# Programs as Data: Function Pointers

By Alex Allain

A function pointer is a variable that stores the address of a function that can later be called through that function pointer. This is useful because functions encapsulate behavior. For instance, every time you need a particular behavior such as drawing a line, instead of writing out a bunch of code, all you need to do is call the function. But sometimes you would like to choose different behaviors at different times in essentially the same piece of code. Read on for concrete examples.

## Example Uses of Function Pointers

### Functions as Arguments to Other Functions

If you were to write a sort routine, you might want to allow the function's caller to choose the order in which the data is sorted; some programmers might need to sort the data in ascending order, others might prefer descending order while still others may want something similar to but not quite like one of those choices. One way to let your user specify what to do is to provide a flag as an argument to the function, but this is inflexible; the sort function allows only a fixed set of comparison types (e.g., ascending and descending).

A much nicer way of allowing the user to choose how to sort the data is simply to let the user pass in a function to the sort function. This function might take two pieces of data and perform a comparison on them. We'll look at the syntax for this in a bit.

### Callback Functions

Another use for function pointers is setting up "listener" or "callback" functions that are invoked when a particular event happens. The function is called, and this notifies your code that something of interest has taken place.

Why would you ever write code with callback functions? You often see it when writing code using someone's library. One example is when you're writing code for a graphical user interface (GUI). Most of the time, the user will interact with a loop that allows the mouse pointer to move and that redraws the interface. Sometimes, however, the user will click on a button or enter text into a field. These operations are "events" that may require a response that your program needs to handle. How can your code know what's happening? Using Callback functions! The user's click should cause the interface to call a function that you wrote to handle the event.

To get a sense for when you might do this, consider what might happen if you were using a GUI library that had a "create_button" function. It might take the location where a button should appear on the screen, the text of the button, and a function to call when the button is clicked. Assuming for the moment that C (and C++) had a generic "function pointer" type called function, this might look like this:

```
void create_button( int x, int y, const char *text, function callback_func );
```

Whenever the button is clicked, callback_func will be invoked. Exactly what callback_func does depends on the button; this is why allowing the create_button function to take a function pointer is useful.

## Function Pointer Syntax

The syntax for declaring a function pointer might seem messy at first, but in most cases it's really quite straight-forward once you **understand** what's going on. Let's look at a simple example:

```
void (*foo)(int);
```

In this example, foo is a pointer to a function taking one argument, an integer, and that returns void. It's as if you're declaring a function called "*foo", which takes an int and returns void; now, if *foo is a function, then foo must be a pointer to a function. (Similarly, a declaration like int *x can be read as *x is an int, so x must be a pointer to an int.)

The key to writing the declaration for a function pointer is that you're just writing out the declaration of a function but with (*func_name) where you'd normally just put func_name.

## Reading Function Pointer Declarations

Sometimes people get confused when more stars are thrown in:

```
void *(*foo)(int *);
```

Here, the key is to read inside-out; notice that the innermost element of the expression is *foo, and that otherwise it looks like a normal function declaration. *foo should refer to a function that returns a void * and takes an int *. Consequently, foo is a pointer to just such a function.

## Initializing Function Pointers

To initialize a function pointer, you must give it the address of a function in your program. The syntax is like any other variable:

```
#include <stdio.h>
void my_int_func(int x)
{
    printf( "%d\n", x );
}

int main()
{
    void (*foo)(int);
    /* the ampersand is actually optional */
    foo = &my_int_func;

    return 0;
}
```

(Note: all examples are written to be compatible with both C and C++.)

## Using a Function Pointer

To call the function pointed to by a function pointer, you treat the function pointer as though it were the name of the function you wish to call. The act of calling it performs the dereference; there's no need to do it yourself:

```
#include <stdio.h>
void my_int_func(int x)
{
```

```
    printf( "%d\n", x );
}


int main()
{
    void (*foo)(int);
    foo = &my_int_func;

    /* call my_int_func (note that you do not need to write (*foo)(2) ) */
    foo( 2 );
    /* but if you want to, you may */
    (*foo)( 2 );

    return 0;
}
```

Note that function pointer syntax is flexible; it can either look like most other uses of pointers, with & and *, or you may omit that part of syntax. This is similar to how arrays are treated, where a bare array decays to a pointer, but you may also prefix the array with & to request its address.

# Function Pointers in the Wild

Let's go back to the sorting example where I suggested using a function pointer to write a generic sorting routine where the exact order could be specified by the programmer calling the sorting function. It turns out that the C function qsort does just that.

From the Linux man pages, we have the following declaration for qsort (from stdlib.h):

```
 void qsort(void *base, size_t nmemb, size_t size,
            int(*compar)(const void *, const void *));
```

Note the use of void*s to allow qsort to operate on any kind of data (in C++, you'd normally use templates for this task, but C++ also allows the use of void* pointers) because void* pointers can point to anything. Because we don't know the size of the individual elements in a void* array, we must give qsort the number of elements, nmemb, of the array to be sorted, base, in addition to the standard requirement of giving the length, size, of the input.

But what we're really interested in is the compar argument to qsort: it's a function pointer that takes two void *s and returns an int. This allows anyone to specify how to sort the elements of the array base without having to write a specialized sorting algorithm. Note, also, that compar returns an int; the function pointed to should return -1 if the first argument is less than the second, 0 if they are equal, or 1 if the second is less than the first.

For instance, to sort an array of numbers in ascending order, we could write code like this:

```
#include <stdlib.h>

int int_sorter( const void *first_arg, const void *second_arg )
{
    int first = *(int*)first_arg;
    int second = *(int*)second_arg;
    if ( first < second )
    {
        return -1;
    }
```

```
    else if ( first == second )
    {
        return 0;
    }
    else
    {
        return 1;
    }
}

int main()
{
    int array[10];
    int i;
    /* fill array */
    for ( i = 0; i < 10; ++i )
    {
        array[ i ] = 10 - i;
    }
    qsort( array, 10 , sizeof( int ), int_sorter );
    for ( i = 0; i < 10; ++i )
    {
        printf ( "%d\n" ,array[ i ] );
    }

}
```

## Using Polymorphism and Virtual Functions Instead of Function Pointers (C++)

You can often avoid the need for explicit function pointers by using virtual functions. For instance, you could write a sorting routine that takes a pointer to a class that provides a virtual function called compare:

```
class Sorter
{
    public:
    virtual int compare (const void *first, const void *second);
};

// cpp_qsort, a qsort using C++ features like virtual functions
void cpp_qsort(void *base, size_t nmemb, size_t size, Sorter *compar);
```

inside cpp_qsort, whenever a comparison is needed, compar->compare should be called. For classes that override this virtual function, the sort routine will get the new behavior of that function. For instance:

```
class AscendSorter : public Sorter
{

    virtual int compare (const void*, const void*)
    {
        int first = *(int*)first_arg;
        int second = *(int*)second_arg;
        if ( first < second )
        {
            return -1;
```

```
        }
        else if ( first == second )
        {
            return 0;
        }
        else
        {
            return 1;
        }
    }
};
```

and then you could pass in a pointer to an instance of the AscendSorter to cpp_qsort to sort integers in ascending order.

### But Are You Really Not Using Function Pointers?

Virtual functions are implemented behind the scenes using function pointers, so you really are using function pointers--it just happens that the compiler makes the work easier for you. Using polymorphism can be an appropriate strategy (for instance, it's used by Java), but it does lead to the overhead of having to create an object rather than simply pass in a function pointer.

## Function Pointers Summary

### Syntax

#### Declaring

Declare a function pointer as though you were declaring a function, except with a name like *foo instead of just foo:

```
void (*foo)(int);
```

#### Initializing

You can get the address of a function simply by naming it:

```
void foo();
func_pointer = foo;
```

or by prefixing the name of the function with an ampersand:

```
void foo();
func_pointer = &foo;
```

#### Invoking

Invoke the function pointed to just as if you were calling a function.

```
func_pointer( arg1, arg2 );
```

or you may optionally dereference the function pointer before calling the function it points to:

```
(*func_pointer)( arg1, arg2 );
```

## Benefits of Function Pointers

- Function pointers provide a way of passing around instructions for how to do something
- You can write flexible functions and libraries that allow the programmer to choose behavior by passing function pointers as arguments
- This flexibility can also be achieved by using classes with virtual functions

*I am grateful to Alex Hoffer and Thomas Carriero for their comments on a draft of this article.*

### Related articles

Lambda functions in C++11, a replacement for function pointers

What are Pointers?

Functors in C++, a better function pointer