MUSC 208 Winter 2014
John Ellinger, Carleton College

Lab 8 Arrays, Classes, Time Sync, and STK Instruments

Note And Rhythm Array

You want to play a melody in ChucK and you already know the notes and rhythms. There are several ways you could structure the note and rhythm data.

The notes are the 8 notes of the C major scale as MIDI note numbers. The rhythm is represented by the numbers 1-8. In your code the rhythm would be a number (float) that represents time, duration, or multiples/fractions of a beat.

Method 1 - Alternating Pairs

We'll first look at a one dimensional array that alternates the scale notes and the fake rhythm numbers.

```
// notesNrhythm1
// John Ellinger, Music 208, Spring2013

// This array alternates MIDINoteNumber-rhythm
[ 60, 1, 62, 2, 64, 3, 65, 4,
67, 5, 69, 6, 71, 7, 72, 8 ] @=> int notesNrhythm[];

//print out all numbers
<<< "====== for loop ======" >>>;
for ( 0 => int ix; ix < notesNrhythm.cap(); ix++ )
{
    <<< "notesNrhythm[", ix, "] =", notesNrhythm[ix] >>>;
}

//another way to print out all numbers
<<< "====== while loop ======" >>>;
0 => int ix;
while ( ix < notesNrhythm.cap() )
{
    <<< "notesNrhythm[", ix, "] =", notesNrhythm[ix] >>>;
    ix++;
}

// still another way to print out all numbers
<<< "====== do while loop ======" >>>;
0 => int jx;
do
{
    <<< "notesNrhythm[", jx, "] =", notesNrhythm[jx] >>>;
    jx++;
} while ( ix < notesNrhythm.cap() );

```

Exercise 1

```
// add code to print notes only
<<< "====== Notes only ======" >>>;
```

Excercise 2

```
// add code to print notes only
<<< "====== Rhythms only ======" >>>;
```

Exercise 3

```
add code to print this
60 1
62 2
64 3
65 4
67 5
69 6
71 7
72 8
// add code to print note rhythm pairs
<<< "====== Note Rhythm Pairs ======" >>>;
```

Method 2 - Multidimensional Array

You could also use a multi dimensional array where elements of the larger array are individual two element arrays.

[…] outer array encloses [n, r] inner arrays
[[n, r], [n, r], [n, r], [n, r], …] multiDimensional array

Chuck array definition syntax

```
[ [60,1], [62,2] ] @=> int nr[][]; // nr[1][0] = 62 and nr[0][1] = 1
```

Method 2 Code

```
// notesNrhythm2
// John Ellinger, Music 208, Spring2013

// This is an array of arrays.
// Two element [ note, rhythm ] arrays are elements
// of a container array.
// Elements are [MIDINoteNumber, rhythm]
[
  [60, 1],
  [62, 2],
  [64, 3],
  [65, 4],
  [67, 5],
  [69, 6],
  [71, 7],
  [72, 8]
] @=> int notesNrhythm[][];


// print all elements
<<< "====== all elements ======" >>>;
<<< "notesNrhythm.cap() = ", notesNrhythm.cap() >>>;
for ( 0 => int ix; ix < notesNrhythm.cap(); ix++ )
{
    <<< notesNrhythm[ix][0], notesNrhythm[ix][1] >>>;
}

```

Exercise 4

```
// add code to print notes only
<<< "====== Notes only ======" >>>;
```

Exercise 5

```
// add code to print rhythms only
<<< "====== Rhythms only ======" >>>;
```

Chuck Classes

A class is an object that contains data and methods that operate on that data. We've already used many ChucK classes like SinOsc, Impulse, NRev, etc.

A ChucK class encapsulates variables and functions. For example

```
class MathClass
{
    // variables go here
    "ERROR DIVIDE BY ZERO exiting" => string errorStr;

    // functions used by the class go here
    function float add2( float n1, float n2 ) {
        return n1 + n2;
    }

    function float mult2( float n1, float n2 ) {
        return n1 * n2;
    }

    function float divide2( float n1, float n2 ) {
        if ( n2 == 0 ) {
            <<< errorStr, "" >>>;
            me.exit();
        } else
          return n1 / n2;
  }
}

// TESTING
MathClass mc; // create an instance of the class
<<< "add2", mc.add2( 4, 3.33 ) >>>;
<<< "mult2", mc.mult2( 25, -25 ) >>>;
<<< "divide2", mc.divide2( 100, 4.25 ) >>>;
<<< "divide2", mc.divide2( 100, 0 ) >>>;
```

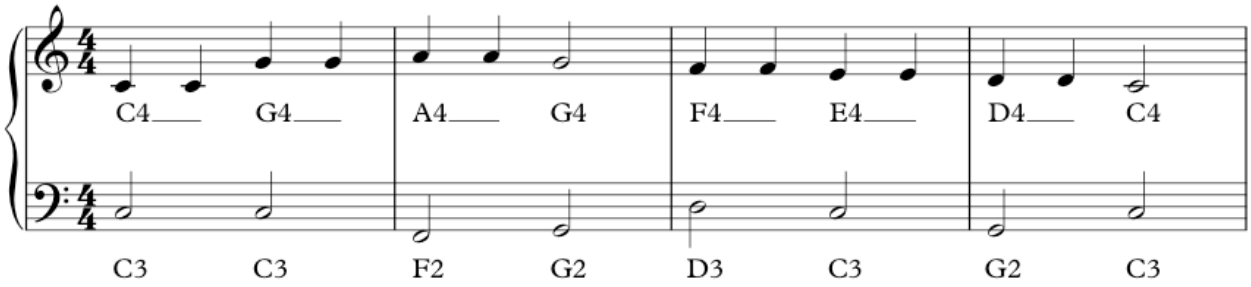
Class vs. Array

A major difference between classes and arrays is that classes can contain mixed data types. All elements in an array must all be the same type.

Method 3 - Note and Rhythm Class

```
// notesNrhythm3
// John Ellinger, Music 208, Spring2013

// Create a class to hold data of different types
class NoteRhythmClass
{
    int note;
    float rhythm;

    function void init( int n, float r )
    {
        n => note;
        r => rhythm;
    }
}

// create an Array class to hold NoteRhythm classes
// in the len variable
class NoteRhythmArrayClass
{
    // we know we have less than 256 notes
    NoteRhythmClass @ nr[ 256 ];

    // keep track of the number of elements added to the class
    int length;

    function void add( int n, float r )
    {
        // use the new keyword to create a new element in the array
        new NoteRhythmClass @=> nr[ length ];
        nr[ length ].init( n, r);
        // update length for next element to be added
        length++;
    }
}

NoteRhythmArrayClass nrArray;
nrArray.add( 60, 1 );
nrArray.add( 62, 2);
nrArray.add( 64, 3);
nrArray.add( 65, 4);
nrArray.add( 67, 5);
nrArray.add( 69, 6);
nrArray.add( 71, 7);
nrArray.add( 72, 8);

// use length, not .cap()
for ( 0 => int ix; ix < nrArray.length; ix++ )
{
    <<< nrArray.nr[ix].note, nrArray.nr[ix].rhythm   >>>;
}
```

Twinkle Twinkle, Melody And Bass In Stereo



Code Outline

1. Define rhythm values
2. Define MIDI note numbers
3. Create the Note-Rhythm two dimensional array for melody
4. Create the Note-Rhythm two dimensional array for bass
5. Define Tempo related variables.
   gTempo is beats per minute
   gSecondsPerBeat is the duration of one beat
6. Write the setTempoVariables(float tempo) function
7. Write the playClarinetMelody(int repeats) function
8. Write the playMoogieBass(int repeats) function
9. Test

Save twinkleLab8.ck. You'll need it for Homework 7-8.

```
// twinkleLab8.ck
// John Ellinger, Music 208, Winter 2014

// STEP 1 Rhythm values half note = 2 quarter notes
1.0 => float nQ;      // quarter note
nQ * 2 => float nH;       // half note

// STEP 2 MIDI note numbers for melody
60.0 => float C4;
62.0 => float D4;
64.0 => float E4;
65.0 => float F4;
67.0 => float G4;
69.0 => float A4;
// MIDI note numbers for bass
41.0 => float F2;
43.0 => float G2;
48.0 => float C3;
50.0 => float D3;
```

```
// STEP 3 melody array first four measures
[
[C4, nQ], // m1
[C4, nQ],
[G4, nQ],
[G4, nQ],
[A4, nQ],
[A4, nQ],
[G4, nH],
[F4, nQ], // m2
[F4, nQ],
[E4, nQ],
[E4, nQ],
[D4, nQ],
[D4, nQ],
[C4, nH]     // no comma
] @=> float melody[][]; // @=> used for arrays

// STEP 4 bass array first four measures
[
[C3, nH],// m1
[G2, nH],
[F2, nH],
[G2, nH],
[D3, nH],// m2
[C3, nH],
[G2, nH],
[C3, nH]     // no comma
] @=> float bassLine[][]; // @=> used for arrays

// STEP 5 global variables prefixed with g ( a common practice )
60 => float gTempo;       // beats per minute
1.0::second => dur gSecondsPerBeat;    // seconds per beat
0.75 => float noteOnTime;

// STEP 6 this function sets gTempo and calculates gSecondsPerBeat
function void setTempoVariables( float tempo )
{
    tempo => gTempo;
    <<< "\t", gTempo >>>;
    gTempo / 60.0 => float beatsPerSecond;
    ( 1.0 / beatsPerSecond )::second => gSecondsPerBeat;

    // time sync formula
    gSecondsPerBeat - ( now % gSecondsPerBeat ) => now;
}
```

```
// STEP 7
function void playClarinetMelody( int repeats )
{
    Clarinet clar => JCRev rev => dac.left;
    clar.gain( 0.3 );

    0 => int count;
    do
    {
        for ( 0 => int ix; ix < melody.cap(); ix++ )
        {
            Std.mtof( melody[ix][0] ) => clar.freq;
            clar.noteOn( 0.8 );
            melody[ix][1] * gSecondsPerBeat * noteOnTime => now;
            clar.noteOff( 0.2 );
            melody[ix][1] * gSecondsPerBeat * ( 1-noteOnTime ) => now;
        }
        count++;
    } while (count < repeats );
}

// STEP 8
function void playMoogieBass( int repeats )
{
    Moog moogie => JCRev rev => dac.right;
    0.9 => moogie.gain;

    0 => int count;
    do
    {
        for ( 0 => int ix; ix < bassLine.cap(); ix++ )
        {
            Std.mtof( bassLine[ix][0] ) => moogie.freq;
            moogie.noteOn(0.8);
            bassLine[ix][1] * gSecondsPerBeat * noteOnTime => now;
            moogie.noteOff( 0.2 );
            bassLine[ix][1] * gSecondsPerBeat*( 1-noteOnTime ) => now;
        }
        count++;
    } while (count < repeats );
}

// STEP 9 Test and debug
setTempoVariables( 100 );
spork ~ playClarinetMelody( 10 );
spork ~ playMoogieBass( 10 );
2::minute => now;  // so shreds can run
```

Add Keyboard Code To Control The Tempo

We'll use code you've seen before to make the up and down arrows increase and decrease the tempo.

```
// STEP 10
Hid hi;
HidMsg msg;
you_find_it => int upArrowCode;
you_find_it => int downArrowCode;

// which keyboard
0 => int device;
// get from command line
if( me.args() ) me.arg(0) => Std.atoi => device;

// open keyboard (get device number from command line)
if( !hi.openKeyboard( device ) ) me.exit();
<<< "keyboard '" + hi.name() + "' ready", "" >>>;

// define the amount the tempo changes in response to the arrow keys
10 => int tempoDelta;
30 => int MIN_TEMPO;
300 => int MAX_TEMPO;

function void handleKeyboard()
{
    gTempo => float tempo;
    // infinite event loop
    while( true )
    {
        // wait on event
        hi => now;

        // get one or more messages
        while( hi.recv( msg ) )
        {
            // check for action type
            if( msg.isButtonDown() )
            {
                if ( msg.which == upArrowCode )
                {
                    tempo + tempoDelta => tempo;
                    if ( tempo > MAX_TEMPO )
                        MAX_TEMPO => tempo;
                    setTempoVariables( tempo );
                }
                else if ( msg.which == downArrowCode )
                {
                    tempo - tempoDelta => tempo;
                    if ( tempo < MIN_TEMPO )
                        MIN_TEMPO => tempo;
```

```
                    setTempoVariables( tempo );
                }
                // <<< "down:", msg.which, msg.key, msg.ascii >>>;
            }
            else
            {
                // <<< "up:", msg.which, msg.key, msg.ascii >>>;
            }
        }
    }
}
```

Copy Step 9 To Here And Add The Handlekeyboard() Function

```
// STEP 9 Test and Debug
// handleKeyboard() function added
setTempoVariables( 100 );
spork ~ handleKeyboard();
spork ~ playClarinetMelody( 10 );
spork ~ playMoogieBass( 10 );

2::minute => now;  // so shreds can run
```

Test and Debug

Run the program and test the tempo controls. You should be able to hear Twinkle respond to up arrow/down arrow tempo changes.

Time Synchronization

Observe ChucK Time in the Console Monitor

This example counts in seconds. You should see decimal places in the output.

```
// timeSync1
// John Ellinger, Music 208, Spring2013

while ( true )
{
    1000::ms => now;
    <<< now / second >>>;
    // Console Monitor will show seconds with decimal places
 }
```

Observe Chuck Synchronized Time

Have ChucK synchronize the time for you. Line 6 does the magic.

```
// timeSync2
// John Ellinger, Music 208, Spring2013

// this is the magic line
1000::ms => dur beat;
beat - ( now % beat ) => now;

while ( true )
{
    beat => now;
    <<< now / second >>>;
    // no decimals
}
```

Example Non Synchronized Sound

This example uses an Impulse and a BiQuad filter to produce "blips".

```
// timeSync3
// John Ellinger, Music 208, Spring2013

Impulse imp => BiQuad bq => dac;
.9999 => bq.prad;
2000 => bq.pfreq;

1000::ms => dur beat;

while (true)
{
    0.4 => imp.next; // amplitude
    beat => now;
}
```

See if you can get the clicks to line up when you repeatedly click the Add shred button. It's almost impossible.

Example Synchronized Sound

```
// timeSync4
// John Ellinger, Music 208, Spring2013

Impulse imp => BiQuad bq => dac;
.9999 => bq.prad;
2000 => bq.pfreq;

// Calculate tempo
83 => float tempo; // in beats per minute
(60 / tempo)::second => dur beat; // in seconds per beat

// this is the magic line
beat - ( now % beat ) => now;

while (true)
{
    0.1 => imp.next; // amplitude
    beat => now;
}
```

Repeatedly click the Add shred button. The clicks get louder but remain in sync. They get louder because the amplitudes of each added shred are additive.

Biquad Filter Experiments

http://chuck.cs.princeton.edu/doc/program/ugen_full.html#BiQuad

[ugen]: BiQuad

- *STK biquad (two-pole, two-zero) filter class.*

```
This protected Filter subclass implements a
two-pole, two-zero digital filter.  A method
is provided for creating a resonance in the
frequency response while maintaining a constant
filter gain.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.
```

(control parameters)

- **.b2** - (float , READ/WRITE) - *b2 coefficient*
- **.b1** - (float , READ/WRITE) - *b1 coefficient*
- **.b0** - (float , READ/WRITE) - *b0 coefficient*
- **.a2** - (float , READ/WRITE) - *a2 coefficient*
- **.a1** - (float , READ/WRITE) - *a1 coefficient*
- **.a0** - (float , READ only) - *a0 coefficient*
- **.pfreq** - (float , READ/WRITE) - *set resonance frequency (poles)*
- **.prad** - (float , READ/WRITE) - *pole radius (less than 1 to be stable)*
- **.zfreq** - (float , READ/WRITE) - *notch frequency*
- **.zrad** - (float , READ/WRITE) - *zero radius*
- **.norm** - (float , READ/WRITE) - *normalization*
- **.eqzs** - (float , READ/WRITE) - *equal gain zeroes*

Experiment With The BiQuad filter.

```
// timeSync5
// John Ellinger, Music 208, Spring2013

Impulse imp => BiQuad bq => Gain g => dac;
.9999 => bq.prad;
2000 => bq.pfreq;
0.2 => g.gain;

1000::ms => dur beat;
// Add and remove shreds
// uncomment the next line for synchronized shreds
// beat - ( now % beat ) => now;

while (true)
{
    Math.random2( 100, 300 ) => int millis;
    Math.random2( 300, 3000 ) => bq.pfreq;
    Math.random2f( .9900, .999999 ) => bq.prad;
    Math.random2f( 0.1, 0.7 ) => imp.next;
    millis::ms => now;
}
```

Random Mandolin In Sync

```
// randomMandolinInSync John Ellinger, Music 208 Spring2013

// Add a reverb to the mix
Mandolin m => NRev reverb => Pan2 p2 => dac;
0.3 => m.gain;
// the reverb wet/dry mix  0 is no reverb, 1.0 is full reverb
0.2 => reverb.mix;

// quarter note in milliseconds, Tempo = 120
500 => float t4;
t4 / 2 => float t8;
t8 / 2 => float t16;
t4::ms => dur beat;

// create an array of pentatonic scale notes
//G   A   C   D   E   G   A   C
[ -5, -3, 0, 2, 4, 7, 9, 12 ] @=> int notes[];

function void playMandolin( int startingC )
{
    dur millis;

    // choose a random index into the notes array
    Math.random2( 0, notes.cap()-1) => int ix;
    // select that note
    Std.mtof( notes[ix] + startingC )  => m.freq;
    // set a random plucking position
    Math.randomf() => m.pluckPos;

    Math.random2(1,100) => int guess;
    if ( guess < 11 )  // 10% chance
        t4::ms => millis;
    else if ( guess > 70 ) // 30% chance
        t8::ms => millis;
    else
        t16::ms => millis;  // 60% chance
    // Play it
    m.pluck(1);
    millis => now;
}

beat - ( now % beat ) => now;
while (true)
{
    0 => p2.pan;
    playMandolin( 60 );

}
```

Try This

Remove all shreds. Change these parameters and add these three shreds.

Shred 1

```
// change ms tempo in line 12 to 2000
0 => p2.pan; // line 54
playMandolin( 48 ); // line 55
```

Shred 2

```
// change ms tempo in line 12 to 1000
-1 => p2.pan; // left speaker
playMandolin( 60 );
```

Shred 3

```
// change ms tempo in line 12 to 500
1 => p2.pan; // right speaker
playMandolin( 72 );
```

STK Instruments

STK is an acronym for Synthesis Tool Kit written by Perry Cook and Gary Scavone. Perry Cook along with Dan Trueman who you met in class were Ge Wang's PhD advisors at Princeton. ChucK is the result of Ge Wang's PhD dissertation.

All STK instruments are created using Physical Modeling Synthesis. The sounds are constructed using mathematical models of physics sound waves in a tube, along a string, on two dimensional membranes, etc. The math is complicated often involving second order differential equations.

Open and play STK_test.ck from the m208Lab8 download folder.

Run it. Several instruments have a limited range and sound terrible outside of that range.