ChucK Assignment 1

29 March 2012

Welcome to ChucK! ChucK is a powerful audio language that allows you to control sound synthesis in a very precise manner. In this first lesson you will learn to use oscillators and control their properties directly.

1. If you have not already done so, go to `http://chuck.cs.princeton.edu/release/` and download the **midiAudicle** for your operating system. (Mac 10.7 "Lion" users should download this: `https://ccrma.stanford.edu/~spencer/chuck/miniAudicle-0.2.1-beta-1.tgz`)

   This is the built-in editor for ChucK called the miniAudicle. Open the program and choose **Start Virtual Machine** to launch ChucK. Then type your code into the blank document and click the + button to run it.

2. Let's begin with a simple Sine Oscillator.

   ```
   SinOsc myOsc => dac;
   ```

   - *ChucK is case-sensitive, so be sure to observe the correct capitalization.*
   - *If you forget to end a line with a semicolon you will get an error message. This is a very common mistake.*

3. This creates a Sine Oscillator called **myOsc** and connects it to the **dac**, the *digital-to-analog convertor*. The dac is the only way to get sound output. Let's start by giving it a frequency:

   ```
   440 => myOsc.freq;
   ```

4. The finally necessary element is **time.** Time is ChucK's real specialty; it lets you deal with precise time down to 1 audio sample, and up to 1 week!

   In ChucK you must explicitly specify how long to generate sound. Let's make it last 2 seconds:

   ```
   2::second => now; // <-- notice the 2 colons
   ```

5. Run this (press the + button), and you should get a fabulous sine oscillator for 2 seconds. (You might want to turn down your computer's volume a bit.)

   SinOsc is a built-in audio object; in computer music these are called Unit Generators or UGens (pronounced "You-Jens").[1]

   Think about a hardware oscillator box. It must have some controls on it, right? Probably a volume control, and certainly a frequency control. In ChucK, these controls are called "methods." The volume control is called **gain** and the frequency control is called **freq**. You can find them listed in the ChucK manual on page 133.

6. If you want to change the value of one of the methods, you need to ChucK a value to it.

```
0.5 => myOsc.gain; // half volume
700 => myOsc.freq; // 700 Hz
500::ms => now; // 500 milliseconds = 0.5 seconds
1400 => myOsc.freq; // one octave higher
800::ms => now;
```

7. It's not very practical to always think in frequency. If you want notes from the equal tempered scale, you must convert them using another method, **Std.mtof**. This stands for MIDI-to-frequency. MIDI key 60 is middle C, 61 is C#, etc.[2]

```
65 => Std.mtof => myOsc.freq; // F above middle C
900::ms => now;
```

8. SinOsc is not the only kind of oscillator available in ChucK. Pages 132-135 of the ChucK Manual list several different oscillators, each with its own unique sound, and sometimes with different control methods. Try some different oscillators by replacing the SinOsc at the top of this code with SawOsc, SqrOsc, or any of the others. Don't worry if things don't work or if you get an error. Just experiment with changing this sample code.

Assignment 1: Experiment with Oscillators

Using the code above as a model, experiment with different oscillators and their control methods. Create a melody with at least 10 pitches. Save your ChucK code as a .ck or .txt file and upload it to the assignment page on Blackboard by this Tuesday, April 3.

[1] Although Max and ChucK look very different, they have a common ancestry, Max Mathew's Music III, and "under the hood" they function quite similarly. Both ChucK and Max create sound by connecting UGens together, ending with the dac. Max's SinOsc object is called cycle~ and it functions in a very similar way as ChucK's.

[2] mtof is also a Max object that does the same thing, converting MIDI key numbers to frequency.

ChucK Assignment 2

5 April 2012

In this lesson you will learn about variables, conditional tests, and how to perform repetitive tasks automatically. You'll also explore the properties of the natural harmonic series.

1 Variables

1. We're going to continue to use oscillators for this lesson. Review the techniques from lesson 1 for connecting and controlling oscillators.

2. Sometimes we need to be able to store and recall numbers, times, and other pieces of information. This is what **variables** are for. Do you remember variables from high school algebra? Well, forget them! The concept of variables in computers is completely different.

   - Variables are just *containers*. Each container holds one thing. That thing might be a number, or a duration, or a piece of text, or even an oscillator.
   - In order to use a variable, you first must create it, give it a name, and indicate what kind of information it will hold.
   - The three most common variable types are **int**, **float**, and **dur**, representing integers, floating-point numbers, and durations. We'll discuss more about the differences between these later.

   Here is the code to create a new integer variable called **myNum**:

   ```
   int myNum;
   ```

   Now we can store a value in that variable:

   ```
   25 => myNum;
   ```

   That variable can now be used in place of a simple number in your code:

   ```
   TriOsc myOsc => dac;
   myNum => myOsc.freq;
   ```

3. Here's the cool twist: variables can be changed, and you can use the *old* value of a variable to set the *new* value.

```
int myFrequency;
55 => myFrequency;
<<< myFrequency >>>; // This prints the value on screen
myFrequency + 100 => myFrequency;
<<< myFrequency >>>; // Now it's 155
```

2 Conditional Tests

1. Now that we have numbers in variables, we sometimes want to use them to make decisions. For this we use the **if** command:

```
float myGain;
0.8 => myGain;
if (myGain > 1)
{
  <<< "myGain is greater than 1">>>;
} else
{
  <<< "myGain is not greater than 1">>>;
}
```

2. The **conditional test** is the part inside the parentheses. In this case, myGain is 0.8, so the test *is myGain greater than 1?* is false.

3. The following are possible conditional tests:

```
x > y          is x greater than y?
x < y          is x less than y?
x >= y         is x greater than or equal to y?
x <= y         is x less than or equal to y?
x == y         does x equal y? (notice 2 equal signs)
x != y         is x not equal to y?
```

3 Repetition

1. If you want to do something many times, you could just copy and paste it in your code, but that's not a very efficient way of working. There are several methods of doing repeated things in ChucK. The first is the **repeat** command.

```
repeat(10)
{
  <<<"Hi, nice to meet you! I have no short term memory!">>>;
}
```

2. We can use repeat to create a series or pattern by changing a variable in every repetition.

```
int myNumber;
5 => myNumber;
repeat(10)
{
  <<< myNumber >>>;
  myNumber + 1 => myNumber; // increase it by 1
}
```

There is a useful shortcut for increasing a number by 1: myNumber++

3. You are probably familiar with the natural harmonic series. It is produced by playing frequencies that are all multiples of the same fundamental.

```
TriOsc myOsc => dac;
0.5 => myOsc.gain;
int fundamental;
int harmonic;
55 => fundamental;
1 => harmonic;
repeat(10)
{
  fundamental * harmonic => myOsc.freq; // multiply
  <<< "frequency:",fundamental * harmonic >>>; // print it
  harmonic++; // increase by 1
  250::ms => now;
}
```

EXTRA CREDIT: There are multiple methods of creating the harmonic series. Rewrite the above code to do the same thing but using a different method of calculating the frequency.

4. More complicated than **repeat**, but much more flexible, is the **for** loop. A **for** loop contains a conditional test, so you can decide whether you want to keep looping or not.

   A **for** loop looks like this:

```
for(int num; num < 10; num++)
{
  <<< num >>>;
}
```

See the three different parts in the parentheses?

```
        1          2         3
for(int num; num < 10; num++)
```

- The first part is the *initialization*. This is something done just before it begins looping. In this case, a new **int** called num is created.
- The second part is the *conditional test*. The commands in the brackets will be followed it it's true. If it's false, the looping ends.
- The third part is the *enumeration*. Simply put, it's what happens at the end of every repetition.

You can remember the form of a for loop with the mnemonic ICE:

```
I: Initialization
C: Conditional test
E: End of Every repetition
```

5. Here is the same harmonic series code as above, but now using a **for** loop instead of **repeat**:

```
TriOsc myOsc => dac;
0.5 => myOsc.gain;
int fundamental;
55 => fundamental;
for (1 => int harmonic; harmonic < 11; harmonic++ )
{
  fundamental * harmonic => myOsc.freq;
  250::ms => now;
}
```

Assignment 2: Harmonic Series Etude

Experiment with the harmonic series code above. Using variables and a repeat() or for() loop, create a short etude that explores the harmonic series. Be creative: try to go beyond the simple idea of an ascending arpeggio. Save your ChucK code as a .ck or .txt file and upload it to the assignment page on Blackboard by this Tuesday, April 10.

ChucK Assignment 3

12 April 2012

In this lesson you will explore some of the filters, envelopes, and effects in ChucK, and how to chain them together to produce interesting musical results.

1. You're probably starting to get a little tired of hearing plain oscillators blasting at you for the last couple of weeks. Well, ChucK provides many methods of coloring your sound.

   - **Envelopes** give your sound a dynamic shape so you can control attacks and decays on individual notes.
   - **Filters** amplify and attenuate different frequency ranges to brighten or darken the tone.
   - **Effects** like reverberation, delay, and chorus give you fine control over timbre.

   You need to have your ChucK Manual open for this lesson.

1 ADSR Envelope

2. The ADSR envelope stands for **Attack, Decay, Sustain, Release.** You can find it listed on pages 164-165 in the ChucK Manual.

   - Attack, Decay, and Release are **durations.**
   - Sustain is a **float** value between 0 and 1.

3. To use an envelope, you must place it between the oscillator and the dac:

   ```
   TriOsc myOsc => ADSR myEnv => dac;
   ```

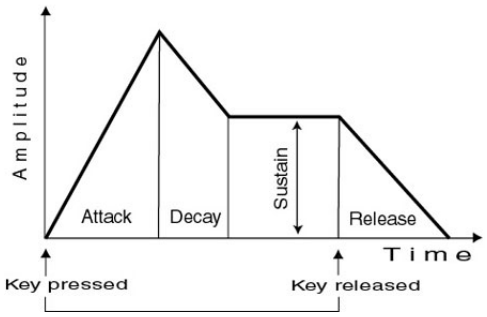

Figure 1: ADSR Envelope

4. Now set up the envelope using the methods on page 165:

```
TriOsc myOsc => ADSR myEnv => dac;
10::ms => myEnv.attackTime; //short attack time
100::ms => myEnv.decayTime;
0.5 => myEnv.sustainLevel;
1000::ms => myEnv.releaseTime; // long release

1 => myEnv.keyOn; // start the attack-decay-sustain
3::second => now;
1 => myEnv.keyOff; // start the release
2::second => now;
```

5. For bell-like or piano-like notes that have no fixed sustain, set the sustainLevel to 0. The release can be arbitrarily short, like 1 millisecond.

```
10::ms => myEnv.attackTime; //short attack time
2000::ms => myEnv.decayTime;
0 => myEnv.sustainLevel;
1::ms => myEnv.releaseTime; // long release

1 => myEnv.keyOn; // start the note
3::second => now;
```

2 Filters

6. Filters change the harmonic spectrum of a sound. They're listed on pages 130-131. They generally have two controls:

   - A main frequency for the filter. For instance, an **LPF** (Low Pass Filter) allows everything below this frequency to be heard, filtering out everything above it.
   - The filter's strength, which is called **Q**.[1] 0 means the filter is off. 1 is moderately strong. Experiment with different Q values to see the effect.

   You can set the frequency and Q separately, or simultaneously with **set**:

```
SawOsc myOsc => ResonZ myFilter => dac;
880 => myFilter.freq;
2 => myFilter.Q;
3::second => now;
myFilter.set(660,10); // 660 Hz, Q=10
3::second => now;
```

[1] No one knows why it's called Q.

3 Effects

7. What sound doesn't benefit from a little reverb? There are a few different reverbs on pages 165-166, each with slightly different sounds.

   - It's most effective to use a reverb on a sound with an envelope.
   - Be sure to allow extra time in your code to allow the reverb to decay

```
SawOsc myOsc => ADSR myEnv => JCRev myReverb => dac;
0.5 => myOsc.gain;
5::ms => myEnv.attackTime;
100::ms => myEnv.decayTime;
0 => myEnv.sustainLevel;
0.2 => myReverb.mix; // Reverb amount
repeat(50)
{
  Std.rand2(400,1200) => myOsc.freq; // random freq between 400 and 1200
  1 => myEnv.keyOn;
  150::ms => now;
}
5000::ms => now; // extra time for reverb
```

8. Experiment with various effects on pages 161-167.

9. Let's try an effect on the sound from your internal microphone instead of an oscillator.

   You should use headphones for this so you don't get feedback.

```
adc => PitShift myPS => dac;
1 => myPS.mix; // all wet sound, no dry
2 => myPS.shift; // up 1 octave (try 0.5 also)
1::minute => now;
```

   Try various effects and filters using adc instead of an oscillator.

Assignment 3: Revisit and Revise

Create a short piece that uses an ADSR envelope and either effects, filters, or both. You may revisit the code you used from assignments 1 or 2, or create something new. Save your ChucK code as a .ck or .txt file and upload it to the assignment page on Blackboard by this Tuesday, April 17.

ChucK Assignment 4

19 April 2012

In this lesson you will explore the Synthesis Toolkit (STK) instruments in ChucK, and learn to use and control randomness.

- **You need to have your ChucK Manual open for this lesson.**

1 STK Instruments

- In the early 1990s Perry Cook began developing the Synthesis Tool Kit (STK) at the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University. It's a collection of software instruments that model physical instruments, or aid in creating synthetic sounds. ChucK includes the STK as UGens. They are listed in the ChucK manual on pages 137-170.
- You may notice that we have already used some of them: ADSR, Echo, PitShift. Today we're going to focus on the physical modeling instruments. These are synthetic instruments that attempt to imitate real instruments, not just in sound but in actual sound production method.

1. Look at the listing for StkInstrument on page 137. All STK instruments share some common parameters:
   - **noteOn** - trigger note on
   - **noteOff** - trigger note off
   - **freq** - set frequency in Hz
   - (We can ignore controlChange, since ChucK give us better controls, as you'll see below.)
2. Try this sample STK code:

```
ModalBar myBar => dac;
440 => myBar.freq;
1 => myBar.noteOn;
1::second => now;
1 => myBar.noteOff;
1::second => now;
```

3. The only significant difference between the above code and the oscillators you were working with before is the .noteOn and .noteOff methods. These are similar to the **ADSR** envelopes we used last week. All STK instruments have their envelopes built-in.
4. In the above example I used ModalBar, one of my favorite STK instruments. Turn to page 146 to read a little more about it. You'll see that there are numerous parameters specific to ModalBar: stickHardness, strikePosition, etc. The **preset** parameter lets you choose from the 9 different Modal Presets listed on page 146.

```
0.15 => myBar.stickHardness;
3 => myBar.preset;
60 => Std.mtof => myBar.freq; //middle C
1 => myBar.noteOn;
1::second => now;
1 => myBar.noteOff;
1::second => now;
```

5. Experiment with several different STK instruments. Don't be disappointed if they don't sound good right away–STK instruments are not meant to be perfect right out of the box! See if you can create interesting new sounds by pushing an instrument to extremes–very low notes on the Flute, for instance.

2 Randomness

6. Computers allow us to control every detail of a musical event, but sometimes that's an overwhelming task. Iannis Xenakis used controlled randomness to determine the low-level details of some of his pieces so he could focus his efforts on the big picture. ChucK has several methods for generating randomness.

   - **Std.rand2(0,10)** returns a random integer between 0 and 10.
   - **Std.rand2f(0,10)** returns a random float between 0.0 and 10.0.
   - **Std.randf()** returns a random float between -1.0 and 1.0.
   - The special keyword **maybe** is sometimes 1, sometimes 0. This is most useful in **if** statements.

7. Randomness is most commonly used to control parameters:

```
TubeBell myBell => dac;

repeat(10)
{
 Std.rand2(72,96) => Std.mtof => myBell.freq;
 1 => myBell.noteOn;
 1500::ms => now;
}
```

8. Randomness can also be used to make decisions. Let's replace the **1500::ms =¿ now;** with a random choice between two times:

```
TubeBell myBell => dac;
repeat(10)
{
 Std.rand2(72,96) => Std.mtof => myBell.freq;
 1 => myBell.noteOn;
 if (maybe) 500::ms => now;
 else 1500::ms => now;
}
```

9. Computer randomness isn't really random at all–it's a fancy algorithm for creating *pseudo-random* numbers. Don't worry–as far as we're concerned, they are good enough! And it gives us one nice advantage–we can choose our own "seed" value, and then every time we use that seed, we get the same repeatable series of random numbers. Try running this a few times:

```
Std.srand(830); // my random seed
repeat(10)
{
 <<< Std.rand2(1, 100) >>>;
}
```

Do you get the same series of "random" numbers each time?

10. The random number generators Std.rand2(), rand2f(), and randf() are designed to create a **flat distribution.** That is, every number is equally likely to occur. But we can gain some control over the randomness with a **weighted distribution.**

11. Let's say we want random numbers over a range of 0-100, but with *low numbers more likely to be chosen than high.* This is a **low-weighted** distribution. One good method is to pick **two** random numbers, then compare them and select the lower of the two:

```
repeat(20)
{
 Std.rand2(0,100) => int a;
 Std.rand2(0,100) => int b;
 if (a<b) <<< a >>>;
 else <<< b >>>;
}
```

Run this a few times. Notice how low numbers are much more common? How would you change this to a **high-weighted** distribution?

12. A **triangular** distribution is weighted toward the middle; that is, over a range of 0-100, numbers close to 50 are most common. To create this distribution, you again choose two random numbers, but now you average them together!

```
repeat(20)
{
 Std.rand2(0,100) => int a;
 Std.rand2(0,100) => int b;
 <<< (a+b)/2 >>>;
}
```

Assignment 4: STK and Randomness

Create a short piece that uses one or more STK instruments and randomness. Save your ChucK code as a .ck or .txt file and upload it to the assignment page on Blackboard by this Tuesday, April 24.

ChucK Assignment 5

10 May 2012

This lesson will discuss using arrays to create pitch collections, concurrency with Machine.add(), and recording to disk.

You've learned much of what ChucK is capable of at the audio level. Now we're going to focus on some techniques to create polyphony (multiple notes at once), recording your ChucK output to an audio file. But first we're goint to talk about arrays.

1 Arrays

- Arrays are variables made up of **ordered lists.** That is, an array allows you to store a list of numbers and access them in order.
- Think of a row of mailboxes for an apartment building. All of the boxes are at the same address, but they have different apartment numbers.

An array is a variable, so it has a type (like int, float, or dur), a name, and a size (the number of "boxes"):

```
int myApartmentBldg[5]; // creates an array with 5 elements
83 => myApartmentBldg[0]; // sets the first element of the array to 83
24 => myApartmentBldg[1]; // sets second element to 24
```

- One of the most common uses of an array is to create a collection of pitches. Take careful note of how this code is laid out:

```
[60, 62, 64, 67, 69] @=> int pentatonic[];
```

There's a new operator above: @=>, called the *assignment operator.* This is a special case where you can't use the usual =>to set something.

- Now that we've created our pentatonic scale, we need to access it. Like so many computer-related things, we start counting from 0, so the first item in the array is **pentatonic[0]**, the second is **pentatonic[1]**, and so on.

```
TubeBell m => dac;
pentatonic[0] => m.freq;
1 => m.noteOn;
1::second => now;
pentatonic[1] => m.freq;
1 => m.noteOn;
1::second => now;
```

- Heads up! The first element in the array is [0]. That means the **last** item is [size-1]. So if you make an array **int num[5]** and try to use num[5] you'll get an ArrayOutOfBounds error. You will make this error. We all do, and often.

- Before we go on, identify the error in each of these four code examples, and suggest how to fix it.

  1. Find the error:

     ```
     int badArray[10];
     0.5 => badArray[0]; // ERROR!
     ```

  2. Find the error:

     ```
     dur timings[10];
     5::second => timings; // ERROR!
     ```

  3. Find the error:

     ```
     [0.8, 0.3, 0.1] => float mallets[]; //ERROR!
     ```

  4. Find the error:

     ```
     float pan[12];
     0.46 => pan[12]; // ERROR!
     ```

- **Arrays** and **for** loops are a classic combination. The **for** loop cycles through the array, retrieving values. The following code is a very typical way to use an array.

```
// REICH MELODY
ModalBar mb => dac;
[64, 66, 71, 73, 74, 66, 64, 73, 71, 66, 74, 73] @=> int pianoPhase[];
for (int i; i<12; i++) // loop from 0 to 11
{
 pianoPhase[i] => Std.mtof => mb.freq; // convert to freq and assign to mb
 1 => mb.noteOn; // start note
 150::ms => now; // pass time
}
```

2 Concurrency

- I have intentionally held off on introducing *concurrency*, the ability to do multiple sounds at once, but I know you're eager to create more complex textures. There are a couple of methods of concurrency, but we're going to start with the simplest. You already know that you can run multiple "shreds" at once in the virtual machine. Well, you can tell your code to do that too.
- You're going to start with a ChucK script that makes sound. Open one of our earlier assignments, or go to the Course Documents page on Blackboard and download concurrency1.ck and concurrency2.ck. **Save them in your home folder, or move them there after you've saved them.**
- Now open a new blank ChucK file in the miniAudicle. Enter the code below and save it in your home folder.

```
Machine.add("concurrency1.ck"); // change the name in quotes if
                                // you're using your own ChucK file
5::second => now;
Machine.add("concurrency2.ck");
```

- Now hit the "add" button and watch as it launches the two files.

3 Recording to audio file

So you want to share your ChucK sounds with others? This requires a new object called **WvOut**, as well as the strange new world of the **blackhole**!

- Open one of your earlier ChucK scripts (not one that runs forever). Add this to the top of your code:

```
dac => WvOut mySound => blackhole;
"prettysounds" => mySound.wavFilename;
```

Hmm... OK, it makes sense that if WvOut records, it should get input from the dac. But what the heck is the blackhole??

Synthesizing audio takes up processor power. The more things running at once, the greater the demands on your computer. But ChucK reduces that as much as it can. You see, if you create this:

```
SinOsc s;
```

ChucK recognizes that it's not connected to the output, so it doesn't even calculate it! We say that *the dac reaches back*, meaning it activates the UGens that output into it.

But you don't want to send the output of WvOut into the dac; you just want it to be activated. That's what the blackhole is for. It's a silent sample calculator. The blackhole activates UGens like the dac does, but it doesn't pass the UGen's audio to the output.

- Now, at the end of your code, add this to close out the audio file:

```
mySound.closeFile();
```

And that's it! Save and run your file and a new audio file will be created in the same directory as your script.

Assignment 3: Piano Phase

Create a version of Steve Reich's Piano Phase using the REICH MELODY code from page 2. Record 1 minute of it using WvFile and upload the audio file to Blackboard by Tuesday, May 15 at 11:59 pm. Feel free to mess around with the melody or timbre, but don't destroy the essential phasing character of the piece.

Tips for realizing Reich's Piano Phase with ChucK

1. You'll need 3 ChucK files, all saved in your home directory. Two of them contain *nearly identical* code (based on the REICH MELODY code from page 2) while the other has the Machine.add() functions.

2. In order for the phasing to occur, you'll need to alter how quickly time passes between notes for one of the REICH MELODY scripts.

3. The REICH MELODY code only plays through the array once. You'll need to create a loop to make it repeat. Put the "main" section of the code in a while(true) loop to make it repeat forever.

4. Since each of the phasing instruments has a gain of 1, when they're both playing the total gain is 2. This will result in unpleasant digital distortion. Reduce the gains of the ModalBar instruments so they don't exceed 1.

5. The WvOut object will be placed in the same file as Machine.add(). At the end of that file, you will advance time by 1 minute (1::minute =>now;). Don't forget to tell ChucK to close the audio file.

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.

ChucK Lesson 7

24 May 2012

This lesson discusses Events and how ChucK responds to external inputs.

ChucK (like Max/MSP, Supercollider, and most other musical computer languages, has two faces.

- The **logical** face of ChucK interprets the instructions in your code.
- The **audio** face of ChucK generates a continuous stream of audio samples.

Here's the thing: ChucK can only do *one or the other* at once! When it's calculating audio, it's not listening to your instructions; and it is forced to cram all of your instructions in a tiny tiny period of time between samples.

1 When is now?

We've been using **now** to pass time, but we haven't really explored how it works. The instruction **10::second =¿ now;** is a built-in shorthand for *"Schedule an event 10 seconds in the future, and then wait until that event happens."* This is a scheduler event, but ChucK allows us to create other types of events as well. An input from a MIDI keyboard, or a keypress from your computer, or a message sent over the internet can be events too.

2 HID Events

HID stands for Human Interaction Device, and it is a blanket term for any of the standard methods we use to interact with our computer: keyboard, mouse/touchpad, joystick, etc.

Here is the code for using the computer keyboard to control ChucK:

```
Hid myHid; // the object to receive the HID data
HidMsg msg; // the data from myHid will get stored in a HidMsg
myHid.openKeyboard (0); // the zero indicates device #0;
        // you may have multiple devices,
        // and you may need to try 1, or 2, etc
while (true) // repeat forever
{
 myHid => now; // wait for keyboard data
 while (hi.recv (msg)) // receive key message
 {
  if (msg.isButtonDown()) <<< "down:", msg.which >>>; // which key
   else <<< "up:", msg.ascii >>>; // ascii is the numeric code
 }
}
```

The ChucK Manual has good examples on using MIDI events as well.

3 Open Sound Control

Open Sound Control, or OSC, is a way to send messages between ChucK and Max, or lots of other programs. You can even send messages to other computers connected on the same wireless network.

OSC is sort of like MIDI, but with a lot of advantages:

- It's faster
- It's wireless capable
- It's capable of carry different kinds of messages

To send an OSC message, you need to know a few things:

1. The IP address of the target computer. (In most cases, you're sending data to yourself, so you use the address "localhost").
2. A port number. This is a number you pick that will be a unique pathway for your data. You can safely choose any number between 10000 and 65535.
3. A message ID or tag. These start with a slash and are descriptive: /chuck or /notedata
4. The data types you're sending. Are you sending two floats? Two ints and a string?
5. And finally, of course, your data!

oscsend: localhost 6449 /demo ff 20.5 0.33

4 Receiving OSC in ChucK

```
OscRecv osc;
6449=> osc.port;
osc.listen();
osc.event("/demo, ff") @=> OscEvent myOSC;

while (true)
{
  myOSC => now;
  while ( myOSC.nextMsg() != 0 ) // looks for multiple msgs
  {
   myOSC.getFloat() => float val1;
   myOSC.getFloat() => float val2;
  }
  // here, do something with the data you received
}
```

Try the webcam example on Blackboard.