Function Pointers

Syntax: `Return_Type (*Pointer_Name)(Parameters)`

Function Pointers

Function pointers are a great C++ feature! Some people have a hard time remembering the syntax though, which is why I've put it at the top of this page. If you ever forget it, just come back here.

A function pointer is a pointer that points to functions. Say we want our game to have a menu system. We'll have a function that controls the main menu, functions that control sub-menus (like options and settings), a function that handles the actual game, and a function that displays a message when the player quits the game.

What we need is some way to call the right function depending on what state the game is in. One option is to keep a variable that stores the state of our game and check it each frame. I'll show you some code that would do this and then explain why we would want to do it another way.

If you need a refresher on pointers, now would be a good time to check out my Pointers Tutorial.

Here's some code that handles a menu system without using function pointers:

```
// First we need to enumerate some values to make our code readable.
// These obviously represent our game states.
enum GameState
{
    MENU,
    OPTIONS,
    GAME,
    EXIT,
    REALLYQUIT
    // the Exit() function checks to see that the user
    // really wants to quit and returns REALLYQUIT
};

// I'll just write the prototypes to our games functions.
// Each returns the game state to switch to. If we're in
// one state and don't want to switch, we can just return
// the same state (if we're in Game(), we can return GAME).
GameState Menu();
GameState Options();
```

```
GameState Game();
GameState Exit();

// This variable keeps track of the current game state.
// It is global so the game functions have access to it.
// Since we're starting our game in the main menu, we'll
// set the current state to MENU.
GameState CurrentState = MENU;

int main()
{
    bool playing = true;

    // Now we start a while loop and check which state we're in
    while (playing)
    {
        // We'll use a switch statement to handle the states.
        // Note that changes to the current state will be made
        // within the state functions. If we're in the menu and
        // the player wants to play the game, Menu() will set
        // CurrentState to GAME. All the switch statement does
        // is figure out which state function to call.
        switch (CurrentState)
        {
            case (MENU):
                Menu();
                break;
            case (OPTIONS):
                Options();
                break;
            case (GAME):
                Game();
                break;
            // Exit will return REALLYQUIT if it's time to quit
            case (Exit):
                if (Exit() == REALLYQUIT)
                playing = false;
                break;
        }
    }
    return 0;
}
```

Alright, so in each iteration of the while loop we check to see what state we're in and run the appropriate function.

One argument against this method is that the switch statement has to perform up to four conditional statements for every frame of our game. My main problem with this method, however, is that it's a little ugly. When we want to add more states to our game, perhaps

an option screen for video modes, we not only have to add the logic to each of our menu functions, we also have to add the new state to our GameState enumeration and switch statement (which also increases the number of conditional operations).

It would nice if there was a way we could call one function from our while loop and never have to change it. Function pointers provide this functionality!

Using Function Pointers

A function pointer is exactly what it sounds like, a pointer that points to a function. When we declare a function pointer, we also declare what type of function it will point to (i.e. the return type and parameters). This means that a function pointer can only point to one type of function. A function pointer to a function that returns an int and takes no parameters can only point to functions that return int's and take no parameters.

We declare a function pointer like this:

```
Return_Type (*Pointer_Name)(Parameters)
```

This means that the function pointer Pointer_Name points to functions that return whatever Return_Type is and take whatever parameters we put in place of Parameters. We assign functions to the function pointer like this:

```
Pointer_Name = myFunction;
```

Note that we do not include the parameters when we assign a function pointer, only the function name. Now that our function pointer points to something, we call it like this (let's assume the function takes an int as a parameter):

```
Pointer_Name(32);
```

This will call the function that Pointer_Name points to with 32 as the parameter. In this case, Pointer_Name points to myFunction so the above line is exactly the same as if we had done this:

```
myFunction(32);
```

Back to our Game State Problem

So we now have a pointer that points to functions, but how does that help us with our game state problem?

Well, we already have the functions and our while loop set up. So what if we point a function pointer to the menu function and call the function pointer from the while loop? We'll assume that when the player decides to start a game, the menu handles pointing

the function pointer to the game function, and that the game function will then handle pointing the pointer to whatever state comes next, and so on.

Let's code this and then discuss what happens.

```
// Note that we no longer need a switch statememt, so we no
// longer need the enumerations or the CurrentState variable.

// Our functions don't need to return the current state anymore.
// We just have them return false if it's time to exit our loop.

bool Menu();
bool Options();
bool Game();
bool Exit();

// Here is where we declare our function pointer. It is
// global so the state functions have access to it. Note
// that the return type and parameters match our game
// state functions.
bool (*StatePointer)();

int main()
{
    bool playing = true;

    // Let's point the function pointer to the menu function.
    // Remember that we leave out the brackets.
    StatePointer = Menu;

    // Now we start a while loop and call our function pointer.
    // When a state changes, it will be handled in our game
    // functions so we never need to deal with this section of
    // code again.
    while (playing)
    {
        // Call the function pointer and assign the result
        // to our playing variable. If a function returns
        // false, our loop will quit.
        playing = StatePointer();

    }

    return 0;
}
```

Now that we've placed all of the responsibility on our game functions, we can forget about main() entirely. When the while loop starts, it calls the function pointer, which points to whatever function handles the current state of our game.

We start with it pointing to the menu program. If the user chooses to play the game, the function pointer will be assigned to the game function from within the menu function. If the user chooses to quit, the function pointer will be assigned to the exit function. If the user makes no selection, the while loop will just keep calling our function pointer.

I've written a simple menu program and heavily commented it for you. I highly recommend that you check it out before continuing with this tutorial.

Click here to download the source code for this section of the tutorial.

State Stacks

The previous example relied on the fact that the game functions will change the function pointer to point to whatever state comes next. This means that the options menu could set the pointer to the main menu, it could set it to the game function, or it could set it to the exit function. But what if we wanted our system to be a bit more restrictive?

For instance, think of some of the games you've played. A lot of them start with the main menu. After the main menu, you can select to start a game, go to the options screen, or exit the game. If you select the options menu, it will pop up with a new menu where you can select more submenus like input, sound, and graphics settings.

Let's say we select the options menu from our main menu. The options menu pops up and we select to see our input settings. Once we're done staring at our input settings, we hit the **escape** key. This causes the input settings to disappear and we're back to where we just were, the options menu. We hit **escape** again and we're back to the menu.

Using our previous method, we would have the menu set our pointer to the the options menu, the options menu set our pointer to the input settings, the input settings set our pointer to the options menu, and finally we would have the options menu set the pointer to the main menu.

A better way to do this is to use a stack. If you're not familiar with stacks, I have written a tutorial that can be found here. Also, I'll be using the STL implementation of a stack; see my STL Tutorial for an introduction to the STL stack (it'll only take a sec, trust me).

So we now all know that a stack stores data like a stack of plates. If we add a plate to the stack, it goes on top, and if we take a plate from the stack, we take it from the top.

Alright, so what does a stack have to do with function pointers? Well, let's say we have a stack of function pointers that control the states of our game. When we start our program, we push a pointer to the menu function onto the stack. The stack now looks like this:

(bottom) **Menu** (top)

When we choose to open the options menu, the stack looks like this:

(bottom) **Menu->Options** (top)

Selecting the input menu, we get this:

(bottom) **Menu->Options->Input** (top)

When we hit **escape**, we don't have to have the input function do anything. We can just tell our stack to pop its top element. We now have this:

(bottom) **Menu->Options** (top)

After hitting escape again, we get:

(bottom) **Menu** (top)

Remember how with our last method we had to have the playing variable keep track of whether the while loop should continue or not? We no longer need that. All we have to do is check to see if our stack is empty and we'll know to quit the loop. If we hit **escape** again with our current stack, the pointer to the menu function will be popped and the stack will be empty.

Adding states to our game is easy now too. We just have to write our function and push a pointer to it on the stack when we want to run it. In each iteration of our while loop we'll just call whatever function is at the top of the stack.

So I guess I better show some code now, eh? See the downloadable source code for this section of the tutorial for a working example. Note that we can't just initialize a stack of function pointers. If we try this:

```
stack<void (*StatePointer)()> StateStack;
```

our compiler will go nuts. STL stacks can't just take function pointers. My solution to this problem is to encapsulate a function pointer within a struct. The std::stack has no problem with storing structs. Here's the final version of our game state example:

```
// Declare our struct which holds a function pointer
struct StateStruct {
    void (*StatePointer)();
};

// Note that our functions no longer need to return false if it's time to
// exit the program. We just have to check if the state stack is empty.
void Menu();
void Options();
void Game();

// I've decided to have exit be a helper function for the MainMenu. If MainMenu is the only
// thing left on the stack, then we know that the user is trying to quit if 0 is pressed.
// MainMenu will call Exit() which will ask the user if it's really time to quit. Note that
// this function returns a boolean, so our function pointers can't point to it. If this is
```

```
// confusing, see the MainMenu implementation in the downloadable source code below.
bool Exit();

// Here's our state stack. We're declaring a std::stack that holds StateStructs.
// Remember that StateStruct is just a struct that contains a function pointer.
stack<StateStruct> g_StateStack;

int main()
{
    // Let's now start by pushing a Menu pointer onto the stack. First, we have to
    // declare a StateStruct structure and point its function pointer to MainMenu.
    // Then we push the whole structure onto the stack.
    StateStruct menu;
    menu.StatePointer = MainMenu;
    g_StateStack.push(menu);

    // We will now continually call the function pointer at the top of our stack.
    // Notice how our while loop's terminating condition is now an empty stack.
    while (!g_StateStack.empty())
    {
        // All we have to do here is call the function at the top of the stack.
        g_StateStack.top().StatePointer();
    }

    return 0;
}
```

Click here to download the source code for this section of the tutorial.