AUDIO C++

Level	Intermediate
Source	
Status	Ongoing
Related to Coding Projects (Files)	
Tags	

JUCE FRAMEWORK:

▼ EQ PROYECT

▼ SETUP NOTES

▼ PluginProcessor.h

In a JUCE plugin, there are two functions that are the most important:

```
// PREPARE TO PLAY FUNCTION:
   void prepareToPlay (double sampleRate, int samplesPerBlock) override;
// Gets called by the host when it's about to start playback


// PROCESS BLOCK FUNCTION:
   void processBlock (juce::AudioBuffer<float>&, juce::MidiBuffer&) override;
// When play button is activated in transport control.
// The host starts sending buffers at a regular rate into the plugin.
// Its the plugin's job to give back any finished audio that is done processing.
/// DO NOT INTERRUPT THE PROCESSBLOCK OR ANYTHING THAT GOES INSIDE IT \
```

IMPORTANT: FIGURE OUT A WAY TO MAKE THE CODE RUN IN A CERTAIN TIMESPAN.

▼ PluginProcessor.cpp

```
//PREPARE TO PLAY
void SimpleEQAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    // Use this method as the place to do any pre-playback
    // initialisation that you need..
}
//PROCESSBLOCK
void SimpleEQAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels  = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    // In case we have more outputs than inputs, this code clears any output
    // channels that didn't contain input data, (because these aren't
    // guaranteed to be empty - they may contain garbage).
    // This is here to avoid people getting screaming feedback
    // when they first compile a plugin, but obviously you don't need to keep
    // this code if your algorithm always overwrites all the output channels.
    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());

    // This is the place where you'd normally do the guts of your plugin's
    // audio processing...
```

```
    // Make sure to reset the state if your inner loop is processing
    // the samples and the outer loop is handling the channels.
    // Alternatively, you can process the samples with the channels
    // interleaved by keeping the same state.
    for (int channel = 0; channel < totalNumInputChannels; ++channel)
    {
        auto* channelData = buffer.getWritePointer (channel);

        // ..do something to the data...
    }
}
```

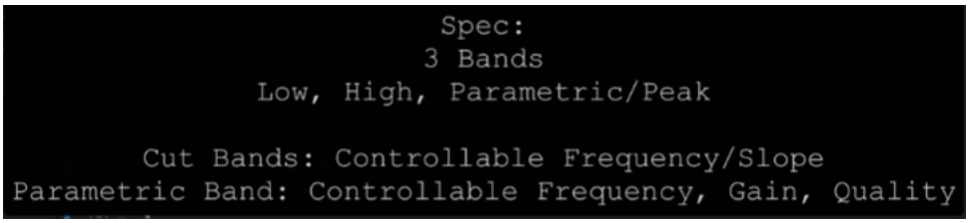
▼ CREATING AUDIO PARAMETERS

▼ About & Parameter Creation

Audio plugins depend on parameters to control the various parts of the DSP. JUCE *uses an object called the* `AudioProcessorValueTreeState` *to coordinate syncing these parameters with the knobs on the GUI and the variables on the processor.*

```
// In Pluginprocessor.h > declare apvts

static juce::AudioProcessorValueTreeState::ParameterLayout
  createParameterLayout();
juce::AudioProcessorValueTreeState apvts /*name*/
{
  *this, nullptr, "Parameters", createParameterLayout()
};
*this: //processor to connect to
nullptr: //no undo manager
"Parameters" // const identifier

// APVTS EXPECTS TO PROVIDE A LIST OF ALL PARAMETERS WHEN
// ITS CREATED >> APVTS PARAMETERLAYOUT
```

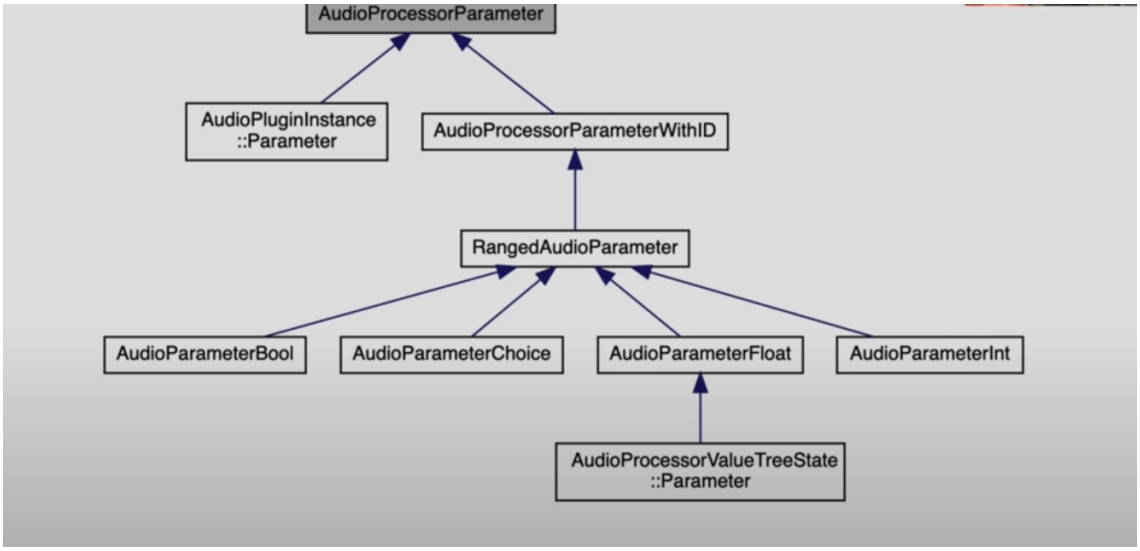
- Declaring create apvts parameter layout in PluginProcessor.cpp:




```
// Keeping DSP and GUI simple:
juce::AudioProcessorValueTreeState::ParameterLayout
  SimpleEQAudioProcessor::createParameterLayout()
{

}
```

Juce APP class: Generic Interface towards all different audio parameter formats that different plugin hosts use.

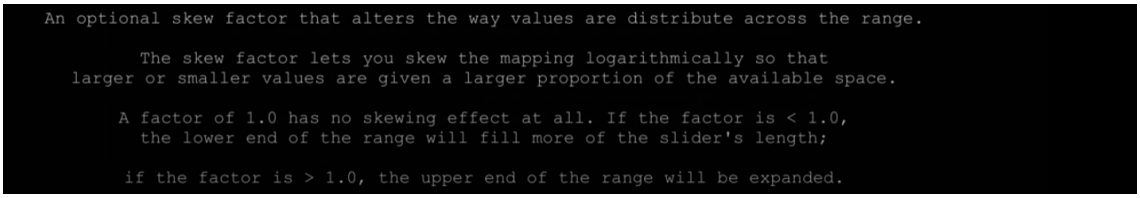
juce class diagram:



AudioParameterFloat: Parameters with a range of values, should be represented with a slider of some kind.

CREATING LOWCUT FREQUENCY PARAMETER:

skew value:




```
juce::AudioProcessorValueTreeState::ParameterLayout SimpleEQAudioProcessor::createParameterLayout()
{
    juce::AudioProcessorValueTreeState::ParameterLayout layout;

    layout.add(std::make_unique<juce::AudioParameterFloat>("LowCut Freq", // Name
    "LowCutFreq", // Parameter Name
    juce::NormalisableRange<float>(20.f, 20000.f, 1.f, 1.f), //Normalisable Range
    20.f)); // Starting Point


    return layout;
}
```

CREATING HIGH CUT AND PEAK FREQUENCY PARAMETERS:

```
layout.add(std::make_unique<juce::AudioParameterFloat>("HighCut Freq",
"HighCut Freq", juce::NormalisableRange<float>(20.f, 20000.f, 1.f, 1.f), 20000.f));

layout.add(std::make_unique<juce::AudioParameterFloat>("Peak Freq",
"Peak Freq", juce::NormalisableRange<float>(20.f, 20000.f, 1.f, 1.f), 750.f));
```

PEAK GAIN:

parameter expressed in dB: range > -24-24

step size: 0.5 > 0.5 dB change per slider step

slider to behave in linear fashion > skew of 1.f

dont add gain/cut by default > set value of 0

```
layout.add(std::make_unique<juce::AudioParameterFloat>("Peak Gain",
"Peak Gain", juce::NormalisableRange<float>(-24.f, 24.f, 0.5f, 1.f), 0.0f));
```

QUALITY CONTROL OF PEAK (Q)

```
layout.add(std::make_unique<juce::AudioParameterFloat>("Peak Quality",
"Peak Quality", juce::NormalisableRange<float>(0.1f, 10.f, 0.05f, 1.f), 1.f));
```

LOW CUT AND HIGH CUT FILTERS:

Ability to change the steepness of the filter cut

FILTER RESPONSES EXPRESSED IN dB PER OCTAVE.

the way the math behind the filter equations works, it ends up expressing these choices in multiples of 6 or 12:

12, 24, 36, 48 > Specific choices and not a range of values, therefore AudioParameterChoice object is used.

```
//String array that represents choices for filter APC object

    juce::StringArray stringArray;
    for (int i = 0; i < 4; i++)
    {
        juce::String str;
        str << (12 + i * 12);
        str << " dB/Oct";
        stringArray.add(str);
    }
```

CREATING LOW CUT AND HIGH CUT FILTERS AS APC OBJECTS

```
// Low Cut Filter Slope
layout.add(std::make_unique<juce::AudioParameterChoice>("LowCut Slope",
"LowCut Slope", stringArray, 0));
// High Cut Filter Slope
layout.add(std::make_unique<juce::AudioParameterChoice>("HighCut Slope",
"HighCut Slope", stringArray, 0))
```

CREATING A GENERIC EDITOR:

Using the GenericAudioProcessorEditor to see what the parameters created in the APVTS Layout look like.

```
juce::AudioProcessorEditor* SimpleEQAudioProcessor::createEditor()
{
    // return new SimpleEQAudioProcessorEditor (*this);
    return new juce::GenericAudioProcessorEditor(*this);
}
```

▼ SETTING UP A DIGITAL SIGNAL PROCESSOR (DSP)

- Add dsp module from projucer>juce global modules>juce_dsp
- Each of the signal processing classes in the juce::dsp:: namespace is set up to process mono by default unless its declared as stereo in the documentation. Need to duplicate everything for stereo processing.

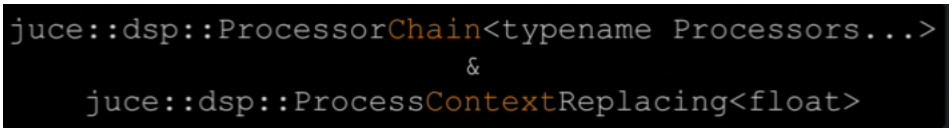
- dsp namespace uses a lot of template metaprogramming and nested namespaces > create type aliases to eliminate all of those namespace and template definitions.

CREATING A FILTER ALIAS:

Each filter type in IIR filter class has a response of 12dB/octave when it’s configured as a high/low pass filter.

```
//in pluginprocessor.h > AudioProcessor class private:

using Filter = juce::dsp::IIR::Filter<float>;
```

If a chain with a response of 48dB/octave is wanted, 4 filters will be needed. A central concept of the juce framework is to define a `chain` and pass in a `processing context` > which will run through each element of the chain automatically.




```
using Filter = juce::dsp::IIR::Filter<float>;

using CutFilter = juce::dsp::ProcessorChain<Filter, Filter,
                                            Filter, Filter>;
```

Now that Filter and Cut Filter are defined as aliases, a `chain` can be defined to represent the whole Mono signal path, `then doubling their instances` if stereo processing is desired, and to have access to the filter instances in order to adjust cutoff, gain, q, and slope.

```
//creating a filter alias:
    using Filter = juce::dsp::IIR::Filter<float>;

    //Creating a processor chain and passing in process context automatically:
    using CutFilter = juce::dsp::ProcessorChain<Filter, Filter, Filter, Filter>;

    //Creating a chain to represent the whole mono signal path.
    using MonoChain = juce::dsp::ProcessorChain<CutFilter, Filter, CutFilter>;
    //Stereo
    MonoChain leftChain, rightChain;
```

Filters must be prepared before use, the way to do this by passing a **`ProcessSpec`** object to the chains which will then pass it to each link in the chain.

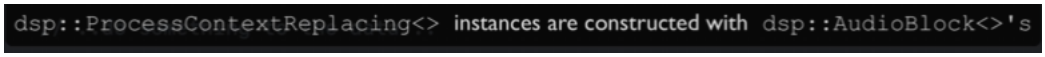
```
**IN PLUGINPROCESSOR.CPP > AudioProcessor::prepareToPlay**

void SimpleEQAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    // Use this met..hod as the place to do any pre-playback
    // initialisation that you need

    //ProcessSpec Object
    juce::dsp::ProcessSpec spec;
    //Needs to know max number of samples that will be processed
    //at one time
    spec.maximumBlockSize = samplesPerBlock;

    //Needs to know the number of channels
    //Mono chains can only handle 1 AudioChannel!
    spec.numChannels = 1;

    //Needs to know sampleRate
    spec.sampleRate = sampleRate;
```

```
    //Now it can be passed to each chain and it will be ready for processing.
    leftChain.prepare(spec);
    rightChain.prepare(spec);
}
```

PROCESS BLOCK:

The processing chain requires a processing context to be passed to it > to run audio through the links in the chain.

In order to create a processing context, it needs to be supplied with an audio block instance.



The process block function is called by the host and it is given a buffer that can have any number of channels.

This means the left/right channels need to be extracted from this buffer (0/1 respectively)

First Step: Create an Audio Block that wraps the buffer.

```
**IN PLUGINPROCESSOR.CPP > AudioProcessor::processBlock**


//creating audio block
juce::dsp::AudioBlock<float> block(buffer);
```

Step 2: Use the helper function in the AudioBlock class to extract individual channels from the buffer which will then be wrapped inside more audio blocks.

```
auto leftBlock = block.getSingleChannelBlock(0);
auto rightBlock = block.getSingleChannelBlock(1);
```

Now that Audio Blocks are present for each channel, processing contexts that wrap each individual channel block can be created.

```
juce::dsp::ProcessContextReplacing<float> leftContext(leftBlock);
juce::dsp::ProcessContextReplacing<float> rightContext(rightBlock);
```

Now that the filters are created, they can be passed to the Mono Filter Chains.

```
leftChain.process(leftContext);
rightChain.process(rightContext);
```

▼ SETTING UP AUDIOPLUGINHOST (APH)

Included in juce > AudioPluginHost application.

Steps for setting up with a juce plugin in Visual Studio.

- Build APH

- Build VST3 plugin
- In APG > Scan for VST3 plugins wherever the plugin folder saved it.
- Save as a .filtergraph file inside the project
- Configure VST3 to open aph when ran > VST3 > Properties > debugging > change $(TargetPath) to:
- D:\AUDIO C++\JUCE\extras\AudioPluginHost\Builds\VisualStudio2022\x64\Debug\App\AudioPluginHost.exe