Functions - Part 2: Beyond the Basics

At a Glance

Const-Parameters

```
void ConstParamFunc(const int val1, int val2)
{
    val1++;  // ERROR: Changing the value of a const parameter is not
allowed!
    val2++;  // OK.
}
```

Function Overloading & Wrapping

```
// Note: The compiler uses function signatures to determine which function
to call.


// Below are 3 valid overloads of the WriteLine() function

// (1) A convenience method, to get rid of the second parameter of
AddDebugMessage()
void WriteLine(string text)
{
    AddDebugMessage(text, false);   // <--- WriteLine() wraps
AddDebugMessage()
}

// (2) Shows the value of 'data' as a debug message, with some extra info
void WriteLine(int data, string text)
{
    WriteLine(text + ": " + data);  // calls (1)
}

// (3) Shows the value of 'data' as a debug message
void WriteLine(int data)
{
    WriteLine(data, "");  // calls (2)
}

// PROBLEM - return type doesn't make a difference
int WriteLine(int data)  // ERROR: signature same as for (3)!
{
    WriteLine(data, "");
    return data;
```

```
}
```

Discussion

Const Function Parameters

Just as you can turn a variable into a constant by prepending it's declaration with the `const` keyword, you can, by applying the same keyword, turn a normal function parameter to a constant parameter (const-parameter). As a scripter, by doing so, you're making a statement (to the compiler, to yourself, and to other people who might use your code in their projects) that the value of that particular parameter will not be modified for the duration of the function. If you try to change it by accident, you'll receive an error message. This can be useful, especially if the code you're writing is complicated or sensitive, where making accidental changes to values that are supposed to remain constant can cause hard-to-solve problems. Declaring parameters const effectively lets the compiler spot any errors for you.

The following example is trivial, and there's no real need for `const` in this function, but it serves the purpose of demonstrating the syntax:

```
float GetCircleArea(const float radius)  // makes sure 'radius' is not
changed
{
    return 3.14f * radius * radius;
}
```

You can choose to make only some parameters constant, and leave others as they are:

```
// makes sure 'lastItemFoundID' is constant, but 'itemsLeft' is free to
change
void UpdateQuest(const int lastItemFoundID, int itemsLeft)
{
    // implementation omitted...
}
```

Function Overloading

Function overloading is a rather useful feature of the language. It enables you to define several functions with the same name, provided that all of them have different parameter lists. You'd want to do this if you have several functions which all do related things, but on different data types, or if all of those functions differ only slightly. Also, if you have a function which accepts a large number of parameters, but most of them are rarely used, or mostly receive the same set of values on a call, you might want to provide a version of the function which does the same thing, but takes fewer parameters. Overloading makes this possible by letting you define such a function, which can then internally call the original version, passing along its own parameters, and filling the rest with default values.

For example, if you've used the predefined AddDebugMessage() function before, you know that this function takes two parameters: the message itself, and a bool parameter which indicates whether the engine should check if the debug message you're trying to show has already been shown, so that duplicates can be avoided. That's a neat feature, but most of the time, you won't care. That being the case, the chore of passing an extra parameter you don't even use quickly becomes annoying. Luckily, you can *overload* the AddDebugMessage() function, by providing your own version which takes only one parameter, like this:

```
void AddDebugMessage(string text)  // your version which accepts a single string
{
    // it simply calls the original version, always passing 'false' as the 2nd param
    AddDebugMessage(text, false);
}
```

Now you have available for use two versions, or two *overloads* of AddDebugMessage(). Their declarations are:

```
void AddDebugMessage(string& text, bool checkForDuplicates);  // original version; ignore the '&' for now
void AddDebugMessage(string text);  // your version
```

When you use one in a script, the compiler determines which one you called by looking at *how many parameters you've passed*, as well as *what are the types of those parameters*. Note that the actual names of the parameters are not important in this regard; only the types, position, and number of parameters make a difference. This is why, when you refer to different overloads in discussions, you can omit the actual parameter names.

```
void AddDebugMessage(string&, bool); // original; ignore the '&' for now
void AddDebugMessage(string);        // your version
```

Using these in code is pretty straightforward. In this case, it's essentially as if the second parameter to the original function was optional:

```
AddDebugMessage("Player collided with Area6", false);  // calls the original version

// vs

AddDebugMessage("Player collided with Area6");        // calls your version
```

So, to determine what function you intended to call, a compiler will take a look at the name of the function, and then at the parameters you've passed to it. Then it will try to match these to one of the declared functions, by checking the *signature* of each function.

Function signature consists of a function name, and a parameter list (where the names of the parameters are irrelevant, as previously discussed).

The function which has the matching signature gets called. This is important to know, because it means that you *can't* overload a function by just specifying a different return type for the new version. It also means that you *cannot* create an overload by simply changing the names of the parameters.

```
// Note: function bodies omitted, only declarations shown

void AddDebugMessage(string& text, bool checkForDuplicates);     // (1)
original version; ignore the '&' for now
void AddDebugMessage(string& msg, bool checkDuplicates);         // (2)
won't work - exactly the same as (1)
string AddDebugMessage(string& text, bool checkForDuplicates);   // (3)
won't work - in a call, cant distinguish from (1)

//All of these can be the target of this call:
AddDebugMessage("Music started.", false);
```

Wrapping Functions

I've demonstrated how to overload a predefined function, but this is not a requirement. You can overload any function, including the ones you made yourself. When writing debug messages, there's often a need to print out the value of some variable you used in your script, so that you can track what's going on as your code executes. Let's say that you're keeping track of the number of items the Player found, and that you've grown tired of always passing `“Items#: ” + numItems` to AddDebugMessage() (whichever version). You can write a specialized function of your own that is easier to use, which then calls AddDebugMessage() in turn. This technique was already demonstrated in the previous section, and it's called *wrapping*. When you wrap a function, you delegate all or most of the work to it, but you enclose it into another function which is than free to have any name and format you chose. Essentially, you adapt the original function to a new format (or *interface*), one that matches your new needs.

The WriteLine() function in the example below accepts an int and a string, and wraps the AddDebugMessage() function. The wrapper displays a text message, followed by the value of the `data` parameter.

```
void WriteLine(int data, string text)
{
    AddDebugMessage(text + ": " + data, false);
}

// Calling the function:
numItems = 3;
WriteLine(numItems, "Items#");   // outputs: "Items#: 3"
```

WriteLine(int, string) is somewhat more convenient than AddDebugMessage(string&, bool) because you don't have to bother with string concatenation, *and* you can use the second parameter to WriteLine(int, string) to pass along any info text you might want.

More Overloading

Now let's make an overload which just accepts the variable itself, and prints it's value alone:

```
void WriteLine(int data)
{
    WriteLine(numItems, "");
}

// Calling the function:
numItems = 3;
WriteLine(numItems);   // outputs: "3"
```

Note that this function simply makes a call to the previously defined overload. Next, maybe you want to print the values of some floats? No problem, add two more overloads:

```
// Note: function bodies omitted, only declarations shown
void WriteLine(float data, string text);
void WriteLine(float data);
```

Here's another overload which can be used to create horizontal separators, which can be used to logically separate various debug messages:

```
void WriteLine(int length, bool wide)
{
    string separator = "";

    string separatorChar = "-";


    if (wide)
        separatorChar = "=";

    for (int i = ; i < length; ++i)
        separator += separatorChar;

    AddDebugMessage(separator, true);
}
```

Using all of these functions your script:

```
int anInt = 8;
float aFloat = 1.0f;

WriteLine(anInt, "Planets");    // WriteLine(int, string) called
WriteLine(200);                 // WriteLine(int) called

WriteLine(3.14f, "Pi");         // WriteLine(float, string) called
WriteLine(aFloat);              // WriteLine(float) called
```

```
WriteLine(40, true);              // WriteLine(int, bool) called
```

For completeness, the output of each of these, respectively, is:

```
Planets: 8
200
Pi: 3.14
1.0
=======================================
```

When to use Overloads? Any Other Tips?

Overloading functions is great, but there are some guidelines to be considered. As always, strive to create functions which, when called, provide a good idea of what they do. In the example above, the call “WriteLine(40, true)” doesn't really tell you what the function does. If you had to guess just by looking at that call, you'd probably get it wrong. This indicates bad function design - in this case, it is much better to rename that overload to something like WriteSeparator(), at which point it would stop being an overload, but it would be much more user friendly. So, overloads can be very useful, but don't go overboard with them.

Provide overloads when you want to simulate optional parameters. With such overloads, always try to maintain the same parameter order in all overloads. Keep the most commonly used parameters near the front of the list, and push those used rarely to the back. Also create overloads to support the same operation on different data types, but be mindful that the purpose of the function remains clear. In different overloads, try to keep the logically equivalent parameters of different types in the same place in the parameter lists. If an overload feels too cryptic when used in code, consider changing it's name to something that better communicates it's purpose.