Basics of Function Pointers in C

This post is very detailed because I am attempting to create a mental model to help beginners understand the syntax and basics of function pointers. If you are ok with detail happy reading.

Function pointers are an interesting and powerful tool but their syntax can be a little confusing. This post will going into C function pointers from the basics to simple usage to some quirks about function names and addresses. In the end it will give you an easy way to think about function pointers so their usage is more clear.

A Simple Function and Function Pointer

Let's start with a very simple function to print out the message hello world and see how we can create a function pointer from there.

```
1  #include <stdio.h>;
2
3  // function prototype
4  void sayHello();
5
6  // function implementation
7  void sayHello() {
8    printf("hello world");
9  }
10
11 // calling from main
12 int main() {
13   sayHello();
14 }
```

Here we have a function called sayHello along with its function prototype. This function returns nothing (void) and doesn't take any parameters. We call the function from main and it prints out "hello world". Pretty simple. Now lets convert main to use a function pointer instead of calling the function directly.

```
1 int main() {
2   void (*sayHelloPtr)() = sayHello;
3   (*sayHelloPtr)();
4 }
```

The syntax `void (*sayHelloPtr)()` on line 2 may look a little weird so let's step by step it.

1. We are creating a function pointer to a function that returns nothing (void) so the return type is `void` . That is the void keyword.
2. We have the pointer name `sayHelloPtr` . This is similar to creating any other pointer it has to have a name.
3. We use the `*` notation to signify that it is a pointer. This is no different than declaring an int pointer or a char pointer.
4. We must have parentheses around the pointer `(*sayHelloPrt)` . If we don't have parentheses it is seen as `void *sayHelloPtr` which is a void pointer instead of a pointer to a void function. This is a key point, function pointers must have parentheses around them.
5. We have the parameter list which in this case there isn't one so it is just empty parentheses `(*sayHelloPrt)()` .
6. Putting it all together we get `void (*sayHelloPtr)()` , a pointer to a function that returns void and takes no parameters.

On line 2 above we are assigning the sayHello function name to our newly created function pointer like this `void (*sayHelloPtr)() = sayHello` . We will go into more detail about function names later, but for now understand that a function name (label) is the address of the function and it can be assigned to a function pointer. This is similar to `int *x = &myint` where we assign the address of myint to an int pointer. Only in the case of a function, the address-of the function is the function name and we dont' need the address-of operator. Simply put, the function name is the address-of the function. On line 3 we dereference and call our function pointer like this `(*sayHelloPtr)()` .

1. Once created on line 2 sayHelloPtr is our function pointer name and can be treated just like any other pointer, assigned, stored.

2. We dereference our sayHelloPtr pointer the same as we dereference any other pointer, by using the value-at-address (*) operator. This gives us `*sayHelloPtr` .
3. Again we must have parentheses around the pointer `(*sayHelloPrt)` . If we don't it isn't a function pointer. We must have parentheses when creating a function pointer and when dereferencing it.
4. The () operator is used to call a function in C. It is no different on a function pointer. If we had a parameter list there would be values in the parentheses similar to any other function call. This gives us `(*sayHelloPrt)()` .
5. This function has no return value so there is no need to assign its return to any variable. The function call can standalone similar to `sayHello()` .

Now that we have show the weird syntax understand that often function pointers are just treated and called as regular functions after being assigned. To modify our previous example.

```
1 int main() {
2   void (*sayHelloPtr)() = sayHello;
3   sayHelloPtr();
4 }
```

As before we assign the `sayHello` function to our function pointer, but now we call the function pointer just like we would call a regular function. We will get into function names later which will show why this works but for now understand that calling a function pointer with full syntax `(*sayHelloPtr)()` is the same as calling the function pointer as a regular function `sayHelloPtr()` .

A Function Pointer with Parameters

Now lets create a function pointer that still doesn't return anything (void) but now has parameters.

```
1 #include <stdio.h>
2
```

```
3  // function prototype
4  void subtractAndPrint(int x, int y);
5
6  // function implementation
7  void subtractAndPrint(int x, int y) {
8    int z = x - y;
9    printf("Simon says, the answer is: %d\n", z);
10 }
11
12 // calling from main
13 int main() {
14   void (*sapPtr)(int, int) = subtractAndPrint;
15   (*sapPtr)(10, 2);
16   sapPtr(10, 2);
17 }
```

As before we have our function prototype, our function implementation and the executing of the fuction from main using a function pointer. The signature of both the prototype and its implementation have changed. Where before our sayHello function didn't have parameters, the subtractAndPrint function takes two parameters, both integers, subtracts one from the other and prints the result.

1. We create our sapPtr function pointer on line 14 with `void (*sapPtr)(int, int)`. The only difference from before is now instead of empty parentheses on the end when creating the function we have `(int, int)` which matches the signature of our new function.
2. On line 15 when dereferecing and executing the function, everything is the same as when we called our sayHello function except now we have `(10, 2)` on the end passing parameters.
3. On line 16 we show executing the function pointer as a regular function.

A Function Pointer with Parameters and Return Value

Let's change our subtractAndPrint function to be called subtract and to return the result instead of printing it.

```
1  #include <stdio.h>
2
3  // function prototype
4  int subtract(int x, int y);
```

```
5
6  // function implementation
7  int subtract(int x, int y) {
8    return x - y;
9  }
10
11 // calling from main
12 int main() {
13   int (*subtractPtr)(int, int) = subtract;
14
15   int y = (*subtractPtr)(10, 2);
16   printf("Subtract gives: %d\n", y);
17
18   int z = subtractPtr(10, 2);
19   printf("Subtract gives: %d\n", z);
20 }
```

Similar to the subtractAndPrint function except now the subtract function returns an int. The prototype and function signatures have changed as would be expected.

1. We create our subtractPtr function pointer on line 13 with `int (*subtractPtr)(int, int)`. The only difference from before is instead of void we have an int return value. This matches our subtract method signature.
2. On line 15 when dereferencing and executing the function pointer, everything is the same as when we called our subtractAndPrint function except now we have `int y =` which assigns the return value of the function to y.
3. On line 16 we print out the return value.
4. On lines 18-19 we execute the function pointer as a regular function and print the results.

Not much difference from before, we just added the int return value. Let's move on to a little more complex example where we pass a function pointer into another function as a parameter.

Passing a Function Pointer as a Parameter

We have stepped through the main parts of the declaring and executing function pointers with and without parameters and return values. Now lets look at using a function pointer to execute different functions based on input.

```
1  #include <stdio.h>
2
3  // function prototypes
4  int add(int x, int y);
5  int subtract(int x, int y);
6  int domath(int (*mathop)(int, int), int x, int y);
7
8  // add x + y
9  int add(int x, int y) {
10   return x + y;
11 }
12
13 // subtract x - y
14 int subtract(int x, int y) {
15   return x - y;
16 }
17
18 // run the function pointer with inputs
19 int domath(int (*mathop)(int, int), int x, int y) {
20   return (*mathop)(x, y);
21 }
22
23 // calling from main
24 int main() {
25
26   // call math function with add
27   int a = domath(add, 10, 2);
28   printf("Add gives: %d\n", a);
29
30   // call math function with subtract
31   int b = domath(subtract, 10, 2);
32   printf("Subtract gives: %d\n", b);
33 }
```

Let's break this down.

1. We have two functions with the same signature `int function(int, int)` , add and subtract. Both return an integer and both take two integers as parameters.
2. On line 6 we have `int domath(int (*mathop)(int, int), int x, int y)` The first parameter `int (*mathop)(int, int)` is a pointer to a function that takes two integers as input and returns an integer. We have seen this before, the syntax is no different here. The last two parameters x and y are just integer inputs into the domath function. So the domath function is takes a function pointer and two integers as parameters.
3. On lines 19-21 the domath function executes the function pointer passed with the x and y integers passed. This could also have been done as `mathop(x, y);` .

4. Lines 27 and 31 are somewhat new. We are calling the domath function and we are passing in the function names. Function names are the address-of the function and can be used in place of function pointers.

The main function calls domath twice, once for add and once for subtract, printing out the results.

Function Names and Addresses

Let's wrap up by talking a bit about function names and addresses as promised. A function name (label) is converted to a pointer to itself. This means that function names can be used where function pointers are required as input. It also leads to some very funky looking code that actually works. Take a look at some examples.

```
1  #include <stdio.h>
2
3  // function prototypes
4  void add(char *name, int x, int y);
5
6  // add x + y
7  void add(char *name, int x, int y) {
8    printf("%s gives: %d\n", name, x + y);
9  }
10
11 // calling from main
12 int main() {
13
14   // some funky function pointer assignment
15   void (*add1Ptr)(char*, int, int) = add;
16   void (*add2Ptr)(char*, int, int) = *add;
17   void (*add3Ptr)(char*, int, int) = &add;
18   void (*add4Ptr)(char*, int, int) = **add;
19   void (*add5Ptr)(char*, int, int) = ***add;
20
21   // execution still works
22   (*add1Ptr)("add1Ptr", 10, 2);
23   (*add2Ptr)("add2Ptr", 10, 2);
24   (*add3Ptr)("add3Ptr", 10, 2);
25   (*add4Ptr)("add4Ptr", 10, 2);
26   (*add5Ptr)("add5Ptr", 10, 2);
27
28   // this works too
29   add1Ptr("add1PtrFunc", 10, 2);
30   add2Ptr("add2PtrFunc", 10, 2);
31   add3Ptr("add3PtrFunc", 10, 2);
32   add4Ptr("add4PtrFunc", 10, 2);
33   add5Ptr("add5PtrFunc", 10, 2);
34 }
```

Run this code and every function pointer will execute. Yes you will get some warnings about char conversion, this is a simple example. But the function pointers still work.

1. Line 15, the function name `add` by itself gives the address of the function. It is implicitly converted to a function pointer. Function names can be used where function pointers are required as input.
2. Line 16, the value-at-address operator `*add` when applied to the function name gives the function at that address, which is converted to a function pointer implicitly just like the function name.
3. Line 17, address-of (&) operators when applied to a function name gives the address of the function. This yields a function pointer too.
4. Lines 18 and 19, the pointers to the function keep yielding themselves over and over again returning the function address which is converted to a function pointer. In the end, same as just the function name.

This code isn't an example of best practice. The takeaway is this. One, function names are converted to function pointers implicitly the same way that array names are converted to pointers implicitly when passed into functions. Function names can be used wherever a function pointer is required. Two, the address-of (&) and value-at-address (*) operators are almost always redundant when used against function names.

Conclusion

I hope this helps clarify some things about function pointers and their usage. When understood, function pointers become a powerful tool in the C toolbox. In future posts I may go into more detailed usage of function pointers for things like callbacks and basic OOP in C.

Update 1: I removed the part about `(*sayHelloPrt)(void)` being the same as `(*sayHelloPrt)()` because that is not correct. There is a very good explanation in the comments by Dave G about this.