Pointer to Function

```
#include <stdio.h>
// A normal function with an int parameter // and void return type
void fun(int a)
{
   printf("Value of a is %d\n", a);
}
 int main()
{
   // fun_ptr is a pointer to function fun()
   void (*fun_ptr)(int) = &fun;

   /* The above line is equivalent of following two
      void (*fun_ptr)(int);
      fun_ptr = &fun;
   */
     // Invoking fun() using fun_ptr
   (*fun_ptr)(10);

   return 0;
}
```

Output:-

Following are some interesting facts about function pointers.

1) Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

2) Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

3) A function's name can also be used to get functions' address. For example, in the below program, we have removed address operator '&' in assignment. We have also changed function call by removing *, the program still works.

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
   printf("Value of a is %d\n", a);
}

int main()
{
   void (*fun_ptr)(int) = fun;  // & removed

   fun_ptr(10);  // * removed
```

```
    return 0;
}
```
Output:

4) Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.

5) Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks.

```
#include <stdio.h>
void add(int a, int b)
{
    printf("Addition is %d\n", a+b);
}
void subtract(int a, int b)
{
    printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b)
{
    printf("Multiplication is %d\n", a*b);
}
int main()
{
    // fun_ptr_arr is an array of function pointers
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, 1 for subtract and 2 "
           "for multiply\n");
    scanf("%d", &ch);

    if (ch > 2) return 0;

    (*fun_ptr_arr[ch])(a, b);

    return 0;
}
```
Enter Choice: 0 for add, 1 for subtract and 2 for multiply
2
Multiplication is:-

6) Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.
For example, consider the following C program where wrapper() receives a void fun() as parameter and calls the passed function.

```
// A simple C program to show function pointers as parameter
#include <stdio.h>
```

```
// Two simple functions
void fun1() { printf("Fun1\n"); }
void fun2() { printf("Fun2\n"); }

// A function that receives a simple function
// as parameter and calls the function
void wrapper(void (*fun)())
{
    fun();
}
  int main()
{
    wrapper(fun1);
    wrapper(fun2);
    return 0;
}
```

Passing address to a Function

```
#include <stdio.h>
void swap(int *n1, int *n2);
int main()
{
    int num1 = 5, num2 = 10;
    // address of num1 and num2 is passed
    swap( &num1, &num2);
    printf("num1 = %d\n", num1);
    printf("num2 = %d", num2);
    return 0;
}
// pointer n1 and n2 stores the address of num1 and num2 respectively
void swap(int* n1, int* n2)
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

When you run the program, the output will be:

```
num1 = 10
num2 = 5
```

Pointer to Array-

Once you store the address of the first element in 'p', you can access the array elements using *p, *(p+1), *(p+2) and so on. Given below is the example to show all the concepts discussed above −

```
#include <stdio.h>

int main () {

   /* an array with 5 elements */
```

```
   double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
   double *p;
   int i;
   p = balance;
    /* output each array element's value */
   printf( "Array values using pointer\n");
   for ( i = 0; i < 5; i++ ) {
      printf("*(p + %d) : %f\n",  i, *(p + i) );
   }
   printf( "Array values using balance as address\n");

   for ( i = 0; i < 5; i++ ) {
      printf("*(balance + %d) : %f\n",  i, *(balance + i) );
   }
    return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Array values using pointer
*(p + 0) : 1000.000000
*(p + 1) : 2.000000
*(p + 2) : 3.400000
*(p + 3) : 17.000000
*(p + 4) : 50.000000
Array values using balance as address
*(balance + 0) : 1000.000000
*(balance + 1) : 2.000000
*(balance + 2) : 3.400000
*(balance + 3) : 17.000000
*(balance + 4) : 50.000000
```