Functions 101

Lingo:

name: how you call the function (e.g., Std.rand2f, myFunction)

parameters / **arguments**: things you give to the function to compute

Math.sin(x): you're supplying x as a parameter

Std.rand2(10, z): you're supplying 10 and z (supposedly an int variable)

s.freq(): requires no parameters

return type: the type a function gives you back

Some functions compute a value for you:

Math.sin(x) gives you a floating point value (the sine of x)

Std.rand2(10,z) gives you an integer (randomly chosen between 10 and z)

Some functions do some behavior, and you don't need a value back:

setTempo(…): would return type "void"

Declaring your own functions

The format:

fun *return_type function_name*(*param1_type param1_name, param2_type param2_name, ...)* {

//function body goes here

return *some_object_of_return_type*; //or just return; if return_type is "void"

}

Calling functions

If you're calling a function you've defined, declare the function (as above) somewhere in your code.

Syntax option 1:

function_name(param1_value, param2_value,…);

Syntax option 2 (equivalent):

(*param1_value, param2_value, ...*) => *function_name;*

Note: if the function is associated with an object or class, use the object or class name before the function, followed by a ".":

For example, s.freq() or Std.sin(3.14)

Functions 102

Parameters and scope

The parameter names you use in your function declaration will be treated as **local variables** within the function. Remember learning about local variables inside loops, where something declared within a set of {} brackets isn't accessible from outside those brackets? This is the same thing, except you don't have to declare the parameters separately (the function declaration takes care of this). Of course, you're free to declare other local variables within the function.

Consider the example:

```
fun int add(int x, int y) {
        int z;
        x + y => z;
        return z;
}
```

x, y, and z will not be accessible anywhere outside this function.

Scope and functions (ChucK specific!)

Anything you declare **outside** the function has global scope (unless it is within some other function or loop, or inside {} of any sort). So, you can access it within the function with no problem.

Consider the example:

```
SinOsc s => dac;
fun void set_s_freq(float f) {
        f => s.freq;
}
```

This is totally valid, since s has global scope.

Masking

What if the name of a global variable is the same as the name of a function parameter or other local variable? This is legal, but it causes the local variable to **mask** the global variable. (In fact, if you have nested scopes (nested {} sets), the inner-most variable of that name will be used.)

For example,

```
0 => int x;

{
      5 => int x;
      <<< x >>>;
}
<<< x >>>;
```

will print out

5 :(int)

0 :(int)

Similarly,

```
3 => int x;
<<< x >>>;
stuff(10);

fun void stuff(int x) {
     <<< x >>>;

}
<<< x >>>;
```

will print out

3 :(int)
10 :(int)
3 :(int)

Concurrency
When you call a function, control will pass to the function, and the whole function will execute up to the return statement before control returns to the code that called it.

For example,

```
fun void play_C() {
     SinOsc s => dac;
     Std.mtof(60) => s.freq;
     1::second => now;
     s =< dac;
}

fun void play_Fsharp() {
     SinOsc s => dac;
     Std.mtof(66) => s.freq;
     1::second => now;
     s =< dac;
}
```

```
play_C();
play_Fsharp();
```

will play a C followed by an F#.

ChucK gives you an easy way to execute multiple pieces of code at once using shreds. To **spork** a function (execute it in parallel with the current code), use spork ~*function_name(param1_value, param2_value,…)*

For example,

```
fun void play_C() {
     SinOsc s => dac;
     Std.mtof(60) => s.freq;
     1::second => now;
     s =< dac;
}

fun void play_Fsharp() {
     SinOsc s => dac;
     Std.mtof(66) => s.freq;
     1::second => now;
     s =< dac;
}

spork ~play_C();
spork ~play_Fsharp();
1::minute => now;
```

will play a C and an F# at the same time.

Tip: Using spork launches the function in its own shred, which is a **child** of the main shred. ChucK is designed so that, whenever a parent shred dies (due to no more time advancing), its children also die. Therefore, the 1::minute=> now; line at the end of the main code ensures that the parent won't die before the child shreds execute.