C++ Real-Time Audio Programming with Bela

Dr Andrew McPherson

Centre for Digital Music
School of Electronic Engineering and Computer Science
Queen Mary University of London

Founder and Director, Bela

Course topics

Programming topics

Working in real time
Buffers and arrays
Parameter control
Classes and objects
Analog and digital I/O
Filtering
Circular buffers
Timing in real time
State machines
MIDI
Block-based processing
Threads
Fixed point arithmetic
ARM assembly language

Today

Music/audio topics

Oscillators
Samples
Wavetables
Filters
Control voltages
Gates and triggers
Delays and delay-based effects
Metronomes and clocks
Envelopes
ADSR
MIDI
Additive synthesis
Phase vocoders
Impulse reverb

Lecture 5: Classes and objects

What you'll learn today:

Basics of C++ classes
Working with arrays of objects
Principles of additive synthesis

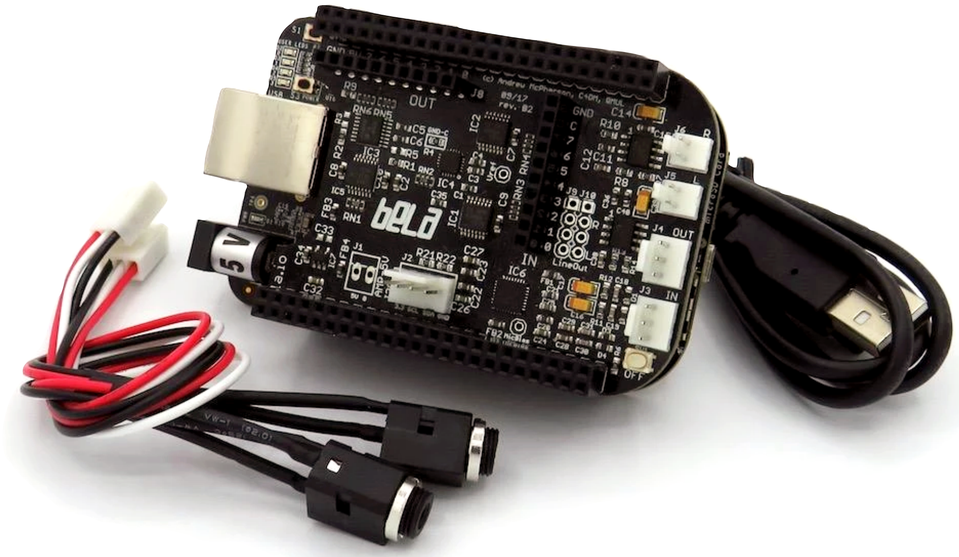
What you'll make today:

An additive synthesis waveform generator

Companion materials:

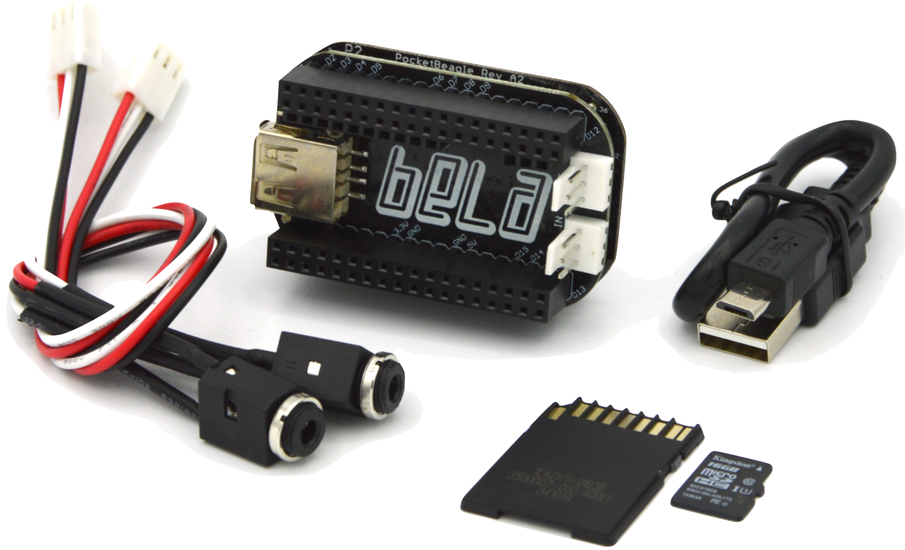
github.com/BelaPlatform/bela-online-course

What you'll need



Bela Starter Kit

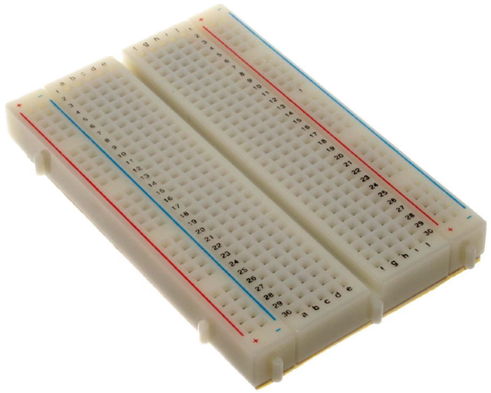
or

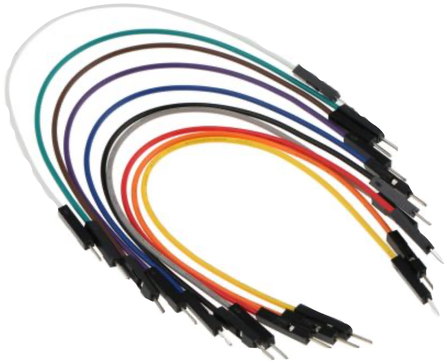


Bela Mini Starter Kit

[shop.bela.io]

Recommended for some lectures:

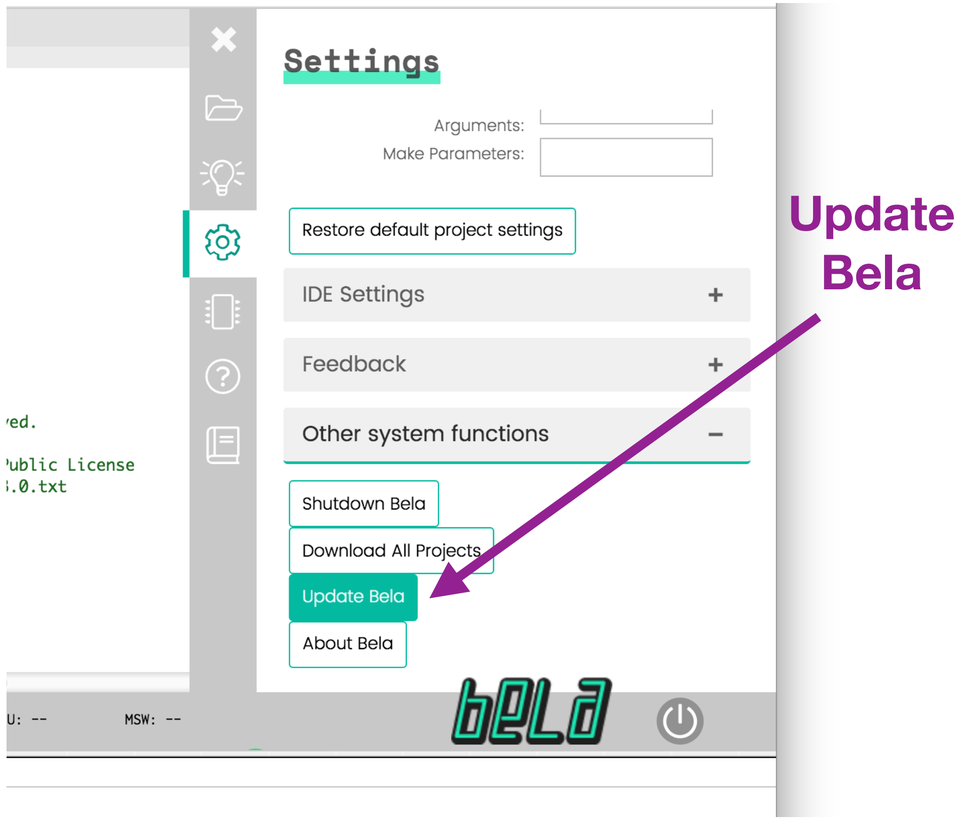




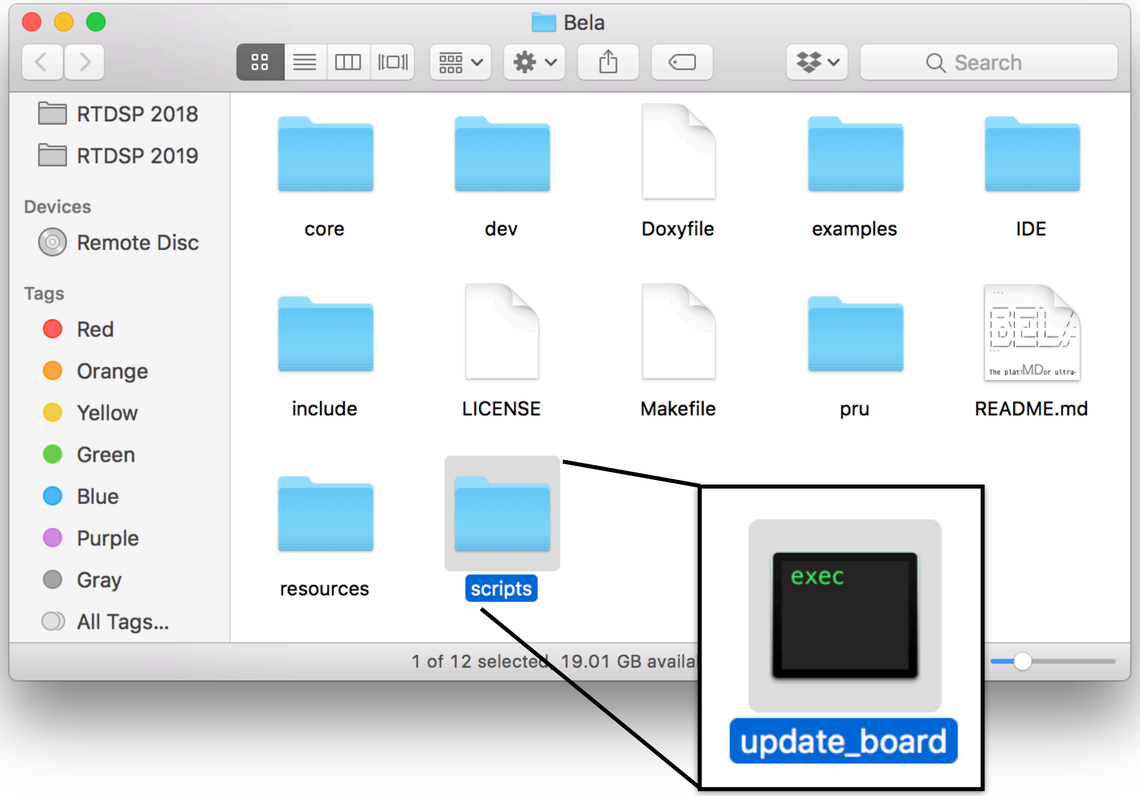
Updating Bela

- Download the latest Bela core software from: **learn.bela.io/update**
- Two ways to update the board:

By IDE (recommended)



By Script (Mac and Linux)
unzip the archive



Multiple oscillators

- Review of what we implemented in Lecture 4:
- On analog synths, it's common to use 2 or more oscillators for a single voice
  - The oscillators are tuned nearly, but not exactly, to the same frequency
  - Slight detuning results in beat frequencies according to the mathematical relationship:

$$\sin(x) + \sin(y) = 2\sin(\frac{x+y}{2})\cos(\frac{x-y}{2})$$

- Practically speaking, we want 2 controls:
  - One for setting the centre frequency of both oscillators (same as before)
  - One for changing the detuning (the slight difference between the oscillators)
  - If the centre frequency is *f* and the detuning amount is *r*, we should have:

$$f_1 = (1+r)f \qquad f_2 = (1-r)f$$

    - In this case, *r* = 0 corresponds to no detuning (both oscillators the same frequency)

Multiple oscillators: some checks

- How many copies of the wavetable do we need (for 2 oscillators)? **1**
- How many copies of the read pointer do we need? **2**
- How many sliders do we need? **3** (amplitude, frequency, detune)
- How do we calculate the frequencies?

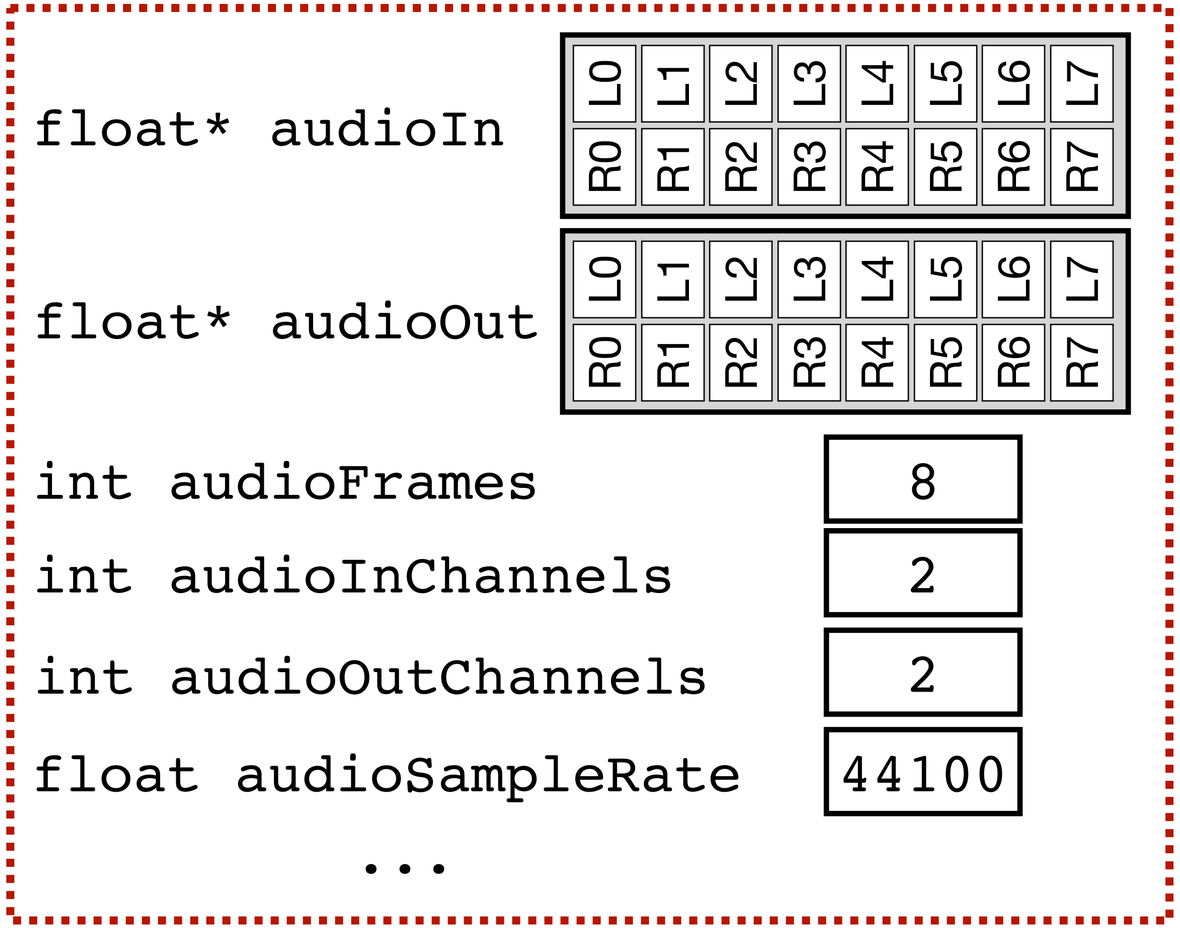
```
float frequency1 = frequency * (1 + detune);
float frequency2 = frequency * (1 - detune);
```

- Which code in `render()` do we need to duplicate?
  - The outer `for()` loop involving `n`? **no**
  - Calculating the output value from integer and fractional indexes? **yes**
  - Updating the read pointer? **yes**
  - The `for()` loop containing `audioWrite()`? **no**
- Copying and pasting code is bad. Is there a cleaner way to deal with this?
  - Let's try to encapsulate everything we need for the oscillator into a class

BelaContext: a C-style struct

- `BelaContext` is an example of a `struct`, a container for named data

`struct BelaContext`



buffer of audio input samples for all channels (we'll look at how this is organised soon)

buffer of audio output samples for all channels

the audio block size

how many audio input channels

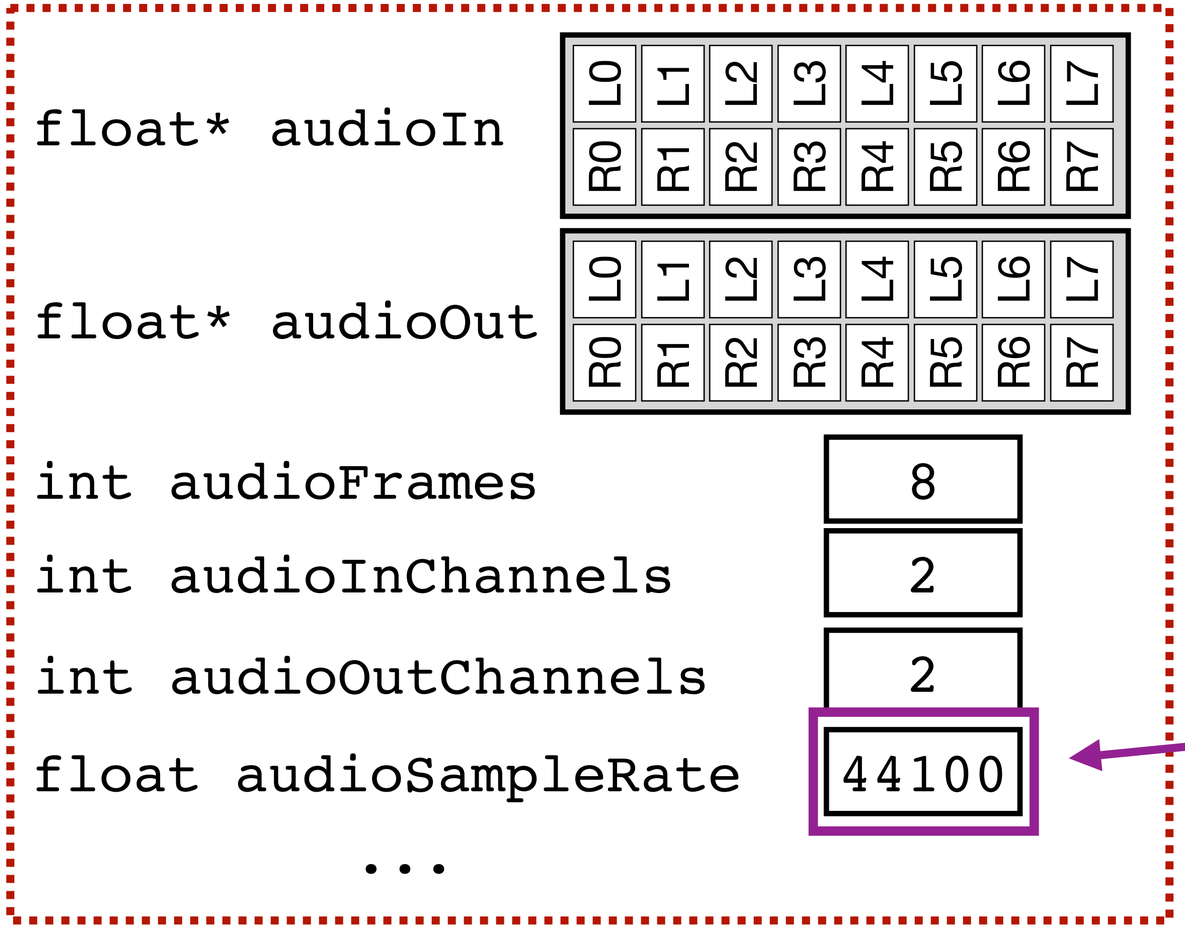
how many audio output channels

sample rate of audio data

BelaContext: a C-style struct

- See the Documentation tab in the IDE or **learn.bela.io** for full description

`struct BelaContext`



all this information is passed to `render()`

```
void render(BelaContext *context, void *userData)
```

`render()` gets a pointer to the data structure. That means the data lives somewhere in memory, and this tells us where to find it

To access an element inside `context`:

```
context->audioSampleRate
```

If `context` wasn't a pointer, it would be:

```
context.audioSampleRate
```

Classes in C++

- A `class` is a type of data structure in C++
  - Like the `struct` in C, it can encapsulate many elements inside a single structure

```
class Wavetable {
      // Useful stuff goes inside here!
};
```

We're going to build a wavetable oscillator example; You can follow along in the wavetable-class project

- Like the C `struct`, a `class` can hold variables:

```
class Wavetable {
    float sampleRate;          // Sample rate of the audio
    float frequency;           // Frequency of the oscillator
    float phase;               // Phase of the oscillator
    // [etc.]
};
```

- We can create objects of the class, then access their elements:
  - Every object has its own copies of the elements

```
Wavetable w;       // Declare an object of type Wavetable
w.frequency;       // <--- access the element "frequency" of the object w
```

Classes and methods

- Unlike a C `struct`, a C++ `class` can also hold functions
  - These functions inside a class are known as methods
  - They have automatic access to all the variables inside the class
    - How do we implement the contents of the method?

This is a method: a function inside a class

```
class Wavetable {
    float process();

    float sampleRate;
    float frequency;
    float phase;
    // [etc.]
};

Wavetable w;
float out = w.process();
```

```
// Get the next sample and update the phase

// Sample rate of the audio
// Frequency of the oscillator
// Phase of the oscillator

// Make an object of type Wavetable
// Call the method for the next sample
```

Classes and methods

- The contents of methods can be implemented inside the class definition
- Notice: we can access all the variables from inside the method
  - The values of these instance variables will persist across calls to the method
  - Very similar to global variables, but encapsulated within each particular object

```
class Wavetable {
    float process() {
        // Increment and wrap the phase
        phase += tableLength * frequency / sampleRate;
        while(phase >= tableLength)
            phase -= tableLength;
        // Read from the table, without interpolation
        return table[(int)phase];
    }

    float *table;              // Data in the wavetable
    int   tableLength;         // Length of the wavetable
    float sampleRate;          // Sample rate of the audio
    float frequency;           // Frequency of the oscillator
    float phase;               // Phase of the oscillator
};
```

We can access the instance variables from within the method

This sets the value that comes back to whoever called the method

Classes and methods

- Or declare the prototype inside the class, and the implementation elsewhere
  - They could even be in different files!
  - Use the `::` symbol to indicate that the implementation belongs to the class
  - It would be normal to have a `wavetable.h` file to hold the class declaration and a `wavetable.cpp` file where the implementations are declared

Method prototype: declares the arguments and return type, but not the implementation

```
class Wavetable {
    float process();

    float sampleRate;
    float frequency;
    float phase;
    // [etc.]
};

float Wavetable::process() {
    // Implementation goes here; can still access the class variables
}
```

```
// Get the next sample and update the phase

// Sample rate of the audio
// Frequency of the oscillator
// Phase of the oscillator
```

Method implementation might go in a separate file

Constructors

- A constructor is a special method that runs when an object is created
  - Remember: we can make lots of objects of the class type
  - The role of the constructor is to initialise the contents of the object
- We can have multiple constructors with different arguments
  - Which constructor gets called depends on how the object is created

```
class Wavetable {
    Wavetable();                        // Constructor: runs when the object is created
    Wavetable(float sampleRate); // Constructor with argument

    float process();                    // Get the next sample and update the phase

    float sampleRate;                   // Sample rate of the audio
    float frequency;                    // Frequency of the oscillator
    float phase;                        // Phase of the oscillator
};

Wavetable w1;                           // Constructor will run when this variable is created
Wavetable w2(44100);                    // Make an object using the second constructor
```

Destructor

- A destructor is a special method that runs when an object is released
  - For example, when the end of the function is reached where the object was declared
  - Or when `delete` is called (for dynamically allocated objects)
  - The role of the destructor is to do any necessary cleanup, e.g. freeing memory

```
class Wavetable {
    Wavetable();                      // Constructor: runs when the object is created
    Wavetable(float sampleRate); // Constructor with argument

    float process();                  // Get the next sample and update the phase

    ~Wavetable();                     // Destructor

    float sampleRate;                // Sample rate of the audio
    float frequency;                 // Frequency of the oscillator
    float phase;                     // Phase of the oscillator
};
```

Access specifiers

- Variables and methods can be declared with several types of access
  - `public:` means that any code can access the elements
  - `private:` means they can only be accessed from methods within the class
  - `protected:` is like `private:` but can also be accessed from subclasses
    - We won't get into inheritance today, but it is a useful feature of object-oriented programming

Anyone can access these methods →

These variables are only accessible within the class →

```
class Wavetable {
public:
    Wavetable();                          // Constructor: runs when the object is created
    Wavetable(float sampleRate); // Constructor with argument

    float process();                      // Get the next sample and update the phase

    ~Wavetable();                         // Destructor

private:
    float sampleRate;                   // Sample rate of the audio
    float frequency;                    // Frequency of the oscillator
    float phase;                        // Phase of the oscillator
};
```

Access specifiers

- A common convention:
  - Make all the variables private (no direct access outside the class)
  - Have methods for getting and setting their values
  - Also common to have a naming convention to identify instance variables

```
class Wavetable {
public:
    Wavetable();                          // Default constructor
    Wavetable(float sampleRate);          // Constructor with argument

    void setFrequency(float f);           // Set the oscillator frequency
    float getFrequency();                 // Get the oscillator frequency

    float process();                      // Get the next sample and update the phase

    ~Wavetable();                         // Destructor

private:
    float sampleRate_;                    // Sample rate of the audio
    float frequency_;                     // Frequency of the oscillator
    float phase_;                         // Phase of the oscillator
};
```

`frequency_` is private, but these public methods read and write its value

the underscore is one possible convention to identify instance variables (versus local variables inside each method)

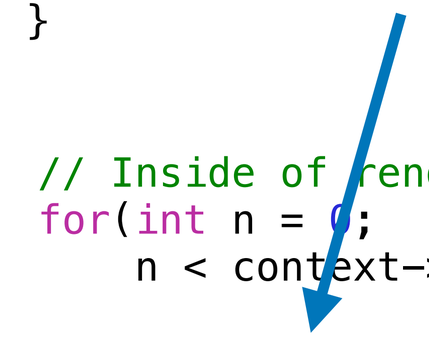
Classes in practice

- In our real-time audio systems, here's a typical approach to using classes:
  - See `wavetable-class` example
- Preparation:
  1. Define the class in a `classname.h` header file (i.e. file name same as the class)
  2. Implement the methods in `classname.cpp`
  3. Use `#include "classname.h"` inside both `classname.cpp` and `render.cpp`
- Instantiation:
  4. Create one or more objects of the class, often as global variables
     - Unless the object only needs to persist for the duration of a single call to `render()`
  5. The constructor runs wherever the object is declared: don't call it directly
- Use:
  6. Access the methods (and/or variables) in `setup()` and `render()` as needed
  7. The destructor will run automatically when the object is freed: don't call it directly

Classes for real-time signals

- In our signal processing code, we often perform the same series of steps repeatedly for each new audio sample
  - e.g. in each iteration of a `for()` loop over a block of samples
- One approach to a class dealing with signals: declare a method we call once per frame
  - The method takes any required input signals as arguments
  - The method will calculate the output and put it in the return value
    - There are ways to return more than one output, which we won't look at now
  - The method will also update the internal state of the object for the next frame
- Can also have other methods for updating the state or parameters of the object
  - These methods would be called sporadically, as needed
  - e.g. filter coefficients, frequencies, etc.

```
// Get the next sample and
// update the phase
float Wavetable::process() {
    // Increment and wrap the phase
    phase += tableLength * frequency
             / sampleRate;
    while(phase >= tableLength)
        phase -= tableLength;
    // Read from the table, without
    // interpolation
    return table[(int)phase];
}
```



```
// Inside of render():
for(int n = 0;
    n < context->audioFrames; n++) {

    float out = w.process();
}
```

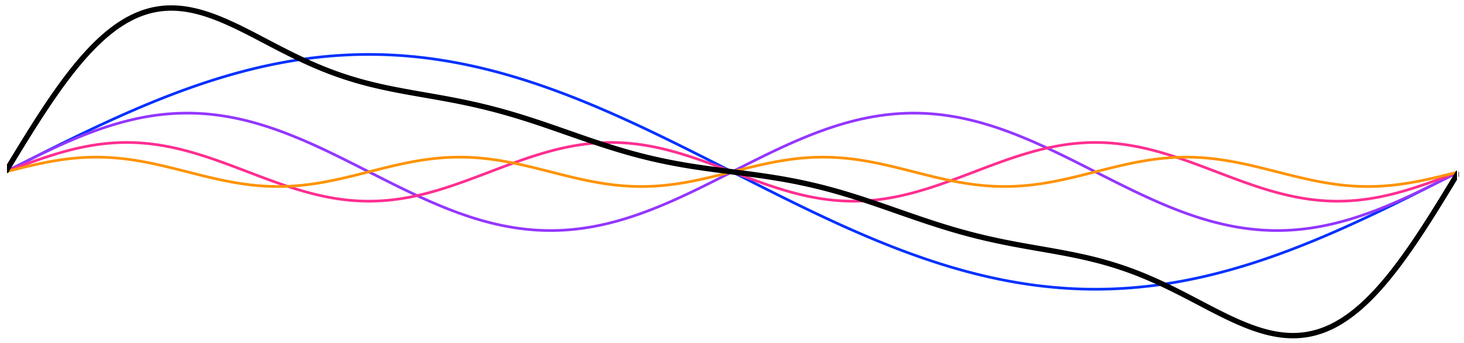
Should a class know about BelaContext?

- If our class is doing generic signal processing, it's better not to have it know anything about the `BelaContext` data structure
  - Keep the signal processing independent of the platform it's implemented on
  - For example: envelopes, oscillators and filters don't need to know anything specific about Bela I/O settings
  - Anything Bela-specific can stay in `render()`
- On the other hand, a class that specifically relates to hardware communication might want to know about `BelaContext`
  - But you'll still need to pass in the `BelaContext` structure each time you call a method
  - Each call to `render()` has different data in `BelaContext`, so we can't cache a copy

Additive synthesis

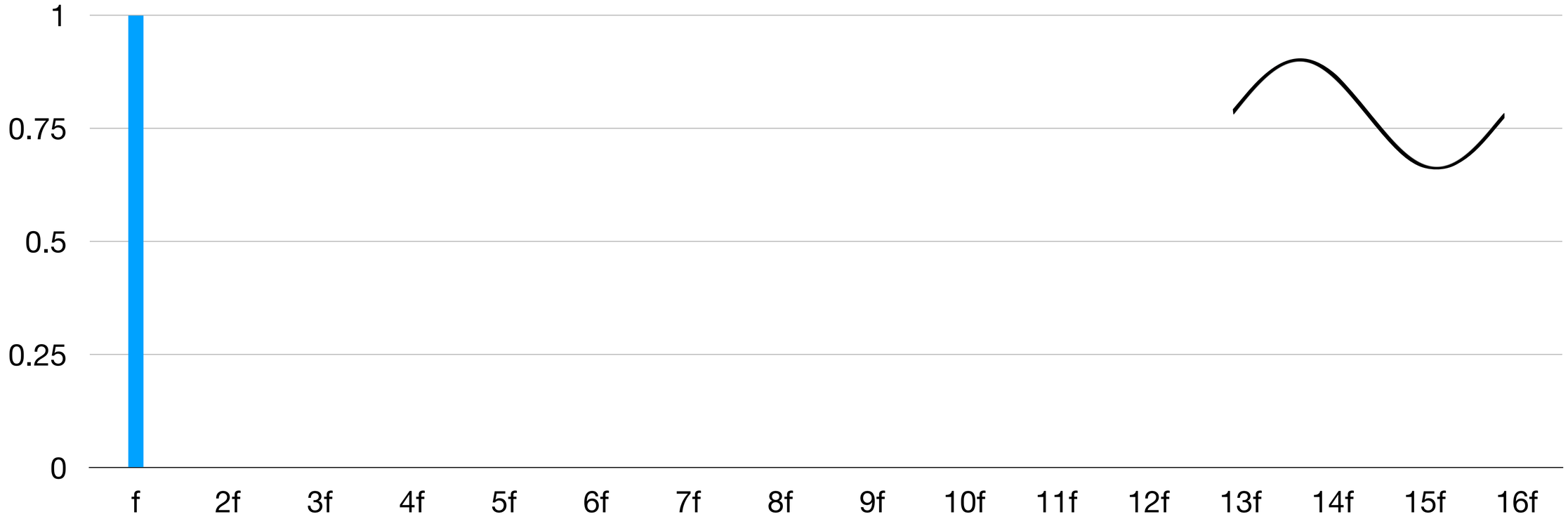
- Additive synthesis is a method of building complex sounds out of simpler components, such as sine waves
- According to Fourier, every periodic signal can be represented as a sum of harmonically-related sinusoids
  - *Harmonically-related* means integer multiples of some fundamental frequency (or integer divisions of the period)
  - Each harmonic could have any amplitude or phase
  - The more general case, for any (non-periodic) signal, is the Fourier Transform

$$w(\phi) = 0.5\sin(\phi) + 0.25\sin(2\phi) + 0.125\sin(3\phi) + 0.0625\sin(4\phi)$$



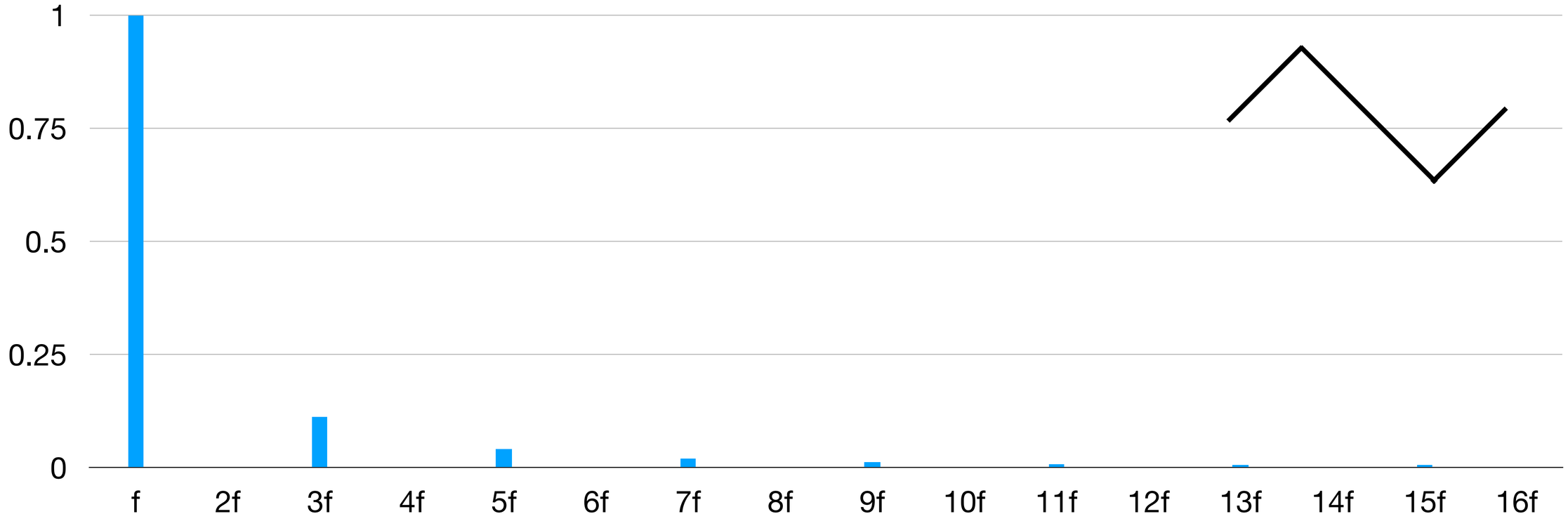
Additive synthesis and timbre

- The spectrum of a sound is one important component of its timbre
  - Timbre is a complex concept, also dependent on a sound's onset transient and its evolution over time
- We can plot the amplitude of each harmonic versus its frequency:



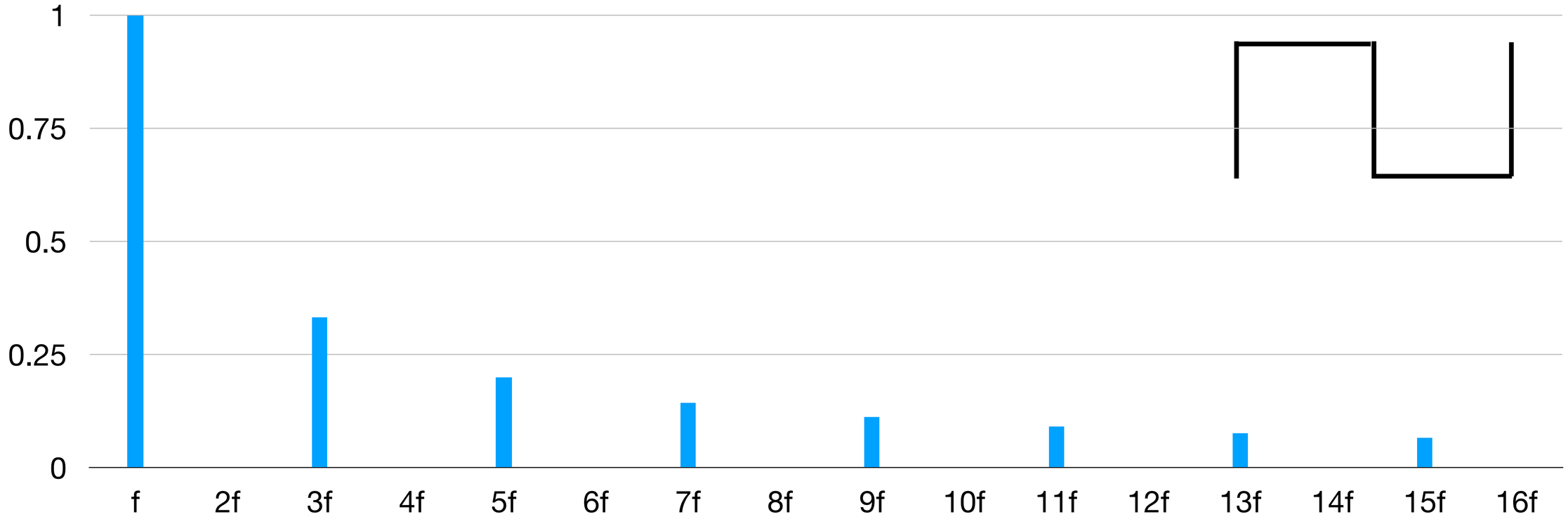
Additive synthesis and timbre

- The spectrum of a sound is one important component of its timbre
  - Timbre is a complex concept, also dependent on a sound's onset transient and its evolution over time
- We can plot the amplitude of each harmonic versus its frequency:



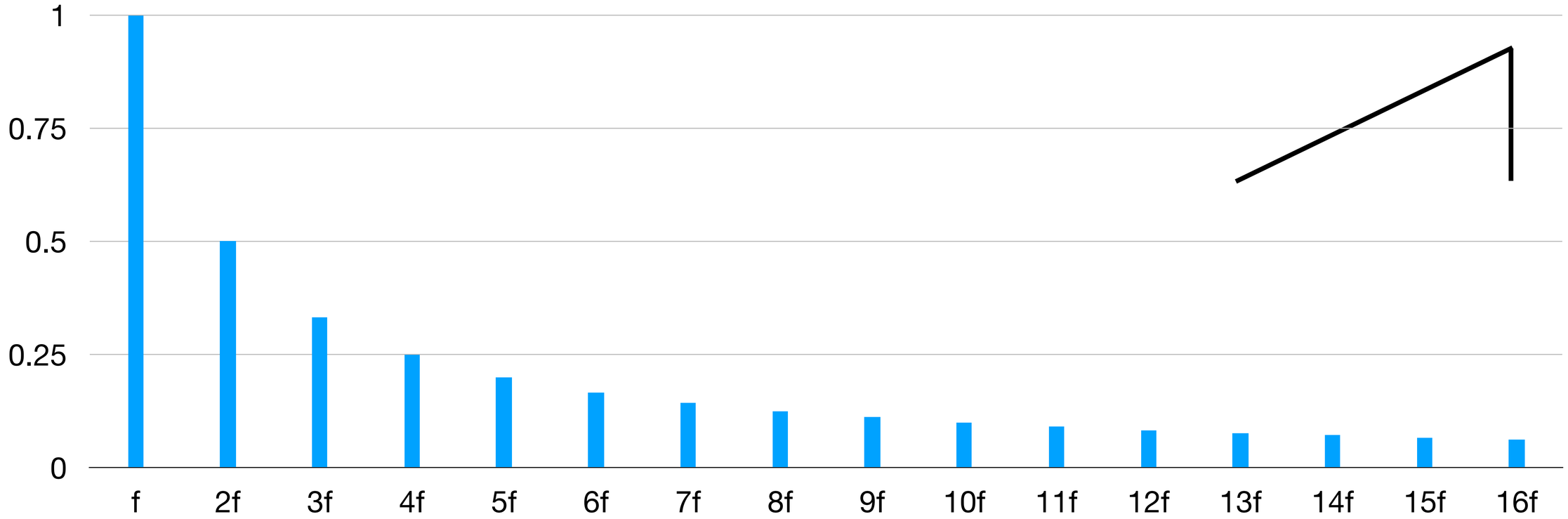
Additive synthesis and timbre

- The spectrum of a sound is one important component of its timbre
  - Timbre is a complex concept, also dependent on a sound's onset transient and its evolution over time
- We can plot the amplitude of each harmonic versus its frequency:



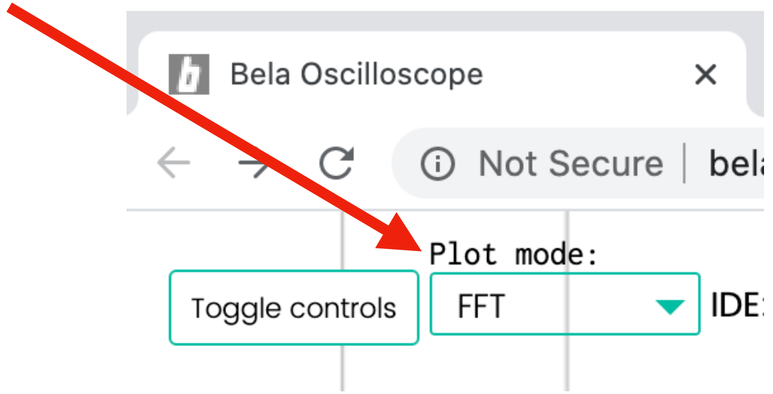
Additive synthesis and timbre

- The spectrum of a sound is one important component of its timbre
  - Timbre is a complex concept, also dependent on a sound's onset transient and its evolution over time
- We can plot the amplitude of each harmonic versus its frequency:



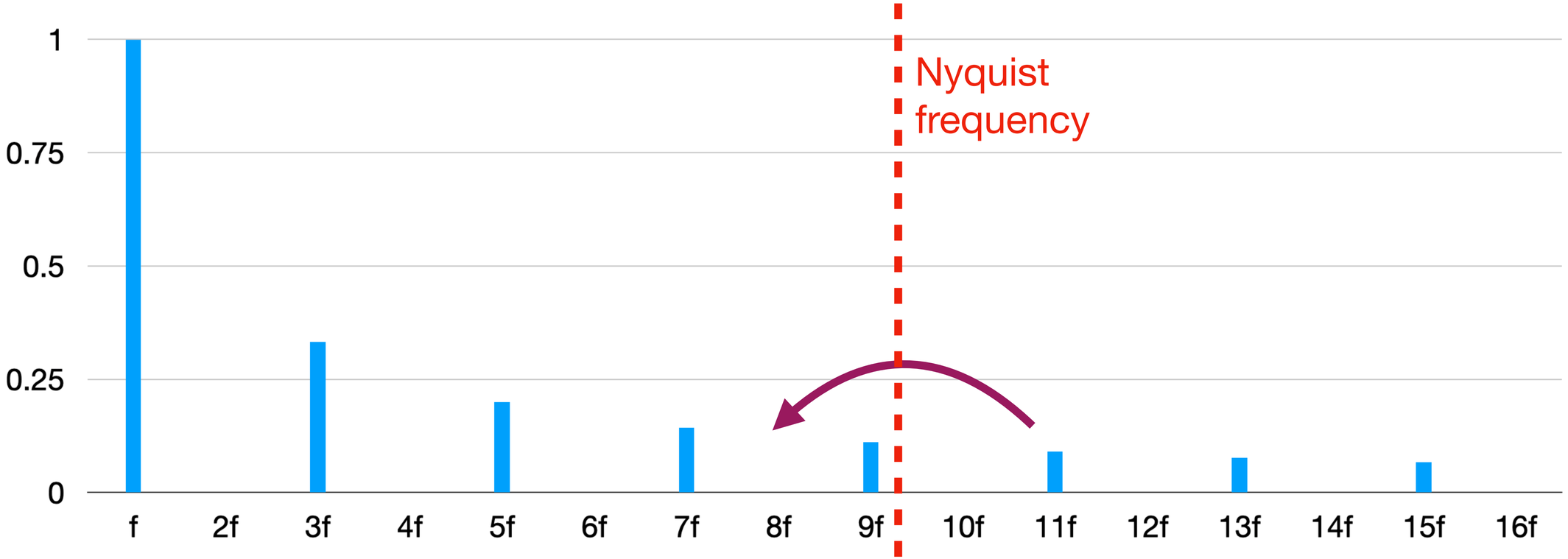
Why do digital oscillators sound bad?

- The harmonic series for triangle, square and sawtooth waves are infinite
  - Gradually diminishing in amplitude, but never reaching 0
  - Why might this be a problem in the digital domain?
- Task: in the wavetable-class example, change the calculation in `setup()` to generate a square wave rather than a sine wave
  - Square wave has the value of 1.0 for the first half of the samples, -1.0 for the second half
- Open the oscilloscope, change it FFT mode
  - In Controls, change to 2048 points, decibel scale Y-axis
  - What happens to the frequency components as you change the fundamental frequency?



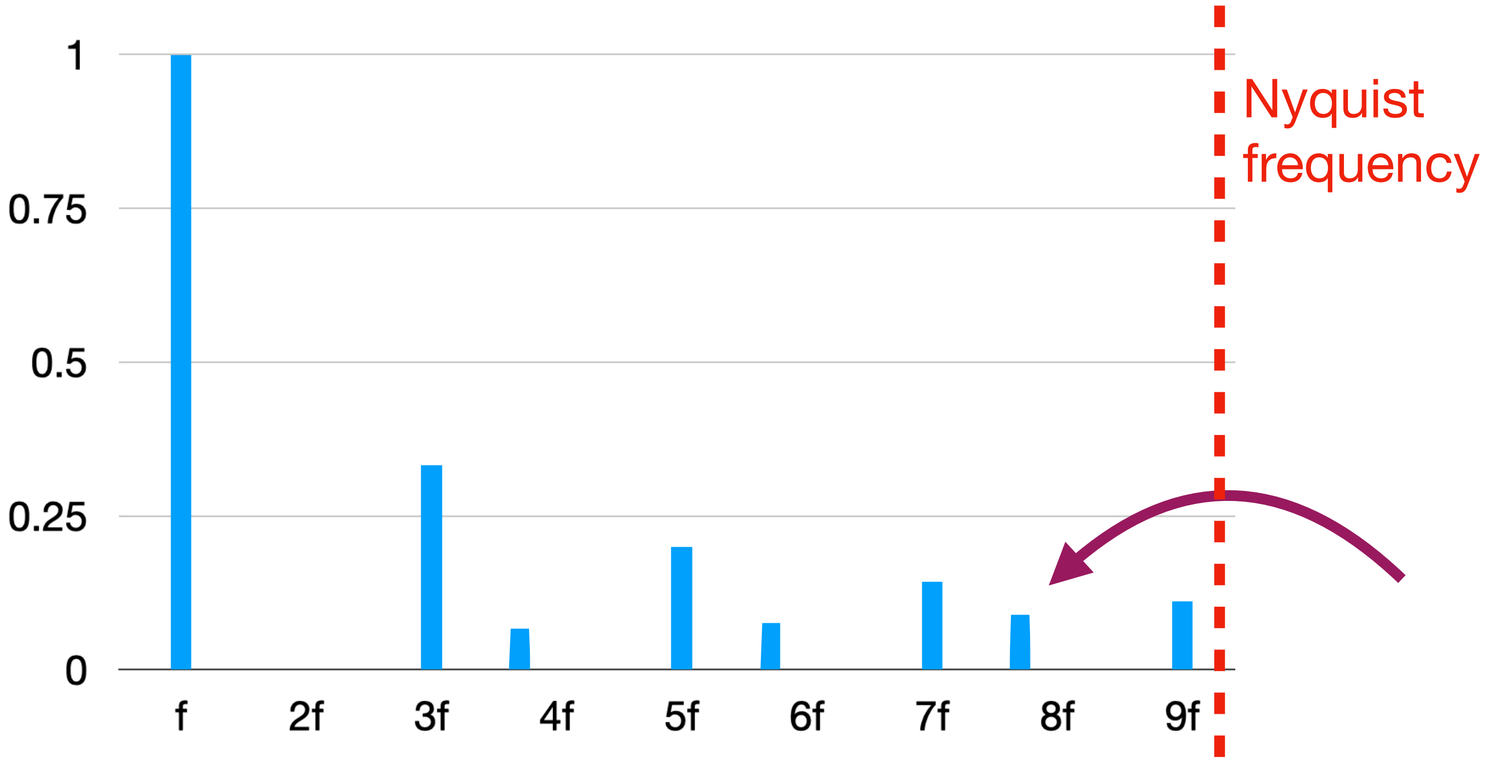
Why do digital oscillators sound bad?

- The harmonic series for triangle, square and sawtooth waves are infinite
  - Gradually diminishing in amplitude, but never reaching 0
- Frequencies in digital signals are limited to the Nyquist rate
  - Above that, we get **aliasing** of frequency components, which will be inharmonic



Why do digital oscillators sound bad?

- The harmonic series for triangle, square and sawtooth waves are infinite
  - Gradually diminishing in amplitude, but never reaching 0
- Frequencies in digital signals are limited to the Nyquist rate
  - Above that, we get **aliasing** of frequency components, which will be inharmonic



An additive synthesis square wave

- With our `Wavetable` class, we could build up our square wave one harmonic at a time
  - One Wavetable per harmonic, up to a certain maximum number
  - Can't make more than a few dozen this way, but Nyquist limits how many we need
- Make an array of `Wavetable` objects
  - See the additive-synth project. We can use the C++ `std::vector` object for the array:

```
const unsigned int kNumOscillators = 24;
std::vector<Wavetable> gOscillators;        // An array of oscillator objects

bool setup(BelaContext *context, void *userData)
{
    // ...
    for(unsigned int n = 0; n < kNumOscillators; n++) {
        // Create a new Wavetable oscillator
        Wavetable oscillator(context->audioSampleRate, wavetable, false);
        // Add it to the end of the vector
        gOscillators.push_back(oscillator);
    }
    // ...
}
```

Declare the type of the vector

The **`push_back()`** method adds an object to the end of the vector, making it one element longer. Can't do this with a standard C array!

An additive synthesis square wave

- We can also add a GUI control for the amplitude of each oscillator:

```
// Add levels for the individual oscillators
gGuiController.addSlider("Harmonic 1", 0, -60, 0, 0);
for(unsigned int n = 1; n < kNumOscillators; n++) {
    std::stringstream name;
    name << "Harmonic " << (n + 1);    // Format the name of the slider

    float dBvalue = -60;               // Calculate the default amplitude
    if((n % 2) == 0) {                 // based on a square wave
        dBvalue = 20.0 * log10f(1.0 / (float)(n + 1));
    }
    gGuiController.addSlider(name.str(), dBvalue, -60, 0, 0);
}
```

C++ string formatting stuff...

Convert linear amplitude to decibels

Create the slider with the given name, default value and range

- Task: in additive-synth, implement the code in `render()` to calculate an array of `Wavetable` oscillators
  - Set the frequency of each oscillator to an integer multiple of the fundamental
  - Get the amplitudes from the sliders (use `gAmplitudes` to store); if -60dB, set amplitude to 0
  - Each frame, get the output of each oscillator and add them, scaled by `gAmplitudes`
- Try bringing up the GUI and scope side by side!

Additive synthesis code

- To gather data from the sliders and set frequencies of each oscillator:

```
for(unsigned int i = 0; i < gOscillators.size(); i++) {
    // Set the frequency as a multiple of the fundamental frequency
    gOscillators[i].setFrequency(frequency * (i + 1));

    // Get the amplitude for this particular oscillator
    float oscAmplitudeDb = gGuiController.getSliderValue(2 + i);
    if(oscAmplitudeDb <= -60)
        gAmplitudes[i] = 0;     // Use the bottom of the slider as "mute"
    else
        gAmplitudes[i] = powf(10.0, oscAmplitudeDb / 20);
}
```

C++ array indexing starts from 0, but harmonics start from 1

Sliders for harmonic amplitudes start at 2

Decibels to linear amplitude

- The above code can go at the beginning of `render()`
  - Doesn't need to be inside the main `for()` loop because the slider values won't change as long as `render()` is running

Additive synthesis code

- To calculate each oscillator's output in `render()`:

```
for(unsigned int n = 0; n < context->audioFrames; n++) {
    float out = 0;

    // Step through all the oscillators in the array
    for(unsigned int i = 0; i < gOscillators.size(); i++) {
        // Mix in the output of this oscillator
        out += gAmplitudes[i] * gOscillators[i].process();
    }

    // Scale global amplitude
    out *= amplitude;

    for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
        // Write the sample to every audio output channel
        audioWrite(context, n, channel, out);
    }

    // Write the output to the oscilloscope
    gScope.log(out);
}
```

Get the next sample of this oscillator

- Problem: even with a limited number of harmonics, we might still have aliasing!
  - Task: fix additive-synth so that oscillators above the Nyquist rate are muted

Conclusions

- Classes are a useful way to encapsulate code
  - Lets us easily make multiple copies of an audio processing unit
  - Simplifies our `setup()` and `render()` functions
  - There's much more to C++ classes we haven't looked at!
- Additive synthesis is the process of building complex waveforms from simple components, such as sine waves
  - In principle, we can create any periodic waveform this way
  - We had to consider aliasing and eliminate frequencies above the Nyquist rate
- There are more efficient ways of generating band-limited waveforms
  - Additive synthesis can be expensive
  - The band-limited impulse train (BLIT) is a useful mathematical function that can be used to generate many classic waveforms without aliasing
  - See companion materials for links

Keep in touch!

Social media:

@BelaPlatform

forum.bela.io

blog.bela.io

More resources and contact info at:

learn.bela.io/resources