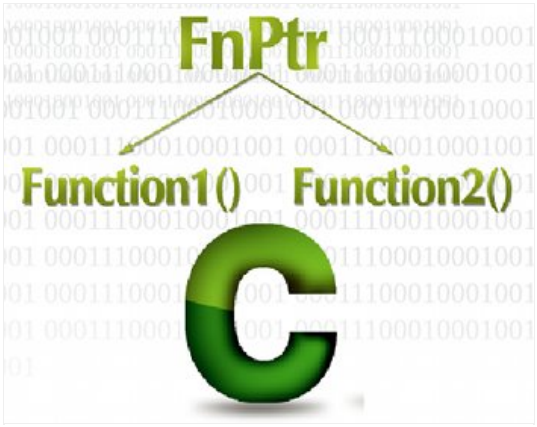
Function Pointers and Callbacks in C — An Odyssey

By Dibyendu Roy on February 28, 2012 in Coding, Developers · 3 Comments

Function pointers are among the most powerful tools in C, but are a bit of a pain during the initial stages of learning. This article demonstrates the basics of function pointers, and how to use them to implement function callbacks in C. C++ takes a slightly different route for callbacks, which is another journey altogether.



A pointer is a special kind of variable that holds the address of another variable. The same concept applies to function pointers, except that instead of pointing to variables, they point to functions. If you declare an array, say, `int a[10];` then the array name `a` will in most contexts (in an expression or passed as a function parameter) "decay" to a non-modifiable pointer to its first element (even though pointers and arrays are not equivalent while declaring/defining them, or when used as operands of the `sizeof` operator). In the same way, for `int func();`, `func` decays to a non-modifiable pointer to a function. You can think of `func` as a `const` pointer for the time being.

But can we declare a non-constant pointer to a function? Yes, we can — just like we declare a non-constant pointer to a variable:

```
int (*ptrFunc) ();
```

Here, `ptrFunc` is a pointer to a function that takes no arguments and returns an integer. DO NOT forget to put in the parenthesis, otherwise the compiler will assume that `ptrFunc` is a normal function name, which takes nothing and returns a pointer to an integer.

Let's try some code. Check out the following simple program:

```
1  #include<stdio.h>
2
3  /* function prototype */
4  int func(int, int);
5  int main(void)
6  {
7      int result;
8      /* calling a function named func */
9      result = func(10,20);
10     printf("result = %d\n",result);
11     return 0;
12 }
13
14 /* func definition goes here */
15 int func(int x, int y)
16 {
17 return x+y;
18 }
```

Search for: Search

Get Connected

RSS Feed Twitter

Facebook

As expected, when we compile it with `gcc -g -o example1 example1.c` and invoke it with `./example1`, the output is as follows:

```
result = 30
```

The above program calls `func()` the simple way. Let's modify the program to call using a pointer to a function. Here's the changed `main()` function:

```
#include<stdio.h>
int func(int, int);
int main(void)
{
    int result1,result2;
    /* declaring a pointer to a function which takes
       two int arguments and returns an integer as result */
    int (*ptrFunc)(int,int);

    /* assigning ptrFunc to func's address */
    ptrFunc=func;

    /* calling func() through explicit dereference */
    result1 = (*ptrFunc)(10,20);

    /* calling func() through implicit dereference */
    result2 = ptrFunc(10,20);
    printf("result1 = %d result2 = %d\n",result1,result2);
    return 0;
}

int func(int x, int y)
{
    return x+y;
}
```

The output has no surprises:

```
result1 = 30 result2 = 30
```

A simple callback function

At this stage, we have enough knowledge to deal with function callbacks. According to Wikipedia, "In computer programming, a callback is a reference to executable code, or a piece of executable code, that is passed as an argument to other code. This allows a lower-level software layer to call a subroutine (or function) defined in a higher-level layer."

Let's try one simple program to demonstrate this. The complete program has three files: `callback.c`, `reg_callback.h` and `reg_callback.c`.

```
/* callback.c */
#include<stdio.h>
#include"reg_callback.h"

/* callback function definition goes here */
void my_callback(void)
{
    printf("inside my_callback\n");
}

int main(void)
{
    /* initialize function pointer to
    my_callback */
    callback ptr_my_callback=my_callback;
    printf("This is a program demonstrating function callback\n");
    /* register our callback function */
    register_callback(ptr_my_callback);
    printf("back inside main program\n");
    return 0;
}
```

```
/* reg_callback.h */
typedef void (*callback)(void);
void register_callback(callback ptr_reg_callback);
```

```
/* reg_callback.c */
#include<stdio.h>
#include"reg_callback.h"

/* registration goes here */
void register_callback(callback ptr_reg_callback)
{
    printf("inside register_callback\n");
    /* calling our callback function my_callback */
    (*ptr_reg_callback)();
}
```

Compile, link and run the program with `gcc -Wall -o callback callback.c reg_callback.c` and

`./callback`:

```
This is a program demonstrating function callback
inside register_callback
inside my_callback
back inside main program
```

The code needs a little explanation. Assume that we have to call a callback function that does some useful work (error handling, last-minute clean-up before exiting, etc.), after an event occurs in another part of the program. The first step is to register the callback function, which is just passing a function pointer as an argument to some other function (e.g., `register_callback`) where the callback function needs to be called.

We could have written the above code in a single file, but have put the definition of the callback function in a separate file to simulate real-life cases, where the callback function is in the top layer and the function that will invoke it is in a different file layer. So the program flow is like what can be seen in Figure 1.

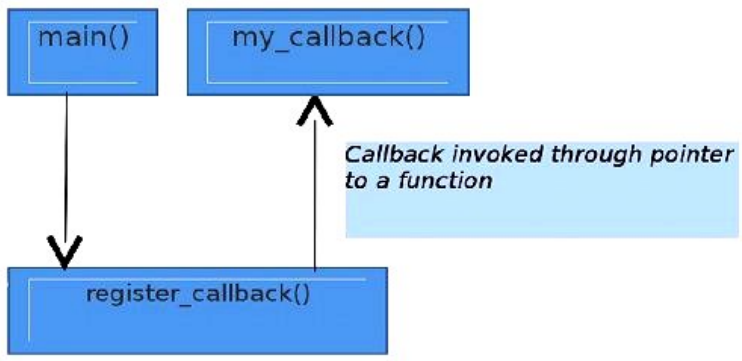


Figure 1: Program flow

The higher layer function calls a lower layer function as a normal call and the callback mechanism allows the lower layer function to call the higher layer function through a pointer to a callback function.

This is exactly what the Wikipedia definition states.

Use of callback functions

One use of callback mechanisms can be seen here:

```
1  / * This code catches the alarm signal generated from the kernel
2      Asynchronously */
3  #include <stdio.h>
4  #include <signal.h>
5  #include <unistd.h>
6
7  struct sigaction act;
8
9  /* signal handler definition goes here */
10 void sig_handler(int signo, siginfo_t *si, void *ucontext)
11 {
12    printf("Got alarm signal %d\n",signo);
13    /* do the required stuff here */
14 }
15
16 int main(void)
17 {
18     act.sa_sigaction = sig_handler;
19     act.sa_flags = SA_SIGINFO;
20
21     /* register signal handler */
22     sigaction(SIGALRM, &act, NULL);
23     /* set the alarm for 10 sec */
24     alarm(10);
25     /* wait for any signal from kernel */
26     pause();
27     /* after signal handler execution */
28     printf("back to main\n");
29     return 0;
30 }
```

Signals are types of interrupts that are generated from the kernel, and are very useful for handling asynchronous events. A signal-handling function is registered with the kernel, and can

be invoked asynchronously from the rest of the program when the signal is delivered to the user process. Figure 2 represents this flow.

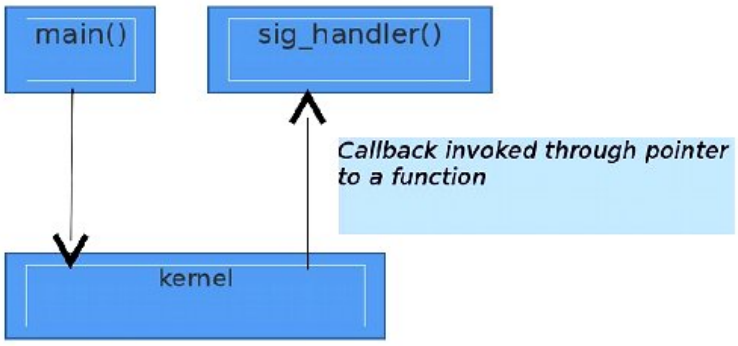


Figure 2: Kernel callback

Callback functions can also be used to create a library that will be called from an upper-layer program, and in turn, the library will call user-defined code on the occurrence of some event. The following source code (`insertion_main.c`, `insertion_sort.c` and `insertion_sort.h`), shows this mechanism used to implement a trivial insertion sort library. The flexibility lets users call any comparison function they want.

```
1  /* insertion_sort.h */
2
3  typedef int (*callback)(int, int);
4  void insertion_sort(int *array, int n, callback comparison);
```

```
1  /* insertion_main.c */
2
3  #include<stdio.h>
4  #include<stdlib.h>
5  #include"insertion_sort.h"
6
7  int ascending(int a, int b)
8  {
9      return a > b;
10 }
11
12 int descending(int a, int b)
13 {
14     return a < b;
15 }
16
17 int even_first(int a, int b)
18 {
19     /* code goes here */
20 }
21
22 int odd_first(int a, int b)
23 {
24     /* code goes here */
25 }
26
27 int main(void)
28 {
29     int i;
30     int choice;
31     int array[10] = {22,66,55,11,99,33,44,77,88,0};
32     printf("ascending 1: descending 2: even_first 3: odd_first 4: quit 5\n");
33     printf("enter your choice = ");
34     scanf("%d",&choice);
35     switch(choice)
36     {
37         case 1:
38             insertion_sort(array,10, ascending);
39             break;
40         case 2:
41             insertion_sort(array,10, descending);
42          case 3:
43             insertion_sort(array,10, even_first);
44             break;
45         case 4:
46             insertion_sort(array,10, odd_first);
47             break;
48         case 5:
49             exit(0);
50         default:
51             printf("no such option\n");
52     }
53
54     printf("after insertion_sort\n");
55     for(i=0;i<10;i++)
56         printf("%d\t", array[i]);
57     printf("\n");
58      return 0;
```

```
}
```

```
/* insertion_sort.c */

#include"insertion_sort.h"

void insertion_sort(int *array, int n, callback comparison)
{
    int i, j, key;
    for(j=1; j<=n-1;j++)
    {
        key=array[j];
        i=j-1;
        while(i >=0 && comparison(array[i], key))
        {
            array[i+1]=array[i];
            i=i-1;
        }
        array[i+1]=key;
    }
}
```

Wrapping it up

The tutorial describes how callbacks are implemented in C. In C++ the same goal can be achieved through template functors. Although callbacks are very hard to trace in large real-world programs, life should be a little easier after going through this introductory article. Just remember, callback is nothing but passing the function pointer to the code from where you want your handler/callback function to be invoked. Happy hacking!

Reference

- *Introduction to Algorithms*, 3/e by Thomas H Cormen, Charles E Leiserson, Ronald L Rivest and Clifford Stein
- *Linux System Programming* by Robert Love
- Function Pointers in C++

Related Posts:

- Loading Library Files in C++
- Analyse Your Network Packets with LibPCAP
- CodeSport
- A Peek Into GNU debugger GDB
- GDB Logging Function Parameters, Part 2

Tags: C, C++, C/C++, callbacks, const pointer, function callbacks, function pointers, LFY February 2012, pointers, pointers and arrays

Article written by:

Dibyendu Roy

The author is a Linux fundamentalist. He works as a Linux developer with a large organisation, and spends most of his time studying and exploring the uncharted spaces of C and Linux. He dreams of becoming a guitar player some day, and spends precious moments with LP, Cradle Of Filth and Children Of Bodom. Being an open source activist, he uses Linux for every silly little thing.

Connect with him: **Website**

Previous Post
Device Drivers, Part 15: Disk on RAM -- Playing with Block Drivers

Next Post
UEFI: Should Linux Users be Worried?