Functions - Part 3: Digging Deeper

At a Glance

Passing by Value vs Passing by Reference

Two kinds of types:

1. Value types (all fundamental types, except string)
2. Reference types (strings, arrays, and classes)

Passing by Value

```
// Reminder: passing by value
int Double(int input)  // 'input' contains a copy of the original value
{
    input = input * 2;   // This is not enough, as 'input' is a copy

    return input;        // The caller must get the new value through the
standard return
}
```

Reference parameters can be:

- **in** - input references - not very useful, similar to passing by value
- **out** - output references
- **inout** - input/output references - only allowed for reference types

Passing by Reference

```
// Input references are best used with const parameters (see discussion)
int GetSquare(const int& in input)
{
    return input * input;
}


// Output references replace the value of the original variable when the
function ends
void GetRandomEvenIntByOutRef(int& out result)  // 'result' starts
uninitialized
{
    result = RandInt(, 1000000) * 2;     // this affects the original
variable
```

```
}

// Input-output references immediately affect the original object:
void AddBrackets(string& inout text)
{
    text = "(" + text + ")";       // this immediately affects the original
object
}

// In/out-reference parameters can be alternatively (and preferably) written
as:
void AddBrackets(string& text)
{
    text = "(" + text + ")";       // this immediately affects the original
object
}

// Input-output references with classes:
void ResetPassCode(PassCode& passcodeObject)
{
    passcodeObject.Reset();       // this immediately affects the original
object
}


// Passing by reference using object handles:
void ResetPassCode(PassCode@ passcodeObject)
{
    passcodeObject.Reset();       // this immediately affects the original
object
}
```

Using References to Support Multiple Return Values

```
void  GetCircleProperties(float radius, float& out circumference, float& out
area)
{
    float pi = 3.14f;
    circumference = 2.0f * pi * radius;
    area = pi * radius * radius;
}
```

Callbacks

Example 1: SetEffectVoiceOverCallback(string& callbackName)

```
// Documentation wiki says:
//     Callback snytax: void MyFunc()  --> return type and parameter list
must match
//                                         for candidate functions
```

```
// "hookup"
SetEffectVoiceOverCallback("OnVoiceOver");

// The callback is defined elsewhere in the code:
void OnVoiceOver()    // Function name chosen by the scripter
{
    // do something...
}
```

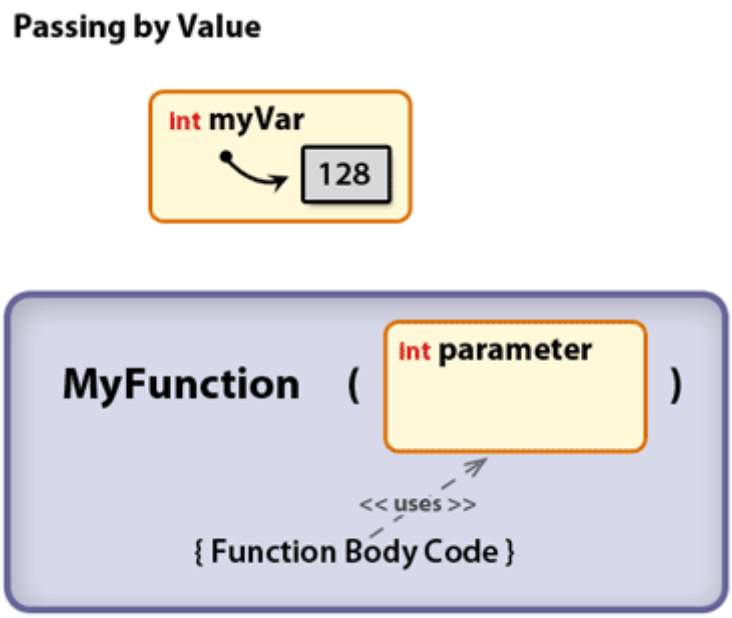
Example 2: AddTimer(string& asTimer, float time, string& callbackName)

```
// Documentation wiki says:
//     Callback snytax: void MyFunc(string &in asTimer)  --> return type and
parameter list must match
//                                                          for candidate
functions

// "hookup"
AddTimer("some.timer.id", 2.0f, "TwoSecondsLater");

// The callback is defined elsewhere in the code:
void TwoSecondsLater(string &in asTimer)    // Function name chosen by the
scripter
{
    // do something...
}
```

Discussion

Passing Parameters by Value

By default, parameters to functions are passed *by value*. This means that *a copy* of the original value is created when the function is entered, and it is this copy that you work with within the scope of the function. If you make any changes to the value, the original is not affected, as it is in no way connected to the parameter. Once the function returns, the parameter goes out of scope, and no changes are preserved.



As the animation demonstrates, when the function makes changes to the parameter (when "{Function Body Code}" flashes in the image), it's the temporary, local value that is affected, and thus any changes within the scope of the function are local and temporary in nature as well - they exist only within the function, and only for the duration of the function.

Passing Parameters by Reference

There are other ways to pass parameters. Collectively, these ways are referred to as *passing by reference*. The word "reference" describes something that *refers to* (or *points to*) something else - just like a phone number is is a reference to someone's physical phone, or a web address is a reference to a web page.
In the case of reference parameters, it means that the parameters themselves are somehow connected back to the original inputs. There are three kinds of references provided by the language:

- input references (`&in`)
- output references (`&out`)
- input-output references (`&inout`, or just `&`)

You can turn a by-value parameter, like `int paramName`, to a reference parameter by appending the symbols listed above to the type of the variable. For example, to declare a parameter as an out-reference, you would use any of these (they are all equivalent and are a matter of preference):

```
int&out paramName
int &out paramName
int& out paramName
```

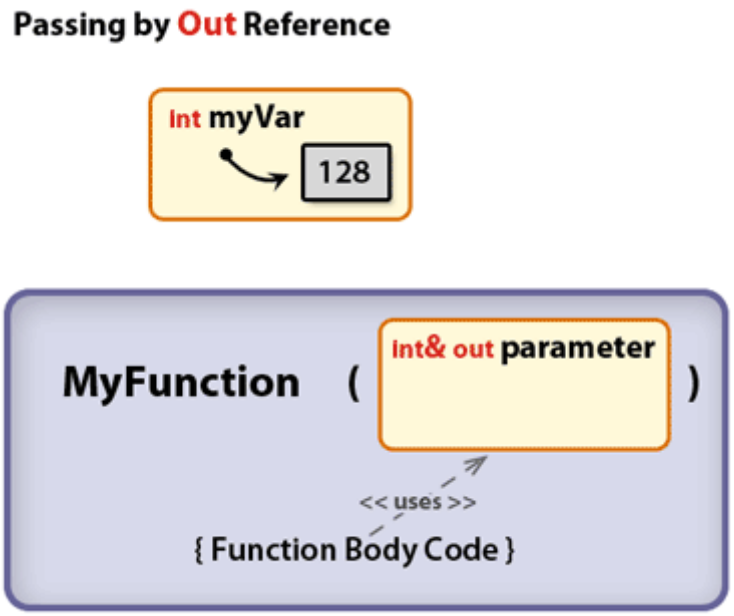
In this guide, the third notation is preferred, since it allows you to think of `int&` as of an "int-reference" *type*, while regarding the `out` keyword as a behavioral modifier, which determines the relation between the original variable that was passed, and the reference parameter itself (see below).

Input References

An in-reference designates the parameter as input-only, denying the possibility of affecting the original, which *mostly* has the exact same effect as passing by value. As such, they are not particularly useful, except with const-parameters (which can't be changed by definition), when they *may* improve performance by not creating a copy, letting the parameter to point to the original memory location instead.

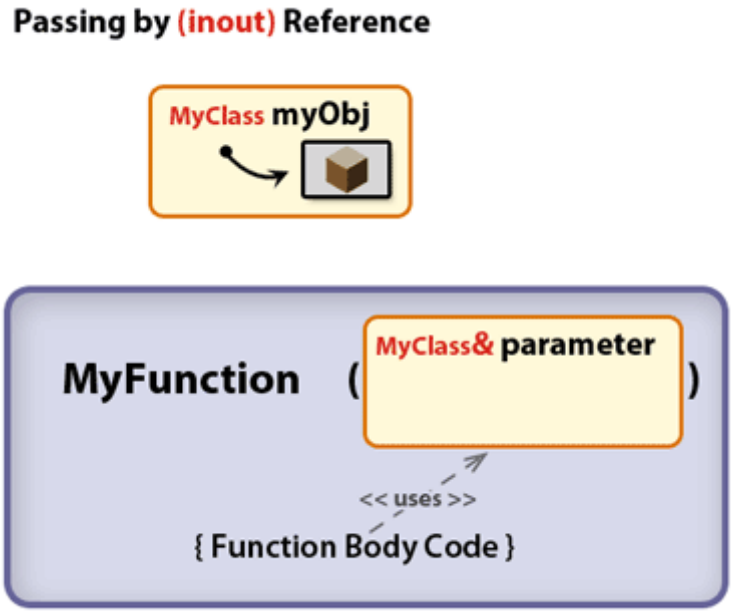
Output References

Marking a parameter as an out-reference designates it as output-only, which has more interesting consequences. Basically, you're declaring that the value of the variable passed as the out-parameter will be set from within the function. When the function starts, the parameter itself begins *uninitialized*; you are then free to perform any calculations required, and set the value of the out-parameter. During the execution of the function, the original is not affected, but *when the function reaches its end*, the the value of the out-parameter replaces the original value.



Input-Output References

Inout-references are probably the most useful, but are only allowed on so-called reference types, which are strings, array types, and user defined classes (you'll learn more about classes and objects in a different part of the guide). As you have guessed, they declare the parameter to be both for input and for output. When the function begins, no copies are made - the parameter instead points directly to the original memory location, effectively becoming an alias for the original variable. Any changes made from within the function are immediately “visible” to the original variable, because both the parameter and the variable point to the same thing.



The animation shows how an inout-reference works on a user defined type (class); the initial state (“value”) of the object is represented by the cube; when the function is called, the function's code changes the state through the inout-reference parameter, which is represented by the cube becoming a cone. The effect on the original object is direct and immediate, and it is retained even after the function ends.

Using Reference Parameters to Support Multiple Return Values

Normally, a function may return only one result; however, with out-references and inout-references, you can return multiple values from a single function. In the example below, `radius` is passed by value as an argument, while `circumference` and `area` represent output parameters, which are to be the return values of the GetCircleProperties() function.

```
void  GetCircleProperties(float radius, float& out circumference, float& out
area)
{
    float pi = 3.14f;
    circumference = 2.0f * pi * radius;
    area = pi * radius * radius;
}

// Somewhere else in the code:
float circleRadius = 10.0f;
float circ, area;

GetCircleProperties(radius, circ, area);

// At this point,
// circ == 62.8f, and
// area == 314.0f
```

Output parameters can also be useful when the function needs to return a success/failure indicator, alongside the result of some calculation. In the example below, the SetEntityActiveRandom() function randomly activates/deactivates an entity. It returns `false` if the specified entity couldn't be found on the map (say, if you misspelled the name); otherwise it returns `true`. The randomly picked activation state is returned via an output parameter, for convenience, in case the caller might want to somehow use it elsewhere. Also, now that you know what reference parameters are, note that the string is passed by an inout-reference, since this is faster, avoids unnecessary copies, and is consistent with the rest of the functions exposed by the engine.

```
bool SetEntityActiveRandom(string& entityID, bool& out activationState)
{
    if(!GetEntityExists(entityID))
        return false;  // no such entity

    // OK, the entity exists
    activationState = (RandInt(, 1) == 1);

    SetEntityActive(entityID, activationState);
    return true;
}

// Usage:
bool wasActivated;
bool success = SetEntityActiveRandom("servant_grunt_corridor",
wasActivated);

if (success && wasActivated)
{
    // do something with the monster
}
```

Note how the code that uses the function can test if the call was sucessfull. Often, functions which return boolean success indicators, like SetEntityActiveRandom() does, are called from within the if-conditionals themselves; the next code snippet is equivalent to the “usage” code above:

```
bool wasActivated;
if(SetEntityActiveRandom("servant_grunt_corridor", wasActivated))
{
    if (wasActivated)
    {
        // do something with the monster
    }
}
```

This is possible because the function returns a `bool`, that is, it *evaluates* to a `bool`, which is all that is required for it to be used as a condition in an if-statement.

Do Reference Parameters Play a Role When Overloading Functions?

Yes they do, but in a *specific way*, and with a few *caveats*. Essentially, adding a reference modifier to a parameter changes its *type*. E.g., `int` is the integer type, while `int& out` is an "integer reference" type. Reference types are *only allowed* as function parameters though. Now, although the different reference modes (in, out, inout) behave differently, as far as the compiler is concerned, all of them give the same function signature. In its eyes, `string&`, `strint& in` and `strint& out` are the same when it comes to deciding which overload to call.

For this reason, except for some special cases and unless you take some special actions, the compiler often won't be able to decide which overload you intended to make the call to. This is why you generally shouldn't overload a function simply by providing another version which uses reference parameters.

This is also one of the reasons this guide prefers writing `int& out` to `int &out`.

Callbacks

If you read through the previous discussion, and through Functions Part 1 and Part 2, you should now have enough knowledge to understand how to use callback functions, and to get a good grasp on how they work.

A *callback* function is, simply put, just a function which is supposed to be, at some later point, called by some other peace of code, when some event of interest happens.

Among the predefined, engine-exposed functions, there's quite a number of functions which let you add a callback to some internal game structure, so that it can be called when a certain in-game event, such as an entity collision, happens. Here are some of them:

```
void AddTimer(string& timerID, float timeSpan, string& timerCallback);  // 'timerCallback' is called when timeSpan elapses
void SetEffectVoiceOverCallback(string& voCallback);  // 'voCallback' is called when voice-over effects end
void SetLanternLitCallback(string& lanternCallback);  // 'lanternCallback' is invoked when the lantern is lit
void SetEntityCallbackFunc(string& entityID, string& interactCallback);   // ''interactCallback'' called when 'entityID' is interacted with
```

However, each of these predefined functions requires a callback of a specific format - the callback has to have a specific return type, and a specific parameter list. The name of the callback is not important, nor are the names of the parameters; the number, types, and the order of the parameters are, though. For each of the engine-defined functions, the wiki documentation, along with the meaning of each parameter, states the required callback format as "Callback syntax:". For example, for the AddTimer() function, it says:

```
void AddTimer(string& timerID, float timeSpan, string& timerCallback);
```

```
Creates a timer which calls a function when it expires.
Callback syntax: void MyFunc(string& in timerID)

timerID - the name of the timer
```

```
timeSpan - time in seconds
timerCallback - the function to call
```

This means that the user-defined callback has to have the return type of `void`, and that it has to accept a single `string` parameter as an in-reference. In other words, the declaration of the callback can be expressed as:

```
void FunctionName(string& in);
```

To assign a function which meets those requirements as a timer callback, all you need to do is to call the AddTimer() function, passing the *name* of your callback as the last string parameter. The game will then internally use the name to find the function itself, and store this information in an internal data structure, so that the function can be invoked when the time comes.
E.g., if you wanted to add a 5-second timer, and make something happen when it expires, you'd do something like this:

```
void OnEnter()
{
    AddTimer("id.timer.screamScare", 5.0f, "OnScreamScare");
}

// the callback
void OnScreamScare(string& in timerID)
{
    PlayGuiSound("05_event_door_bang.snt", 1.0f);
}
```

The format of the callback serves a specific purpose; in the case of a timer callback, the `timerID` parameter is an input-reference parameter which identifies the timer (useful, as it is possible for several timers to use the same callback). The parameter is passed in *by the game itself*, when the timer expires, and when the *game* makes the call (or “calls the function back”). You can check its value from within the callback if you so desire.

Here's an example from the game (from 00_rainy_hall.hps):

```
// NOTE: The code has been formated to match the style of this guide;
//       some comments are added as well.

// somewhere in the code...
AddTimer("door_gallery", 0.01f, "TimerSwingDoor");

// someplace else...
AddTimer("door_gust", 0.3f, "TimerSwingDoor");

// callback
void TimerSwingDoor(string& in timerID)
{
    // This makes sure that the AddPropForce() function is
    // applied 10 times in short (0.03-second) intervals,
    // simulating the effect of a wind blowing
    if(GetLocalVarInt("SwingDoor") == 10)
```

```
    {
        SetLocalVarInt("SwingDoor", );

        // when the force is applied for the 10th time,
        // simply return from the function
        return;
    }

    // This code check which timer invoked the callback,
    // and then applies a different force based on that
    if(timerID == "door_gallery")
        AddPropForce(timerID, 70.0f, , , "World");
    else
        AddPropForce(timerID, -95.0f, , , "World");

    // This increments the "SwingDoor" indicator by 1,
    // (or sets it to 1 on the first call)
    AddLocalVarInt("SwingDoor", 1);

    // This re-assigns this same function as a callback,
    // restarting the same timer by passing 'timerID'
    AddTimer(timerID, 0.03f, "TimerSwingDoor");

    // (a debug message ommited here...)
}
```

To be qualified as a callback of a specific kind, it is enough for a function to match the required callback format (return type and parameter list). It doesn't even have to be defined by you - it is perfectly legitimate to add a predefined engine function as a callback for some event (although there aren't as many cases where you might want to do this). The code below demonstrates this:

```
void OnEnter()
{
    SetEffectVoiceOverCallback("StartRandomInsanityEvent");
}

// someplace else in the script...
AddEffectVoice("CH01L00_DanielsMind01_01.ogg", "", "Flashbacks",
"CH01L00_DanielsMind01_01", false, "", , );
```

The SetEffectVoiceOverCallback() function defines a callback to be called when a voice-over sequence that can be started during the execution of the game reaches its end. The AddEffectVoice() voice function starts the sequence at some point, and when the sequence finishes, the predefined StartRandomInsanityEvent() is called, and an insanity event (a random “voice in the head”) is played out.

This is possible because SetEffectVoiceOverCallback() requires a callback of the following format:

```
void FunctionName()
```

And from the declaration of StartRandomInsanityEvent(), you can see that the format matches:

```
void StartRandomInsanityEvent();
```

Function Handles (Function Pointers)

Coming soon... In the meantime, see: Funcdef & Function Pointers