Functions - Part 1: The Basics

At a Glance

Syntax:

```
returnType FunctionName(parameter_list)
{
    // function body goes here
}
```

Where *returnType* is a name of a type, or `void` if the function doesn't return a value,
and where (*parameter_list*) stands for a list of zero or more variables, called function parameters:

- ()
- (*variableType variableName*)
- (*variableType1 variableName1*, *variableType2 variableName2*, … , *variableTypeN variableNameN*)

Example:

```
// A function which returns a bool value, and takes no parameters
bool RandBool()
{
    int temp = RandInt(, 1);    // a call to another function
    bool result = (temp == 1);

    return result;        // <--- The RETURN statement
}

// Calling the function:
bool isLampLit = RandBool();



// A function which doesn't return a value, and takes two parameters:
void SetLampStates(bool lit, bool useFadingEffect)
{
    SetLampLit("lamp1", lit, useFadingEffect);
    SetLampLit("lamp2", lit, useFadingEffect);
    SetLampLit("lamp3", lit, useFadingEffect);
}

// Calling the SetLampStates() function:
SetLampStates(false, true);

// Passing in a variable:
```

```
bool isLampLit = RandBool();
SetLampStates(isLampLit, true);

// Or simply:
SetLampStates(RandBool(), true);
```

Discussion

A function is a peace of code that does something; it has a specific job. Your script will generally be composed of several functions, and it will be calling a lot of functions, either the ones you made yourself, or the ones already provided by the game engine.

A function can be *called* (or *invoked*); by calling a function you are “asking” it to execute, and do it's job. Calling a function causes something to happen - either the function makes some modifications to the game (e.g. lits/unlits lights, creates particle systems, starts/stops music, adds a game timer), or does some calculation and returns a value (e.g. the RandInt() and RandFloat() calculate and return pseudorandom numbers), or makes some changes to the elements of the script itself, where a value may or may not be returned. When a call is made, the execution jumps from the line which made the call, and goes “inside” the function, continuing there until the function exits, either by reaching it's end or a `return` statement.
After that, the execution jumps back to where it left off, and goes on to the next line of code.

HPL2 engine exposes a bunch of predefined functions, but you can (and often will) define your own. A function declaration & definition has the following format:

```
returnType FunctionName(parameter_list)
{
    // function body goes here
}
```

The placeholder *returnType* can be replaced by the name of any data type, if the function wants to return a value, or by `void`, if it does not. (You can think of `void` as of “empty” or “nothing”.) If a non-void return type is specified, the function must return a value, otherwise, it will not compile. This is achieved by using the `return` keyword inside the function body:

```
returnType FunctionName(parameter_list)
{
    return return_value;
}
```

Here, *return_value* has to be of the type specified by *returnType*, and can be any expression that can be *evaluated to a concrete value*, like a constant, or a string literal, or a variable, or a call to another function which returns a value of the appropriate type.

When the return statement is reached during execution, the value is returned and the *function is exited* - if there are more lines of code *after* the `return`, they will *not be executed*.

You should replace the *FunctionName* placeholder with a descriptive name for your function. The name should provide a clue about what the function does, but you are free to chose any name. The rules are similar to that used for variable names - a function name cannot start with a numeric character, can't have white space within it, etc.

The *function body* is everything between { and }. It's contents defines what the function does. It can contain local variable declarations, simple expressions, various control-flow statements, calls to other functions, etc.

Finally, *parameter_list* placeholder stands for a list of variables that are called function parameters, and which are used by the calling code to pass data into the function. The funcion doesn't have to take any parameters, though. In that case, the name of the funcion is simply followed by an empty parameter list `()`. Otherwise, one or more parameters are listed, in the form of variable declarations, separated by commas:

- (variableType variableName)

or

- (variableType1 variableName1, variableType2 variableName2, ... , variableTypeN variableNameN)

The types of the parameters don't have to be the same. The names of the parameters are chosen by the scripter; once again, it is a good practice to chose meaningful, descriptive names.

For example, this function returns a float value, and takes 3 parameters of the same type, using those parameters in the function body to perform a calculation:

```
float Calculate(float x1, float x2, float offset)
{
    return (x1 * (1.0f - offset)) + (x2 * offset);
}
```

This is both the declaration and the definition. Often, when talking about functions, we are not so much interested in what the function does (definition), as we are interested in how to *use* the function (which is visible from the declaration.) If we omit the body of the function, we get the *declaration* of the function, which we can show to other people who might need to use it:

```
float Calculate(float x1, float x2, float offset);    // declaration
```

All the essential information is in there.
Here, the return type is `float`, and the function name is `Calculate`. The parameter list is `(float x1, float x2, float offset)`.
Note: When talking about functions in this guide, I'll allways append `()` to the end of a function name, regardless of the actual parameter list, so that functions can be easily distinguished form other concepts which might appear in the discussion, such as types and variables.

If you visit the Engine Scripts page, you'll see that all the predefined functions over there are listed as function declarations.

Function parameters are also called *function arguments*.

Calling a Function

To make a call to a function, you must type in a function name, and also pass in all the required parameters, by listing them in the `()`. The parameter list in the declaration defines the meaning, the order, and the types of the parameters. When making the call, you need to pass *actual* values (or anything that evaluates to such values). For example, to make a call to the Calculate() function defined above, you'd type:

```
Calculate(0.1f, 10.0f, 0.75f);
```

Here, the value of `0.1f` is passed as the x1 parameter, `10.0f` is passed as the x2 parameter, and `0.75f` as the offset parameter. I'll repeat the declaration of Calculate for easy comparison:

```
float Calculate(float x1, float x2, float offset);
```

However, notice that this function returns a `float` value - and we usually want to store the result of a function. We can do that by assigning it to a variable:

```
float calculationResult;
calculationResult = Calculate(0.1f, 10.0f, 0.75f);
```

Or simply:

```
float calculationResult = Calculate(0.1f, 10.0f, 0.75f);
```

We can also pass in *variables* as parameters:

```
float min = 0.1f;
float max = 10.0f;
float offset = 0.75f;

float calculationResult = Calculate(min, max, offset);
```

Since functions can return values, a result of a function can be passed in as well:

```
float min = 0.1f;
float max = 10.0f;
float offset = RandFloat(0.0f, 1.0f);

float calculationResult = Calculate(min, max, offset);

// or shorter:
float calculationResult = Calculate(min, max, RandFloat(0.0f, 1.0f));
```

The Scope of Function Parameters

Function parameters are associated with the function body, as if they were variables declared inside it. Thus, function parameters - the variables in the declaration - are local in scope, that is, they are visible (usable) only from within the function itself.

When variables are used in a *function call* as input parameters, these input variables, and their names, are external to the function (and are generally not visible to it); it is *the data* they contain that gets passed in, not their names. The data values, upon entering the function they were passed to, *become assigned* to the corresponding names in the parameter list.

Where to Make the Call From?

Once defined, a function is visible from the entire script, so it can be called from any other function. It can even call itself - this is called *recursion*! Be careful, though. If a function calls itself, it has to have a way to stop doing that, or bad things will happen (more on that in part 2). Similarly, if two functions call each other, this will go on “forever” unless certain preventive measures are taken (more on this also in part 2).

Why Use Functions?

Generally speaking, it is possible to write a map script that consists only of a single function. However, there are benefits to breaking your code down in several smaller peaces, organized as funcions. What's more, sometimes you don't have a choice, due to the design of the script engine - a script has 3 basic entry points all of which are separate functions (OnStart(), OnEnter() and OnLeave() - see Execution Flow). Also, when you use timers, or want to respond to interactions among entities in the game world, you must use *callbacks*, which are functions that are called timer ticks, or object collisions and such, functions that you write yourself (more on this in Part 2).
So, the HPL2 engine encourages you to split your code into functions anyway.

However, the benefit of using functions is that it allows you to organize your code into self-contained functional parts, which are easier to *maintain, debug and reuse*. For example, you'll often need to perform the same set of operations several times, at several places in your script. Without functions, this leads to code duplication, difficulties when it comes to making changes, hard-to-track errors, and an ugly looking wall of text. With functions all of this is avoided: duplicated code is elegantly replaced with function calls, which, as you know, can accept context-specific variables. If any changes to the way the function works need to be made, all of the relevant code is *in one place*.
Provided that functions use descriptive names, the practice also makes your code more readable, and easier to understand. It's a difference between facing a wall of difficult-to-track text, and a short list of descriptive function calls. For example, if I present you with a completely unfamiliar peace of code, you'll probably be able to guess what it does just by examining the names of the functions called, without ever knowing *how* those functions actually work:

```
void OnCollideArea_DramaticEvent(string &in entity1, string &in entity2, int collisionDescription)
{
    SetPlayerActive(false);
    StartPlayerHeadAnimation();
    StartDramaSoundFXsAndMusic();
    RemoveAllItems();
}
```

```
// SetPlayerActive() is exposed by the game, other functions
// are custom, and are omitted from this snippet.
```

If you haven't read Part 2, ignore the strange “&in” qualifiers for now.

Just by looking, you can tell, judging by the name OnCollideArea_DramaticEvent(), that this function is called when the Player collides with a specific script area, and that this function initiates some in-game dramatic event: first it takes control from the player, then it calls another function that animates the camera, and another to start music, and yet another to make the player loose all of the items. (Maybe the ground gave way, and the Player fell into a river! The exact details are not relevant, the important thing is that you can get a pretty good idea of what is the function supposed to do just by looking at it.)

Sandbox Map Exercise

TO DO