Funcdef & Function Pointers

This tutorial explains how to set up and use funcdefs & function pointers in level scripts. The first two parts of this tutorial cover some very basic examples on use, with the last mixing arrays and function pointers to allow both calling sequences and a function at random.

Introduction to the "funcdef" statement

The Angelscript documentation states that *"A function pointer is a data type that can be dynamically set to point to a global function that has a matching function signature as that defined by the variable declaration."* In other words, you're making a "type" - something like *int* or *string* - but instead of this type's variables containing *text* or *numbers*, it contains functions - or more specifically, pointers to functions .
To understand better what a function pointer is: Consider a box, and this box contains a note telling you to look in another box somewhere else - this is your function pointer, it just contains information on where to look. The other box, the one *referenced,* contains the actual functional information. We can use these "extra boxes" to create variables that can be called just like functions.

The first stage in the process, is to make the type which will used to create variables later. This type is dependant on what the signature of your function is (that is, what arguments are taken in by the function, and what is returned by the function) - Below are some example funcdef statements:

```
funcdef void fdSimpleFunction(); // Used for pointers to functions which take no arguments and return no vales
funcdef int  fdReturningFunction(); // Used for pointers to functions which return an int and take no arguments.
funcdef int  fdComplexFunction(int,int);  // Used for pointers to functions which of return type int and take two ints as arguments
```

Taking the top statement, here is a sample function which could later be pointed to:

```
void sfHelloWorld() {
    AddDebugMessage("Hello World!",false);  // Outputs: Hello World!
}
```

With a function pointer type defined, and a function to point to - the next step is to make the function pointer:

```
fdSimpleFunction@ functionVar;
```

This is similar to just normal variable creation, except the "@" sign - The "@" symbol in angelscript literally means "Handle of" / "Address of", and is just stating that our variable is a pointer. However it currently point anywhere - it literally *"is null"*. To make it point to a function, we use an assignment statement just like a normal variable, except the "@" symbol appears again:

```
// We know this line declares our functionVar
fdSimpleFunction@ functionVar;
```

```
// We now will make this point to sfHelloWorld()!
@functionVar = @sfHelloWorld;
```

This new line is making *functionVar*point to the address of *sfHelloWorld* (in other words, we just put a note in our box saying look into the specific other box for *sfHelloWorld*). You are actually saying **"set the handle of function var to the handle of sfHelloWorld"** - this is why *functionVar* can be called just like the function, even though it is actually a variable:

```
// If all has gone well:
functionVar();

// Does exactly the same as:
sfHelloWorld();
```

Below is a sample map script file using the code covered in this section, make sure to be in developer mode to see the output.

```
// Make the type "fdSimpleFunction"
funcdef void fdSimpleFunction();

// Output hello world
void sfHelloWorld() {
    AddDebugMessage("Hello World!",false);
}

void OnStart() {
    // Call the function normally!
    AddDebugMessage("Calling the function normally!",false);
    sfHelloWorld();

    // Call the function using our variable!
    AddDebugMessage("Calling the function using the pointer!",false);
    fdSimpleFunction@ functionVar;
    @functionVar = @sfHelloWorld;
    functionVar();
}
```

So far, all we have managed to achieve is something we could have done already! Why go through all this hassle, to just call a function we could have called anyway? The next section shows how the variable aspect of the function can be exploited to solve a basic problem.

Solving a basic problem with function pointers

This following example can easily be solved without function pointers - however it compactly demonstrates that function pointers can point to any function with a matching signature, and gives a glimpse of what can be achieved using this. If a function pointer can change where it points to - we can effectively make one function call actually call different things to suit what we want. Take for example, the following two functions:

```
void bigFunction() {
    subFunction();
}
void subFunction() {
    // do something...
}
```

The specification is as follows: At the end of *bigFunction* we want to call a function called *output1*. However, *subFunction* should have a random (1/4) chance of making *bigFunction* call *output2* instead. Both the output functions are defined below:

```
void output1() {
    AddDebugMessage("Yo!",false);
}
void output2() {
    AddDebugMessage("Dawg!",false);
}
```

This problem can be solved using a function pointer. First again, the type which matches are output functions is defined:

```
// Create a function definition (which is actually the same as
fdSimpleFunction)
// Which will matches the signature of both our output functions
funcdef void fdOutput();
```

With the second step creating the variable which shall initially point to *output1* at the start of *bigFunction.* Next we call*subFunction* as before, which will change the function pointer. Finally, we will call the function pointed to by the pointer:

```
fdOutput @outputChoice;
void bigFunction() {
    @outputChoice = @output1;
    subFunction();
    outputChoice();
}
```

The subFunction implementation is as follows:

```
void subFunction() {
    if(RandInt(, 3) == 1)
    {
        @outputChoice = @output2;  // A 1 in 4 chance of this code being
reached
    }
}
```

Problem solved. You can test out the full code as with the previous section below:

```
// Create a function definition (which is actually the same as
```

```
fdSimpleFunction)
// Which will matches the signature of both our output functions
funcdef void fdOutput();

// Our output functions ///////////
void output1() {
    AddDebugMessage("Yo!",false);
}
void output2() {
    AddDebugMessage("Dawg!",false);
}
//////////////////////////////////

// The actual stuff that does the descision making
fdOutput @outputChoice;
void bigFunction() {
    // Initially point to output 1
    @outputChoice = @output1;        // 1 in 4 of output 2...
    subFunction();

    outputChoice();   // Call whichever output has been chosen
}

void subFunction() {
    if(RandInt(,3)==1)
    {
        @outputChoice = @output2;    // A 1 in 4 chance of this code being
reached
    }
}

void OnStart() {
    for(int i=; i<20; i++) bigFunction();  // Call bigFunction 20 times -
1/4 chance of output2?
}
```

It's time to move onto solving a much bigger problem: Calling a random function.

Arrays, function pointers, and you

We shall extend the above problem to have an arbitrary number of functions, and still manage calling one at random. To make things easier though, we will have each function called with an equal probablility. Obviously we could do a massive set of ifs, or a huge switch-case block (what if there are 100 choices? who would want to write that out that switch-case?). We shall re-use most of the code from the first section to get started, however, this time there are two new signature matching functions, giving us the following script file so far:

```
// This creates a signature called "SimpleFunction"
// Which matches functions which take no arguments, and return nothing.
funcdef void fdSimpleFunction();
```

```
// Such as these example functions
void sfHelloWorld() {
    AddDebugMessage("Hello World!",false);   // Output hello world
}
// This function will output how many times it has been called
void sfDisplayTimesCalled() {
    AddLocalVarInt("DTC_TimesCalled",1);
    AddDebugMessage("DisplayTimesCalled, called: " +
GetLocalVarInt("DTC_TimesCalled") + " times", false);
}
void sfPlayScarySound() {
    PlayGuiSound("enemy\\brute\\notice.snt",1.0f);   // Play the sound of an
angry brute!
}
```

The problem of calling one of the functions at random is going to be solved by making an array. Each index is going to contain one of the above functions, we just then pick a random index, and call the function at that index. This solution scales really nicely when there is are a good number of functions to call.

An array of function pointers is declared in the following code segment - note that the {} part is not required, but merely used for the sake of compact-ness, all this section does is initialise the array with some values in it already. If you are not familiar with arrays, take a scan through the Angelscript documentation.

```
fdSimpleFunction@[] simpleFunctions = { @sfHelloWorld,
@sfDisplayTimesCalled, @sfPlayScarySound };
```

Now all is left to do is create a function that picks an index at random, and calls the function at that index:

```
void callRandomSimpleFunction() {
    uint index = RandInt(,simpleFunctions.length()-1);           // Pick a
random index from the array
    fdSimpleFunction @functionToCall = simpleFunctions[index];    // Select
that function
    functionToCall();                                             // Call
that function

    // Note this can be simplified down to one line:
    // simpleFunctions[RandInt(0,simpleFunctions.length()-1)]();
}
```

We are also in a prime position to solve the problem of sequences here too, the following code will call each function in order in the array:

```
void callEachFunction(string &in asTimerName) {
    uint index = GetLocalVarInt("simpleFunctionIndex");    // Get the index
    if(index>= simpleFunctions.length()) return;           // Don't go any
further if we have called all the functions
```

```
    // Access, and call like before:
    fdSimpleFunction @functionToCall = simpleFunctions[index];
    functionToCall();

    AddLocalVarInt("simpleFunctionIndex",1);        // Increment the index
    AddTimer(asTimerName,1.0f,"callEachFunction");  // Call this function
again in 1 second
}
```

One potential extension for use with sequences is making each function return a float, which can then be used as the delay time before calling the next function. Below is a sample script file for this section, don't forget to visit the OnStart routine and uncomment one of the two lines - or you won't see anything!

```
// This creates a signature called "SimpleFunction"
// Which matches functions which take no arguments, and return nothing.
funcdef void fdSimpleFunction(); // Such as these example functions
void sfHelloWorld() {
    AddDebugMessage("Hello World!",false);    // Output hello world
}
// This function will output how many times it has been called
void sfDisplayTimesCalled() {
    AddLocalVarInt("DTC_TimesCalled",1);
    AddDebugMessage("DisplayTimesCalled, called: " +
GetLocalVarInt("DTC_TimesCalled") + " times", false);
}
void sfPlayScarySound() {
    PlayGuiSound("enemy\\brute\\notice.snt",1.0f);    // Play the sound of
an angry brute!
}

// We now can create an array of these simple functions for further use.
// Note that fdSimpleFunction@ is like a type now - like string, or int!
fdSimpleFunction@[] simpleFunctions = { @sfHelloWorld,
@sfDisplayTimesCalled, @sfPlayScarySound };

// Some example uses of this:
// Calling a random function
void callRandomSimpleFunction() {
    uint index = RandInt(,simpleFunctions.length()-1);           // Pick a
random index from the array
    fdSimpleFunction @functionToCall = simpleFunctions[index];    // Select
that function
    functionToCall();                                             // Call
that function

    // Note this can be simplified down to one line:
    // simpleFunctions[RandInt(0,simpleFunctions.length()-1)]();
}
// Using a timer to call one function per second in sequence.
void callEachFunction(string &in asTimerName) {
```

```
    uint index = GetLocalVarInt("simpleFunctionIndex");    // Get the index
    if(index>= simpleFunctions.length()) return;           // Don't go any
further if we have called all the functions

    // Access, and call like before:
    fdSimpleFunction @functionToCall = simpleFunctions[index];
    functionToCall();

    AddLocalVarInt("simpleFunctionIndex",1);        // Increment the index
    AddTimer(asTimerName,1.0f,"callEachFunction");  // Call this function
again in 1 second
}
// Using a timer to repeatedly call a random function
void callRandTimer(string &in asTimerName) {
    callRandomSimpleFunction();
    AddTimer(asTimerName,1.0f,"callRandTimer");
}
void OnStart() {
    // Uncomment one of the following to test it out!
    //callEachFunction("testTimer1");
    //callRandTimer("testTimer2");
}
```

Closing comments

A quick note on an earlier point - when the function pointer is defined initially, it can point nowhere. This can be tested for with the following code:

```
if( functionPointer is null )
```

Finally, that state of a function pointer **is not saved when the map is -** there is no way to save the state of a function pointer - so keep them the same (which is fine in the case of an array - like the last example), or assume they haven't changed within the**scope** of one function (as in the second example).

You should now understand a little about funcdefs. Have a play around, you can use this to make sequences, callbacks, and all sorts of fun stuff.

Check out the documentation at: www.angelcode.com/angelscript/sdk/docs/manual/doc_script.html