

# Build Your First Audio Plug-in with JUCE

ADC Japan - 2026-06-01

Anthony Nicholls, Attila Szarvas, Reuben Thomas, Tom Poole

<https://audio.dev/adc-japan-26/schedule/>

<https://data.audio.dev/workshops/2026/build-first-plugin-with-juce/materials.zip>



# Overview

- Introduction to JUCE
- Creating JUCE-based projects
- Building audio plug-ins
- Testing audio plug-ins

<https://audio.dev/adc-japan-26/schedule/>

<https://data.audio.dev/workshops/2026/build-first-plugin-with-juce/materials.zip>

Put the materials somewhere that is not managed by iCloud, OneDrive, DropBox, or similar.

Backup systems can interfere with the build process!

# Ask us questions!

- We will have breaks between the main sections
- Please interrupt us
- The slides are available in the workshop materials link in the online schedule

<https://audio.dev/adc-japan-26/schedule/>

<https://data.audio.dev/workshops/2026/build-first-plugin-with-juce/materials.zip>





```
{  
    setColour (blue);  
    drawRect (0, 0, 100, 50);  
  
    setColour (red);  
    drawRect (100, 0, 100, 50);  
}
```

```
{  
    setColour (blue);  
    drawRect (0, 0, 100, 50);  
  
    setColour (red);  
    drawRect (100, 0, 100, 50);  
}
```



```
{  
    setColour (blue);  
    drawRect (0, 0, 100, 50);  
  
    setColour (red);  
    drawRect (100, 0, 100, 50);  
}
```

## Application has crashed

An unexpected error occurred forcing the application to stop. Please help us fix this by sending us error data, all you have to do is click 'Report'.

Report

Exit

C++



C++



C++



Obj-C++



Win32



Native  
Activity



POSIX/  
X11



AppKit/UIKit



## macOS/iOS

```
{
    CGRect rc = { 0, 0, 100, 50 };
    CGContextSetRGBFillColor (context, 0.0f, 0.0f, 1.0f, 1.0f);
    CGContextFillRect (context, &rc);

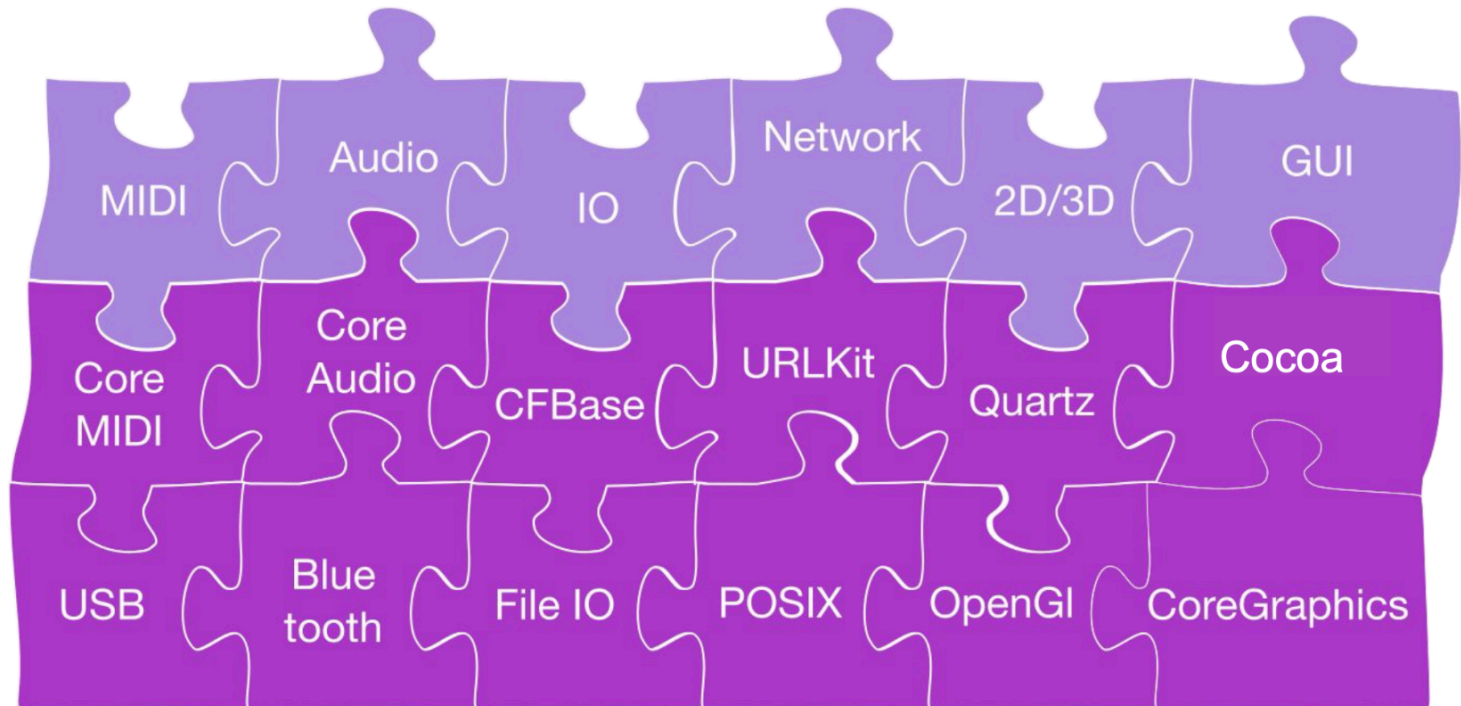
    rc = { 100, 0, 100, 50 };
    CGContextSetRGBFillColor (context, 1.0f, 0.0f, 0.0f, 1.0f);
    CGContextFillRect (context, &rc);
}
```

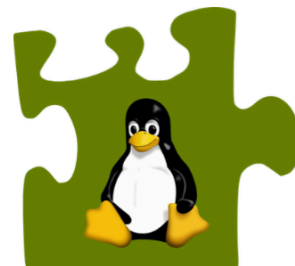
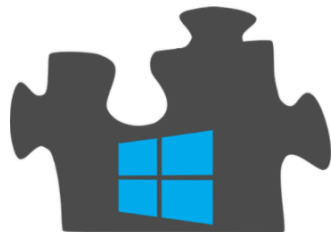
# Windows

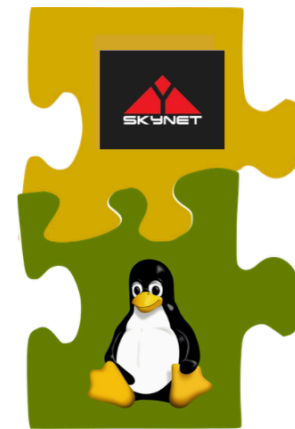
```
{  
    Rect rc = { 0, 0, 100, 50 };  
    FillRect (context, &rc, RGB (0, 0, 255));  
  
    Rect rc = { 100, 0, 100, 50 };  
    FillRect (context, &rc, RGB (255, 0, 0));  
}
```

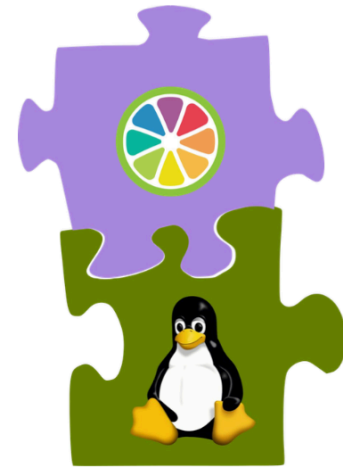
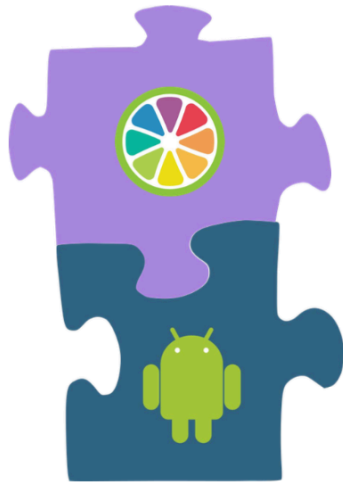


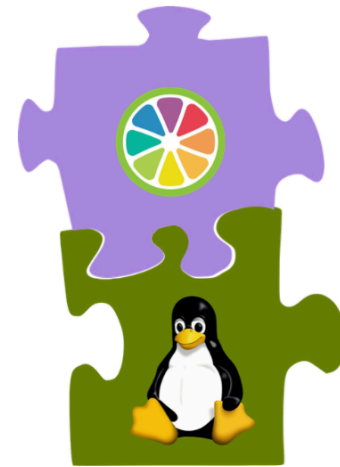
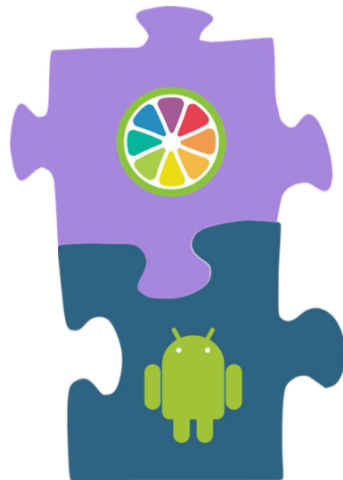


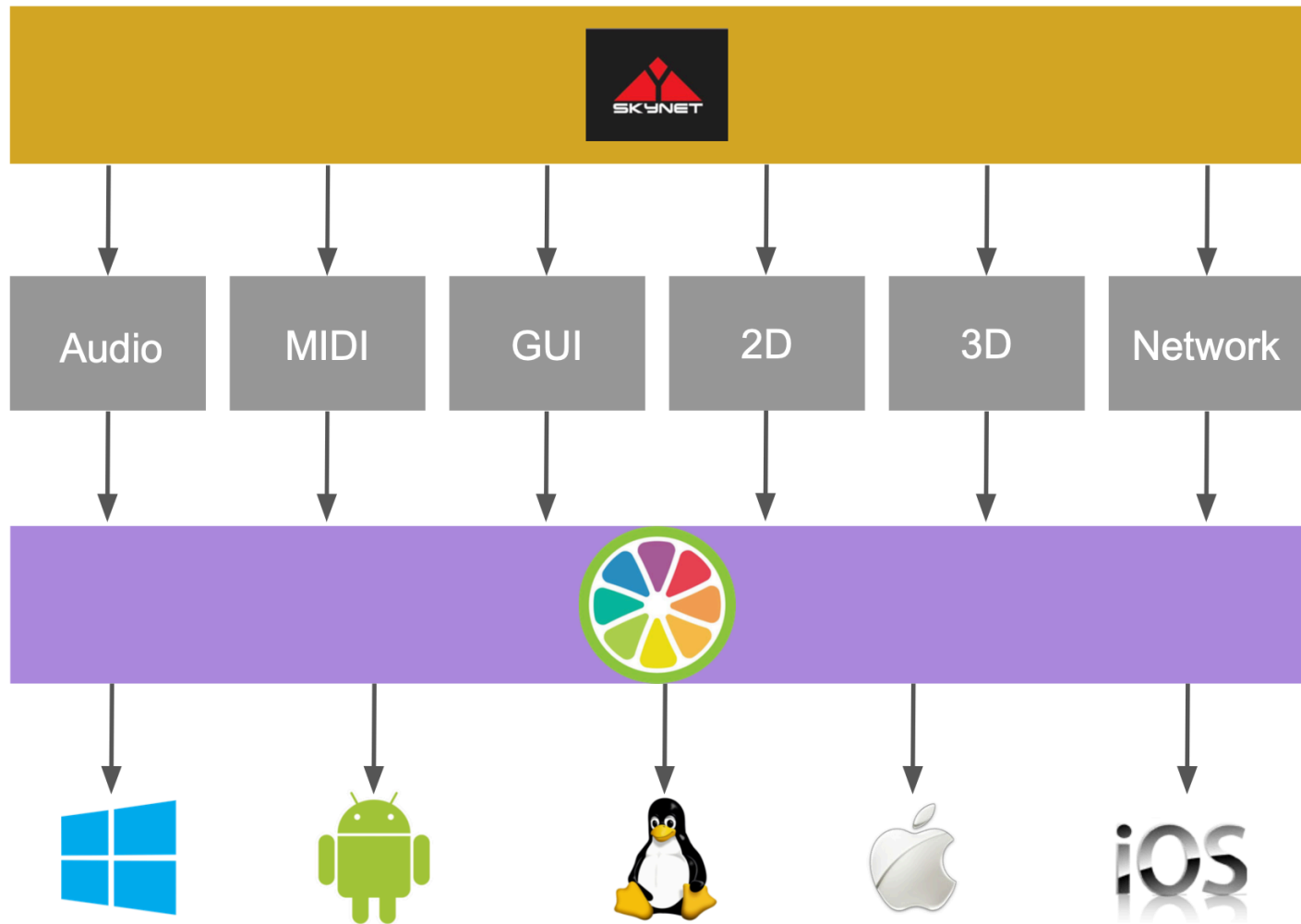


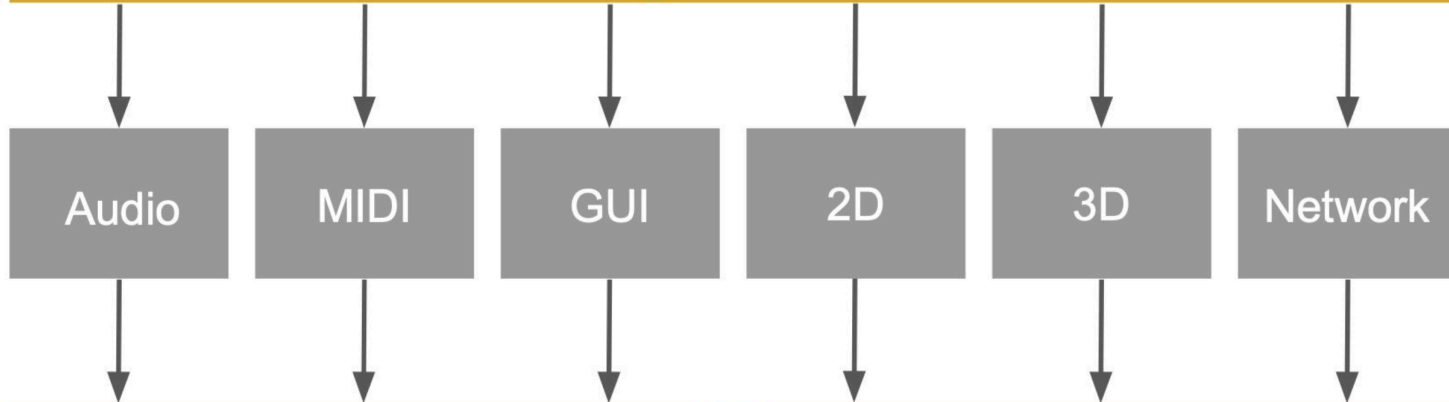














```
{  
  g.setColour (juce::Colours::blue);  
  g.drawRect (0, 0, 100, 50);  
  
  g.setColour (juce::Colours::red);  
  g.drawRect (100, 0, 100, 50);  
}
```



# Creating a plug-in using JUCE

# Creating a plug-in using JUCE

- Set up a JUCE project using the Projucer
- Write the C++ plug-in code in an IDE
- Compile the different plug-in formats
- Add plug-in parameters
- Create a GUI
- Test the plug-in

# Workshop materials

- workspace is where we'll edit the source code of our plug-ins
- Numbered directories contain source code snapshots for each section
- Plugins is where you will find your compiled plug-ins
- TestHost is where you will find the TestHost application
- Projucer is where you will find the Projucer application

# #01: Creating a plug-in project

We're now using the workshop checkpoints

The #01 in the title signals that the corresponding checkpoint for the beginning of the section is the 01 directory

To return to this point in the workshop you can delete the contents of the workspace folder and copy the contents of the 01 folder into the workspace folder

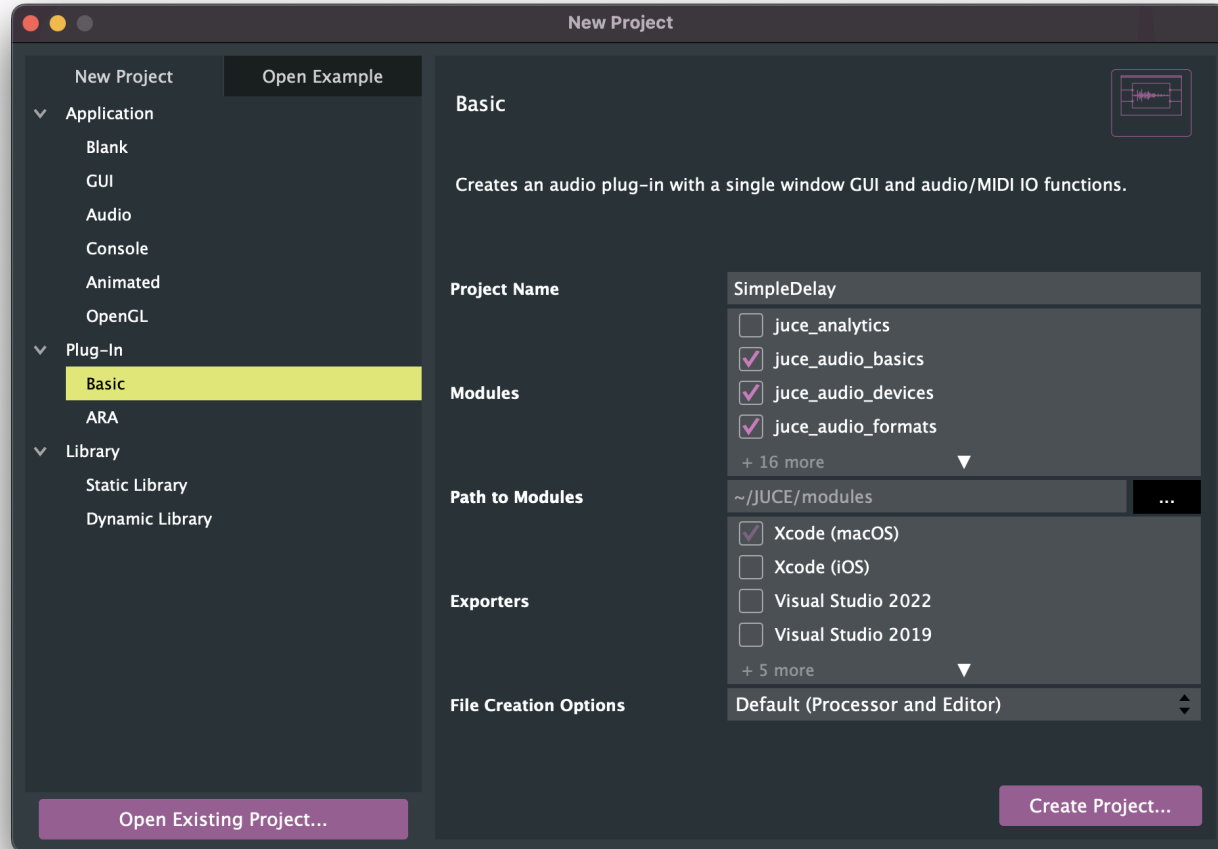
We've already done this for step 01, so we're all starting from the same place

# #01: Creating a plug-in project

Objectives of this section:

- Go through the common project configurations settings in the Projucer
- Export the project and open it in your IDE of choice (Xcode, Visual Studio, ...)
- Build an empty project
- Load the plug-in in the TestHost

# #01: Creating a plug-in project



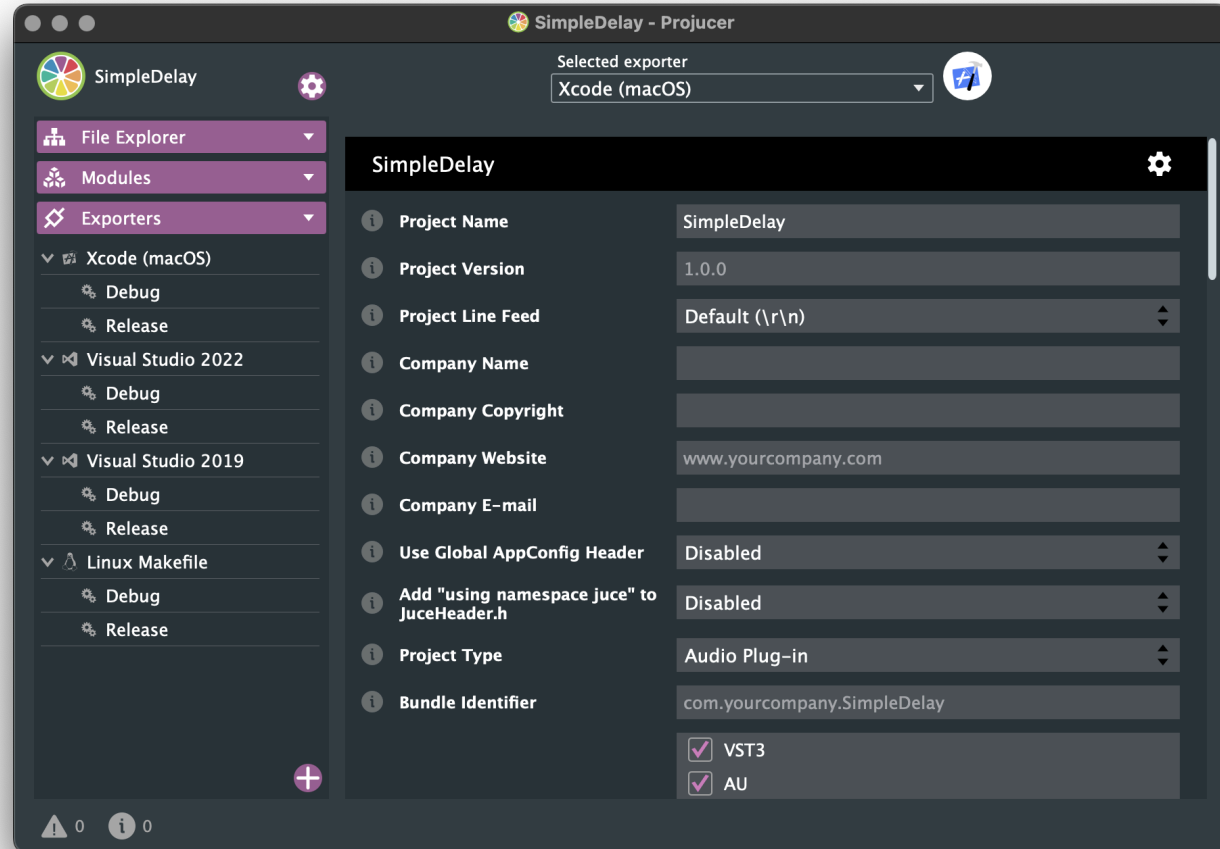
# #01: Creating a plug-in project

Open the Projucer: Projucer/[your platform]/Projucer(.app/.exe)

Open the workspace/SimpleDelay.jucer project

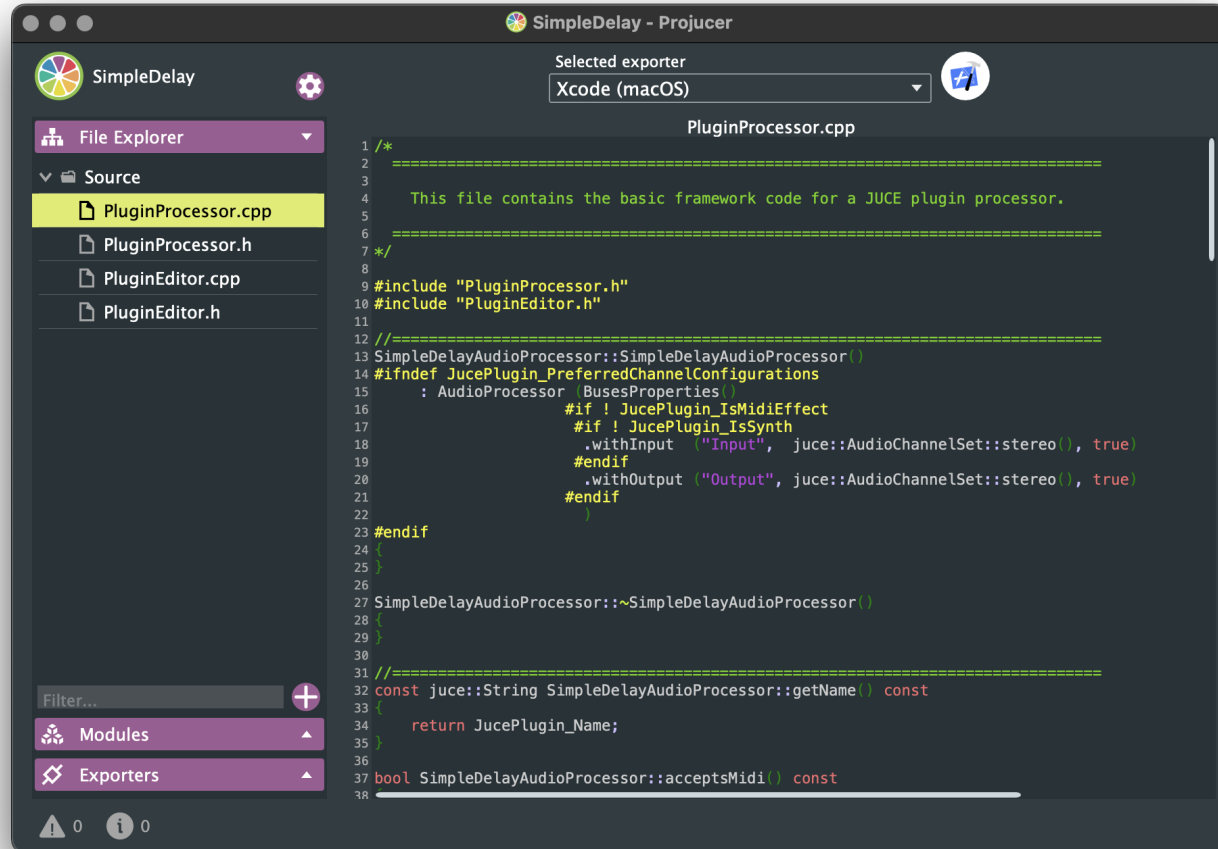
- File -> Open -> Select SimpleDelay.jucer in the workspace folder

# #01: Project settings

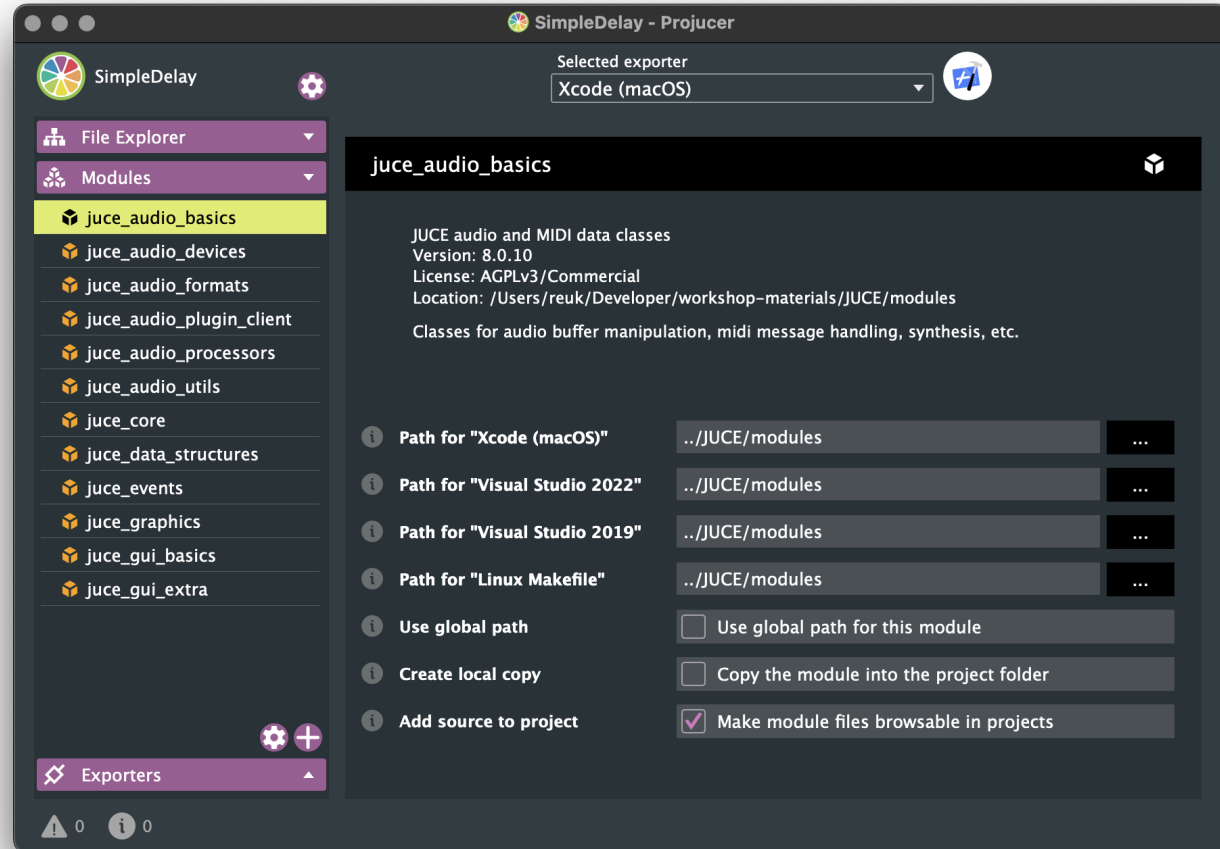




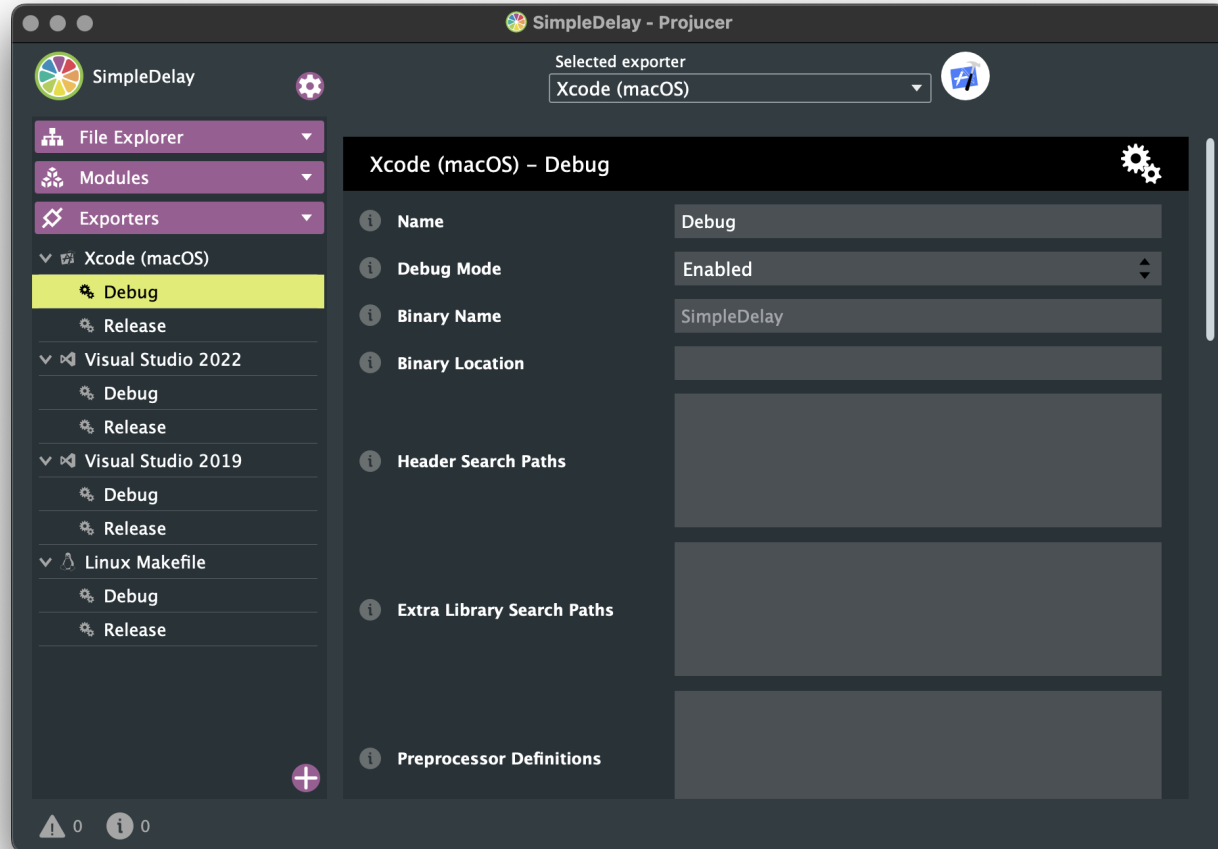
# #01: File browser



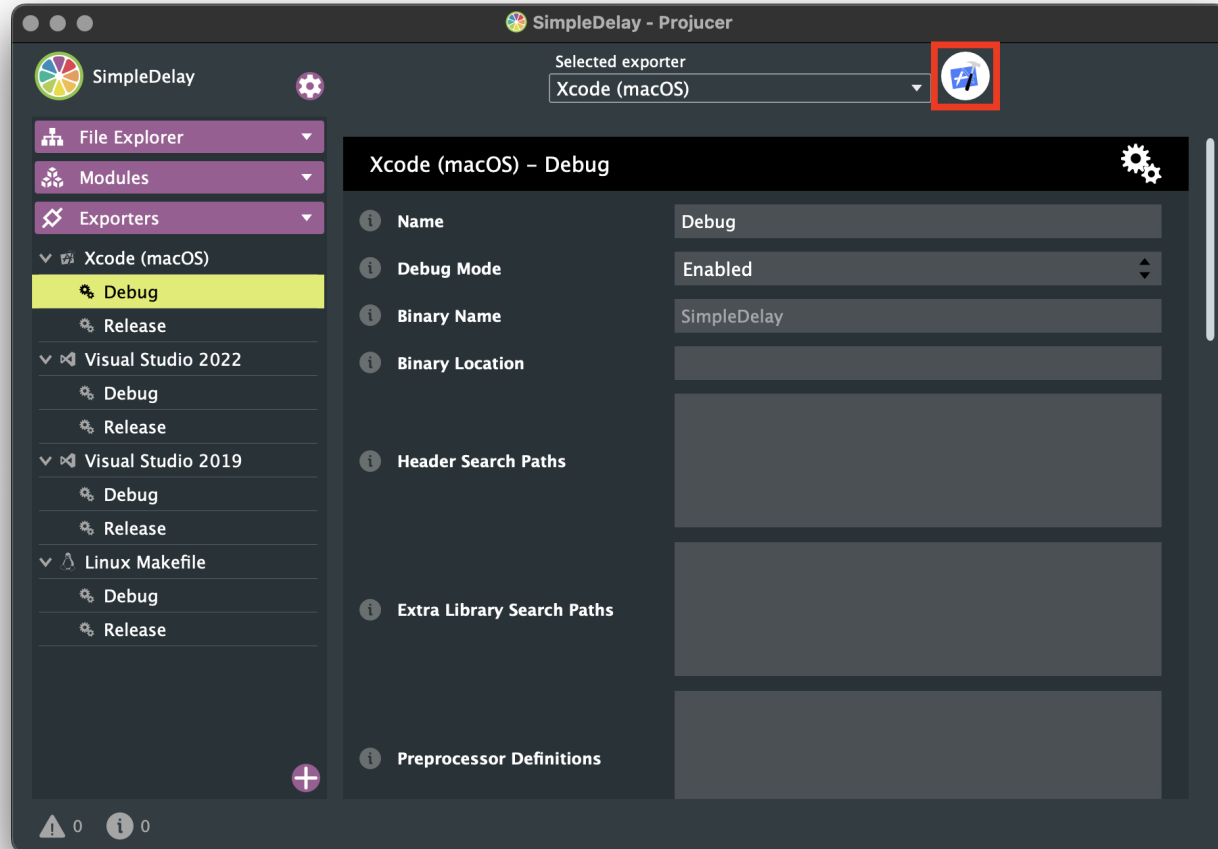
# #01: Module settings



# #01: Exporter Settings



# #01: Generate IDE project files



# #01: Generate IDE project files

We've seen a lot of Projucer settings but we don't need to change any for this workshop!

Creating a Linux Makefile from the Projucer is a little different - there are no IDEs associated with Makefiles so after saving your project you need to run make from the command line

```
cd workspace/Buils/LinuxMakefile  
make
```

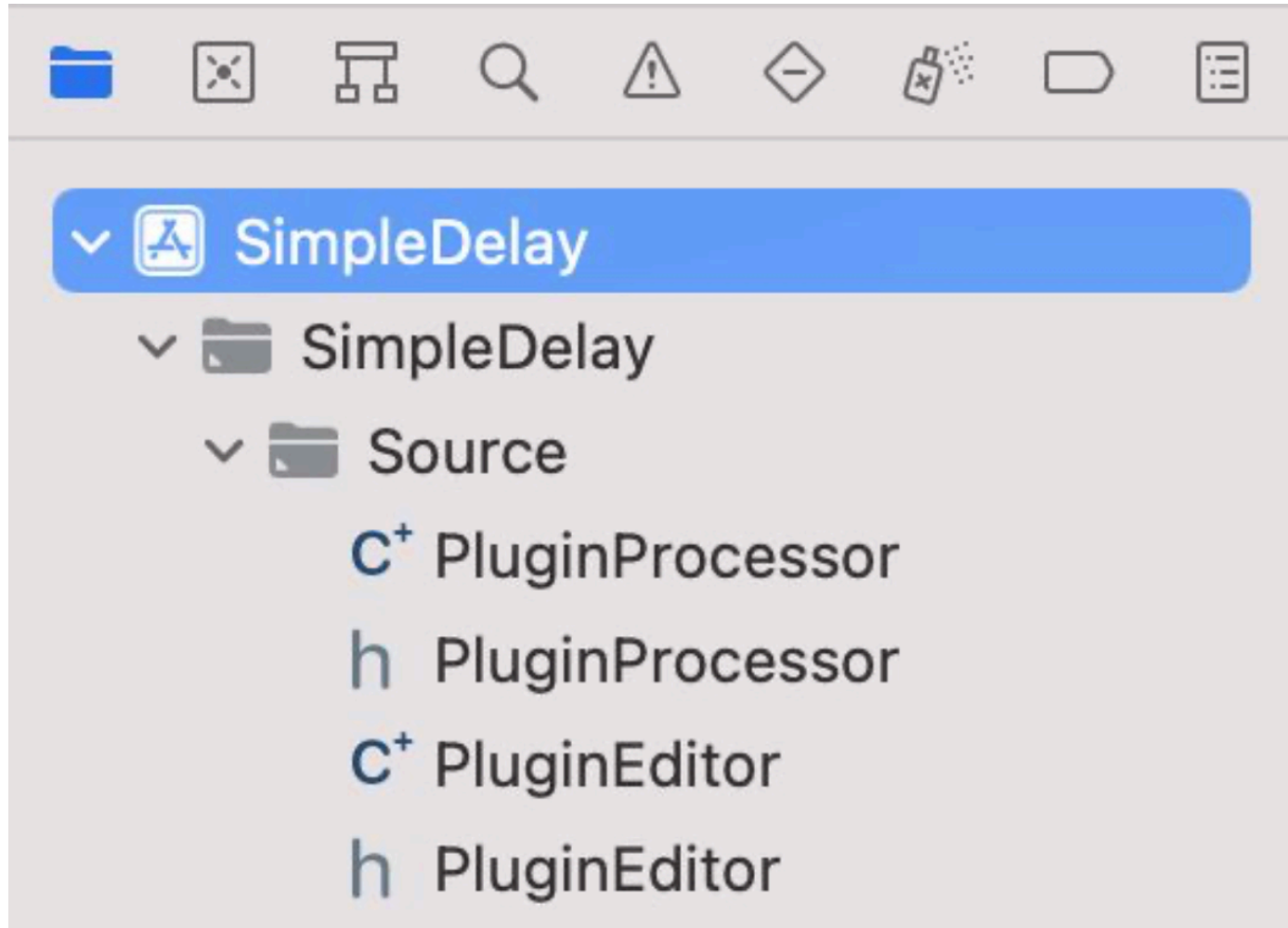
# #01: CMake

If you are comfortable using CMake then there is a CMakeLists.txt in the root directory.

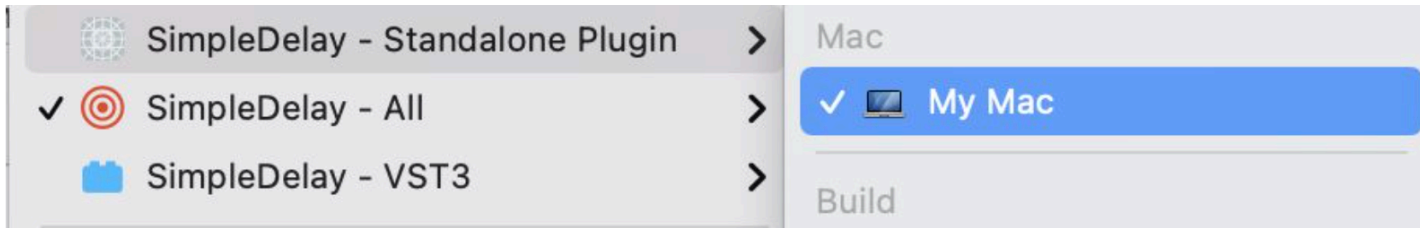
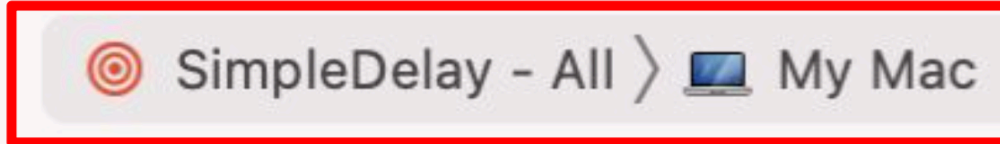
# on macOS, add [-G Xcode] to generate an Xcode project `cmake -B./build`

```
cmake --build ./build --config Debug --target workspace_SimpleDelay_All
```

# #01: Project structure

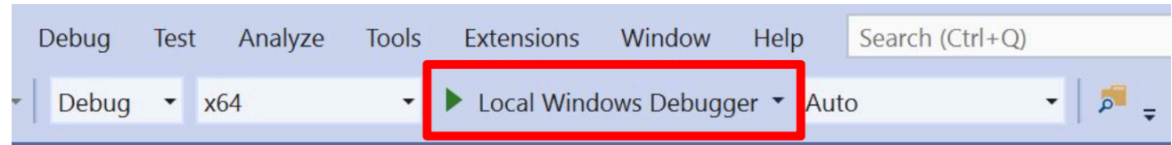
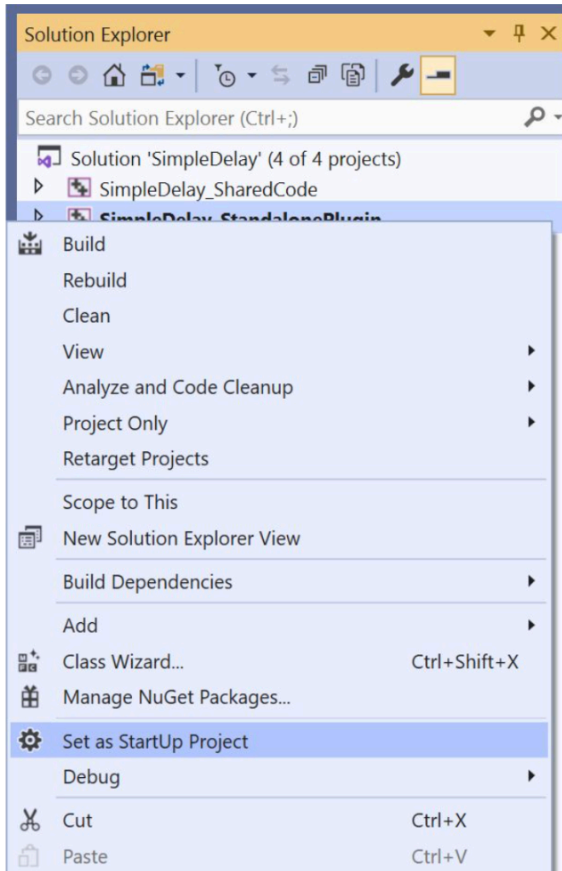


# #01: Running Standalone (macOS)





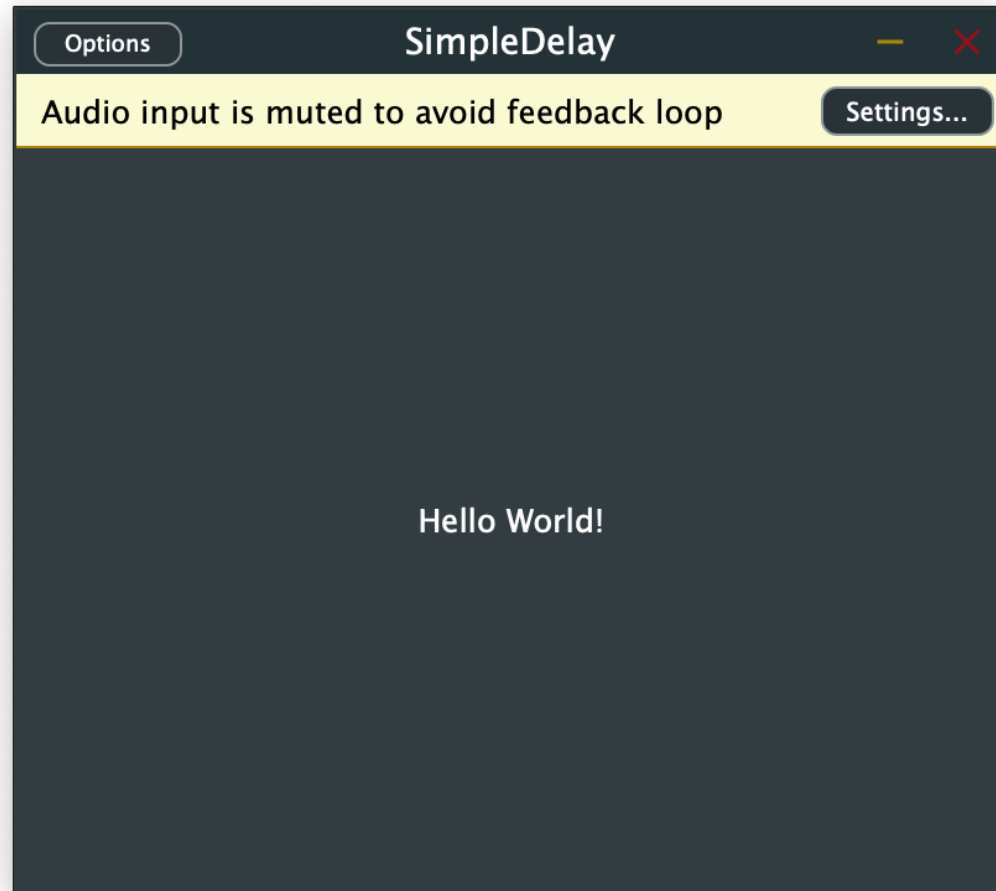
# #01: Running Standalone (Windows)



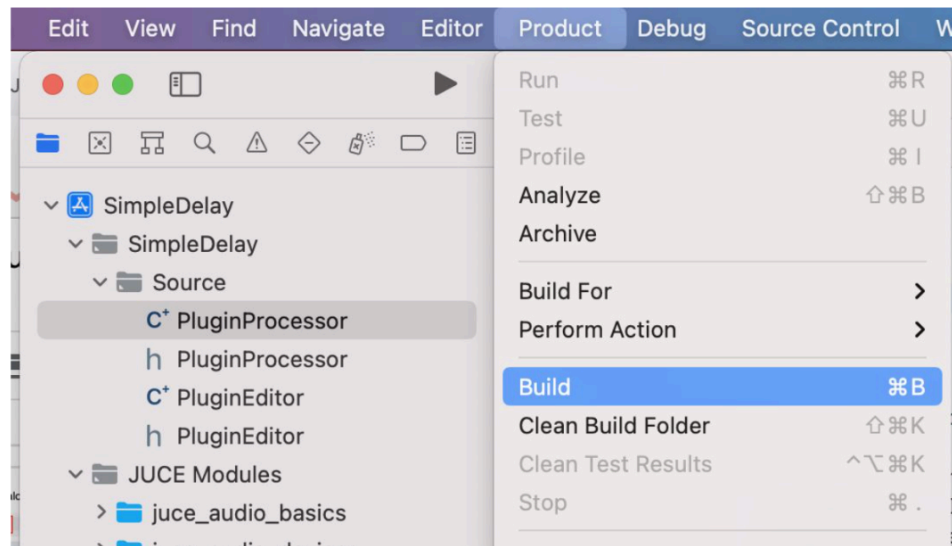
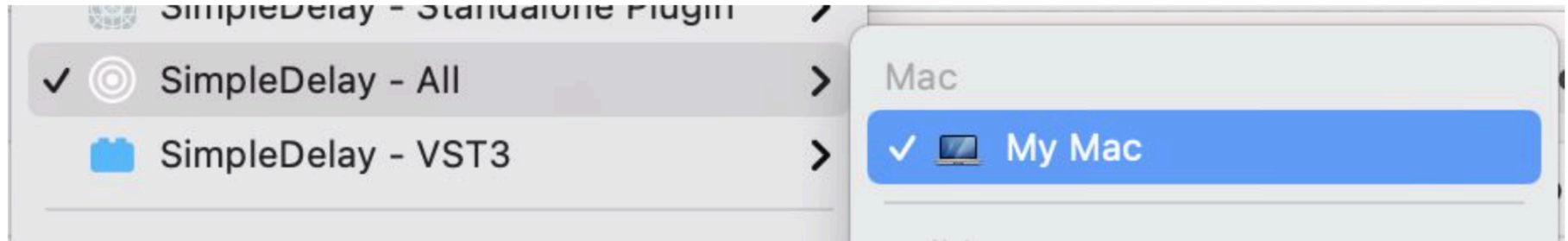
# #01: Running Standalone (Linux)

```
cd workspace/Buils/LinuxMakefile  
make  
./build/SimpleDelay
```

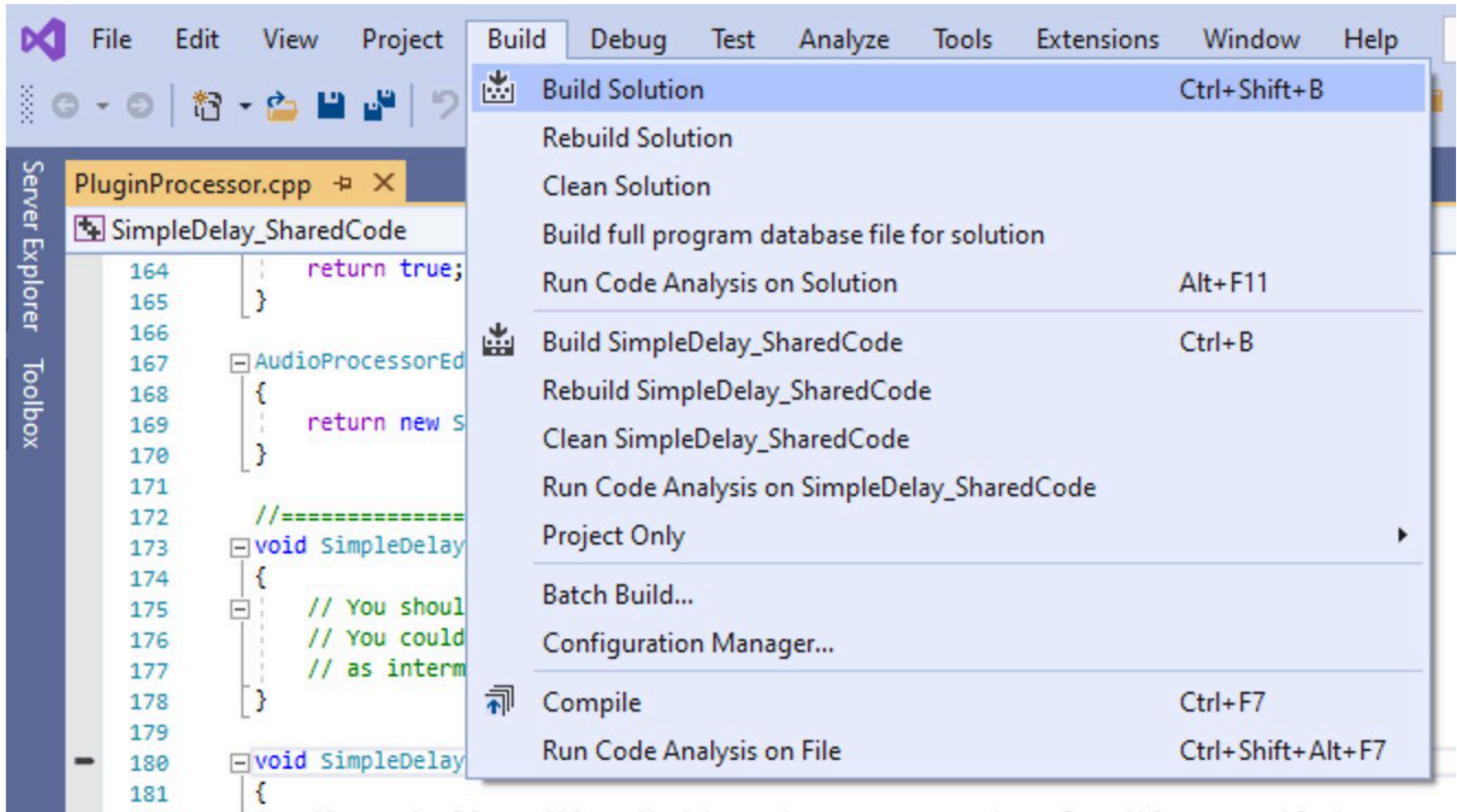
# #01: Running the Standalone App



# #01: Building plug-ins (macOS)



# #01: Building plug-ins (Windows)



# #01: Building plug-ins (Linux)

```
cd workspace/Buils/LinuxMakefile  
make
```

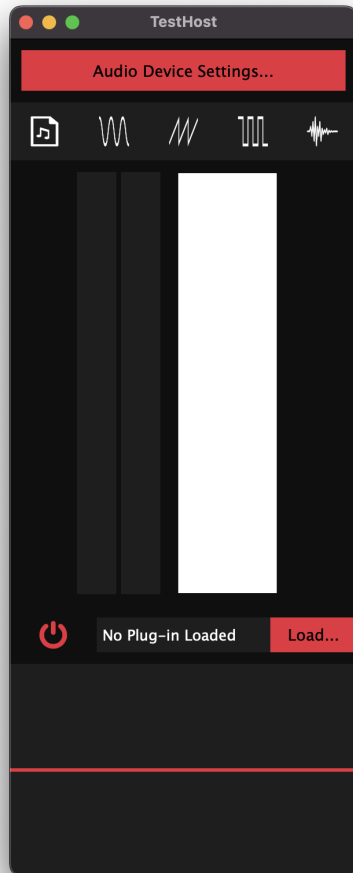
# #01: Running in the TestHost

A simple host application for testing our plug-in can be found in the workshop materials:

- TestHost/macOS/TestHost.app
- TestHost/Win64/TestHost.exe
- TestHost/Linux/TestHost

Launch the one appropriate for your platform.

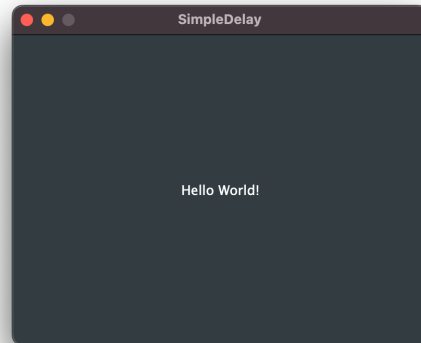
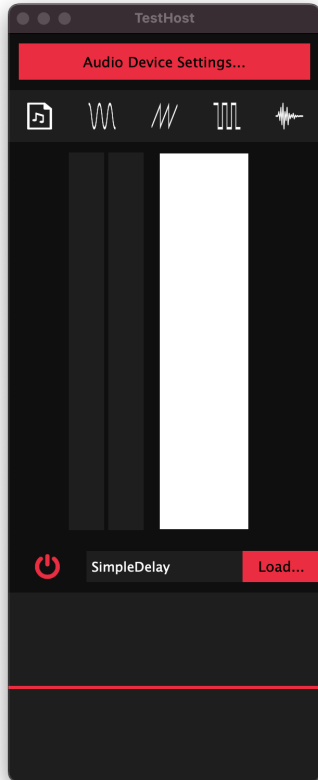
# #01: Running in the TestHost



- Play different sounds
- Plug-in section
- Meters
- Oscilloscope



# #01: Running in the TestHost



- Find the SimpleDelay plug-in that you've built in the Plugins folder and drop it onto the plug-in section of the TestHost app.
- Click on the plug-in name to open the UI.

**QUICK BREAK**

## #02: Modifying audio

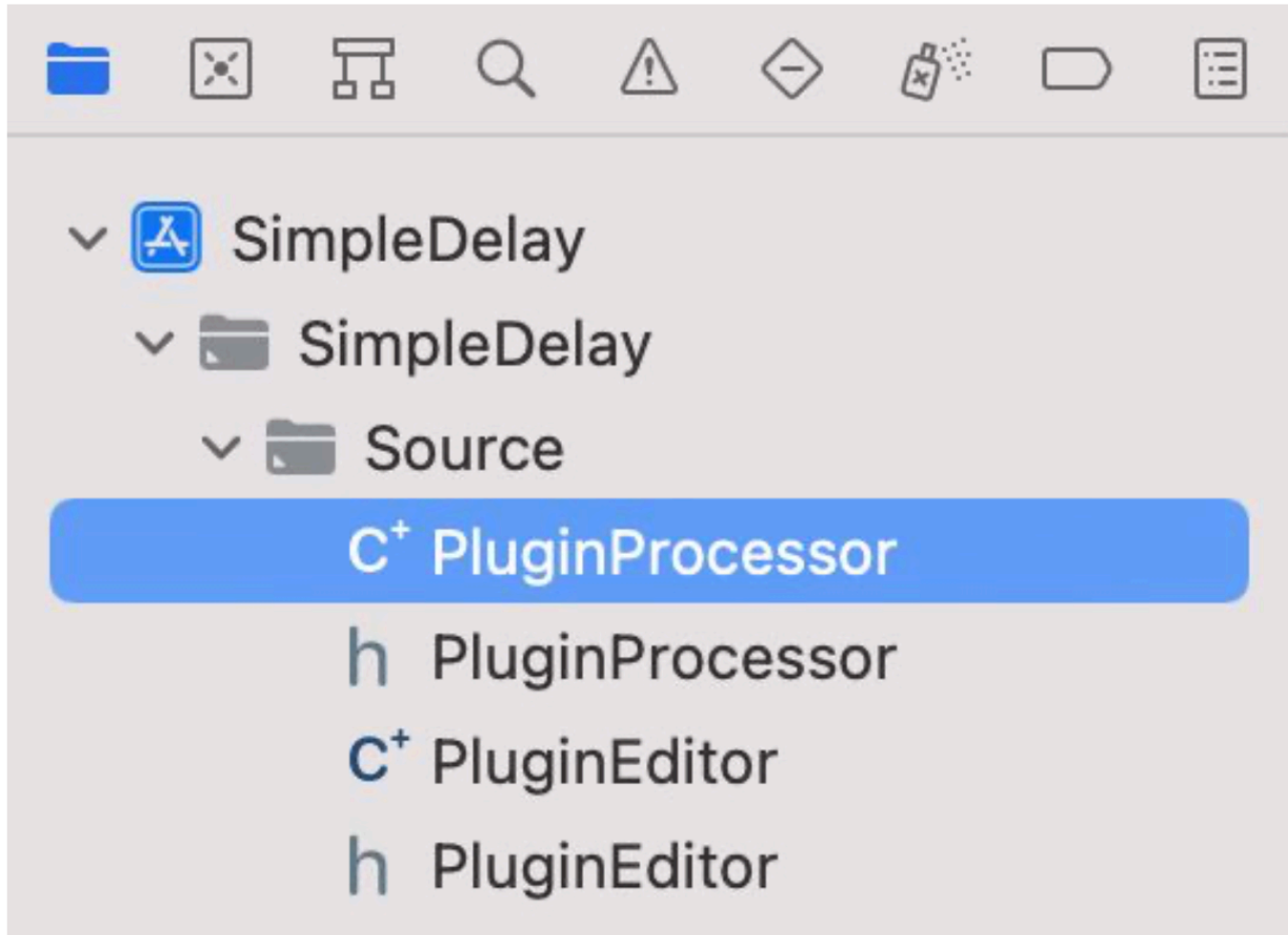
Open your IDE project file:

- `workspace\Builds\VisualStudio2022\SimpleDelay.sln`
- `workspace/Builds/macOS/SimpleDelay.xcodeproj`

On Linux open the source file directly:

- `workspace/Source/PluginProcessor.cpp`

## #02: Edit the AudioProcessor



## #02: Modifying audio

The SimpleDelayAudioProcessor class has a lot of methods

- These have been automatically created by a Projucer template
- Defines a lot of the behavior of the plug-in

The only one needed for this section is processBlock

## #02: What does processBlock do?

processBlock is called repeatedly by the plug-in host with a chunk of audio to process

Be careful here ! processBlock will be called on the audio thread, which is not the same as that used for drawing a GUI or handling mouse events (more on this a little later)

Don't use headphones when developing

## #02: juce::AudioBuffer<float>

`AudioBuffer` is a class containing non-interleaved channels of audio data represented as floats, and methods to access and modify them.

- Non-interleaved : continuous lengths of audio data for each channel
- Floats: each sample is represented by a floating point number in the range -1.0 to 1.0

`processBlock` is passed a reference to an `AudioBuffer` containing the incoming audio data. We create an audio effect by modifying this data.

For plug-ins the number of samples to process each block usually corresponds to the “buffer size” setting of the host (1024, 512, ...)

# #02: Modifying audio

```
void SimpleDelayAudioProcessor::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...
    }
}
```



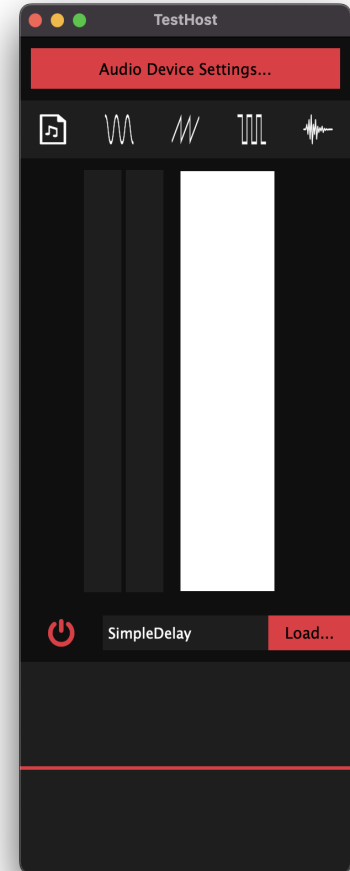
## #02: Fixed gain reduction

```
for (int channel = 0; channel < totalNumInputChannels; ++channel)
{
    auto* channelData = buffer.getWritePointer (channel);

    for (int i = 0; i < buffer.getNumSamples(); ++i)
        channelData[i] *= 0.2f;
}
```

# #02: A functional plug-in!

- Compile your projects
- Open TestHost
- This should reload the last-opened plug-in location
- If this doesn't work then find the plug-in created in the Plugins directory and drag it onto the plug-in section
- Play some sounds
- Toggle the bypass
- Hear the gain reduction



# #03: Plug-in parameters

Parameters are how plug-in hosts control plug-ins

They are exposed as part of the plug-in format's interface

- Hosts can query plug-ins to find out what parameters they offer

Parameters can be changed via:

- The plug-in's GUI
- Host-provided plug-in views
- Automation (although you can mark parameters as being non-automatable)
- Preset switching

# #03: Parameter types

Use classes derived from `AudioProcessorParameter`

- Provide methods for getting and setting parameter values and properties
- Added to, and then managed by, your `AudioProcessor`

JUCE provides some basic types to get you started

- `AudioParameterFloat`
- `AudioParameterBool`
- `AudioParameterChoice`
- `AudioParameterInt`

## #03: Adding a parameter

```
SimpleDelayAudioProcessor::SimpleDelayAudioProcessor()  
#ifndef JUCE_PLUGIN_PREFERRED_CHANNEL_CONFIGURATIONS  
    : AudioProcessor (BusesProperties()  
        #if ! JUCE_PLUGIN_IS_MIDI_EFFECT  
        #if ! JUCE_PLUGIN_IS_SYNTH  
            .withInput ("Input",  juce::AudioChannelSet::stereo(), true)  
        #endif  
            .withOutput ("Output", juce::AudioChannelSet::stereo(), true)  
        #endif  
    )  
#endif  
{  
}
```

## #03: Adding a parameter

```
SimpleDelayAudioProcessor::SimpleDelayAudioProcessor()
#ifdef JUCE_PLUGIN_PREFERRED_CHANNEL_CONFIGURATIONS
    : AudioProcessor (BusesProperties()
        #if ! JUCE_PLUGIN_IS_MIDI_EFFECT
        #if ! JUCE_PLUGIN_IS_SYNTH
            .withInput ("Input",  juce::AudioChannelSet::stereo(), true)
        #endif
            .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
        #endif
    )
#endif
{
    addParameter (new juce::AudioParameterFloat ({ "gain", 1 }, "Gain", 0.0f, 1.0f, 1.0f));
}
```

## #03: Using a parameter in processBlock

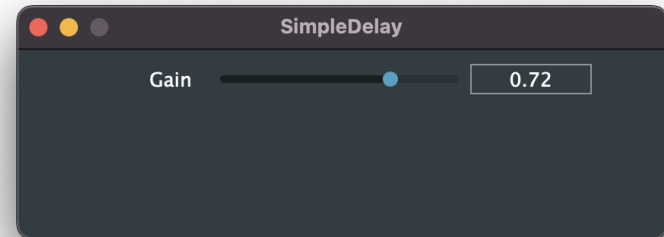
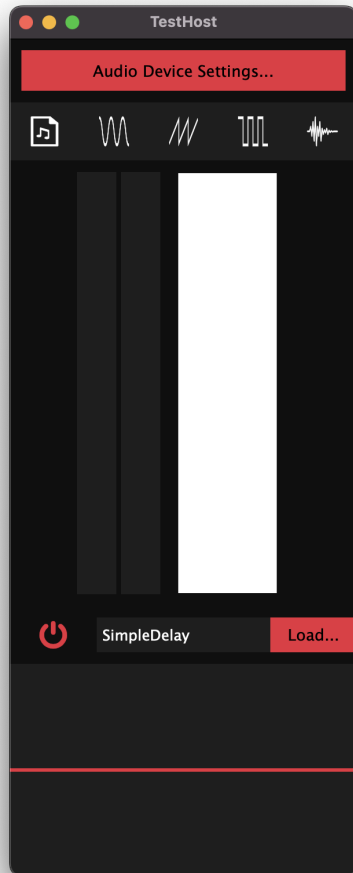
```
juce::AudioProcessorParameter* gainParameter = getParameters()[0];  
float gain = gainParameter->getValue();  
  
for (int channel = 0; channel < totalNumInputChannels; ++channel)  
{  
    float* channelData = buffer.getWritePointer (channel);  
  
    for (int i = 0; i < buffer.getNumSamples(); ++i)  
        channelData[i] *= gain;  
}
```

## #03: Use an automatically generated user interface

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



# #03: Compile the plug-in and load it



# #04: Creating the delay effect

Let's add some less trivial audio processing

- Implement a basic delay effect
- Add some more parameters
- Configure our audio processing algorithm in `prepareToPlay`

This section requires quite a lot of typing to complete. Start from the contents of the 05 folder and review the changes rather than typing along.

## #04: What is a basic delay effect?

An echo of the incoming audio

The audio at time  $t$  is combined with a recursively attenuated audio at time  $(t - nD)$

where  $D$  is the delay time and  $n = 1, 2, 3, 4, \dots$

Increasing the delay time  $D$  increases the time between echos

Increasing the attenuation decreases the time taken for the echos to fade away

Our SimpleDelay plug-in will feature a fixed delay time, but a variable feedback (opposite of attenuation) with a dry/wet mix control

# #04: Add some more parameters

```
SimpleDelayAudioProcessor::SimpleDelayAudioProcessor()
#ifdef JUCE_PLUGIN_PREFERRED_CHANNEL_CONFIGURATIONS
    : AudioProcessor (BusesProperties()
        #if ! JUCE_PLUGIN_IS_MIDI_EFFECT
        #if ! JUCE_PLUGIN_IS_SYNTH
            .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
        #endif
            .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
        #endif
    )
#endif
{
    addParameter (new juce::AudioParameterFloat ({ "gain",      1 }, "Gain",          0.0f, 1.0f, 1.0f));
    addParameter (new juce::AudioParameterFloat ({ "feedback",  1 }, "Delay Feedback", 0.0f, 1.0f, 0.35f));
    addParameter (new juce::AudioParameterFloat ({ "mix",        1 }, "Dry / Wet",      0.0f, 1.0f, 0.5f));
}
```

# #04: The delay algorithm

A simple circular buffer of audio history

- Audio samples are continuously written to the buffer in processBlock
- When the capacity is exceeded we go back to the start
- Existing audio data is attenuated, new audio data added on top
- The size of the circular buffer will determine the (fixed) delay

# #04: Add a circular buffer to PluginProcessor.h

```
private:
//=====
int delayBufferPos = 0;
juce::AudioBuffer<float> delayBuffer;

JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (SimpleDelayAudioProcessor)
```

## #04: Configure the buffer in prepareToPlay

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

    int delayMilliseconds = 200;
    auto delaySamples = (int) std::round (sampleRate * delayMilliseconds / 1000.0);
    delayBuffer.setSize (2, delaySamples);
    delayBuffer.clear();
    delayBufferPos = 0;
}
```

# #04: Implement delay algorithm in processBlock

```
auto& parameters = getParameters();
float gain      = parameters[0]->getValue();
float feedback  = parameters[1]->getValue();
float mix       = parameters[2]->getValue();

int delayBufferSize = delayBuffer.getNumSamples();

for (int channel = 0; channel < totalNumInputChannels; ++channel)
{
    float* channelData = buffer.getWritePointer (channel);
    int delayPos = delayBufferPos;

    for (int i = 0; i < buffer.getNumSamples(); ++i)
    {
        float drySample = channelData[i];

        float delaySample = delayBuffer.getSample (channel, delayPos) * feedback;
        delayBuffer.setSample (channel, delayPos, drySample + delaySample);

        delayPos++;
        if (delayPos == delayBufferSize)
            delayPos = 0;

        channelData[i] = (drySample * (1.0f - mix)) + (delaySample * mix);
        channelData[i] *= gain;
    }
}

delayBufferPos = (delayBufferPos + buffer.getNumSamples()) % delayBufferSize;
```



# #05: Parameter management and state

Use an `AudioProcessorValueTreeState` to manage your parameters

- Improved parameter handling
- Serialise and deserialise your plug-ins state
- (Later) Link parameters to GUI elements

This section requires quite a lot of typing to complete. Start from the contents of the 06 folder and review the changes rather than typing along.

# #05: Adding an AudioProcessorValueTreeState

```
private:
    //=====
    juce::AudioProcessorValueTreeState state;
    int delayBufferPos = 0;
    juce::AudioBuffer<float> delayBuffer;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (SimpleDelayAudioProcessor)
```

# #05: Adding an AudioProcessorValueTreeState

```
SimpleDelayAudioProcessor::SimpleDelayAudioProcessor()
#ifdef JUCE_PLUGIN_PREFERRED_CHANNEL_CONFIGURATIONS
    : AudioProcessor (BusesProperties()
        #if ! JUCE_PLUGIN_IS_MIDI_EFFECT
        #if ! JUCE_PLUGIN_IS_SYNTH
            .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
        #endif
            .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
        #endif
    )
#endif
, state (*this, nullptr, "STATE", {
    std::make_unique<juce::AudioParameterFloat> (juce::ParameterID { "gain",      1 }, "Gain",          0.0f, 1.0f, 1.0f),
    std::make_unique<juce::AudioParameterFloat> (juce::ParameterID { "feedback",  1 }, "Delay Feedback", 0.0f, 1.0f, 0.35f),
    std::make_unique<juce::AudioParameterFloat> (juce::ParameterID { "mix",       1 }, "Dry / Wet",      0.0f, 1.0f, 0.5f)
})
{
}
```

# #05: Parameter management

```
auto& parameters = getParameters();  
float gain      = parameters[0]->getValue();  
float feedback  = parameters[1]->getValue();  
float mix       = parameters[2]->getValue();
```

becomes

```
float gain      = state.getParameter ("gain")->getValue();  
float feedback  = state.getParameter ("feedback")->getValue();  
float mix       = state.getParameter ("mix")->getValue();
```

## #05: Saving and restoring plug-in state

When plug-in hosts save and load projects, each plug-in must save and restore its state

- `getStateInformation (juce::MemoryBlock& destData)`
- `setStateInformation (const void* data, int sizeInBytes)`

The plug-in's state must be serialised to, and deserialised from, a block of memory managed by the host.

## #05: Saving plug-in state

```
void SimpleDelayAudioProcessor::getStateInformation (juce::MemoryBlock& destData)
{
    // You should use this method to store your parameters in the memory block.
    // You could do that either as raw data, or use the XML or ValueTree classes
    // as intermediaries to make it easy to save and load complex data.

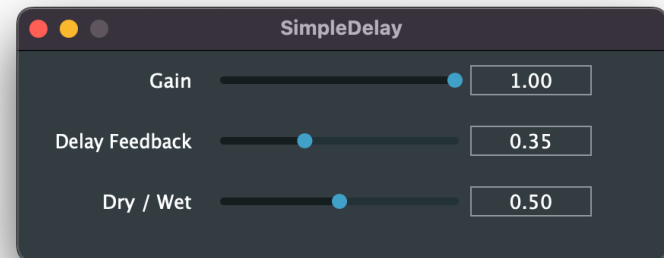
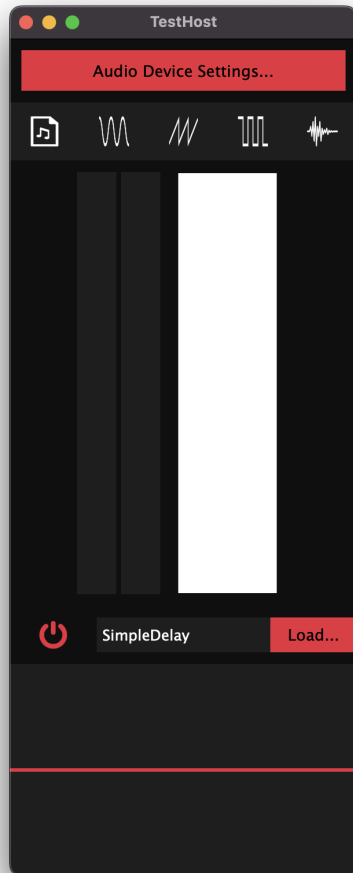
    if (auto xmlState = state.copyState().createXml())
        copyXmlToBinary (*xmlState, destData);
}
```

## #05: Restoring plug-in state

```
void SimpleDelayAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    // You should use this method to restore your parameters from this memory block,
    // whose contents will have been created by the getStateInformation() call.

    if (auto xmlState = getXmlFromBinary (data, sizeInBytes))
        state.replaceState (juce::ValueTree::fromXml (*xmlState));
}
```

# #05: Have a play!





**BREAK**

# #06: Adding a GUI

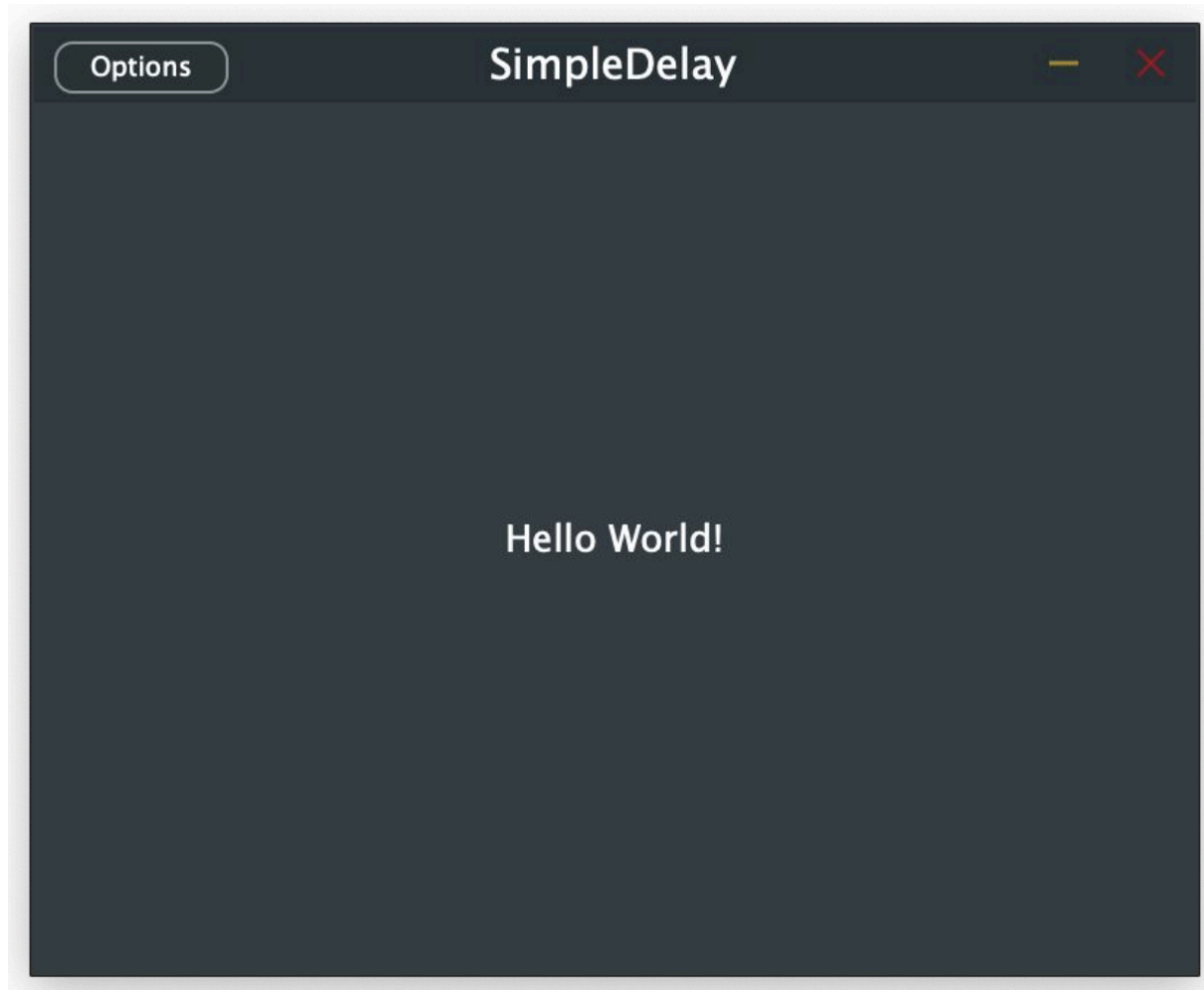
Let's display a custom GUI and draw some graphics

- Basic shapes
- Text

## #06: Remove the generic GUI

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

## #06: Our custom GUI



# #06: Drawing in paint

```
void SimpleDelayAudioProcessorEditor::paint (juce::Graphics& g)
{
    // (Our component is opaque, so we must completely fill the background with a solid colour)
    g.fillAll (getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId));

    g.setColour (juce::Colours::white);
    g.setFont (15.0f);
    g.drawFittedText ("Hello World!", getLocalBounds(), juce::Justification::centred, 1);
}
```

# #06: Threads

Be careful here!

- The GUI is rendered on the “main” (GUI, message) thread
- `processBlock` is called on the audio thread

It's very easy to create race conditions, where one thread is modifying a bit of memory whilst another thread is reading it.

This is easily the most complicated aspect of working with real-time audio.

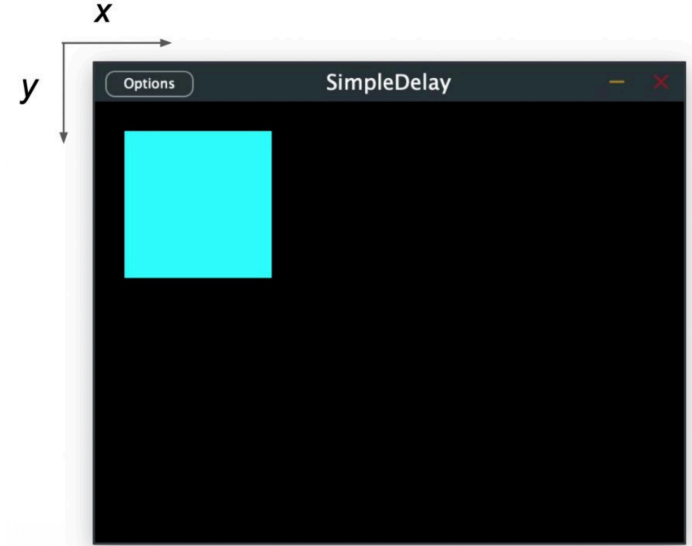
Using JUCE's `AudioParameter` classes and an `AudioProcessorValueTreeState` makes things much simpler.

# #06: Draw a square

```
void SimpleDelayAudioProcessorEditor::paint (juce::Graphics& g)
{
    g.fillAll (juce::Colours::black);

    g.setColour (juce::Colours::cyan);

    //          x    y    width  height
    g.fillRect (20, 20, 100,  100);
}
```



# #06: Lots of GUI code incoming

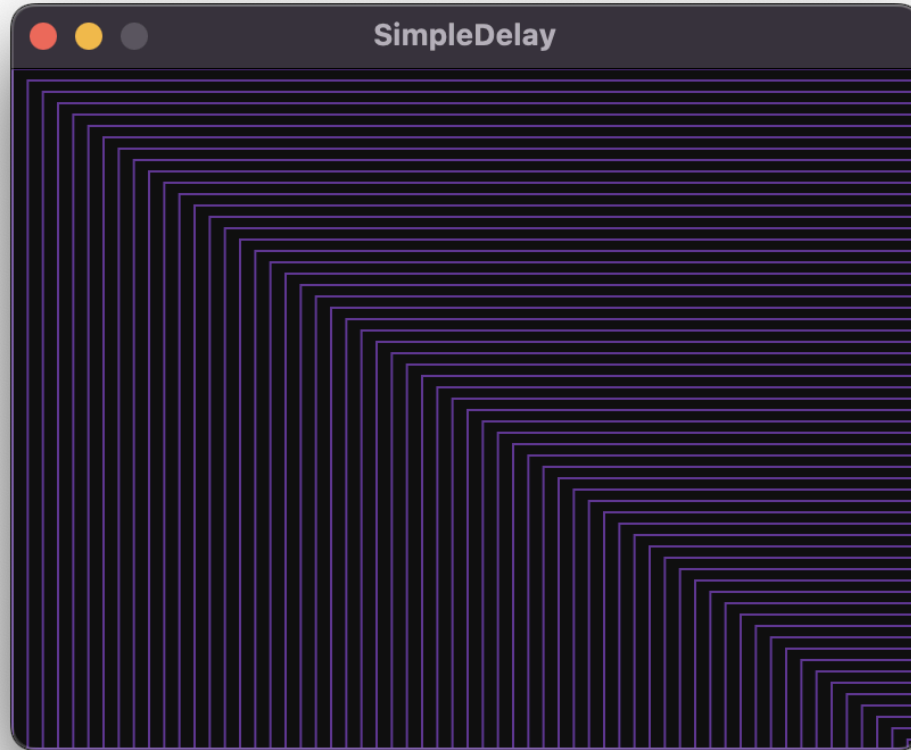
From now until the next break there will be a lot of code to add as we put together a GUI that's more than a few basic shapes

Don't worry about keeping up!

During and after the break there will be time to experiment with your own GUIs



## #06: Preview: Something more advanced



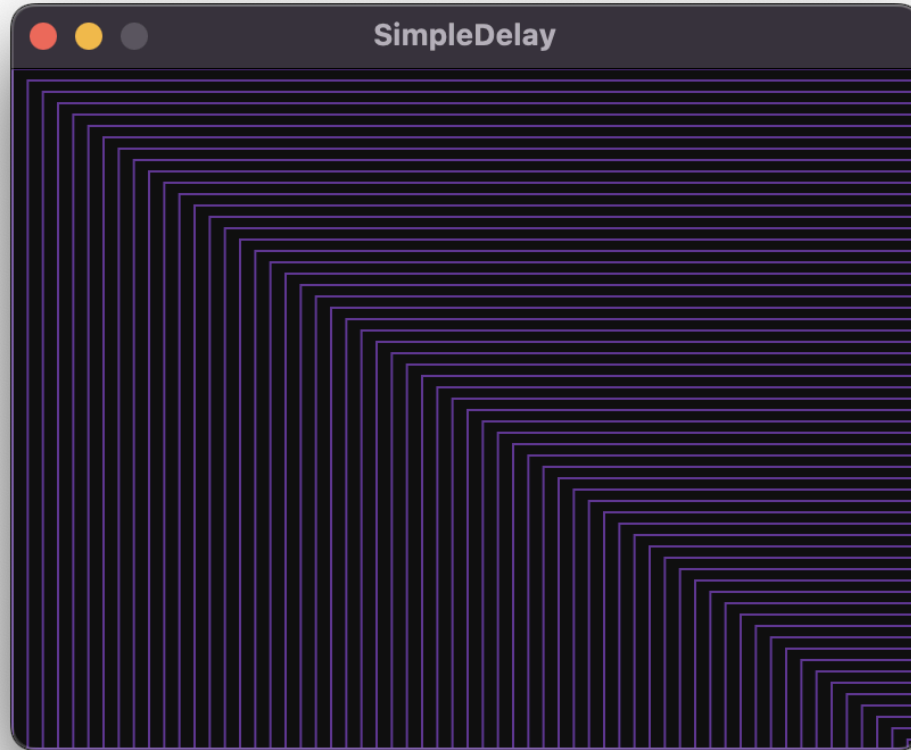
## #06: Something more advanced

```
g.fillAll (juce::Colour (0xff121212));

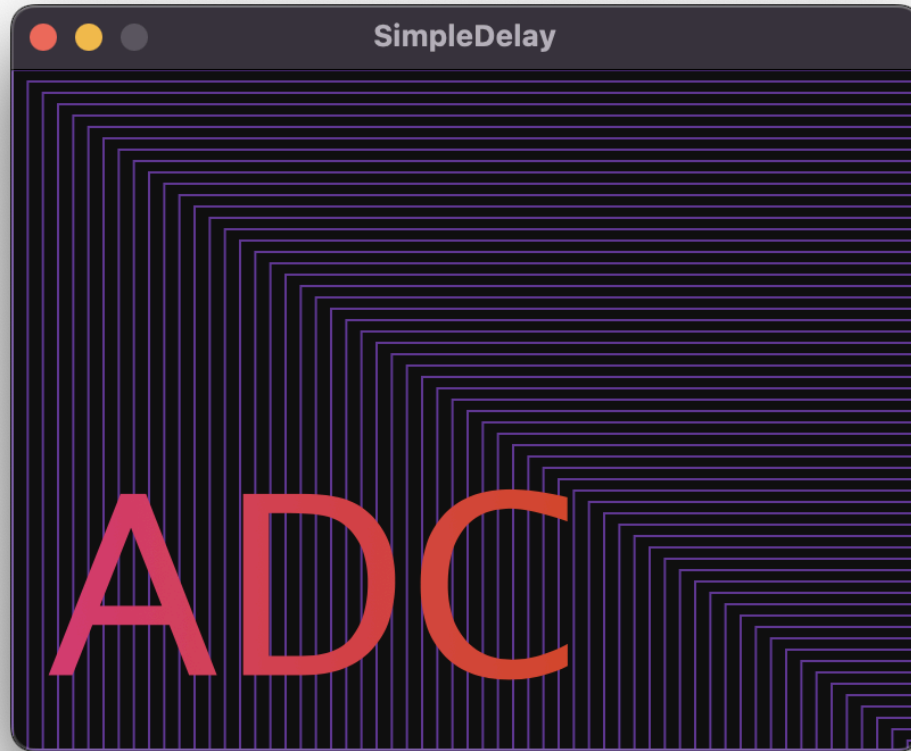
g.setColour (juce::Colours::rebeccapurple);
juce::Rectangle<float> backgroundRect = getLocalBounds().toFloat();
int numBackgroundRects = 60;
juce::Point<float> offset = backgroundRect.getBottomRight() / numBackgroundRects;

for (int i = 0; i < numBackgroundRects; ++i)
{
    g.drawRect (backgroundRect);
    backgroundRect += offset;
}
```

## #06: Something more advanced



## #06: Preview: Something more advanced



## #06: Something more advanced

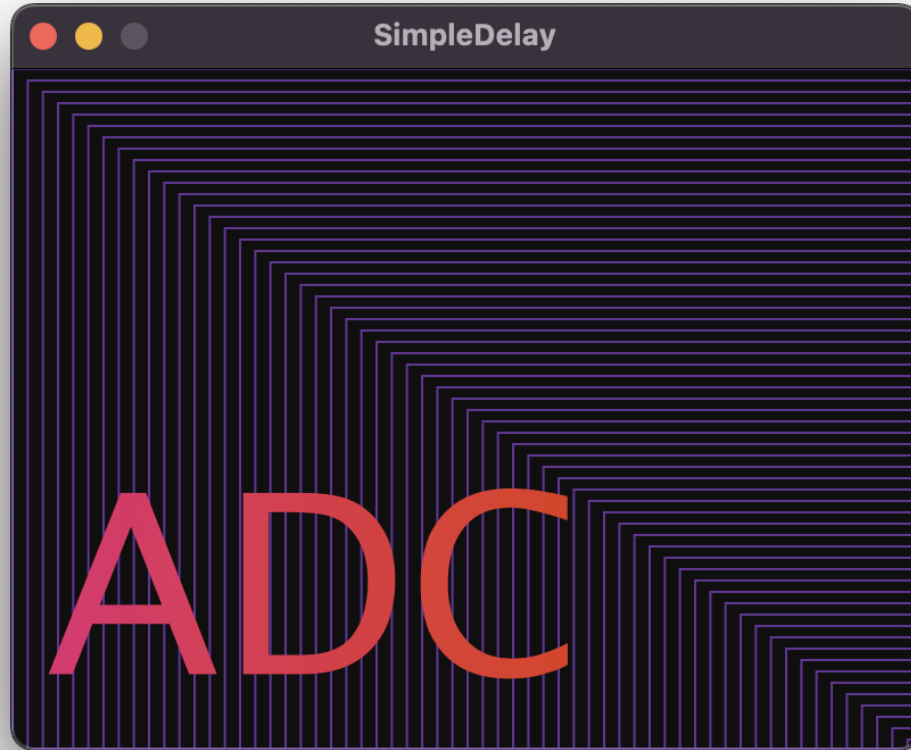
```
juce::Rectangle<int> bounds = getLocalBounds();
juce::Rectangle<int> textArea = bounds.removeFromLeft ((bounds.getWidth() * 2) / 3)
    .removeFromBottom (bounds.getHeight() / 2)
    .reduced (10);

juce::ColourGradient gradient (juce::Colour (0xffe62875),
    textArea.toFloat().getTopLeft(),
    juce::Colour (0xffe43d1b),
    textArea.toFloat().getTopRight(),
    false);

g.setGradientFill (gradient);

g.setFont (textArea.toFloat().getHeight());
g.drawFittedText ("ADC", textArea, juce::Justification::centred, 1);
```

## #06: Something more advanced



# #07: Components

Let's create some interactive GUI elements

- Add some sliders
- Handle dynamic layout changes in resized

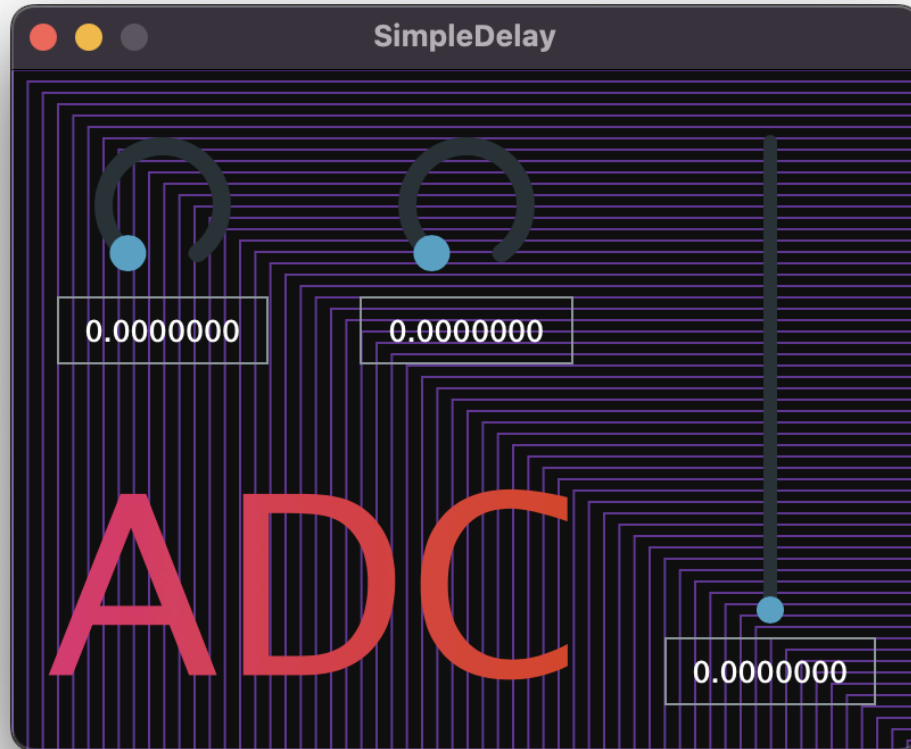
# #07: JUCE Components

JUCE GUIs are trees of components

- You can create your own; the `AudioProcessorEditor` is a `Component`
- Parent components are responsible for laying out child components
- Mouse events and keyboard focus can be passed between them
- JUCE has a selection of common widget components you can use



## #07: Preview: GUI with Sliders



# #07: Workflow for adding Components

Adding Sliders, or any other Component requires the following basic steps

1. Create a component member inside your editor class
  - Place the instance in the Editor class
  - Initialise it in the Editor constructor
2. Attach it to the component tree by calling `addAndMakeVisible()`
3. Specify the size and position in the Editor's `resized()` → flexible layout

# #07: Add some sliders to PluginEditor.h

**private:**

```
// This reference is provided as a quick way for your editor to  
// access the processor object that created it.
```

```
SimpleDelayAudioProcessor& audioProcessor;
```

```
juce::Slider gainSlider, feedbackSlider, mixSlider;
```

```
JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (SimpleDelayAudioProcessorEditor)
```

# #07: Configure the sliders in PluginEditor.cpp

```
SimpleDelayAudioProcessorEditor::SimpleDelayAudioProcessorEditor (SimpleDelayAudioProcessor& p)
    : AudioProcessorEditor (&p), audioProcessor (p)
{
    gainSlider    .setSliderStyle (juce::Slider::SliderStyle::LinearVertical);
    feedbackSlider.setSliderStyle (juce::Slider::SliderStyle::Rotary);
    mixSlider     .setSliderStyle (juce::Slider::SliderStyle::Rotary);

    for (auto* slider : { &gainSlider, &feedbackSlider, &mixSlider })
    {
        slider->setTextBoxStyle (juce::Slider::TextBoxBelow, true, 200, 30);
        addAndMakeVisible (slider);
    }

    // Make sure that before the constructor has finished, you've set the
    // editor's size to whatever you need it to be.
    setSize (400, 300);
}
```

# #07: Layout the sliders in resized

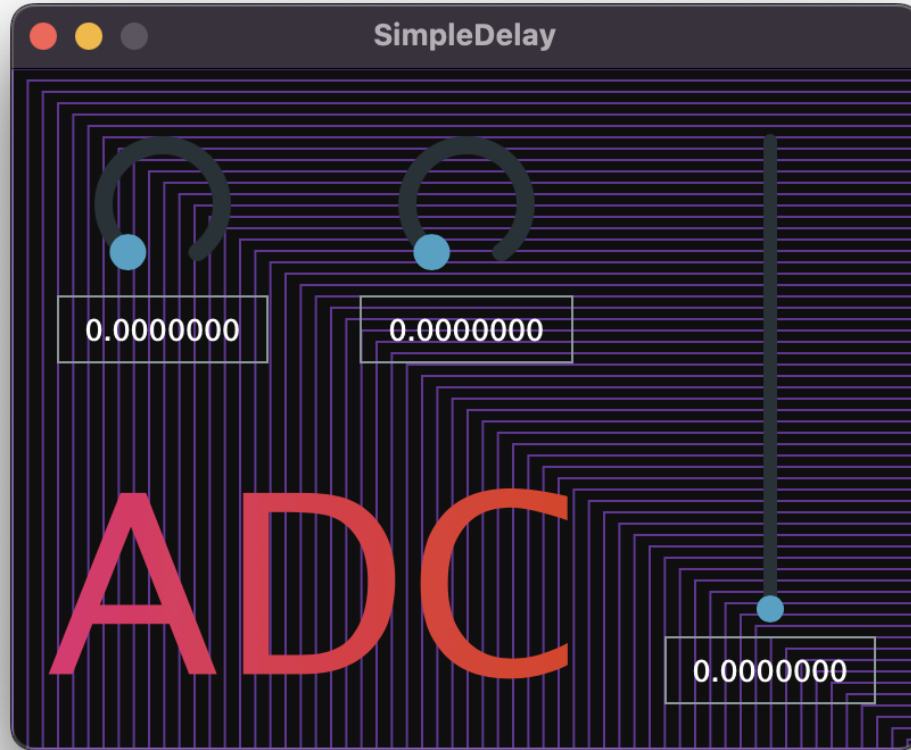
```
void SimpleDelayAudioProcessorEditor::resized()
{
    // This is generally where you'll want to lay out the positions of any
    // subcomponents in your editor..

    juce::Rectangle<int> bounds = getLocalBounds();
    int margin = 20;

    juce::Rectangle<int> gainBounds = bounds.removeFromRight (getWidth() / 3);
    gainSlider.setBounds (gainBounds.reduced (margin));

    juce::Rectangle<int> knobsBounds = bounds.removeFromTop (getHeight() / 2);
    juce::Rectangle<int> feedbackBounds = knobsBounds.removeFromLeft (knobsBounds.getWidth() / 2);
    feedbackSlider.setBounds (feedbackBounds.reduced (margin));
    mixSlider.setBounds (knobsBounds.reduced (margin));
}
```

## #07: GUI with Sliders



# #08: Connecting GUI controls to plug-in parameters

We would like to control the plug-in from the GUI

- Use the `AudioProcessorValueTreeState` attachment classes to link Sliders to plug-in parameters

# #08: Add some attachments to PluginEditor.h

```
private:
    // This reference is provided as a quick way for your editor to
    // access the processor object that created it.
    SimpleDelayAudioProcessor& audioProcessor;

    juce::Slider gainSlider, feedbackSlider, mixSlider;
    juce::AudioProcessorValueTreeState::SliderAttachment gainAttachment, feedbackAttachment, mixAttachment;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (SimpleDelayAudioProcessorEditor)
```



# #08: We need to pass the state to our editor

```
// Encapsulate this better in production code!  
juce::AudioProcessorValueTreeState state;
```

**private:**

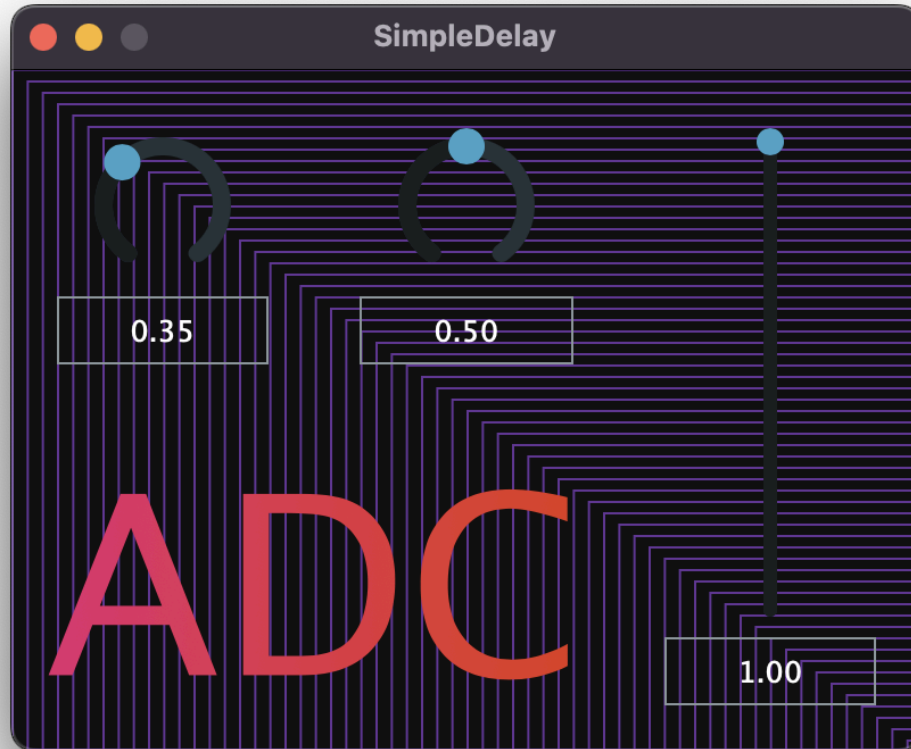
```
//=====
int delayBufferPos = 0;
juce::AudioBuffer<float> delayBuffer;

JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (SimpleDelayAudioProcessor)
```

# #08: Linking Slider to plug-in parameters

```
SimpleDelayAudioProcessorEditor::SimpleDelayAudioProcessorEditor (SimpleDelayAudioProcessor& p)
: juce::AudioProcessorEditor (&p), audioProcessor (p),
  gainAttachment      (p.state, "gain",      gainSlider),
  feedbackAttachment  (p.state, "feedback", feedbackSlider),
  mixAttachment        (p.state, "mix",      mixSlider)
```

## #09: The interactive plug-in



**BREAK**

# Testing in Hosts

# Debugging

What is a debugger?

- The most useful tool in a programmer's arsenal!
- GDB, LLDB, Microsoft Visual Studio Debugger
- CLI/GUI
- Examine program state, pause when conditions are met

What is a breakpoint?

- Can be set via the CLI or GUI
- Program execution pauses when hit

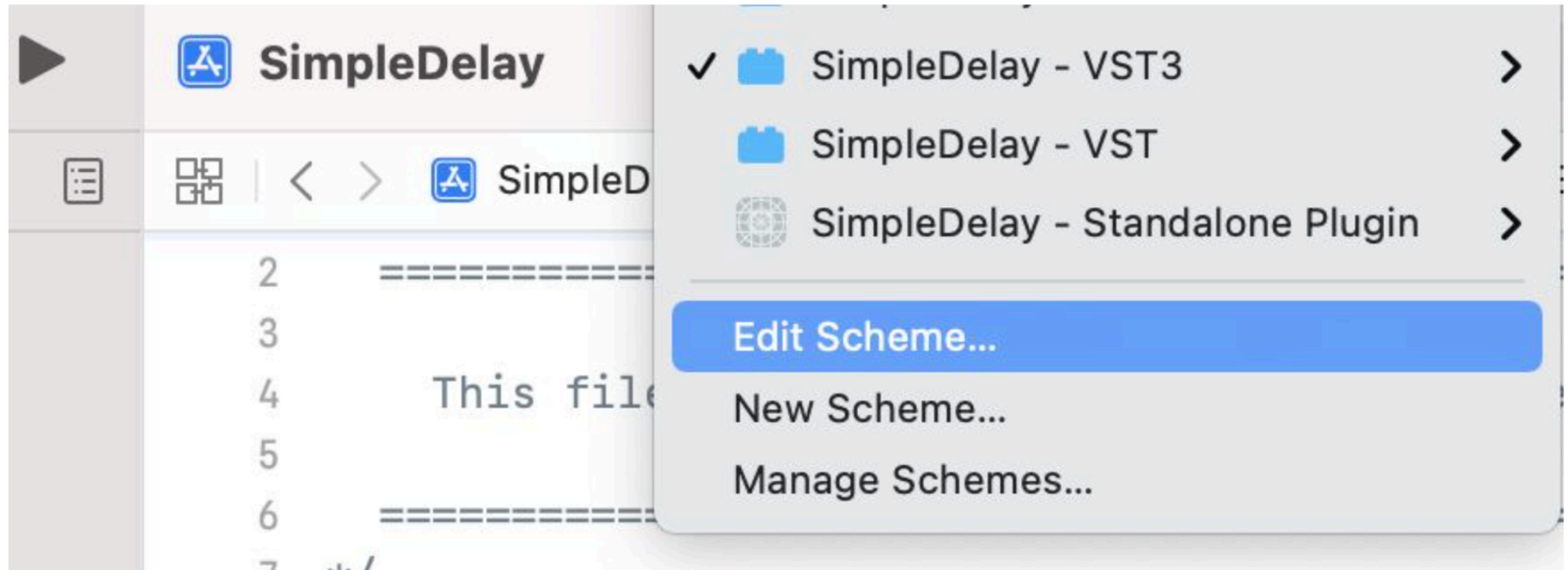
# Debugging

Debugging standalone plug-in is simple as we control the whole process

Debugging the plug-in inside an actual host is slightly more complicated as we are running inside a different process (the host)

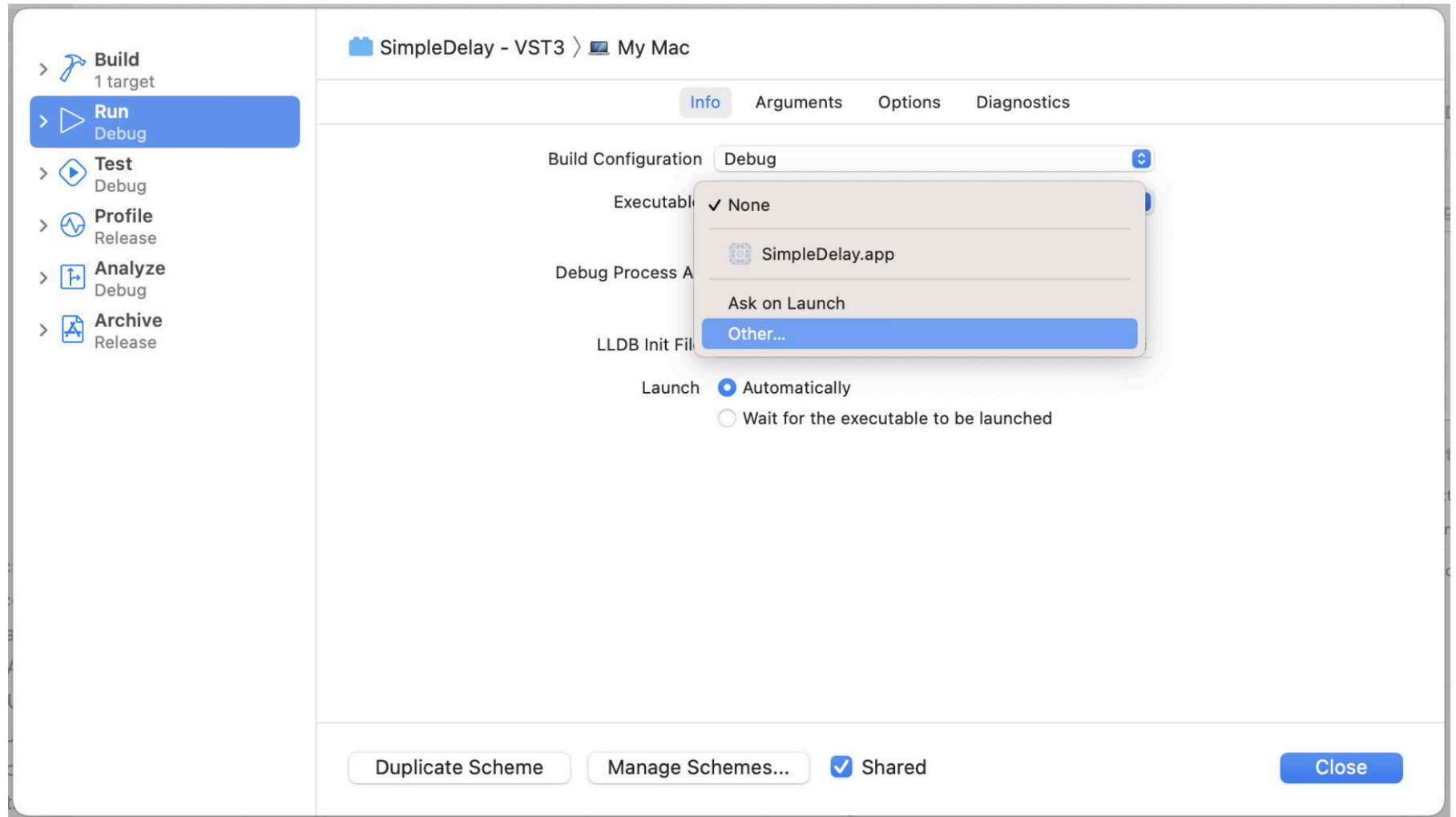
Debuggers can attach to a separate process to allow you to debug and set breakpoints in your code

# macOS

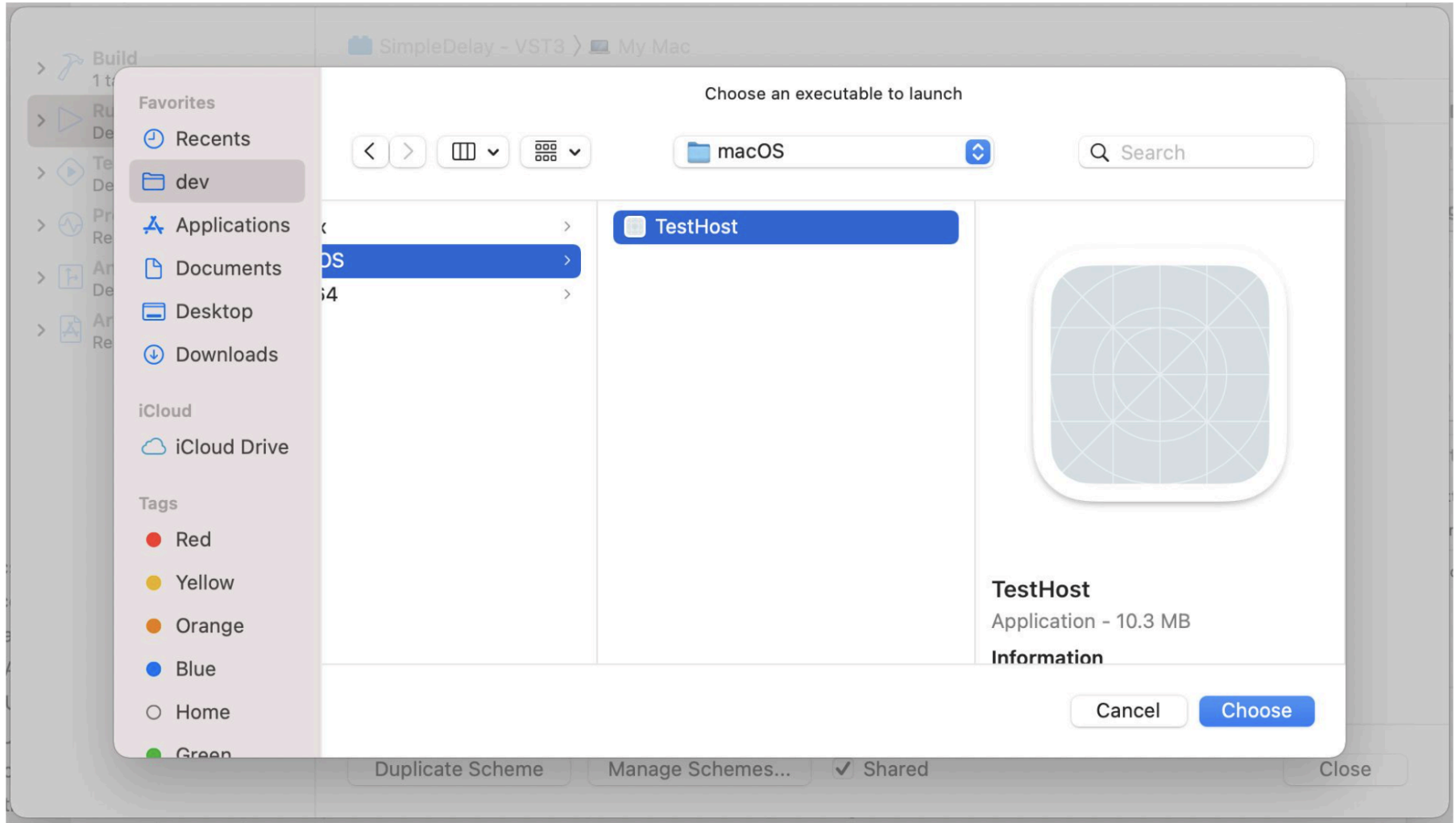




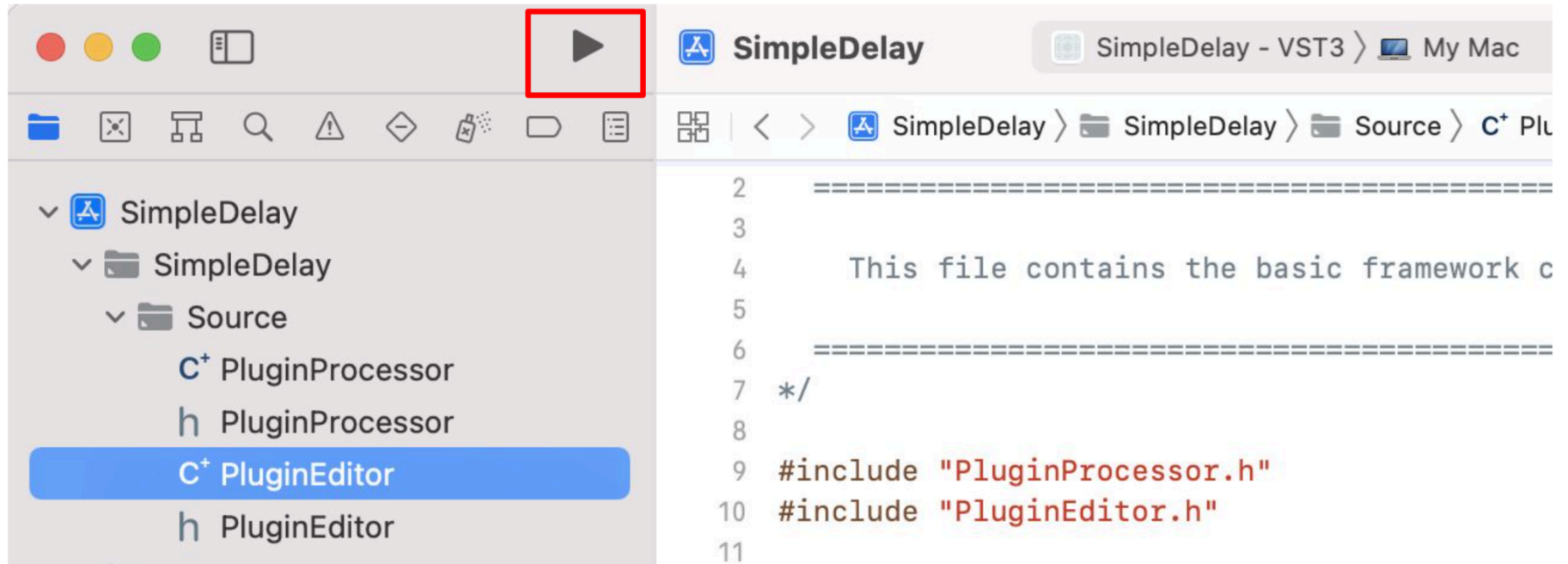
# macOS



# macOS



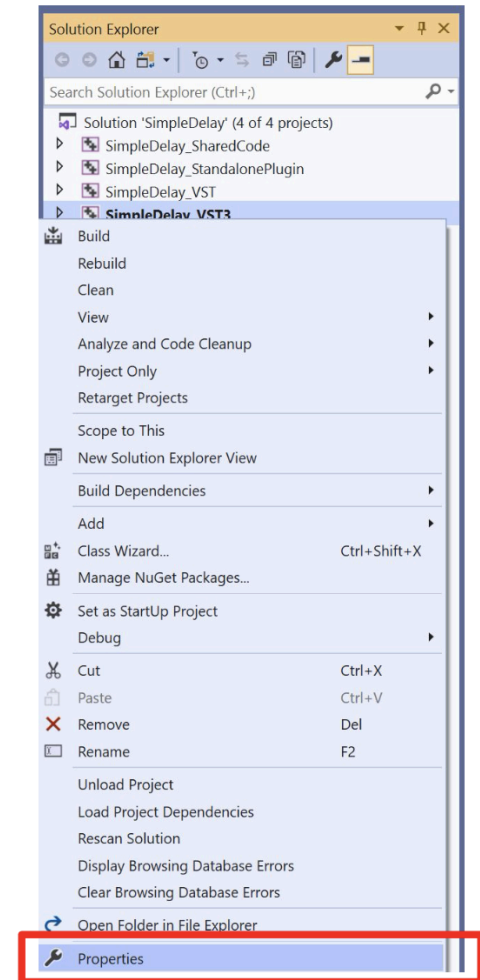
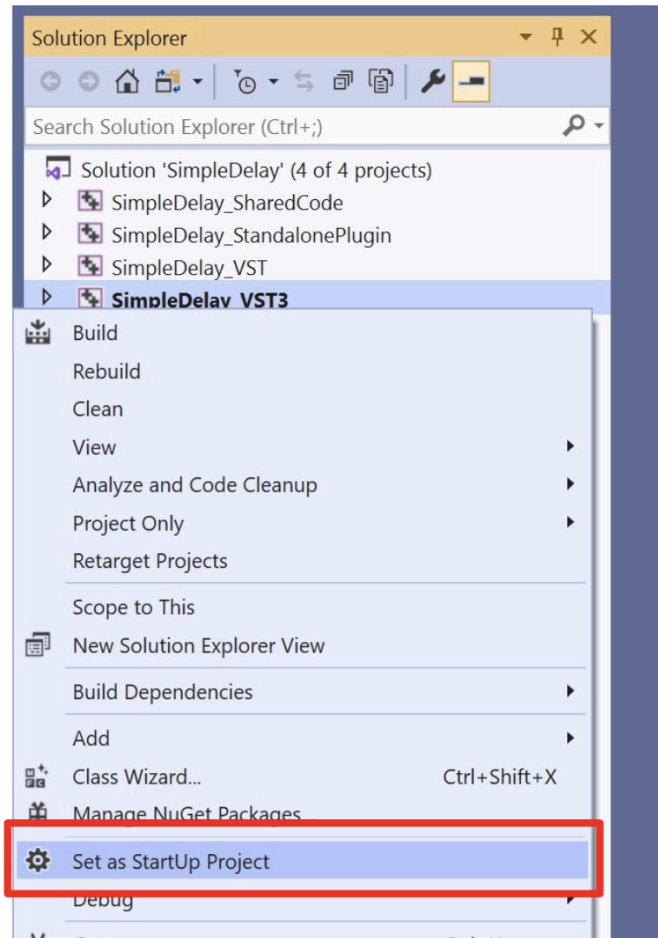
# macOS



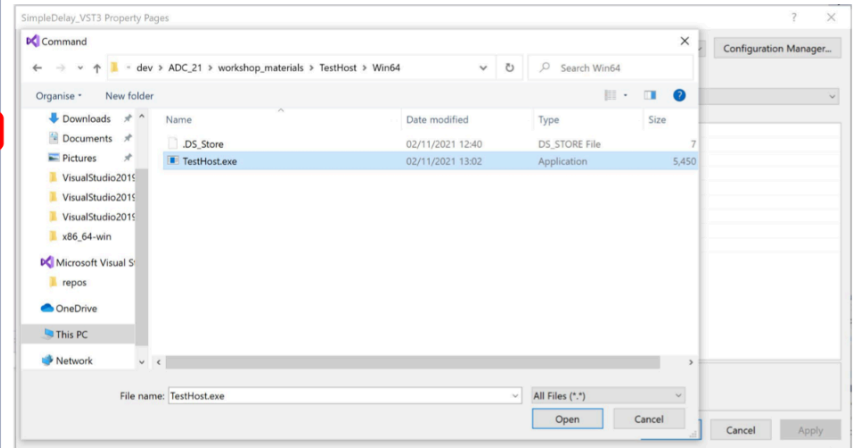
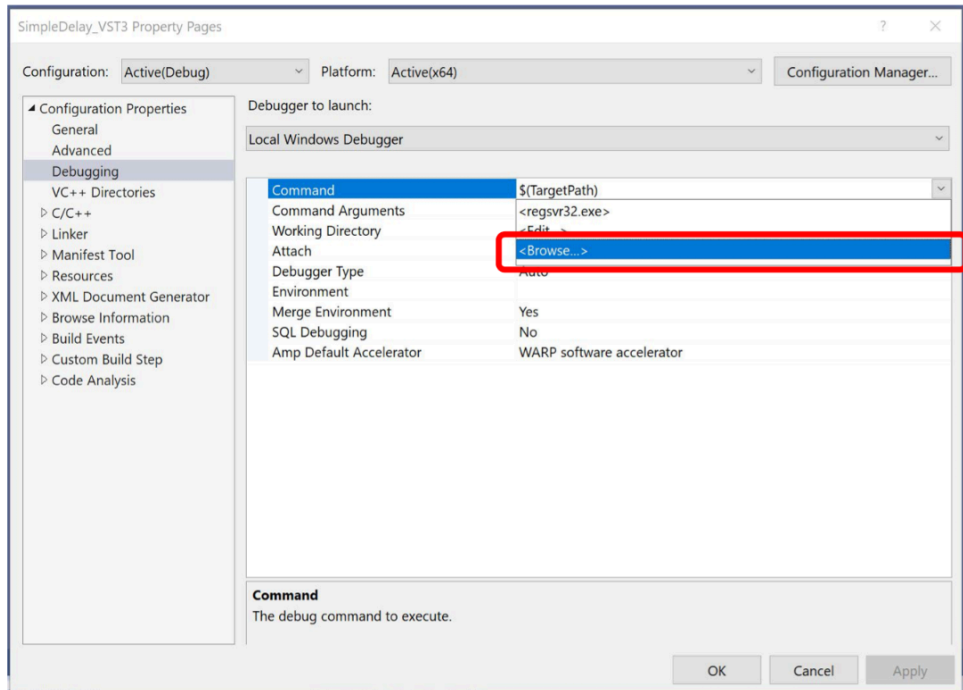
# macOS

```
142
143 void SimpleDelayAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
144                                              juce::MidiBuffer& midiMessages)
145 {
146     juce::ScopedNoDenormals noDenormals;
147     auto totalNumInputChannels = getTotalNumInputChannels();
148     auto totalNumOutputChannels = getTotalNumOutputChannels();
149
150     // In case we have more outputs than inputs, this code clears any output
151     // channels that didn't contain input data, (because these aren't
152     // guaranteed to be empty – they may contain garbage).
153     // This is here to avoid people getting screaming feedback
154     // when they first compile a plugin, but obviously you don't need to keep
155     // this code if your algorithm always overwrites all the output channels.
156     for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
157         buffer.clear (i, 0, buffer.getNumSamples());
158 }
```

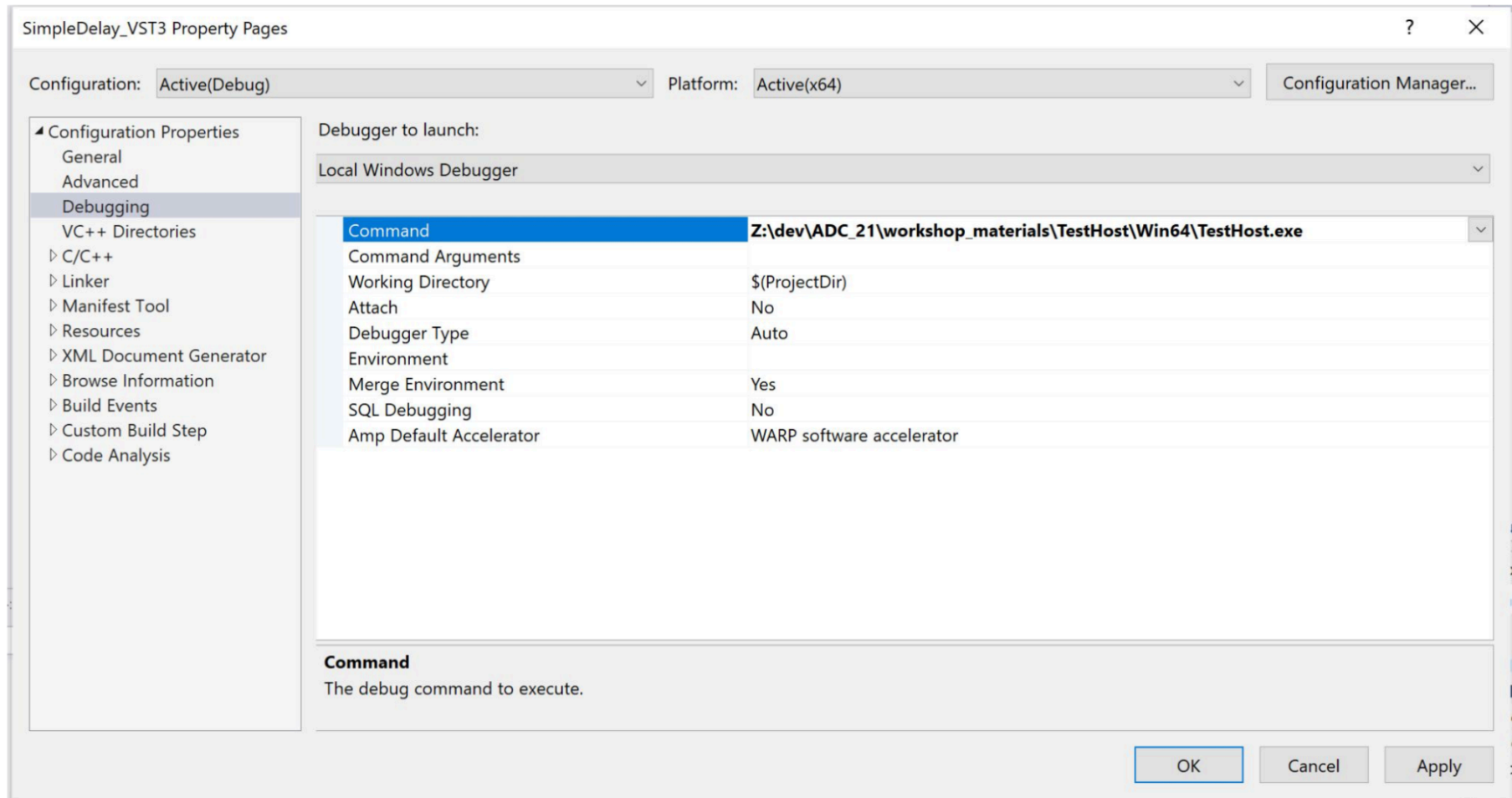
# Windows



# Windows

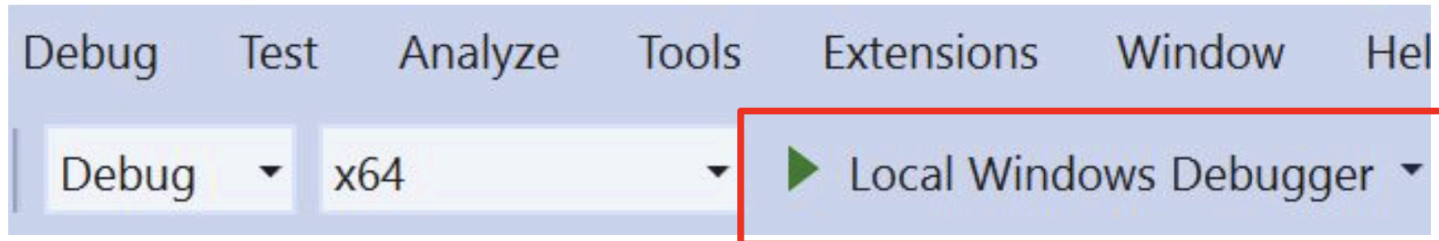


# Windows





# Windows



```
132 void SimpleDelayAudioProcessor::processBlock(juce::AudioBuffer<float>& buffer,
133 .....juce::MidiBuffer& midiMessages)
134 {
135     ..juce::ScopedNoDenormals noDenormals;
136     ..auto totalNumInputChannels = getTotalNumInputChannels();
137     ..auto totalNumOutputChannels = getTotalNumOutputChannels();
138
139     ..// In case we have more outputs than inputs, this code clears any output
140     ..// channels that didn't contain input data, (because these aren't
141     ..// guaranteed to be empty -- they may contain garbage).
142     ..// This is here to avoid people getting screaming feedback
143     ..// when they first compile a plugin, but obviously you don't need to keep
144     ..// this code if your algorithm always overwrites all the output channels.
145     ..for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
146     ..    buffer.clear(i, 0, buffer.getNumSamples());
147
```



# Linux

```
cd workspace/Builds/LinuxMakefile  
make CONFIG=Debug  
gdb ./TestHost/Linux/TestHost  
break SimpleDelayAudioProcessor::processBlock  
run
```

# Linux

```
ed@ubuntu: /mnt/hgfs/dev/ADC_21/workshop_materials/09/Builds/LinuxMakefile
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./build/SimpleDelay...
(gdb) b SimpleDelayAudioProcessor::processBlock(juce::AudioBuffer<float>&, juce::MidiBuffer&)
Breakpoint 1 at 0xa6a41: file ../../Source/PluginProcessor.cpp, line 144.
(gdb) r
Starting program: /mnt/hgfs/dev/ADC_21/workshop_materials/09/Builds/LinuxMakefile/build/SimpleDelay
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
JUICE v6.1.2
[New Thread 0x7ffff227b640 (LWP 45117)]
[Thread 0x7ffff227b640 (LWP 45117) exited]
[New Thread 0x7ffff227b640 (LWP 45118)]
[Thread 0x7ffff227b640 (LWP 45118) exited]
[New Thread 0x7ffff227b640 (LWP 45119)]
[New Thread 0x7ffff1a7a640 (LWP 45120)]
[New Thread 0x7ffff1279640 (LWP 45121)]
[New Thread 0x7ffff0a78640 (LWP 45122)]
[Switching to Thread 0x7ffff1279640 (LWP 45121)]

Thread 6 "JUICE ALSA" hit Breakpoint 1, SimpleDelayAudioProcessor::processBlock (this=0x555555f838a0, buffer=..., mi
diMessages=...) at ../../Source/PluginProcessor.cpp:144
144      {
(gdb) █
```

# Out of process loading

- Some hosts load plug-ins in a separate process
  - Logic Pro on Apple Silicon
  - AUv3 plugins (mostly)
  - Bitwig/Reaper depending on settings
- Need to attach to the plug-in process not the host process:
  - Xcode: Debug->Attach to Process by PID or Name...
  - Visual Studio: Debug->Attach to Process...
  - GDB: attach <PID>

# macOS Notarised Hosts

- Since macOS Catalina (10.15), apps distributed outside the App Store must be notarised
- Apps can only be debugged with if they have the `com.apple.security.get-task-allow` entitlement set to true
- Must be false for notarisation to succeed
- However it is possible to re-sign host binaries!

# macOS Notarised Hosts

- Get app's entitlements using:

```
codesign -d --entitlements :- /path/to/host.app
```

- Modify them to include

```
<key>com.apple.security.get-task-allow</key>  
  <true/>
```

- Set the new entitlements using:

```
codesign --force --options runtime --sign - --entitlements /path/to/plist "/path/to/app"
```

# Testing with tools

# pluginval

- Cross-platform plug-in validation tool for testing AU/VST/VST3
- <https://github.com/Tracktion/pluginval> for source code and tagged releases
- Can be run on the command line or with a GUI

# pluginval

```
pluginval --strictness-level 10 --validate SimpleDelay.vst3
```

All tests completed successfully

Finished validating: SimpleDelay.vst3  
ALL TESTS PASSED



# auval

- Apple's command line AU verification tool
- Tests basic AudioUnit functionality

```
auval -al # list known plugins
```

```
auval -v aufx DLAY JUCE # test a specific plugin
```

```
* * PASS
```

```
-----  
AU VALIDATION SUCCEEDED.  
-----
```

# Developing SimpleDelay further

General improvements:

- Parameter change smoothing
- A LookAndFeel to style the widgets
- Accessibility
- Presets
- CMake

For this particular effect:

- A variable delay length
- Fractional delay lengths

# Links



GitHub: <https://github.com/juce-framework/JUCE>

Forum: <https://forum.juce.com>

Course: <https://juce.com/learn/course>

Tutorials: <https://juce.com/learn/tutorials>