ChucK Assignment 6

15 May 2012

In this lesson you will learn about functions and sporking.

1. Last week we learned how to do concurrency by launching multiple scripts at once. If that seemed a little clunky to you, well it is. There is a much more powerful method of doing concurrency, but first we need to talk about one of the most useful code structures: functions.

1 Functions

A **function** is a subroutine or subprogram. It is a semi-independent part of your code that performs a task. Take a look at the following code.

```
sayHello();

// FUNCTION
fun void sayHello()
{
  <<< "Hello there!" >>>;
}
```

The last four lines define a new function called sayHello.

- The **fun** keyword indicates that we're defining a function.
- **void** indicates that this function doesn't return a value (more on this later.)
- We'll be doing a little more with the parentheses in a moment.
- The first line is where we actually *call* (use) the function.

2. One of the really nice things about functions is they don't have to appear at the top of your code. It's very useful to put them at the end of your script, to keep it uncluttered.

3. Functions are most useful when you call them more than once in the same script. It's easier to read than copy-and-paste.

2 Arguments

4. Let's make this function a little more versatile. This version takes an **argument.** In this case the argument is a string (text):

```
sayHello("Keith");
sayHello("Ty");

fun void sayHello (string myName)
{
  <<< "Hello there", myName >>>;
}
```

5. In the function definition, the argument always needs to be preceded by a type (int, float, dur, string). You can have multiple arguments separated by commas.

```
sayHello("Keith",3);

fun void sayHello (string myName, int num)
{
  repeat (num)
  {
   <<< "Hello there", myName >>>;
  }
}
```

6. I hinted above that a function can return a value. This allows you to create functions that do calculations.

```
add(2,3) => int myNum;
<<< myNum >>>;

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

   There are two changes here. The first is that the type is now **int**, meaning that the output will be an int. The second is the **return** keyword, which tells the function what to output.

7. Here's a more useful function. It takes two integers as inputs and calculates a random float between those numbers, favoring numbers in the middle. (See the triangle distribution from lesson 4.)

```
fun float triRand (int x, int y)
{
   Std.rand2(x,y) => int random1;
   Std.rand2(x,y) => int random2;
 return (random1 + random2) * 0.5;
}
```

   This example illustrates that the output type (float) is independent of the types of the arguments.

8. Another nice feature of functions is that the variables you create in them (including the names of your arguments) exist *only inside that function.* That means that if you use x and y for variables in functions, you can still use them elsewhere without conflict. Functions are like Las Vegas–what happens in functions, stays in functions! But functions also have access to your global variables.

I'm going to use "foo" for this next example. This is one of those classic programming examples that demonstrates how functions encapsulate their variables.

```
3 => int foo; // GLOBAL foo
<<< "foo before function:", foo >>>;
bar();
<<< "foo after function:", foo >>>;

fun void bar ()
{
 5 => int foo; // LOCAL foo
 <<< "foo in function:", foo >>>;
}
```

3 Audio functions

9. You can include audio code in your functions too. Here's a function that plays one note:

```
playNote (60, 1::second);
playNote (62, 500::ms);
playNote (67, 500::ms);

fun void playNote (int midiNote, dur noteLen)
{
 ModalBar m => dac;
 0.25 => m.gain;
 midiNote => Std.mtof => m.freq;
 1 => m.noteOn;
 noteLen => now;
}
```

See how readable and logical it is to encase the details of your code in functions?

4 Spork!

10. Functions can be launched as new threads with the **spork** keyword. Change the first few lines of the above code to:

```
spork ~ playNote (60, 1::second); // Don't forget the tilde
spork ~ playNote (62, 500::ms);
spork ~ playNote (67, 500::ms);
2::second => now; // need to keep parent thread alive
```

We're sporking 3 threads that each consist of a single note. The last line above is needed because the sporked threads are child threads of your main thread, which exit as soon as the parent thread finishes.

11. Let's redo the Reich Phase code, only this time we'll use sporked threads.

```
[64, 66, 71, 73, 74, 66, 64, 73, 71, 66, 74, 73] @=> int pianoPhase[];
spork ~ reich(150::ms);
spork ~ reich(149::ms);
1::minute => now;

fun void reich (dur phaseLen)
{
 ModalBar mb => dac;
 0.5 => mb.gain;
 while (true) // repeat forever
 {
  for (int i; i<12; i++) // loop from 0 to 11
  {
   pianoPhase[i] => Std.mtof => mb.freq;
   1 => mb.noteOn;
   phaseLen => now; // uses argument to set time
  }
 }
}
```

This time the sporked threads are each full independent musical lines. In this example, even though the reich threads are set up to repeat forever, they will close after 1 minute because the main (parent) thread will finish.

12. Another way to use sporking is to create a GLOBAL audio object, and then modify it in threads.

```
adc => PitShift p => NRev r => dac; // GLOBAL
spork ~ pitchWave(0.01);
1::minute => now;

fun void pitchWave (float step)
{
 0 => float x;
 while (true)
 {
  Math.sin(x) => float s; // sine is between [-1,1]
  Math.pow (2, s) => p.shift; 2 raised to the s power
  x + step => x;
  10::ms => now;
 }
}
```

13. It's good to keep the code inside your functions simple. Remember, functions can call other functions! If your function definition starts to get too long, it's probably time to break it up into smaller functions.

Assignment 6: Sporked Polyphony

Make a polyphonic piece with sporked threads. Save your ChucK code as a .ck or .txt file and upload it to the assignment page on Blackboard by this Tuesday, May 22.