[1]) (x,y);             // Calling sub() function.
    printf("\nDifference of two values is : %f",r);
      r=(*fp[2]) (x,y);            // Calliung sub() function.
    printf("\nMultiplication of two values is : %f",r);
     r=(*fp[3]) (x,y);            // Calling div() function.
    printf("\nDivision of two values is : %f",r);
    return 0;
}

float add(int x,int y)
{
    float a=x+y;
    return a;
}
float sub(int x,int y)
{
    float a=x-y;
    return a;
}
float mul(int x,int y)
{
    float a=x*y;
    return a;
}
float div(int x,int y)
{
    float a= (float) x/y;
    return a;
}
```

In the above code, we have created an array of function pointers that contain the addresses of four functions. After storing the addresses of functions in an array of function pointers, we call the functions using the function pointer.

OUTPUT

```
Enter the values of x and y : 5 7
Sum of two values is : 12.000000
Difference of two values is :-2.000000
Multiplication of two values is : 35.000000
Division of two values is : 0.714286
```