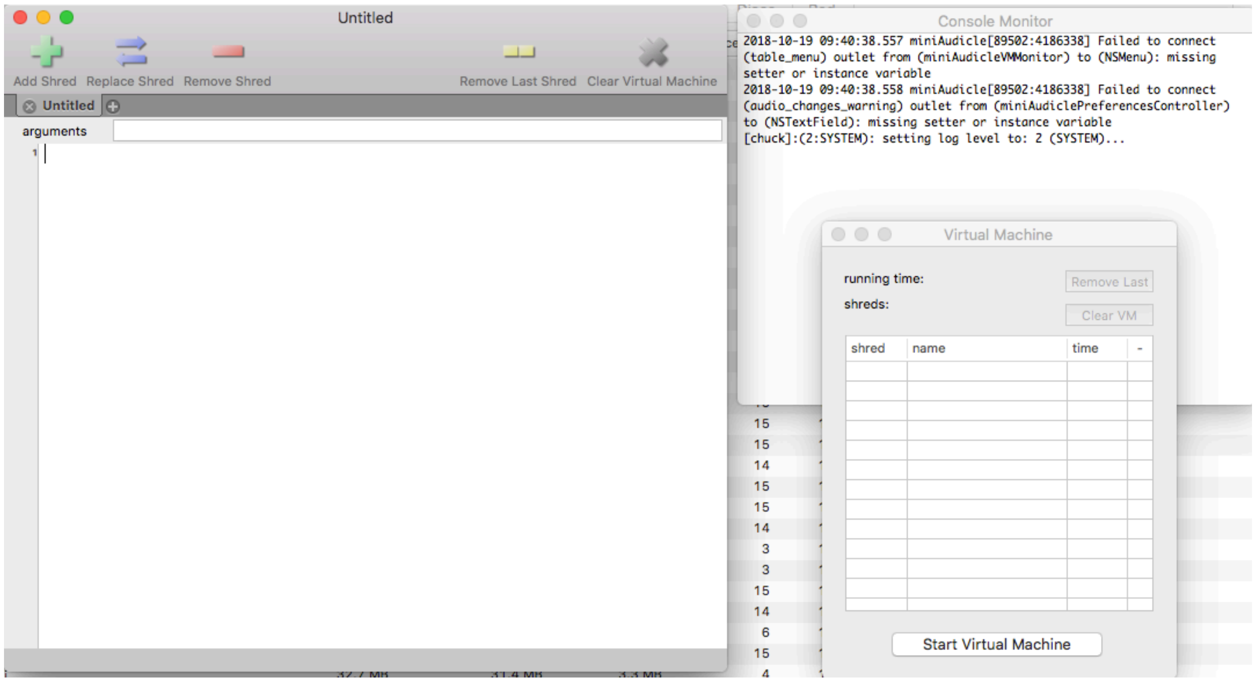
ChucK Labs

My Friend for Programming Music

ChucK is a programming language for Sound and Music, for can develop electronic music with it, we will use miniAudicle as IDE (Interface Development Environment) for programming, test and run our code.



```
2018-10-19 09:40:38.557 miniAudicle[89502:4186338] Failed to connect
(table_menu) outlet from (miniAudicleVMMonitor) to (NSMenu): missing
setter or instance variable
2018-10-19 09:40:38.558 miniAudicle[89502:4186338] Failed to connect
(audio_changes_warning) outlet from (miniAudiclePreferencesController)
to (NSTextField): missing setter or instance variable
[chuck]:(2:SYSTEM): setting log level to: 2 (SYSTEM)...
```

1. Hello World! with A middle frequency

We will say Hello World! with chuck reproducing the sound of A middle (440 Hertz) using the Sine Wave Oscillator.

First, in the miniAudicle editor, we will start creating two variables of float type to represent our frequency and gain, we need to add the next code:

```
440 => float frequency;
0.6 => float gain;
```

Second, we will define a Sine Wave Oscillator and assigned it to the DAC (Digital-to-Analog Converter), we need to add the next code:

```
SinOsc wave => dac;
```

To reproduce a sound we need to define the frequency of it and the gain (volume), for that we will add the next code:

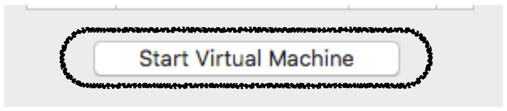
```
frequency => wave.freq;
gain => wave.gain;
```

Finally, we will establish how many time we will reproduce the sound and we will do it now, we will add the next code:

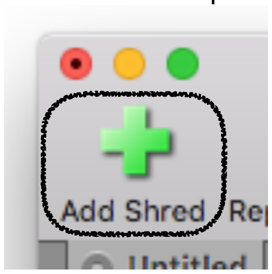
```
3::second => now;
```

Running our Hello World!

We need to start the Virtual Machine, something that we can do clicking the “Start Virtual Machine” button.



Then, we will add our code to the Virtual Machine stack clicking “Add Shared” button, the sound will be reproduced.



2. Playing the same note (A) in different octaves

Do you know what makes that we can call notes in different octaves with the same name instead when they sound different (but similar)?

Exist an interesting relationship between notes with the same name but in different octaves, is a mathematical relationship between the note's frequency, in each octave the frequency doubles. So, for example, A middle has a frequency of 440 Hz, the A in the above octave has a frequency of 880 Hz, and the A in the under octave has a frequency of 220 Hz.

In this lab, we will play 5 different A note starting at 110 frequency, and we will start in a high gain and decrease it in each note played.

First, create a new project and then define a Sine Wave Oscillator and assigned it to the DAC (Digital-to-Analog Converter), we need to add the next code:

```
SinOsc wave => dac;
```

Now, we will play our first A with a frequency of 110 Hz, a gain of 0.9, and we will play it during 1 second, we need to add the next code:

```
110 => wave.freq;
0.9 => wave.gain;
1::second => now;
```

We will repeat this process four more times, but now with:

- a frequency of 220 and a gain of 0.8
- a frequency of 440 and a gain of 0.7
- a frequency of 880 and a gain of 0.6
- a frequency of 1760 and a gain of 0.5

all then will be played during 1 second, the rest of the code will look like the next:

```
220 => wave.freq;
0.8 => wave.gain;
1::second => now;

440 => wave.freq;
0.7 => wave.gain;
1::second => now;

880 => wave.freq;
0.6 => wave.gain;
1::second => now;

1760 => wave.freq;
0.5 => wave.gain;
1::second => now;
```

Finally, add it to the Virtual Machine Stack and listening to the result.

3. Playing with the others waveforms oscillators

Now is time to play with the other oscillators and listening to the difference between the sound produced for the same note. We will change the Oscillator type we are assigning to the DAC.

```
SinOsc wave => dac;
```

Instead to use the ***SinOsc*** use:

- ***SqrOsc*** for Square Oscillator
- ***TriOsc*** for Triangle Oscillator
- ***SawOsc*** for Sawtooth Oscillator

Don't forget to add each test to the Virtual Machine Stack for taste after changing the Oscillator type.

4. Generating random sounds with random gains

First, create a new project and then define a Sine Wave Oscillator and assigned it to the DAC (Digital-to-Analog Converter), we need to add the next code:

```
SinOsc wave => dac;
```

Now, we will create an infinite loop to produce random sounds with random gains during the track stay in the virtual machine stack, we need to add the next code:

```
while( true ){

}
```

Now is time to generate random values, for that we will use one of the standard libraries that ChucK provides to us, the Math class (you can find a few documentation about some standard libraries in the next link http://chuck.cs.princeton.edu/doc/program/stdlib.html). Math class has the *random2f* method, with which one we can obtain float random numbers and define the range of numbers that we will use to generate them.

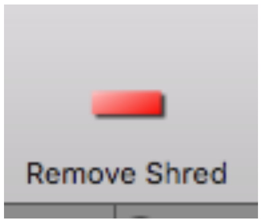
For generate a random frequency value, we will use a range between 220 to 880, and for the random gain value, we will use a range between 0.2 to 0.8, we will add the next lines of code inside the while loop.

```
Math.random2f(220, 880) => wave.freq;
Math.random2f(0.2, 0.8) => wave.gain;
```

Finally, we will be established our sound will have a duration of a quarter second (0.25 seconds), we will add the next code behind the random generation.

```
0.25::second => now;
```

Don't forget to add it to the Virtual Machine Stack for taste it. You can change the Oscillator type and test how sound. He will be playing always while the machine is on, and you can add many tracks, to remove them to the Virtual Machine stack, you only need to click the "Remove Shared" button.



5. Converting MIDI notes

In this lab, we will generate random MIDI notes and will transform them into their frequency value. For that, we will use the *mtof* method in the Std class, which receives a MIDI note.

First, create a new project and then define a Sine Wave Oscillator and assigned it to the DAC (Digital-to-Analog Converter), we need to add the next code:

```
SinOsc wave => dac;
```

Second, we will define a variable of int type in which one we will save the random MIDI note generated inside the infinite loop, we need to add the next code for it:

```
0 => int midiNote;
```

Third, we will create an infinite loop to produce random sounds with random gains during the track stay in the virtual machine stack, we need to add the next code:

```
while( true ){

}
```

Now is time to generate random values, but this time we need int values not float ones like we generated for frequency and gain at lab four. We will use the random2 method in the Math class, and we will establish a range between 60 to 71 that correspond to the MIDI notes between C middle to B middle. Inside the while cicle we will add the next code:

```
Math.random2(60, 71) => midiNote;
```

Now that we have our random MIDI note assigned to the midiNote variable, we need to transform them to frequency for can reproduce them. For that, we will use the *mtof* method at Std class, we will add the next code:

```
Std.mtof(midiNote) => wave.freq;
```

Finally, we will use the same code we used on Lab 4 to generate random gains and establish a quarter second to each sound.

```
Math.random2f(0.2, 0.8) => wave.gain;
0.25::second => now;
```

Don't forget to add it to the Virtual Machine Stack for taste it. You can change the Oscillator type and test how sound.

6. Playing Twinkle - Using arrays

First, create a new project and then define a Sine Wave Oscillator and assigned it to the DAC (Digital-to-Analog Converter), we need to add the next code:

```
SinOsc wave => dac;
```

Second, we will define our MIDI notes array. We will call it notes, and we will start defining it as an empty array of int types, we need to add the next code for it:

```
[ ] @=> int notes[];
```

Next step is to add MIDI notes to the array, here are the tables with the notes and MIDI notes.

C	C	G	G	A	A	G	F	F	E	E	D	D	C
60	**60**	**67**	**67**	**69**	**69**	**67**	**65**	**65**	**64**	**64**	**62**	**62**	**60**

G	G	F	F	E	E	D	G	G	F	F	E	E	D
67	**67**	**65**	**65**	**64**	**64**	**62**	**67**	**67**	**65**	**65**	**64**	**64**	**62**

C	C	G	G	A	A	G	F	F	E	E	D	D	C
60	**60**	**67**	**67**	**69**	**69**	**67**	**65**	**65**	**64**	**64**	**62**	**62**	**60**

After transcribing the notes to our array, our code will look like the next:

```
[60, 60, 67, 67, 69, 69, 67, 65, 65, 64, 64, 62, 62, 60,
 67, 67, 65, 65, 64, 64, 62, 67, 67, 65, 65, 64, 64, 62,
 60, 60, 67, 67, 69, 69, 67, 65, 65, 64, 64, 62, 62, 60] @=> int notes[];
```

Now we need to reproduce each note stored in the notes array, for that we will create a for loop for with a counter that will start at 0, and will increment in 1 until it is N. If we don't know the value of n, arrays in ChucK has the *cap* method, that returns the N value (amount of elements in the array). We need to add the next code for it:

```
for(0 => int i; i < notes.cap(); i++){

}
```

The counter i is saving the index to which we are going to access in the array, the note that follows. We will obtain the note using the index, then we will convert it to a frequency and finally, we will assign it to the wave.freq. We need to add the next code for it:

```
Std.mtof(notes[i]) => wave.freq;
```

Finally, we only need to define the gain and how many time will be the duration of each sound (try using a half second, 0.5).

```
0.8 => wave.gain;
0.5::second => now;
```

Don't forget to add it to the Virtual Machine Stack for taste it. You can change the Oscillator type and test how sound.

7. Playing Twinkle - Remarking every note

Our current code makes when we play the same note consecutively sounds like we are playing it only once, but with more duration. To avoid it we only need to add a shortest silence between each note. To do that we will establish the frequency with 0, and will reproduce it during one-tenth of a second (0.1). We need to add the next code for it:

```
0.0 => wave.freq;
0.1::second => now;
```

Don't forget to add it to the Virtual Machine Stack for taste it. You can change the Oscillator type and test how sound.

8. Playing Twinkle - Representing figures

We have two types of figures in Twinkle, quarter and half. We will create two variables, one for represent the duration of a quarter (0.5 seconds), and other for representa the duration of half (two quarters), we need to add the next code for it:

```
0.5 => float qt; //quarter
qt*2 => float hw; //half
```

The next step is create an array for save the duration of every note, for the notes that have a quarter figure we will use the qt variable, and for the notes that have a half figure we will use the hw variable, only the last note will be played for a longer time (for made the end), like 1.5 seconds). We need to add the next code for it:

```
[qt, qt, qt, qt, qt, qt, hw, qt, qt, qt, qt, qt, qt, hw,
 qt, qt, qt, qt, qt, qt, hw, qt, qt, qt, qt, qt, qt, hw,
 qt, qt, qt, qt, qt, qt, hw, qt, qt, qt, qt, qt, qt, 1.5]
 @=> float duration[];
```

Now we can change the following code:

```
0.5::second => now;
```

For the next one, in which we are specifying the duration of each note through the duration array.

```
duration[i]::second => now;
```

Don't forget to add it to the Virtual Machine Stack for taste it. You can change the Oscillator type and test how sound.

9. Playing Twinkle - Handle dynamics

To finish these labs, we will add dynamics, basically we will play each note with a different gain. You can try different values (remember the gain can be a value between 0.1 to 1). The next code is our suggestion, we need to have one gain per note.

```
[.3, .3, .6, .6, .7, .7, .6, .5, .5, .4, .4, .3, .3, .2,
 .6, .6, .5, .5, .4, .4, .3, .6, .6, .5, .5, .4, .4, .3,
 .3, .3, .6, .6, .7, .7, .6, .5, .5, .4, .4, .3, .3, .2] @=> float gains[];
```

After add the gains array, we will change the following code:

```
0.8 => wave.gain;
```

and will replace it for the next one, we we indicate the gain through the gains array.

```
gains[i] => s.gain;
```

Don't forget to add it to the Virtual Machine Stack for taste it. You can change the Oscillator type and test how sound.