SuperCollider serverplugins in C++

A tutorial

Mads Kjeldgaard and the SuperCollider community

28-07-2023

Contents

Introduction

In SuperCollider, a unit generator (UGen) is a building block for creating sound patches. These building blocks may be something as simple as a sine wave generator or as complex as an algorithmic reverb. With these, the user of SuperCollider may put together synthesis and sound manipulation patches for creating almost any sound imaginable.

SuperCollider comes prepackaged with a plethora of different UGens.

Expanding the selection of UGens is done by installing *plugins* - collections of user contributed UGens (the words UGen, unit generators and plugins are often used interchangeably in SuperCollider).

The sc3-plugins repository contains a large library of community developed plugins (of varying age and quality) and other notable examples of plugins include IEM's vstplugin plugin and The Fluid Corpus Manipulation Project.

This is all made possible by SuperCollider's plugin API that provides a simple way to write plugins in C or C++ and hook them in to SuperCollider's synthesis server as one of many building blocks. Each addition of a plugin makes possible an abundance of new possibilities for the user when combined with the existing plugins.

In this tutorial series, we will cover some of the basics of writing such plugins using the C++ interface available to plugin authors (if you prefer writing your plugin in pure C, then I would advise reading Dan Stowell's excellent chapter on just that in the MIT SuperCollider book).

Before continuing I would like to extend a big thank you to the ever friendly

and generous SuperCollider community for helping me understand these things a bit better, and to my workplace Notam for allowing me time to work on this.

On that note: If you need help with plugin development, there is an entire subforum dedicated to the subject over on the SuperCollider forum.

The code to accompany this tutorial may be found here.

Why write a plugin for SuperCollider?

Most users of SuperCollider will never experience the need for writing a plugin simply because the library of UGens that comes with SuperCollider is so rich in selection and effective in performance.

That said, there are some reasons for doing it anyway:

1. *Learning*: It's a great way to learn or get better at C++ or DSP (Digital signal processing) because it lets you focus on the sound and the algorithms, with fairly short compilation times and the ability to quickly test it out in SuperCollider.
2. *Contributing*: You get to expand the ever evolving world of sonic possibilities in SuperCollider and contribute to a lively and creative community that has been going for more than 25 years.
3. *Expanding sonic possibilities*: There may be some functionality you would love to see added to SuperCollider (see the flucoma project for a really impressive example of this).

Part 1: Demystifying the SuperCollider plugin

Before actually creating a plugin, let's take a bit of a detour to have a look at what a plugin project contains and how to use the CMake build generator system to turn our code into UGen objects for SuperCollider.

Quick starting a plugin project

The fastest way to create a plugin project is to use the SuperCollider development team's cookiecutter template. By running this script in a terminal and answering the questionnaire it presents you with, you should be able to quickly

generate all the files and project structure necessary for a plugin. See the cookiecutter template's project page for the most updated information about how to use this, but in short it boils down to first installing the cookiecutter tool which is a python library:

```
$ python3.7 -m pip install cookiecutter
```

Then you can run the cookiecutter command line tool and supply it with the URL of SuperCollider's cookiecutter template:

```
$ cookiecutter https://github.com/supercollider/cookiecutter-supercollider-plugin
```

A typical project structure

A typical project structure for a plugin looks something like this:

```
|-- CMakeLists.txt

|-- cmake_modules
|   |-- SuperColliderCompilerConfig.cmake
|   |-- SuperColliderServerPlugin.cmake
|
|-- LICENSE
|
|-- plugins
|   |-- SimpleGain
|       |-- SimpleGain.cpp
|       |-- SimpleGain.hpp
|       |-- SimpleGain.sc
|       |-- SimpleGain.schelp
|
|-- README.md
```

Let's take it from the bottom up:
First there is a readme. This explains what the plugin is and how to install it. Then there is a subfolder called *plugins* - this is where we will be spending most of our time writing plugins. This folder contains another subfolder called *SimpleGain* which is the name of this particular plugin, and then the four files you need for a plugin:

1. A .hpp file that declares your C++ code and imports all the necessary

files.
2. A .cpp file - this C++ implementation file is where we define our algorithms (typically in a setup function (a constructor), an optional teardown function (a destructor) and a calculation function that is called on each block of sound samples we put in to our plugin).
3. A .sc SuperCollider class file that bridges our C++ code to the SuperCollider language
4. A .schelp file for the SuperCollider help system which explains the usage of the plugin and demonstrates its usage in one or more examples.

The *cmake_modules* subfolder and the CMakeLists.txt file are used for building and compiling our project (we will get back to this later).

And then lastly, there is a LICENSE file containing the license for the plugin's code (SuperCollider itself is licensed under the GPL-3 license).

CMake is your friend

Building, compiling and installing your plugin(s) is *one process* handled by a program called CMake.

It is easy to get freaked out by CMake. At first, using it may seem like a semi-esoteric experience but *CMake is actually your friend*. In a SuperCollider plugin project, the main job of CMake is to keep track of your build options, what platform your are on and where the files needed are - and then, with that information at hand, construct a build system, compile your code and install it for you.

It is not strictly necessary to go deep in to the world of CMake to write a SuperCollider plugin but familiarizing yourself with some of the absolute basics can be helpful, which is why we will spend a bit of time doing just that before actually starting on our plugin (should you feel the need to go deep, then Craig Scott's book "Professional CMake" is a nice resource).

CMake'ing a plugin project

CMake needs a copy of the SuperCollider source code to be able to build a plugin project.

The path to this is supplied to CMake by using the `-DSC_PATH` flag like so:

`-DSC_PATH=/path/to/supercollider/sourcecode`

The reason for this is that the SuperCollider source code contains all the actual library code you need to make your plugin work - including the plugin API, so the compiler needs these to be able to create the necessary objects.

CMake also needs to know where your plugin code is. This is done in the `CMakeLists.txt` file at the root of your project where the paths (and other useful options) are defined. You generally only need to change this file if you rename your files, move them or want to add more files.

This is done using two CMake commands:

- `set(variablename ...variable_contents)`
- `sc_add_server_plugin(destination name cpp_files sc_files schelp_files)`

Here is an example of what that might look like in your plugin project:

```
set(SimpleGain_cpp_files
    plugins/SimpleGain/SimpleGain.hpp
    plugins/SimpleGain/SimpleGain.cpp
)
set(SimpleGain_sc_files
    plugins/SimpleGain/SimpleGain.sc
)
set(SimpleGain_schelp_files
    plugins/SimpleGain/SimpleGain.schelp
)

sc_add_server_plugin(
    "${project_name}" # destination directory
    "SimpleGain" # target name
    "${SimpleGain_cpp_files}"
    "${SimpleGain_sc_files}"
    "${SimpleGain_schelp_files}"
)
```

When you add more plugin files to your project you simply have to repeat the chunk of CMake code above in your CMakeLists.txt, modified to contain information about the newly added files.

Build and install using CMake

What makes CMake really nice is that it makes it possible to build, compile and install your project with the help of a few terminal commands.

The workflow for this is divided into these steps:

Step 1: Create a build directory and generate a build system.

From the root of your project, run:

```
mkdir build # Create a build sub directory
cmake -B "build" -S . -DSC_PATH="/path/to/sc/sourcecode" \
    -DCMAKE_INSTALL_PREFIX="/path/to/your/extensions/dir"
```

Once this is done, CMake will fill the subfolder `build` with a bunch of files and data that is needed every time you compile your code (and it also serves as a sort of cache or memory).

Note *always wrap the path arguments in quotes, CMake is very particular about this.*

Step 2: Compile (and optionally install) the plugin(s).

To compile and install all you need to do is run the following command from the root of your project (repeat this command every time you change your code and want to see if it compiles and installs correctly):

```
cmake --build build --config "Release" --target install
```

Step 3: Try it out in SuperCollider

If compilation was successful, you should now have compiled and installed your project to your SuperCollider extensions directory.

If you have SuperCollider open, recompile the class library and search the help system for your plugin to verify that it showed up.

In the next part of this tutorial series, we will go deeper into the actual code of a plugin.

Part 2: Creating a ramp generator plugin

In this part of the tutorial we will start work on a plugin project. We will create a very simple oscillator that ramps from 0.0 to 1.0. This is sometimes known as a phasor as it is often used internally in code to read through lookup tables or as a way to have an internal clock that steps through data. It is also known as a sawtooth oscillator and may be used at audio rate to create sound signals with many (harsh) overtones.

Control, audio and scalar rate: A note on calculation rates

But first: We need to talk about calculation rates.

You may or may not be aware that UGens in SuperCollider often have multiple different output rates at which they may run, these are denoted by the class method they are called with. Some run at audio rate, some at control rate, others at demand rate (which we will not cover here) and others again only at scalar rate (abbreviated `ir` sometimes for initialization rate).

Here are examples of how that might look in SuperCollider:

```
// Control rate
SinOsc.kr(100);

// Audio rate
SinOsc.ar(100);

// Scalar rate
SampleRate.ir();
```

These three different calculation rates are optimized for different purposes:

1. **Scalar rate**, also known as *initialization rate* is useful if your plugin should output a value when it is initialized and then do nothing else.
2. **Audio rate** is self-explanatory, but what it means in practice is your calculation function will receive a block of audio samples, loop over each sample, do something with the data and then return an output block of samples. Every time your plugin's calculation function is called it is supplied with an argument containing the value of the number

of samples to process. This is typically 64 samples, but the user may change this in the server options.

3. **Control rate**. This is a more performant alternative to audio rate calculation that only produces one value per block of samples. It is equivalent to setting the number of samples in the calculation function to 1.

Here is a little table to help you remember this:

Rate	**sclang method name**	**Update rate**
Audio rate	*ar	numSamples/sampleblock
Control rate	*kr	1/sampleblock
Scalar	*ir	at UGen initialization

Ramping values

A ramp generator is a simple algorithm that counts up to a certain threshold, then wraps back to where it began. This technique is seen in sawtooth and phasor oscillators. Not only is the signal itself useful in modulating parameters as an LFO, but the core algorithm is often used in other oscillator functions to index into wavetables and similar things.

The idea is simple, really: Imagine you are counting from 0 to 10 and then every time you reach 10, you start over at 0. If you were to plot this on a graph you would see a ramp signal (which looks like a sawtooth).

We will be using the same concept in our ramp UGen but instead of counting whole numbers we will be counting floats, which is the type of number that comes out of a UGen.

So, instead of counting from 0 to 10, we will be counting from 0.0 to 1.0.

Creating the RampUpGen SuperCollider class

If you followed the previous tutorial in this series, you will know that the easiest way to generate the scaffolding for a plugin is using the supercollider plugin cookiecutter template. Running this will generate all the files needed

for a plugin and going forward I will assume you've done that and that you've called your plugin *RampUpGen*.

The first thing to write is the SuperCollider class that will be responsible for calling our C++ code. The file is called something like `RampUpGen.sc`.

It defines the UGen's audio rate (`*ar`) and control rate (`*kr`) methods and sets the frequency argument's default value.

Another important (and often overlooked) aspect of the SuperCollider class interface for plugins is that they are tasked with checking the rates of the inputs to the parameters and ensure an error is raised if for example a user inputs an audio rate signal in an input that was written for control rate.

```
RampUpGen : UGen {
    *ar { |frequency=1.0|
      ^this.multiNew('audio', frequency);
    }

    *kr { |frequency=1.0|
      ^this.multiNew('control', frequency);
    }

    checkInputs {

        // If you want to do custom rate checking...
        if(this.rate == \control and: { inputs.at(0).rate == \audio }, {
          ^"An audio-rate frequency argument isn't allowed when RampUpGen runs at
        });

        // Checks if inputs are valid UGen inputs
        // And not a GUI slider or something...
        ^this.checkValidInputs;
    }
}
```

Important note on multichannel UGens: *Our SuperCollider class above inherits from UGen - this results in a 1 channel UGen. If you need more outputs than that, you need to inherit from MultiOutUGen instead (don't*

forget to call `this.initOutputs(numChannels, rate)` somewhere in your class if you do end up using this, it will (silently) fail otherwise).

Setting up the header file

Next step is to open up `RampUpGen.hpp` in a text editor. This is your header file for your plugin where the correct files are imported and your new C++ class `RampUpGen` inherits its functionality from the SCUnit class. The class should look something like this:

```
class RampUpGen : public SCUnit {
public:
    RampUpGen();
private:
    // Calc function
    void next(int nSamples);
};
```

As you can see above, it defines a public constructor for the class `RampUpGen()` and a private calculation function called `next(int nSamples)`. If you create a UGen that allocates memory (for example delay based UGens), then you should also define a destructor function which is the same name as the constructor but with a tilde prefixed: `~RampUpGen()`. It is omitted here since we don't need it.

Under the `private:` keyword, declare member variables that we can use to store the state of our UGen between processing blocks: the ramp's phase value and frequency.

```
// State variables
// Ramp generator phase. Initialized to 0.0.
double m_phase{0.0};
// Ramp generator frequency. Uninitialized - remember to initialize in the const
float m_frequency;
```

Creating the calculation function

It's now time to define the behaviour of our plugin. Move into the `RampUpGen.cpp` file.

The core of our plugin will be the calculation function `RampUpGen::next(int nSamples)`.

Let's define this function.

First, we need to figure out how much to increment our counter at every clock tick.

This is done by dividing the frequency of our ramp generator by the sample rate of our UGen. The sample rate is available to us using the `sampleRate()` function that comes with the plugin API.

We will let the user of our UGen decide what frequency our ramp generator should be running at.

This is done by reading the value that is supplied via the `frequency` argument in the SuperCollider class defined above.

For this, we use the `in0(int argumentInputNum)` function - it takes the index (`argumentInputNum`) of the parameter we want to read as an argument. In this case the (first and only) parameter, `frequency`, is at index `0`.

One last thing to note here is that we will read this argument at *control rate*, that is: at a rate of one value per calculation block (see explanation above). This single value is the first value in the block at this input position, hence the "`0`" in the function name `in0`.

```
// First UGen input is the frequency parameter
const float frequency = in0(0);

// Calculate increment value.
// Double precision is important in phase values
// because division errors are accumulated as well
double increment = static_cast<double>(frequency) / sampleRate();
```

If we instead want to read all samples in the calculation block from this input, we would use the `in(int argumentInputNum)` function, which instead returns a *pointer* to the first sample of our input, which in turn is used to interate with in the calculation function's processing loop. This is demonstrated below.

Anyway, every time our calculation function is called, we need to advance the phase of our ramp by an amount, `increment`.

```
m_phase += increment;
```

Then, once our ramp generator reaches `1.0`, we reset it to `0.0` to make it wrap back around at the beginning of the ramp.

```
const double minvalue = 0.0;
const double maxvalue = 1.0;

// Wrap the phasor if it goes above maxvalue or below minvalue
if (m_phase > maxvalue) {
    m_phase = minvalue + (m_phase - maxvalue);
} else if (m_phase < minvalue) {
    m_phase = maxvalue - std::fabs(m_phase);
}
```

That's it for the calculation function.

Documentation

The last thing we need to do is to document the functionality of our UGen so that it is usable for others. This is done in SuperCollider help file that will go with our plugin called `RampUp.schelp`. See the writing help and / or scdoc syntax help files for more information on how to do this. Be sure to add some good and clear examples to this.

The schelp file could look something like this:

```
class:: RampUpGen
summary:: Cyclically ramping up values
related:: Classes/LFSaw
categories:: UGens>Generators

description::

Generate cyclically ramping values from teletype::0.0:: to teletype::1.0::.
teletype::RampUpGen:: behaves similarly to link::Classes/LFSaw::.

classmethods::

method:: ar, kr
```

```
argument:: frequency
Frequency of the ramping signal, in Hertz.

examples::

code::

// Scope the ramp generator at 1 hz
{ RampUpGen.kr(1.0) }.scope

::
```

It's good practice to explain the functionality and usage of your UGen concisely but completely. For example, list all possible rates at which your UGen runs, and the supported rates of the arguments. This is useful not only for users but also for your future self! (*What was I thinking?? Why didn't I support all argument rates??*)

Modal tags It's encourages to use the `emphasis::` tag when referring to your UGen parameters, so they have the same visual emphasis of the rendered parameter name. The `code::` tag will both colorize and make the enclosed text executable. For visual emphasis and referring to variables or numbers, etc., `teletype::` will render text in a monospace font, without colorizing or making the text executable. When referring to other UGens, Classes, Tutorials, etc., it's helpful to `link::` to them, as in the example `description::` above.

Contextual information Additional contextual information is helpful—if, for example, you're implementing a mathematical function or chaos algorithm, you can describe its behavior, range bounds and even link to web pages for more information. The `discussion::` modal tag is useful for this.

If there are known undesireable behavior (will your filter blow up if a parameter is modulated too quickly?), it's kind to warn the user of this with the `warning::` modal tag.

Examples (and tests) Examples are crucial! You'll undoubtedly (hopefully) be thoroughly testing your UGen in the course of development. Your test snippets would make great examples! (In fact, see the examples in the help file for `RampUpGen` for useful tests.) The more ways you can show the use of your UGen the better. Keep examples relatively simple to highlight the core function of *this* UGen, but don't be afraid to show off a bit, especially with a musical example ;-). If you have more elaborate demonstrations, consider writing a tutorial page.

Compile the plugin and try it out in SuperCollider

Now a first version of your plugin should be finished. Build and compile your plugin using the CMake commands discussed earlier in this tutorial, recompile the SuperCollider class library and try messing around with it.

Part 3: Finishing touches

Before our plugin is done, there are some nice little things we can do to touch it up and make it even better.

Using enums to keep track of inputs

Inputs and outputs in the C++ side of your UGen are represented as integers. The `frequency` parameter in our `RampUpGen` UGen is represented by `0` since it's the first input to the UGen. To make it more readable and easier to keep track of parameters when you add more of them later on, a simple trick is to use enums. At its most basic, an enum may be used as a collection of aliases for integers. Using this for your parameters makes your code easier to read and maintain.

In your header file, under the `private:` keyword, add the following:

```
enum Inputs { Frequency };
```

Then, in your plugin code, whenever you need the input number for frequency you simply type `Frequency`. Another nice thing is that you only have to reorder the items in this enum if you reorder your parameters, and the changes will automatically propagate.

Delegating functionality to functions

To keep your code clean and modular, it can be helpful to delegate some of the functionality to "helper" functions outside of the calculation function. Because we are about to add a second calculation function, we can extract a bit of the ramp generation code to its own function to be used by both calculation functions.

To do this, add in your header file:

```
// A helper function
inline float progressPhasor(double phase, float frequency);
```

And then implement it in your *.cpp*-file:

```
// The ramp generator
inline float RampUpGen::progressPhasor(double phase, float frequency) {
  // Calculate increment value.
  // Double precision is important in phase values
  // because division errors are accumulated as well
  double increment = static_cast<double>(frequency) / sampleRate();

  phase += increment;

  const double minvalue = 0.0;
  const double maxvalue = 1.0;

  // Wrap the phasor if it goes above maxvalue or below minvalue
  if (phase > maxvalue) {
      phase = minvalue + (phase - maxvalue);
  } else if (phase < minvalue) {
      phase = maxvalue - std::fabs(phase);
  }

  return phase;
}
```

Interpolating control rate parameters

Sometimes the user of a plugin may wish to modulate one of the UGen's parameters. In an environment such as SuperCollider, this is to be expected. When doing this, you may end up hearing crunchy or glitchy effects in the sound of your UGen. This is caused by a lack of interpolation in the control signal between each sample being processed. This results in steppy jumps between values instead of smooth trajectories. Let's fix this!

The plugin API contains a very useful function for exactly this purpose. The trick is to use the type `SlopeSignal<float>` to represent and contain our input parameter's value. This is produced using the `makeSlope(current_value, previous_value)` function. And then, on each iteration of our calculation function's `for`-loop, we use the `.consume()` method to slowly step from the previous block's sample value to the current one. Then, after the `for`-loop we store the final value in a member variable that will be used next time the block is processed.

This is what our calculation function looks like with parameter interpolation:

```
void RampUpGen::next(int nSamples) {
    const float frequencyParam = in(Frequency)[0];
    SlopeSignal<float> slopedFrequency =
        makeSlope(frequencyParam, m_frequency);
    float *outbuf = out(0);
    double current_phase = m_phase;

    for (int i = 0; i < nSamples; ++i) {
        // Write out the current phase
        outbuf[i] = current_phase;

        // Calculate increment value.
        // Double precision is important in phase values
        // because division errors are accumulated as well
        double increment =
            static_cast<double>(slopedFrequency.consume()) / sampleRate();

        // Advance the phase
        current_phase += increment;
```

```
        // Wrap the phase if it goes above maxvalue or below minvalue
        if (current_phase > maxvalue) {
            current_phase = minvalue + (current_phase - maxvalue);
        } else if (current_phase < minvalue) {
            current_phase = maxvalue - std::fabs(current_phase);
        }
    }

    // Store final value of frequency and phase to be used next time the
    // calculation function is called
    m_frequency = slopedFrequency.value;
    m_phase = current_phase;
}
```

Setting calculation function based on input rates

Let's take this a step further. We actually only need to interpolate the frequency parameter if the parameter's input is control rate. If the input is audio rate, we get a full block's worth of `frequency`-parameter and thus we don't need the interpolation.

The way to work around this and similar problems is with multiple cases for the same plugin to use different calculation functions, one for each calculation rate input for example. In our example we only have one parameter so we can make a `next_a` and `next_k` version of that function, one for audio rate inputs to the `frequency` parameter and one for control (and scalar) rate inputs.

The Server Plugin API comes with a function you can use to poll an input number to see what rate it is running at:

```
inRate(int inputNumber)
```

This is useful when used in combination with the constants (`calc_ScalarRate` for *scalar*, `calc_BufRate` for *control* and `calc_FullRate` for *audio* rate) that represent each of the input rates (these also come with the API):

```
RampUpGen::RampUpGen() {

  // Initialize the state of member variables that depend on input aruguments
```

```
    m_frequency = in0(Frequency);

    // Set the UGen's calculation function depending on the rate of the first
    // argument (frequency).
    // Also, call that function for one calculation cycle, which  generates an
    // initialization sample for downstream UGens to use for their initialization.
    if (inRate(Frequency) == calc_FullRate) {
        mCalcFunc = make_calc_function<RampUpGen, &RampUpGen::next_a>();
        next_a(1);
    } else {
        mCalcFunc = make_calc_function<RampUpGen, &RampUpGen::next_k>();
        next_k(1);
    };

    // Reset the initial state of member variables.
    // This is so the initialization sample calulated above by 'next' matches
    // the first output sample when the synth is run and 'next' is called again.
    m_phase = 0.0;
    // m_frequency is not reset because it's initial value is unaffected by
    // the next_k(1)
}
```

Note that in the final implementation, we replace `if (inRate(Frequency) == calc_FullRate)` with the equivalent function call `if (isAudioRateIn(Frequency))`. Explore the API for different ways to do what you need!

Finally, we have our two calculation functions `next_a` and `next-k` (for audio- and control-rate `frequency` inputs, respectively):

```
// Calculation function for audio-rate frequency input
void RampUpGen::next_a(int nSamples) {
    const float *frequency = in(Frequency);
    float *outbuf = out(0);
    double current_phase = m_phase;

    for (int i = 0; i < nSamples; ++i) {
        // Be sure to read from UGen's inputs BEFORE writing to outputs,
        // they share a buffer by default!
        const float freq = frequency[i];
```

```
        // Write out the phase
        outbuf[i] = current_phase;

        // Advance the phase
        current_phase = progressPhasor(current_phase, freq);
    }

    // Store final value of phase to be used next time the
    // calculation function runs
    m_phase = current_phase;
}

// Calculation function for control-rate frequency input
void RampUpGen::next_k(int nSamples) {
    const float frequencyParam = in(Frequency)[0];
    SlopeSignal<float> slopedFrequency =
        makeSlope(frequencyParam, m_frequency);
    float *outbuf = out(0);
    double current_phase = m_phase;

    for (int i = 0; i < nSamples; ++i) {
        const float freq = slopedFrequency.consume();

        // Write out the phase
        outbuf[i] = current_phase;

        // Advance the phase
        current_phase = progressPhasor(current_phase, freq);
    }

    // Store final value of frequency and phase to be used next time the
    // calculation function is called
    m_frequency = slopedFrequency.value;
    m_phase = current_phase;
}
```

General tips and troubleshooting

Tools that may come in handy

- Quick bench: An online benchmarking tool for C++. Use it to write small algorithms to compare their performance.
- Get familiar with compiler optimizations. There is a nice online tool godbolt.org that can help you with this. See this as an example of using it in a SuperCollider plugin workflow.

Make the compiler more strict

Adding the following section to your CMakeLists.txt will make the compiler a lot more strict and give you warnings on unused variables etcetera. This may be overwhelming at first but it helps make your code cleaner.

```
# Set compiler pickyness for Unix builds
# ... This picks up a ton of extra little things
if(CMAKE_COMPILER_IS_GNUCXX)
  add_compile_options(-Wall -Wextra -pedantic)
endif(CMAKE_COMPILER_IS_GNUCXX)
```

Common issues

One of the difficulties of coding plugins in C++ is that when something is wrong, you have to debug - you rarely get nice error messages to help you out. That said, here are some of the problems and causes that have caused me trouble in 9/10 situations.

Problem: *The server exits as soon as the plugin is initialized in a patch.*

Whenever I have experienced this, it has come down to problems in the way I have allocated memory.

This could also be caused by something dividing by zero somewhere in your code.

Another common mistake I make is to inherit the wrong class in the `.sc`-file containing the SuperCollider interface for my C++ code. This error may occur if, for example, you are inheriting from `MultiOutUGen` but haven't defined how many channels to use.

Problem: *The sound is crunchy when I modulate the parameters of my plugin*

You probably haven't used interpolation for control rate parameters. See the previous part of this tutorial about smoothing control rate signals using `SlopeSignal`.

Problem: *[Esoteric things] are happening*

Make sure that all variables are initialized correctly. C++ allows you to use variables even though you haven't initialized them with any values - this will lead to the variable being filled with random junk from your computer's memory that can easily cause strange behaviour.

This tutorial was written with the support of Notam - The Norwegian Centre for Technology in Arts and Music.