# Function pointer in C

In **C**, it is also possible to define and use **function pointers**, i.e. **pointer** variables which point to **functions.** Function pointer is a pointer which stores the address of another function.

In C, like [normal data pointers](https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/) (int \*, char \*, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

#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;

}

[How to declare a pointer to a function?](https://www.geeksforgeeks.org/how-to-declare-a-pointer-to-a-function/)

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.

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>

// 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;

}

**Array of Function Pointer**

#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;

}

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;

}

As we have learn about function pointer from basic to advance level in article [Function Pointers in C/C++](http://www.firmcodes.com/basics-function-pointers-cc/), now its time to learn What are practical uses of function pointers in c ?

There are many use of function pointer but all they are summing around callback construct, so here i write two use cases for function pointers upon callback construction:

=> Implement Callback functions – used for Event Handlers, parser specialization, comparator function passing.

=> Dynamically function calling(One kind of callback use case) – Create plugins and extension, Enable-Disables some features upon certain events, creating Finite State Machines(FSM), etc

# What is a Callback function?

In simple terms, a Callback function is one that is not called explicitly by the programmer. Instead, there is some mechanism that continually waits for events to occur, and it will call selected functions in response to particular events.

                Thus, callbacks allow the user of a function to fine-tune it at runtime(Dynamically function calling).

# Practical scenario of Callback

Error Signaling in UNIX – A Unix program, for example, might not want to terminate immediately when it receives SIGTERM, to make sure things get taken care of, it would register the cleanup function as a callback.

                Typing certain key combinations at the controlling terminal of a running process in UNIX causes the system to send certain signals:

* Ctrl+C sends an INT signal (SIGINT); by default, this causes the process to terminate.
* Ctrl+Z sends a TSTP signal (SIGTSTP); by default, this causes the process to suspend execution.
* Ctrl+\ sends a QUIT signal (SIGQUIT); by default, this causes the process to terminate and dump core.

                There are more than 30 signals which are generated based on certain situation arise. Here is small example of SIGINT handler to clarify things more.

/\* signal example \*/

#include <stdio.h>

#include <signal.h> /\* signal, raise, sig\_atomic\_t \*/

void my\_handler (int param)

{

printf (" Signal raised by pressing Ctrl+C.\n");

}

int main ()

{

void (\*prev\_handler)(int);

prev\_handler = signal (SIGINT, my\_handler);

while(1)

{

//Do some stuff, Wait to signal arise

}

return 0;

}

# Finite State Machines(FSM)

Finite State Machinesm or simply state machine where the elements of (multi-dimensional) arrays indicate the routine that processes/handles the next state.

               Enabling features and disabling of features can be done using function pointers. You may have features that you wish to enable or disable that do similar yet distinct things. Instead of populating and cluttering your code with if-else or switch constructs testing variables, you can code it so that it uses a function pointer, and then you can enable/disable features by changing/assigning the [function pointer](http://www.firmcodes.com/what-are-practical-uses-of-function-pointers-in-c/). If you add new variants, you don’t have to track down all your if-else or switch cases (and risk missing one); instead you just update your function pointer to enable the new feature, or disable the old one.

Reducing code cluster

/\* Finite State Machine FSM \*/

switch(a)

{

case 0:

func0();

break;

case 1:

func1();

break;

case 2:

func2();

break;

case 3:

func3();

break;

default:

funcX();

break;

}

Can be simplified to …

/\* This declaration may be off a little, but I am after the essence of the idea \*/

void (\*funcArray)(void)[] = {func0, func1, func2, func3, funcX};

... appropriate bounds checking on 'a' ...

funcArray[a]();

we have seen that functions have addresses, so we can have pointers that can contairi these addresses and hence point to them.

The syntax for declaration of a pointer to a function is as-

return type (\*ptr\_name )(type1, type2,..............................);

**For Example-**

float (\*fp)( int,int );

char (\*func\_p)(float, char);

Now let us see how to invoke a function using a function pointer.

r = func( a, b); I\*Calling function in usual way\*1

r = (\*fp)(a, b); I\*Calling function via function pointer \*1

The effect and result of calling a function by its name or by a function pointer is exactly the same.

**Structure:**

* A structure that contains pointers to structures of its own type is known as self referential structure.
* The list is a collection of elements.
* The way of implementing a list in memory is by using a self referential structure. These types of lists are known as linked lists.