20.1 — Function Pointers

learncpp.com/cpp-tutorial/function-pointers/

In lesson 12.7 -- 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:

```
int foo()
{
    return 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:

```
int foo() // code for foo starts at memory address 0x002717f0
{
    return 5;
}

int main()
{
    foo(); // jump to address 0x002717f0

    return 0;
}
```

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

```
#include <iostream>

int foo() // code starts at memory address 0x002717f0
{
    return 5;
}

int main()
{
    std::cout << foo << '\n'; // we meant to call foo(), but instead we're printing
foo itself!

    return 0;
}
```

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?

`operator<<` does not know how to output a function pointer (because there are an infinite number of possible function pointers). The standard says that in this case, `foo` should be converted to a `bool` (which `operator<<` does know how to print). And since the function pointer for `foo` is a non-void pointer, it should always evaluate to Boolean `true`. Thus, this should print:

```
1
```

Tip

Some compilers (e.g. Visual Studio) have a compiler extension that prints the address of the function instead:

```
0x002717f0
```

If your platform doesn't print the function's address and you want it to, you may be able to force it to do so by converting the function to a void pointer and printing that:

```
#include <iostream>

int foo() // code starts at memory address 0x002717f0
{
    return 5;
}

int main()
{
    std::cout << reinterpret_cast<void*>(foo) << '\n'; // Tell C++ to interpret
function foo as a void pointer (implementation-defined behavior)

    return 0;
}
```

This is implementation-defined behavior, so it may not work on all platforms.

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++:

```
// 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 parentheses 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:

```
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). Like with pointers to variables, we can also use &foo to get a function pointer to foo.

```
int foo()
{
    return 5;
}

int goo()
{
    return 6;
}

int main()
{
    int (*fcnPtr)(){ &foo }; // fcnPtr points to function foo
    fcnPtr = &goo; // fcnPtr now points to function goo

    return 0;
}
```

One common mistake is to do this:

```
fcnPtr = goo();
```

This tries to assign the return value from a call to function goo() (which has type `int`) to fcnPtr (which is expecting a value of type `int(*)()`), 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 parentheses 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:

```
// function prototypes
int foo();
double goo();
int hoo(int x);

// function pointer initializers
int (*fcnPtr1)(){ &foo };    // okay
int (*fcnPtr2)(){ &goo };    // wrong -- return types don't match!
double (*fcnPtr4)(){ &goo }; // okay
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, function pointers will not convert to void pointers, or vice-versa (though some compilers like Visual Studio may allow this anyway).

```
// function prototypes
int foo();

// function initializations
int (*fcnPtr5)() { foo }; // okay, foo implicitly converts to function pointer to foo
void* vPtr { foo };       // not okay, though some compilers may allow
```

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

```
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:

```
int foo(int x)
{
    return x;
}

int main()
{
    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:

```
int foo(int x)
{
    return x;
}

int main()
{
    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 cannot 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 leads to undefined behavior.

```
int foo(int x)
{
    return x;
}

int main()
{
    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;
}
```

Passing functions as arguments to other functions

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:

```
#include <utility> // for std::swap

void SelectionSort(int* array, int size)
{
    // Step through each element of the array
    for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
    {
        // smallestIndex is the index of the smallest element we've encountered so
far.
        int smallestIndex{ startIndex };

        // Look for smallest element remaining in the array (starting at
startIndex+1)
        for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++currentIndex)
        {
            // If the current element is smaller than our previously found smallest
            if (array[smallestIndex] > array[currentIndex]) // COMPARISON DONE HERE
            {
                // This is the new smallest number for this iteration
                smallestIndex = currentIndex;
            }
        }

        // Swap our start element with our smallest element
        std::swap(array[startIndex], array[smallestIndex]);
    }
}
```

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:

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

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

```
#include <utility> // for std::swap

void SelectionSort(int* array, int size)
{
    // Step through each element of the array
    for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
    {
        // smallestIndex is the index of the smallest element we've encountered so
far.
        int smallestIndex{ startIndex };

        // Look for smallest element remaining in the array (starting at
startIndex+1)
        for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++currentIndex)
        {
            // If the current element is smaller than our previously found smallest
            if (ascending(array[smallestIndex], array[currentIndex])) // COMPARISON
DONE HERE
            {
                // This is the new smallest number for this iteration
                smallestIndex = currentIndex;
            }
        }

        // Swap our start element with our smallest element
        std::swap(array[startIndex], array[smallestIndex]);
    }
}
```

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:

```
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:

```
#include <utility> // for std::swap
#include <iostream>

// Note our user-defined comparison is the third parameter
void selectionSort(int* array, int size, bool (*comparisonFcn)(int, int))
{
    // Step through each element of the array
    for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
    {
        // bestIndex is the index of the smallest/largest element we've encountered
so far.
        int bestIndex{ startIndex };

        // Look for smallest/largest element remaining in the array (starting at
startIndex+1)
        for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++currentIndex)
        {
            // If the current element is smaller/larger than our previously found
smallest
            if (comparisonFcn(array[bestIndex], array[currentIndex])) // COMPARISON
DONE HERE
            {
                // This is the new smallest/largest number for this iteration
                bestIndex = currentIndex;
            }
        }

        // Swap our start element with our smallest/largest element
        std::swap(array[startIndex], array[bestIndex]);
    }
}

// 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)
{
    return x > y; // swap if the first element is greater than the second
}

// Here is a comparison function that sorts in descending order
bool descending(int x, int y)
{
    return x < y; // swap if the second element is greater than the first
}

// This function prints out the values in the array
void printArray(int* array, int size)
{
    for (int index{ 0 }; index < size; ++index)
    {
        std::cout << array[index] << ' ';
    }
    std::cout << '\n';
}

int main()
{
    int array[9]{ 3, 7, 9, 5, 6, 1, 8, 2, 4 };

    // Sort the array in descending order using the descending() function
    selectionSort(array, 9, descending);
    printArray(array, 9);

    // Sort the array in ascending order using the ascending() function
    selectionSort(array, 9, ascending);
    printArray(array, 9);

    return 0;
}
```

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:

```
bool evensFirst(int x, int y)
{
        // if x is even and y is odd, x goes first (no swap needed)
        if ((x % 2 == 0) && !(y % 2 == 0))
                return false;

        // if x is odd and y is even, y goes first (swap needed)
        if (!(x % 2 == 0) && (y % 2 == 0))
                return true;

        // otherwise sort in ascending order
        return ascending(x, y);
}

int main()
{
    int array[9]{ 3, 7, 9, 5, 6, 1, 8, 2, 4 };

    selectionSort(array, 9, evensFirst);
    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:

```
void selectionSort(int* array, int size, bool (*comparisonFcn)(int, int))
```

can be equivalently written as:

```
void selectionSort(int* array, int size, bool comparisonFcn(int, int))
```

This only works for function parameters, and so is of somewhat limited use. On a non-function parameter, the latter is interpreted as a forward declaration:

```
bool (*ptr)(int, int); // definition of function pointer ptr
bool fcn(int, int);    // forward declaration of function fcn
```

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:

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

In this case, as long as the user calls selectionSort normally (not through a function pointer), the comparisonFcn parameter will default to ascending. You will need to make sure that the `ascending` function is declared prior to this point, otherwise the compiler will complain it doesn’t know what `ascending` is.

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:

```
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:

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

You can do this:

```
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:

```
#include <functional>
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 parentheses. If there are no parameters, the parentheses can be left empty.

Updating our earlier example with std::function:

```
#include <functional>
#include <iostream>

int foo()
{
    return 5;
}

int goo()
{
    return 6;
}

int main()
{
    std::function<int()> fcnPtr{ &foo }; // declare function pointer that returns an
int and takes no parameters
    fcnPtr = &goo; // fcnPtr now points to function goo
    std::cout << fcnPtr() << '\n'; // call the function just like normal

    std::function fcnPtr2{ &foo }; // can also use CTAD to infer template arguments

    return 0;
}
```

Type aliasing std::function can be helpful for readability:

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

Also note that std::function only allows calling the function via implicit dereference (e.g. `fcnPtr()`), not explicit dereference (e.g. `(*fcnPtr)()`).

When defining a type alias, we must explicitly specify any template arguments. We can’t use CTAD in this case since there is no initializer to deduce the template arguments from.

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.

```
#include <iostream>

int foo(int x)
{
        return x;
}

int main()
{
        auto fcnPtr{ &foo };
        std::cout << fcnPtr(5) << '\n';

        return 0;
}
```

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 divide(). 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, and include the appropriate header.

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