Core Reflection

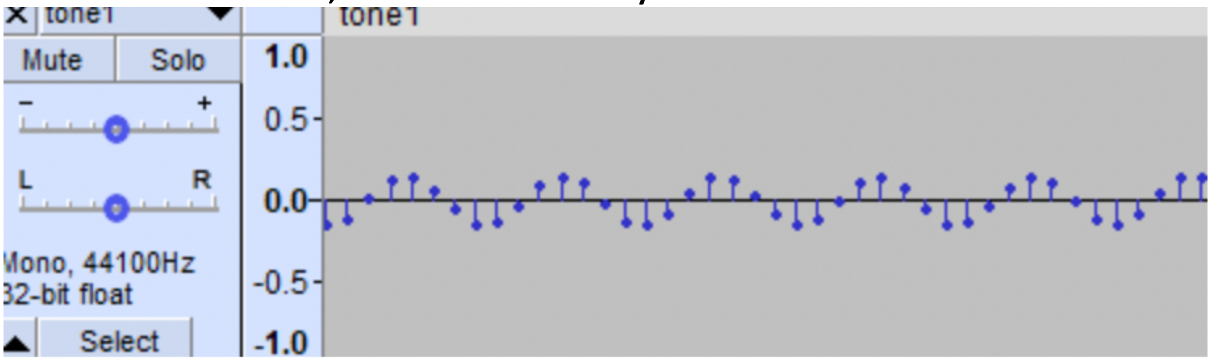
Reason for selected sampling rate.
The sampling rate I used is 44100 due to the Nyquist Theorem. This shows that a periodic signal must be sampled at least twice at the highest frequency of the signal. To have a high frequency in digital form you need to sample twice as fast as the high frequency on the signal. The human ear can hear acoustic signals up to 20,000 Hz which is fundamentally the highest frequency component. This is the reason why I have doubled it as this is the normal sampling rate in music also.

How you calculated total number of samples
Total number of samples *p.s* x number of seconds the wave runs for.

Reasoning for selection of the types of all variables

- Duration I put at 5 seconds. It is the core of the assignment, so I didn't produce a long sample.
- Sample Rate I put at 44100 since it is the normal sampling rate for music.
- Int – "n_samples" – number of samples is an integer.
- Double – "dt" – dt is a fractional number (very small)
- Dt – taken sound window of 1 second and divided it by number of samples (44100)
- Frequency is 6000hz. Comfortable frequency for hearing
- Volume is 5000hz. Easy volume to hear sound
- Waveform – utilizes air pressure as a function for the time equation. Exchanged suitable values into formula.

Screenshot of waveform, does it look like what you intended?



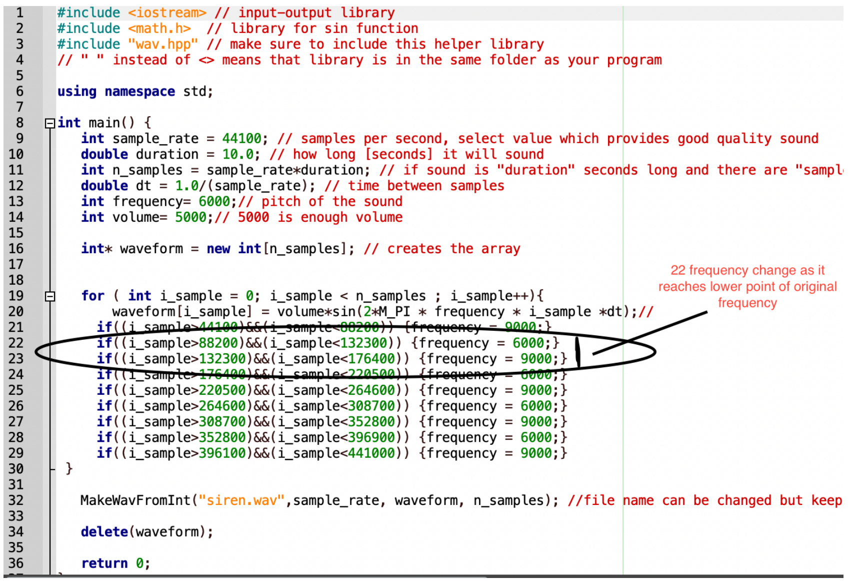
Mono, 44100Hz
32-bit float

It looks like what I intended. I did not vary frequency within the code. This has shown to construct a vibrations on a repeating wave.

Down for Completion Reflection

Completion Reflection

Frequency remains constant depending on the amount of speed it is going at. If the speed were to slow down that is when the wavelength also alternates to fit, it. The high point of the sound wave takes time to adjust after change within the frequency. A good time to change the frequency is at the lower point so the sound and wave can then adjust as the wave increases upwards again.



```
1   #include <iostream> // input-output library
2   #include <math.h>  // library for sin function
3   #include "wav.hpp" // make sure to include this helper library
4   // " " instead of <> means that library is in the same folder as your program
5
6   using namespace std;
7
8   int main() {
9      int sample_rate = 44100; // samples per second, select value which provides good quality sound
10     double duration = 10.0; // how long [seconds] it will sound
11     int n_samples = sample_rate*duration; // if sound is "duration" seconds long and there are "sampl
12     double dt = 1.0/(sample_rate); // time between samples
13     int frequency= 6000;// pitch of the sound
14     int volume= 5000;// 5000 is enough volume
15
16     int* waveform = new int[n_samples]; // creates the array
17
18
19     for ( int i_sample = 0; i_sample < n_samples ; i_sample++){
20         waveform[i_sample] = volume*sin(2*M_PI * frequency * i_sample *dt);//
21       if((i_sample>44100)&&(i_sample<88200)) {frequency = 9000;}
22       if((i_sample>88200)&&(i_sample<132300)) {frequency = 6000;}
23       if((i_sample>132300)&&(i_sample<176400)) {frequency = 9000;}
24       if((i_sample>176400)&&(i_sample<220500)) {frequency = 6000;}
25       if((i_sample>220500)&&(i_sample<264600)) {frequency = 9000;}
26       if((i_sample>264600)&&(i_sample<308700)) {frequency = 6000;}
27       if((i_sample>308700)&&(i_sample<352800)) {frequency = 9000;}
28       if((i_sample>352800)&&(i_sample<396900)) {frequency = 6000;}
29       if((i_sample>396100)&&(i_sample<441000)) {frequency = 9000;}
30  }
31
32     MakeWavFromInt("siren.wav",sample_rate, waveform, n_samples); //file name can be changed but keep
33
34     delete(waveform);
35
36     return 0;
```

22 frequency change as it reaches lower point of original frequency

An example of an algorithm:
If((i_sample>______) &&(i_sample<______) {frequency: can change here as it reaches lower frequency}
If(i_sample>______) &&(i_sample<_____) {frequency: can change here as per prior change in line above}

The code I used for completion is naïve (bunch of ifs). Another way to decide when to change the frequency is by modulo division. This returns the remainder of a division after one number is divided by another. The result is the remainder of integer division of the given numbers.

An example algorithm for the modulo division method:

35/3 = 11, remainder 2 → 35 mod 3 = 2

A formula to work this out is [dividend-{(dividend/divisor) *divisor}]

→35 % 3
→35- {(35/3) *3}
→35- {2*3)
→=2

Modulo division is effective as we can determine an even result (the remainder) to construct frequencies used in our code.