

11.7 — Function Pointers

ALEX SEPTEMBER 7, 2021

In lesson [10.8 -- Introduction to pointers](#), you learned that a pointer is a variable that holds the address of another variable. Function pointers are similar, except that instead of pointing to variables, they point to functions!

Consider the following function:

```
1 int foo()
2 {
3     return
4     5;
5 }
```

Identifier `foo` is the function's name. But what type is the function? Functions have their own l-value function type -- in this case, a function type that returns an integer and takes no parameters. Much like variables, functions live at an assigned address in memory.

When a function is called (via the `()` operator), execution jumps to the address of the function being called:

```
1 int foo() // code for foo starts at memory address
2 0x002717f0
3 {
4     return 5;
5 }
6
7 int main()
8 {
9     foo(); // jump to address 0x002717f0
10
11     return 0;
12 }
```

At some point in your programming career (if you haven't already), you'll probably make a simple mistake:

```
1 #include <iostream>
2
3 int foo() // code starts at memory address 0x002717f0
4 {
5     return 5;
6 }
7
8 int main()
9 {
10     std::cout << foo << '\n'; // we meant to call foo(), but instead we're printing foo
11     itself!
12
13     return 0;
14 }
```

Instead of calling function `foo()` and printing the return value, we've unintentionally sent function `foo` directly to `std::cout`. What happens in this case?

On the author's machine, this printed:

```
0x002717f0
```

...but it may print some other value (e.g. 1) on your machine, depending on how your compiler decides to convert the function pointer to another type for printing. If your machine doesn't print the function's address, you may be able to force it to do so by converting the function to a void pointer and printing that:

```
1 | #include <iostream>
2 |
3 | int foo() // code starts at memory address 0x002717f0
4 | {
5 |     return 5;
6 | }
7 | int main()
8 | {
9 |     std::cout << reinterpret_cast<void*>(foo) << '\n'; // Tell C++ to interpret function foo as a void
10 | pointer
    |
    |     return 0;
    | }
```

Just like it is possible to declare a non-constant pointer to a normal variable, it's also possible to declare a non-constant pointer to a function. In the rest of this lesson, we'll examine these function pointers and their uses. Function pointers are a fairly advanced topic, and the rest of this lesson can be safely skipped or skimmed by those only looking for C++ basics.

Pointers to functions

The syntax for creating a non-const function pointer is one of the ugliest things you will ever see in C++:

```
1 | // fcnPtr is a pointer to a function that takes no arguments and returns an
   | integer
   | int (*fcnPtr)();
```

In the above snippet, fcnPtr is a pointer to a function that has no parameters and returns an integer. fcnPtr can point to any function that matches this type.

The parenthesis around *fcnPtr are necessary for precedence reasons, as `int *fcnPtr()` would be interpreted as a forward declaration for a function named fcnPtr that takes no parameters and returns a pointer to an integer.

To make a const function pointer, the const goes after the asterisk:

```
1 | int (*const
   | fcnPtr)();
```

If you put the const before the int, then that would indicate the function being pointed to would return a const int.

Assigning a function to a function pointer

Function pointers can be initialized with a function (and non-const function pointers can be assigned a function). In the above example, we have used foo directly, and it has been converted to a function pointer. Like with pointers to variables, we can also use &foo to get a function pointer to foo.

```

1  int foo()
2  {
3      return 5;
4  }
5  int goo()
6  {
7      return 6;
8  }
9  int main()
10 {
11     int (*fcnPtr)() { &foo }; // fcnPtr points to function
12     foo
13     fcnPtr = &goo; // fcnPtr now points to function goo
14
15     return 0;
16 }

```

One common mistake is to do this:

```

1  fcnPtr =
   goo();

```

This would actually assign the return value from a call to function `goo()` to `fcnPtr`, which isn't what we want. We want `fcnPtr` to be assigned the address of function `goo`, not the return value from function `goo()`. So no parenthesis are needed.

Note that the type (parameters and return type) of the function pointer must match the type of the function. Here are some examples of this:

```

1  // function prototypes
   int foo();
2  double goo();
3  int hoo(int x);
4
5  // function pointer assignments
6  int (*fcnPtr1)() { &foo }; // okay
   int (*fcnPtr2)() { &goo }; // wrong -- return types don't match!
   double (*fcnPtr4)() { &goo }; // okay
7  fcnPtr1 = &hoo; // wrong -- fcnPtr1 has no parameters, but hoo()
   does
   int (*fcnPtr3)(int) { &hoo }; // okay

```

Unlike fundamental types, C++ *will* implicitly convert a function into a function pointer if needed (so you don't need to use the address-of operator (&) to get the function's address). However, it will not implicitly convert function pointers to void pointers, or vice-versa.

Function pointers can also be initialized or assigned the value `nullptr`:

```

1  int (*fcnptr)() { nullptr }; //
   okay

```

Calling a function using a function pointer

The other primary thing you can do with a function pointer is use it to actually call the function. There are two ways to do this. The first is via explicit dereference:

```

1 | int foo(int x)
2 | {
3 |     return x;
4 | }
5 |
6 | int main()
7 | {
8 |     int (*fcnPtr)(int){ &foo }; // Initialize fcnPtr with function
   |     foo
   |     (*fcnPtr)(5); // call function foo(5) through fcnPtr.
   |
   |     return 0;
   | }

```

The second way is via implicit dereference:

```

1 | int foo(int x)
2 | {
3 |     return x;
4 | }
5 |
6 | int main()
7 | {
8 |     int (*fcnPtr)(int){ &foo }; // Initialize fcnPtr with function
   |     foo
   |     fcnPtr(5); // call function foo(5) through fcnPtr.
   |
   |     return 0;
   | }

```

As you can see, the implicit dereference method looks just like a normal function call -- which is what you'd expect, since normal function names are pointers to functions anyway! However, some older compilers do not support the implicit dereference method, but all modern compilers should.

One interesting note: Default parameters won't work for functions called through function pointers. Default parameters are resolved at compile-time (that is, if you don't supply an argument for a defaulted parameter, the compiler substitutes one in for you when the code is compiled). However, function pointers are resolved at run-time. Consequently, default parameters can not be resolved when making a function call with a function pointer. You'll explicitly have to pass in values for any defaulted parameters in this case.

Also note that because function pointers can be set to nullptr, it's a good idea to assert or conditionally test whether your function pointer is a null pointer before calling it. Just like with normal pointers, dereferencing a null function pointer lead to undefined behavior.

```

1 | int foo(int x)
2 | {
3 |     return x;
4 | }
5 |
6 | int main()
7 | {
8 |     int (*fcnPtr)(int){ &foo }; // Initialize fcnPtr with function
   |     foo
   |     if (fcnPtr) // make sure fcnPtr isn't a null pointer
   |         fcnPtr(5); // otherwise this will lead to undefined behavior
   |
   |     return 0;
   | }

```

One of the most useful things to do with function pointers is pass a function as an argument to another function. Functions used as arguments to another function are sometimes called **callback functions**.

Consider a case where you are writing a function to perform a task (such as sorting an array), but you want the user to be able to define how a particular part of that task will be performed (such as whether the array is sorted in ascending or descending order). Let's take a closer look at this problem as applied specifically to sorting, as an example that can be generalized to other similar problems.

Many comparison-based sorting algorithms work on a similar concept: the sorting algorithm iterates through a list of numbers, does comparisons on pairs of numbers, and reorders the numbers based on the results of those comparisons. Consequently, by varying the comparison, we can change the way the algorithm sorts without affecting the rest of the sorting code.

Here is our selection sort routine from a previous lesson:

```
1 | #include <utility> // for std::swap
2 | void SelectionSort(int *array, int size)
3 | {
4 |     // Step through each element of the array
5 |     for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
6 |     {
7 |         // smallestIndex is the index of the smallest element we've encountered so
8 |         far.
9 |         int smallestIndex{ startIndex };
10 |
11 |         // Look for smallest element remaining in the array (starting at
12 |         startIndex+1)
13 |         for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++currentIndex)
14 |         {
15 |             // If the current element is smaller than our previously found smallest
16 |             if (array[smallestIndex] > array[currentIndex]) // COMPARISON DONE HERE
17 |             {
18 |                 // This is the new smallest number for this iteration
19 |                 smallestIndex = currentIndex;
20 |             }
21 |         }
22 |
23 |         // Swap our start element with our smallest element
24 |         std::swap(array[startIndex], array[smallestIndex]);
25 |     }
26 | }
```

Let's replace that comparison with a function to do the comparison. Because our comparison function is going to compare two integers and return a boolean value to indicate whether the elements should be swapped, it will look something like this:

```
1 | bool ascending(int x, int y)
2 | {
3 |     return x > y; // swap if the first element is greater than the
4 |     second
5 | }
```

And here's our selection sort routine using the ascending() function to do the comparison:

```

1 #include <utility> // for std::swap
2 void SelectionSort(int *array, int size)
3 {
4     // Step through each element of the array
5     for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
6     {
7         // smallestIndex is the index of the smallest element we've encountered so far.
8         int smallestIndex{ startIndex };
9
10        // Look for smallest element remaining in the array (starting at startIndex+1)
11        for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++currentIndex)
12        {
13            // If the current element is smaller than our previously found smallest
14            if (ascending(array[smallestIndex], array[currentIndex])) // COMPARISON DONE
15            {
16                // This is the new smallest number for this iteration
17                smallestIndex = currentIndex;
18            }
19        }
20
21        // Swap our start element with our smallest element
22        std::swap(array[startIndex], array[smallestIndex]);
23    }
24 }

```

Now, in order to let the caller decide how the sorting will be done, instead of using our own hard-coded comparison function, we'll allow the caller to provide their own sorting function! This is done via a function pointer.

Because the caller's comparison function is going to compare two integers and return a boolean value, a pointer to such a function would look something like this:

```
1 | bool (*comparisonFcn)(int,  
  | int);
```

So, we'll allow the caller to pass our sort routine a pointer to their desired comparison function as the third parameter, and then we'll use the caller's function to do the comparison.

Here's a full example of a selection sort that uses a function pointer parameter to do a user-defined comparison, along with an example of how to call it:

```

1  #include <utility> // for std::swap
   #include <iostream>
2
3  // Note our user-defined comparison is the third parameter
4  void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int))
   {
5      // Step through each element of the array
       for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
       {
6          // bestIndex is the index of the smallest/largest element we've encountered so far.
           int bestIndex{ startIndex };
7
8          // Look for smallest/largest element remaining in the array (starting at
           startIndex+1)
           for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++currentIndex)
           {
9              // If the current element is smaller/larger than our previously found smallest
10             if (comparisonFcn(array[bestIndex], array[currentIndex])) // COMPARISON DONE
               HERE
               {
11                 // This is the new smallest/largest number for this iteration
                     bestIndex = currentIndex;
12             }
13
14             // Swap our start element with our smallest/largest element
               std::swap(array[startIndex], array[bestIndex]);
           }
       }
15
16     // Here is a comparison function that sorts in ascending order
       // (Note: it's exactly the same as the previous ascending() function)
       bool ascending(int x, int y)
       {
17         return x > y; // swap if the first element is greater than the second
       }
18
19     // Here is a comparison function that sorts in descending order
       bool descending(int x, int y)
       {
20         return x < y; // swap if the second element is greater than the first
       }
21
22     // This function prints out the values in the array
       void printArray(int *array, int size)
       {
23         for (int index{ 0 }; index < size; ++index)
24         {
25             std::cout << array[index] << ' ';
26         }
27         std::cout << '\n';
28     }
29
30     int main()
       {
31         int array[9]{ 3, 7, 9, 5, 6, 1, 8, 2, 4 };
32
33         // Sort the array in descending order using the descending() function
           selectionSort(array, 9, descending);
           printArray(array, 9);
34
35         // Sort the array in ascending order using the ascending() function
           selectionSort(array, 9, ascending);
           printArray(array, 9);
36
37         return 0;
       }
38

```

This program produces the result:

```

9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9

```

Is that cool or what? We've given the caller the ability to control how our selection sort does its job.

The caller can even define their own "strange" comparison functions:


```

1 | bool evensFirst(int x, int y)
  | {
2 |     // if x is even and y is odd, x goes first (no swap
3 |     needed)
  |     if ((x % 2 == 0) && !(y % 2 == 0))
  |         return false;
4 |
  |     // if x is odd and y is even, y goes first (swap needed)
  |     if (!(x % 2 == 0) && (y % 2 == 0))
  |         return true;
5 |
6 |     // otherwise sort in ascending order
  |     return ascending(x, y);
7 | }
8 |
9 | int main()
  | {
10 |     int array[9]{ 3, 7, 9, 5, 6, 1, 8, 2, 4 };
11 |
12 |     selectionSort(array, 9, evensFirst);
13 |     printArray(array, 9);
  |
  |     return 0;
  | }

```

The above snippet produces the following result:

```
2 4 6 8 1 3 5 7 9
```

As you can see, using a function pointer in this context provides a nice way to allow a caller to “hook” their own functionality into something you’ve previously written and tested, which helps facilitate code reuse! Previously, if you wanted to sort one array in descending order and another in ascending order, you’d need multiple versions of the sort routine. Now you can have one version that can sort any way the caller desires!

Note: If a function parameter is of a function type, it will be converted to a pointer to the function type. This means

```

1 | void selectionSort(int *array, int size, bool (*comparisonFcn)(int,
  | int))

```

can be equivalently written as:

```

1 | void selectionSort(int *array, int size, bool comparisonFcn(int,
  | int))

```

This only works for function parameters, not stand-alone function pointers, and so is of somewhat limited use.

Providing default functions

If you’re going to allow the caller to pass in a function as a parameter, it can often be useful to provide some standard functions for the caller to use for their convenience. For example, in the selection sort example above, providing the ascending() and descending() function along with the selectionSort() function would make the caller’s life easier, as they wouldn’t have to rewrite ascending() or descending() every time they want to use them.

You can even set one of these as a default parameter:

```

1 | // Default the sort to ascending sort
  | void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int) =
2 | ascending);

```

In this case, as long as the user calls selectionSort normally (not through a function pointer), the comparisonFcn parameter will default to ascending.

Making function pointers prettier with type aliases

Let’s face it -- the syntax for pointers to functions is ugly. However, type aliases can be used to make pointers to functions look more like regular variables:

```

1 | using ValidateFunction = bool (*)(int,
  | int);

```

This defines a type alias called "ValidateFunction" that is a pointer to a function that takes two ints and returns a bool.

Now instead of doing this:

```
1 | bool validate(int x, int y, bool (*fcnPtr)(int, int)); //  
   | ugly
```

You can do this:

```
1 | bool validate(int x, int y, ValidateFunction pfcn) //  
   | clean
```

Using std::function

An alternate method of defining and storing function pointers is to use std::function, which is part of the standard library <functional> header. To define a function pointer using this method, declare a std::function object like so:

```
1 | #include <functional>  
2 | bool validate(int x, int y, std::function<bool(int, int)> fcn); // std::function method that returns a  
   | bool and takes two int parameters
```

As you see, both the return type and parameters go inside angled brackets, with the parameters inside parenthesis. If there are no parameters, the parentheses can be left empty. Although this reads a little more verbosely, it's also more explicit, as it makes it clear what the return type and parameters expected are (whereas the typedef method obscures them).

Updating our earlier example with std::function:

```
1 | #include <functional>  
   | #include <iostream>  
2 |  
3 | int foo()  
4 | {  
5 |     return 5;  
6 | }  
7 |  
8 | int goo()  
9 | {  
10 |     return 6;  
11 | }  
12 |  
13 | int main()  
14 | {  
15 |     std::function<int> fcnPtr{ &foo }; // declare function pointer that returns an int and takes no  
16 |     parameters  
   |     fcnPtr = &goo; // fcnPtr now points to function goo  
   |     std::cout << fcnPtr() << '\n'; // call the function just like normal  
   |  
   |     return 0;  
   | }
```

Note that you can also type alias std::function:

```
1 | using ValidateFunctionRaw = bool(*)(int, int); // type alias to raw function  
   | pointer  
   | using ValidateFunction = std::function<bool(int, int)>; // type alias to  
   | std::function
```

Type inference for function pointers

Much like the *auto* keyword can be used to infer the type of normal variables, the *auto* keyword can also infer the type of a function pointer.

```

1  #include <iostream>
2  int foo(int x)
3  {
4      return x;
5  }
6  int main()
7  {
8      auto fcnPtr{ &foo };
9      std::cout << fcnPtr(5) <<
10     '\n';
11     return 0;
12 }

```

This works exactly like you'd expect, and the syntax is very clean. The downside is, of course, that all of the details about the function's parameters types and return type are hidden, so it's easier to make a mistake when making a call with the function, or using its return value.

Conclusion

Function pointers are useful primarily when you want to store functions in an array (or other structure), or when you need to pass a function to another function. Because the native syntax to declare function pointers is ugly and error prone, we recommend using `std::function`. In places where a function pointer type is only used once (e.g. a single parameter or return value), `std::function` can be used directly. In places where a function pointer type is used multiple times, a type alias to a `std::function` is a better choice (to prevent repeating yourself).

Quiz time!

1. In this quiz, we're going to write a version of our basic calculator using function pointers.

1a) Create a short program asking the user for two integer inputs and a mathematical operation ('+', '-', '*', '/'). Ensure the user enters a valid operation.

[Show Solution](#)

1b) Write functions named `add()`, `subtract()`, `multiply()`, and `division()`. These should take two integer parameters and return an integer.

[Show Solution](#)

1c) Create a type alias named `ArithmeticFunction` for a pointer to a function that takes two integer parameters and returns an integer. Use `std::function`.

[Show Solution](#)

1d) Write a function named `getArithmeticFunction()` that takes an operator character and returns the appropriate function as a function pointer.

[Show Solution](#)

1e) Modify your `main()` function to call `getArithmeticFunction()`. Call the return value from that function with your inputs and print the result.

[Show Solution](#)

Here's the full program:

[Show Solution](#)



Next lesson

11.8 The stack and the heap



Back to table of
contents



Previous lesson

11.6 Inline functions

Leave a comment... Put C++ code between triple-backticks (markdown style):````Your C++ code here````



Name*



Email*



Avatars from <https://gravatar.com/> are connected to your provided email address.

Notify me about replies:



POST COMMENT

