On the Development of C++ Instruments

Victor LAZZARINI
Maynooth University
Maynooth
Co. Kildare
Ireland,
Victor.Lazzarini@nuim.ie

Abstract

This paper brings together some ideas regarding computer music instrument development with respect to the C++ language. It looks at these from two perspectives, that of the development of self-contained instruments with the use of a class library and that of programming of plugin modules for a music programming system. Working code examples illustrate the paper throughout.

Keywords

Computer Music Instruments, C++, Music Programming

1 Introduction

Whatever we do, if we are in the business of making music solely or primarily with computers, one way or another, at some point, we will meet computer music instruments[Lazzarini, 2017a] . Whether we are making electroacoustic music, algorithmic composition, live coding, tracking, creating pop tunes, we will find ourselves manipulating these. They can present themselves through music programming systems [Lazzarini, 2013] , such as Csound [Lazzarini et al., 2016] or Faust [Orlarey et al., 2004], or as software synthesizers, plugins, audio processing programs, etc. There is a wide variety of forms. In this paper, I would like to contemplate one of these that involves libraries, compilers, and the C++ language.

C++ was once described as having "the elegance, the power, and the simplicity of a hand grenade", which to me, as a die-hard pure C programmer sounds about right. However, I must admit that its latest standards, ISO C++11[ISO/IEC, 2011] , C++14[ISO/IEC, 2014] , and the forthcoming C++17,[ISO/IEC, 2017] arriving in quick succession as they are, are making this monstrous language more attractive. Now finally we can write a nice lambda and pass it to a map to process a list, for example. The standard library, borne out of the much appreciated, much maligned, standard template library (STL), has actually become quite usable. There is still enough complexity for one to get entangled, however, but with moderation and good design, we can make it work for us.

This paper will examine two approaches of C++ instrument making. The first one is based on employing a signal processing library to write simple, straightforward, programs that can be ported to various platforms. The second is to create components, plugins, for Csound using a framework that sits atop the system implementation in C. It is mostly directed at computer music practitioners who can converse in C/C++, and it will be fully illustrated by working code, which can also be found somewhere in an online repo (links will be given).

2 AuLib Instruments

Towards the end of 2016, I decided to collect a number of digital signal processing (DSP) algorithms that I had been writing or studying throughout the years into a simple, lightweight, flexible C++ library, called AuLib[1]. One of my aims was to document these uniformly in efficient and readable code so that they could become somewhat of a reference for me and others. I was also rewriting some of my teaching materials and this became part of them. Following a number of refactoring steps, I settled on a design that followed modern C++ standards in employing the standard library as much as possible to handle resources and keeping the code as simple and lightweight as possible.

When designing a class library, there are two distinct possibilities (amongst the various decisions we have to make) with respect to object hierarchies. One is that we can define a base class for DSP objects that has a shared processing interface, that is one (or more) DSP methods that

[1] `github.com/vlazzarini/aulib/`

are specialised in derived classes. A means of connecting objects is provided separately from this, and once connected, we can place the objects in a list of references or pointers to the base class and call the processing method of each in turn to get an output signal. This is what is at the heart of processing engines such as the one in PD, Csound, Faust. If the aim is to create a library whose main objective is to be employed as an engine for some higher-level programming or patching system, this is the way to go. The Sound Object Library[Lazzarini, 2000] was designed this way and it really paid off when it was later wrapped up in Python.

The alternative is to relax this constraint and not provide a unified processing interface, leave it to derived classes to define their own. The advantage of this is that each class can have different ways to handle input parameters to processing methods, depending on what they are supposed to do. So an oscillator might have amplitude and frequency as parameters, in scalar or vectorial forms, or no parameters at all (for say fixed values of amplitude and frequency). It can provide a bunch of overloads to handle each case. A filter will have an input signal and optional parameters, depending on the type. A frequency-domain object might take a spectral frame. This, on one hand, simplifies connections (we can define them at the processing point, rather than separately), and on the other makes it hard to use in sound engine applications where the interface needs to be shared.

Given that the objective here for this library was to provide a working context for a diverse set of algorithms, and to provide a flexible means of using them in programs, I have opted for the second approach. This would provide greater freedom to create exactly the right form to hold each DSP formulation. Now, given this context, it is still desirable to use the class structure afforded by C++ to re-use code fully. This meant to design a base class that was a container for an audio signal, providing the typical fundamental operations we would like to perform on it. For me, this meant: scaling (multiplying by a scalar), offsetting (adding a scalar), mixing (adding a vector/signal) and ring modulating (multiplying by a vector/signal). Granted, in an audience of music and audio developers, we are likely to find multiple definitions of what fundamental operations on signals are, but I am drawing the line here (ok maybe not quite, but let's keep at this for the moment). Attributes such as number of channels (interleaved), sampling rate and vector size are also needed, and of course the audio signal vector itself.

This makes up the `AudioBase` class of the library, which begins like this:

```
class AudioBase {
protected:
  uint32_t m_nchnls;
  uint32_t m_vframes;
  std::vector<double> m_vector;
  double m_sr;
  uint32_t m_error;
```

Having a fundamentally neutral base, with no hint of what a DSP object might want to implement allows me to use it for absolutely everything I can think of, or almost. So of the 50-odd classes currently sitting in the library, only four are not derived from `AudioBase` (fig. 1). It is specialised for common time-domain operations (oscillators, filters, envelopes, etc.), for spectral processing (short-time Fourier transform, phase vocoder), for function tables, for audio input/output, and even for higher-level instrument models. Code re-use is truly maximised.

3 Developing Instruments

A detailed description of the library design is offered elsewhere [Lazzarini, 2017c]. In this paper, we will to look at using it for C++ instrument development. So let's explore some cases[2].

3.1 Basic examples

We begin with a trivial case: a ping instrument, written as a command-line program. This just plays a 440Hz, -6dB sine wave to the output for a couple of seconds. The code, without its safety checks etc, can be abbreviated as this seven-liner:

```
int main() {
  Oscil sig(0.5,440);
  SoundOut output("dac");
  for (int i = 0; i < def_sr * 2;
           i += def_vframes)
    output.write(sig.process());
  return 0;
}
```

Frequency and amplitude are not changing, so I pick the `process()` overload with no parameters and stick its return value straight into the output `write()` method. The two classes

[2]all examples available in the `examples` directory of the AuLib repository.

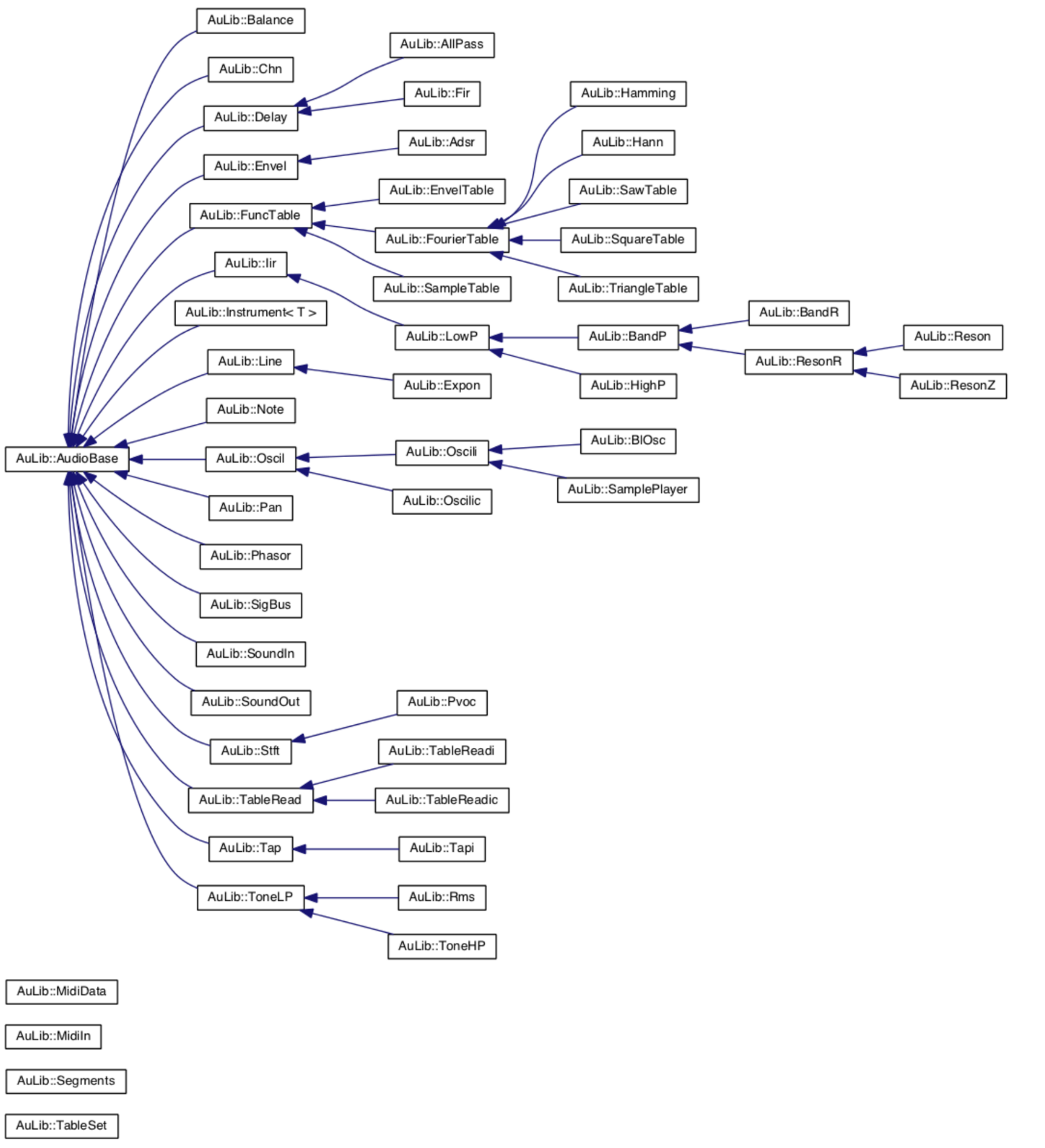

Figure 1: The AuLib class library

share the base, but they have distinctly-named and defined processing methods.

Let's try something slightly less simplistic. A similarly-placed instrument but now with a sweeping resonant filter acting on a sawtooth wave:

```
int main() {
  TableSet saw(SAW);
  BlOsc sig(0.5, 440., saw);
  ResonR fil(1000, 1.);
  Balance bal;
  SoundOut output("dac");
  for (int i = 0; i < def_sr*10;
    i += def_vframes) {
     sig.process();
     fil.process(sig,
      1000. + 400. * i / def_sr);
     bal.process(fil, sig);
      output.write(bal);
   }
  return 0;
}
```

TableSet creates a set of tables for a band-limited oscillator. The filter centre frequency is varied over time, and we feed its output into a balancing operator that uses a comparator to keep amplitudes under control.

To demonstrate how the base class-defined operations can be useful, we have a simple FM example

```
int main() {
  double fm = 440., fc = 220., ndx = 5.;
  Oscili mod, car;
  SoundOut output("dac");
  for (int i = 0; i < def_sr*10;
         i += def_vframes) {
    mod.process(ndx * fm, fm);
    car.process(0.5, mod += fc);
    output.write(car);
  }
  return 0;
}
```

Note the use of the overloaded sum-assignment operator in `mod += fc` to add the modulator signal to the carrier (scalar) frequency.

3.2 Instrument Models

Clearly, the examples above are more demonstrations of how instruments can be set up. This would, in a more realistic scenario, be placed in plugin or GUI application wrapping code, where they can become useful. The AuLib also provides some modelling of instruments and instances of these. We can show how these work in a straightforward application case: a polyphonic MIDI synthesiser.

The AuLib class `Note` provides the base for an instance of a sound object, which can be for example, a synthesiser voice. This holds basic parameters such as amplitude, cps pitch, etc. that we can use to control a sound object. To use it, we derive our own, and specialise its `dsp` method, placing our sound processing code there.

```
class SineSyn : public Note {
  // signal processing objects
  Adsr m_env;
  Oscili m_osc;

  // DSP override
  virtual const SineSyn &dsp() {
    if (!m_env.is_finished())
      set(m_osc(m_env(), m_cps));
    else clear();
    return *this;
  }
...
```

The sound synthesis is again, trivial, to keep the example focused: an envelope and a sine wave oscillator. But note that we have a new convenient interface: using the classes `operator()`, we connect objects more easily one into another. This syntax reinforces the connection metaphor, envelope, alongside pitch, into oscillator. Given that the class is derived from `AudioBase`, we set its vector to the result of the processing.

Additionally, we want to specialise two other methods: for sound onset and sound termination:

```
// note off processing
virtual void off_note() {
    m_env.release();
}

// note on processing
virtual void on_note() {
   m_env.reset(m_amp, 0.01,
     0.5, 0.25 * m_amp, 0.01);
}
```

This plus the constructor completes our `Note`-derived class. Now we want to model the whole synthesiser, not just its voices. To do this, we can use the `Instrument` template class, instantiated with the required number of voices and our note class:

```
Instrument<SineSyn> synth(8);
```

An important aspect of this class is that it has a `dispatch()` method that takes in five parameters (message type, channel, data1, data2, time stamp) and responds to two message types (NOTE ON, NOTE OFF). While these are the same as the MIDI channel messages, we are just re-using the metaphor here. The call to `dispatch()` does not need to originate from MIDI or be limited to the usual MIDI data ranges. Specialisations of instrument can re-implement message handling to allow for other types. `Instrument` also handles polyphony using last-note priority, and this can also be overriden in derived classes.

Given that the example will use MIDI input, the library supports a simple MIDI listener class that takes an `Instrument` object (or from any type implementing `dispatch()` and `process()` and responds to messages. The complete program becomes very straightforward (trivial signal handler implementation omitted):

```
int main() {
 int dev;
 Instrument<SineSyn> synth(8);
 SoundOut out("dac");
 MidiIn midi;
 std::signal(SIGINT, signal_handler);

 std::cout <<
```

```
   "Available MIDI inputs:\n";
 for(auto &devs:
        midi.device_list())
   std::cout << devs << std::endl;
 std::cout << "choose a device: ";
 std::cin >> dev;

 if (midi.open(dev) ==
     AULIB_NOERROR) {
   std::cout <<
    "running...
     (use ctrl-c to close)\n";
   while (running)
      // listen to midi on
      // behalf of synth
      out(midi.listen(synth));
 } else
   std::cout <<
    "error opening device...\n";
  std::cout << "...finished \n";
  return 0;
}
```

Again, with a few lines of code, we can get a basic MIDI synthesiser instrument. Although the synthesis is simple, it can be shown that the effort involved in more complex examples scales well. It is just a case of using other signal processing objects in different arrangements.

4 Csound Plugins

The second case of C++ instrument development we will look at focuses on creating components (plugins) that can be employed in a music programming language. Unit generators in Csound are known as *opcodes* and the system has a well-document C interface for the purpose of adding new ones of these to it. It also has a C++ base class that has been used for a small number of opcode plugin libraries that come with the system.

With the intention of enabling a more complete and well-integrated C++ support for plugin opcode development, I have introduced the Csound Plugin Opcode Framework[3] CPOF (pronounced *see-pough* or *cipó = vine* in Portuguese[4]). The actual framework part of it is fairly light, consisting of two template base classes, but it also contains an extensive set of utility classes that wrap Csound C code for C++ use in a very idiomatic way (table 1). CPOF is discussed extensively in [Lazzarini, 2017b].

[3]available as part of Csound, `github.com/csound/csound`, with code examples in the `examples/plugin` directory.

[4]as in: C++ gives you enough vine, or rope, for you to either hoist yourself up a tree, or hang yourself fairly decently.

Class	Description
`Csound`	The Csound engine
`Params`	Opcode parameters
`AudioSig`	Audio signals
`Fsig`	Spectral signals
`Pvframe<T>`	Spectral data frames
`Pvbin<T>`	Spectral data bins
`Vector<T>`	Array variables
`Table`	Function tables
`AuxMem<T>`	Dynamic memory
`Thread`	Multithreading
`Plugin<N,M>`	Plugin base class
`FPlugin<N,M>`	Spectral plugin base class

Table 1: Classes provided by CPOF.

5 Plugin Examples

The Csound language has a variety of internal data types that its opcodes can process. We will look at each one of these with a programming example.

5.1 Init-time opcodes

In Csound, code that is run only once per instantiation (or again on explicit re-initisation) employs init-time variables. These are scalar types holding a floating-point number (the `MYFLT` type defined by the system). Plugin opcodes for these types are derived from `Plugin` and are instantiated templates taking the number of output and input arguments (respectively) as parameters. The following examples uses the standard library Gaussian generator to produce a random number using the normal distribution. The first input argument is the mean, followed by the deviation, and the seed:

```
#include <plugin.h>
struct Gauss :
  csnd::Plugin<1, 3>{
  std::normal_distribution<MYFLT> norm;
  std::mt19937 gen;

  init init(){
   csnd::constr(&norm, inargs[0],
      inargs[1]);
   csnd::constr(&gen, inargs[2]);
   outargs[0] = norm(gen);
   csnd::destr(&norm);
   csnd::destr(&gen);
  }
};
```

Note that because Csound instantiates the plugin object and it does not know anything about C++ constructors, we need to explicitly construct the objects `norm` and `gen`. When we are done, we need to destruct them as they are likely to have allocated resources, which we do not want to be left dangling. The `Plugin` base class gives us the `inargs` and `outargs` objects, which contain the input and output arguments respectively.

In order for the plugin to be added to Csound's collection of opcodes, we need to register it. To do this, we implement the `csnd::on_load()` function, where we place a call to the `csnd::plugin<T>()` template method, passing the argument types (`"i"`) and the action time of the opcode (`thread::i`), as well as the opcode name we will use ("guassian"):

```
#include <modload.h>
void csnd::on_load(Csound
     *csound) {
csnd::plugin<Gauss>(csound, "gaussian",
  "i", "iii", csnd::thread::i);
}
```

5.2 Control-rate opcodes

The next data type we can tackle is the one used control-rate variables (k). This is also a scalar, but now the opcode is active at performance time (as well as init). A control-rate version of the `gaussian` opcode would look like this:

```
struct GaussP :
 csnd::Plugin<1, 3>{
  std::normal_distribution<MYFLT>
    norm;
  std::mt19937 gen;

  int init(){
   csnd::constr(&norm, inargs[0],
      inargs[1]);
   csnd::constr(&gen, inargs[2]);
   csound->plugin_deinit(this);
   return OK;
  }

  int deinit(){
   csnd::destr(&norm);
   csnd::destr(&gen);
   return OK;
  }

  int kperf() {
    outargs[0] = norm(gen);
    return OK;
  }
};
```

We can see that we now supplied the `kperf()` that will be called repeatedly during performance. Another difference is that we have to provide a `deinit()` to call the destructors, which will be called when performance ends. This method needs to be registered separately with Csound through the `plugin_deinit()` template function. We register this version of the opcode with:

```
csnd::plugin<GaussP>(csound,"gaussian",
 "k", "iii", csnd::thread::ik);
```

5.3 Audio-rate opcodes

For audio signals, we need to implement the `aperf()` method. The variable now is a vector, so we have to use an `AudioSig` object to hold it. The following example shows an `aperf()` method that can be added to `GaussianPerf` to implement an audio rate opcode:

```
int aperf(){
 csnd::AudioSig out(this, outargs(0));
 for(auto &sample : out)
      sample = norm(gen);
 return OK;
}
```

The same class can then be registered for an audio-rate output:

```
csnd::plugin<GaussP>(csound, "gaussian",
 "a", "iii", csnd::thread::ia);
```

5.4 Spectral signals

Spectral signals in Csound are carried from opcode to opcode using fsig variables. These are self-describing variables holding one frame of frequency-domain data, plus associated information about the stream. In CPOF, we manipulate these using the `pv_stream` class. Similarly to audio signals we can get the fsig data off arguments into objects of these types for processing. An opcode is responsible for initialising its own output stream, which we can do at init time. Stream frames can be decomposed in separate bins held by `pv_bin` objects.

The example below shows a plugin that implements spectral tracing [Wishart, 1996] defined as retaining only the loudest N bins in each frame. Some important aspects to note about this code: (a) spectral processing occurs at a rate determined by the frame analysis rate, so we run it a k-rate and process frames as they become available; (b) a framecount, a member variable of the FPlugin base class, is kept for this. (c) The `AuxMem` is used to manage a heap-allocated block of memory to keep bin amplitudes; and (d) we add the types as a static constant member of the class, which simplifies the plugin registration call.

The basic algorithm is as follows:

1. get the amplitudes from each bin;

2. find the nth loudest;
3. use this as a threshold to filter the frame date, keeping only the bin holding amplitudes above it.

```
#include <plugin.h>
#include <algorithm>

struct PVTrace : csnd::FPlugin<1, 2> {
 csnd::AuxMem<float> amps;
 static constexpr
  char const *otypes = "f";
 static constexpr
  char const *itypes = "fk";

 int init() {
  if(inargs.fsig_data(0).isSliding())
   return csound->init_error(
     Str("sliding not supported"));

  if(inargs.fsig_data(0).fsig_format()
     !=csnd::fsig_format::pvs &&
     inargs.fsig_data(0).fsig_format()
     !=csnd::fsig_format::polar)
      return csound->init_error(
      Str("fsig format not supported"));

  amps.allocate(csound,
    inargs.fsig_data(0).nbins());

  csnd::Fsig &fout =
   outargs.fsig_data(0);
  fout.init(csound,
   inargs.fsig_data(0));

  framecount = 0;
  return OK;
 }

 int kperf() {
  csnd::pv_frame &fin =
   inargs.fsig_data(0);
  csnd::pv_frame &fout =
   outargs.fsig_data(0);

  if(framecount < fin.count()) {
   int n =  fin.len() - (int)inargs[1];
   float thrsh;

   std::transform(fin.begin(),fin.end(),
     amps.begin(), [](csnd::pv_bin f){
      return f.amp(); });

   std::nth_element(amps.begin(),
     amps.begin()+n, amps.end());
   thrsh = amps[n];

   std::transform(fin.begin(), fin.end(),
    fout.begin(),
     [thrsh](csnd::pv_bin f){
      return f.amp() >= thrsh ?
        f : csnd::pv_bin(); });

   framecount = fout.count(fin.count());
  }
   return OK;
 }
};

#include <modload.h>
void csnd::on_load(Csound *csound) {
 csnd::plugin<PVTrace>(csound,
   "pvstrace", csnd::thread::ik);
}
```

The standard library algorithms are very well suited to implementing these steps. The code becomes very compact and fairly readable.

5.5 Array variables

Csound has a container type, array, which can be used to create vectors of built in types. CPOF provides a template class `Vector<T>` to wrap array arguments conveniently for manipulation. The typedef `myflt_vector` is an instantiation of this template for real values (`MYFLT`). The following example combines the use of lambdas and templates to create a whole family of binary (two-operand) operators for numeric (scalar) arrays. It can be used for init and k-rate opcodes. The processing is placed on a separate function to avoid code duplication. It is just a matter of mapping the inputs into the outputs through the application of a given function.

```
template <MYFLT (*bop)(MYFLT, MYFLT)>
struct ArrayOp2 : csnd::Plugin<1, 2> {

  int process(csnd::myfltvec &out,
              csnd::myfltvec &in1,
              csnd::myfltvec &in2) {
   std::transform(in1.begin(), in1.end(),
    in2.begin(), out.begin(),
    [](MYFLT f1, MYFLT f2) {
     return bop(f1, f2); });
    return OK;
  }

  int init() {
    csnd::myfltvec &out =
     outargs.myfltvec_data(0);
    csnd::myfltvec &in1 =
     inargs.myfltvec_data(0);
    csnd::myfltvec &in2 =
     inargs.myfltvec_data(1);

    if (in2.len() < in1.len())
     return csound->init_error(
      Str("second input array"
          " is too short\n"));

    out.init(csound, in1.len());
    return process(out, in1, in2);
  }

  int kperf() {
   return
```

```
    process(outargs.myfltvec_data(0),
         inargs.myfltvec_data(0),
         inargs.myfltvec_data(1));
  }
};
```

This class template then is instantiated to create various opcodes based on different two-operand functions:

```
csnd::plugin<ArrayOp2<std::atan2>>
 (csound, "taninv",
  "i[]", "i[]i[]", csnd::thread::i);
csnd::plugin<ArrayOp2<std::atan2>>
 (csound, "taninv",
  "k[]", "k[]k[]", csnd::thread::ik);
csnd::plugin<ArrayOp2<std::pow>>
 (csound, "pow",
  "i[]", "i[]i[]", csnd::thread::i);
csnd::plugin<ArrayOp2<std::pow>>
 (csound, "pow",
  "k[]", "k[]k[]", csnd::thread::ik);
csnd::plugin<ArrayOp2<std::hypot>>
 (csound, "hypot",
  "i[]", "i[]i[]", csnd::thread::i);
csnd::plugin<ArrayOp2<std::hypot>>
 (csound, "hypot",
  "k[]", "k[]k[]", csnd::thread::ik);
csnd::plugin<ArrayOp2<std::fmod>>
 (csound, "fmod",
  "i[]", "i[]i[]", csnd::thread::i);
csnd::plugin<ArrayOp2<std::fmod>>
 (csound, "fmod",
  "k[]", "k[]k[]", csnd::thread::ik);
csnd::plugin<ArrayOp2<std::fmax>>
 (csound, "fmax",
  "i[]", "i[]i[]", csnd::thread::i);
csnd::plugin<ArrayOp2<std::fmax>>
 (csound, "fmax",
  "k[]", "k[]k[]", csnd::thread::ik);
csnd::plugin<ArrayOp2<std::fmin>>
 (csound, "fmin",
  "i[]", "i[]i[]", csnd::thread::i);
csnd::plugin<ArrayOp2<std::fmin>>
 (csound, "fmin",
  "k[]", "k[]k[]", csnd::thread::ik);
```

This is a good example of how we can apply modern a C++ idiom to create compact code for the generation of a family of related opcodes.

6 Conclusions

Perhaps one of the conclusions of this paper is that C++ is not such a terrible choice for the implementation of computer music instruments. While C is still the preeminent language for audio signal processing, the latest C++ standards have made that language somewhat more interesting, providing almost a blend of high-level scripting with a (hopefully) efficient implementation.

References

ISO/IEC. 2011. ISO international standard ISO/IEC 4882:2011, programming language C++.

ISO/IEC. 2014. ISO international standard ISO/IEC 14882:2014, programming language C++.

ISO/IEC. 2017. Working draft, standard for programming language C++.

V. Lazzarini, J. ffitch, S. Yi, J. Heintz, Ø. Brandtsegg, and I. McCurdy. 2016. *Csound: A Sound and Music Computing System.* Springer Verlag.

V. Lazzarini. 2000. The SndObj sound object library. *Organised Sound*, (5):35–49.

V. Lazzarini. 2013. The development of computer music programming systems. *Journal of New Music Research*, (42):97–110.

V. Lazzarini. 2017a. *Computer Music Instruments.* Springer Verlag.

V. Lazzarini. 2017b. The csound plugin opcode framework. In *SMC 2017 (under review)*, Helsinki.

V. Lazzarini. 2017c. The design of a lightweight dsp programming language. In *SMC 2017 (under review)*, Helsinki.

Y. Orlarey, D. Fober, and S. Letz. 2004. Syntactical and semantical aspects of faust. *Soft Computing*, 8(9):623?632.

T. Wishart. 1996. *Audible Design.* Orpheus The Pantomine.