How to write audio cross-environment

LEVERAGING C++20

FOR

DECLARATIVE AUDIO PLUG-IN SPEC

FR

Jean-Michaël Celerier

jcelerier.name
ossia.io
celtera.dev

@jcelerie / @ossia_io

jcelerie / jcelerier / ossia_io

jcelerier / celtera / ossia

ADC'21

Takeaway of this talk

1. Reflection-based design enables :

- Better reusability
- Lower (zero!) overhead
- Simpler code

Some ideas in common with Nick's ta

Than class-based polymorphism for media frameworks

No undefined behaviour or "non-p

Takeaway of this talk

2. Current C++ (20) enables

   reflection-based design
   (RBD ?)
   struct-oriented declarative specification
   (SODS?)

... if anyone can come up with a better acronym, call me :D

MOTIVATION

```
26
27  const char* LowPass::name = "LowPass";
28  const char* LowPass::category = "Filters";
29  const char* LowPass::description = DOC("This algorithm implements a 1st order IIR low-p
30  "References:\n"
31  "  [1] U. Zölzer, DAFX - Digital Audio Effects, p. 40,\n"
32  "  John Wiley & Sons, 2002");
33
34
35  void LowPass::configure() {
36
37    Real fs = parameter("sampleRate").toReal();
38    Real fc = parameter("cutoffFrequency").toReal();
39
40    Real c = (tan(M_PI*fc/fs) - 1) /
41             (tan(M_PI*fc/fs) + 1);
42
43    vector<Real> b(2, 0.0);
44    b[0] = (1.0+c)/2.0;
45    b[1] = (1.0+c)/2.0;
46
47    vector<Real> a(2, 0.0);
48    a[0] = 1.0;
49    a[1] = c;
50
51    _filter->configure("numerator", b, "denominator", a);
52  }
53
54  void LowPass::compute() {
55    _filter->input("signal").set(_x.get());
56    _filter->output("signal").set(_y.get());
57    _filter->compute();
```

```
	};
	enum OutputIds {
		INTERP_OUTPUT,
		NUM_OUTPUTS
	};
	enum LightIds {
		ENUMS(PHASE_LIGHT, 3),
		NUM_LIGHTS
	};

	LowFrequencyOscillator<float_4> oscillators[4];
	dsp::ClockDivider lightDivider;

	LFO2() {
		config(NUM_PARAMS, NUM_INPUTS, NUM_OUTPUTS, NUM_LIGHTS);
		configParam(OFFSET_PARAM, 0.f, 1.f, 1.f, "Offset");
		configParam(INVERT_PARAM, 0.f, 1.f, 1.f, "Invert");
		configParam(FREQ_PARAM, -8.f, 10.f, 1.f, "Frequency", " Hz", 2, 1);
		configParam(WAVE_PARAM, 0.f, 3.f, 1.5f, "Wave");
		configParam(FM_PARAM, 0.f, 1.f, 0.5f, "Frequency modulation", "%", 0.f, 100.f);
		lightDivider.setDivision(16);
	}

	void process(const ProcessArgs& args) override {
		float freqParam = params[FREQ_PARAM].getValue();
		float fmParam = params[FM_PARAM].getValue();
		float waveParam = params[WAVE_PARAM].getValue();

		int channels = std::max(1, inputs[FM_INPUT].getChannels());

		for (int c = 0; c < channels; c += 4) {
			auto* oscillator = &oscillators[c / 4];
			oscillator->invert = (params[INVERT_PARAM].getValue() == 0.f);
```

```
, mGain(1)
, mGainSlider(nullptr)
{
}

void Amplifier::CreateUIControls()
{
   IDrawableModule::CreateUIControls();
   mGainSlider = new FloatSlider(this,"gain",5,2,110,15,&mGain,0,4);
}

Amplifier::~Amplifier()
{
}

void Amplifier::Process(double time)
{
   PROFILER(Amplifier);

   if (!mEnabled)
      return;

   SyncBuffers();
   int bufferSize = GetBuffer()->BufferSize();

   IAudioReceiver* target = GetTarget();
   if (target)
   {
      ChannelBuffer* out = target->GetBuffer();
      for (int ch=0; ch<GetBuffer()->NumActiveChannels(); ++ch)
      {
         auto getBufferChannelCh = GetBuffer()->GetChannel(ch);
         for (int i=0; i<bufferSize; ++i)
         {
```

```
	TTFloat64	mLowBound;		///< Attribute: low bound for clipping
	TTFloat64	mHighBound;		///< Attribute: high bound for clipping

	TTErr calculateValue(const TTFloat64& x, TTFloat64& y, TTPtrSizedInt channel)
	{
		y = x;
		TTLimit(y, mLowBound, mHighBound);
		return kTTErrNone;
	}

	/** Audio Processing Method */
	TTErr processAudio(TTAudioSignalArrayPtr inputs, TTAudioSignalArrayPtr outputs)
	{
		TT_WRAP_CALCULATE_METHOD(calculateValue);
	}
};

TT_AUDIO_CONSTRUCTOR_EXPORT(Clipper)
{
	addAttribute(		LowBound,	kTypeFloat64);
	addMessageProperty(	LowBound,	description,	TT("Sets the minimum amplitude."));

	addAttribute(		HighBound,	kTypeFloat64);
	addMessageProperty(	HighBound,	description,	TT("Sets the maximum amplitude."));

	setAttributeValue(TT("lowBound"), -1.0);
	setAttributeValue(TT("highBound"), 1.0);
	setProcessMethod(processAudio);
}


TTClipper::~TTClipper()
{;}
```

```
class Gain : public AudioObject {
public:
	static constexpr Classname classname = { "gain" };
	static constexpr auto tags = { "dspEffectsLib", "audio", "processor", "dynamics" };


	Parameter<double, Limit::None<double>, NativeUnit::LinearGain> gain = { this, "gain", 1.0 };  ///< Linear gain


	/** Process one sample.
		@param x		Sample to be processed.
		@return		Processed sample
	*/
	Sample operator()(const Sample x)
	{
		return x * gain;
	}


	/** Process a SharedSampleBundleGroup.
		@param x		SharedSampleBundleGroup to be processed.
		@return		Processed SharedSampleBundleGroup.
	*/
	SharedSampleBundleGroup operator()(const SampleBundle& x)
	{
		auto out = adapt(x);

		for (int channel=0; channel<x.channelCount(); ++channel)
			std::transform(x[channel].begin(), x[channel].end(), out[0][channel].begin(), *this);
		return out;
	}
```

```
	class_addmethod(c, (method)myObj_del, "int", A_LONG, 0);
	class_addmethod(c, (method)myObj_assist,"assist",A_CANT,0);
	class_dspinit(c);
	class_register(CLASS_BOX, c);
	myObj_class = c;

	post("vb.fbosc~ by volker böhm, version 1.0.1");

	return 0;
}

void myObj_float(t_myObj *x, double f) {

	switch (proxy_getinlet((t_object *)x)) {
		case 0:
			myObj_freq(x, f); break;
		case 1:
			myObj_fb(x, f); break;
		case 2:
			myObj_cf(x, f); break;
	}
}

void myObj_freq(t_myObj *x, double input) {
	x->incr = input * x->r_sr;		// calculate pointer increment from input freq
	if(input<0.0) {
		x->incr *= -1;
	}
}

void myObj_fb(t_myObj *x, double input) {
	if(input>=-1 && input<1)
		x->fb = input;
}

void myObj_cf(t_myObj *x, double input) {

	CLIP_ASSIGN(input, 10, 0.43/x->r_sr);
	x->b1 = exp(-2*PI*input * x->r_sr);
	x->a0 = 1 - x->b1;
}
```

```
AudioParam::AudioParam(const std::string & name, const std::string & shortName, double defaultValue, double minValue, double max
    : AudioSummingJunction()
    , m_name(name)
    , m_shortName(shortName)
    , m_value(defaultValue)
    , m_defaultValue(defaultValue)
    , m_minValue(minValue)
    , m_maxValue(maxValue)
    , m_units(units)
    , m_smoothedValue(defaultValue)
    , m_smoothingConstant(DefaultSmoothingConstant)
{
}

AudioParam::~AudioParam() {}

float AudioParam::value() const
{
    return static_cast<float>(m_value);
}

void AudioParam::setValue(float value)
{
    if (!std::isnan(value) && !std::isinf(value))
        m_value = value;
}

float AudioParam::smoothedValue()
{
    return static_cast<float>(m_smoothedValue);
}

bool AudioParam::smooth(ContextRenderLock & r)
{
```

Experimental observation

class-based inheritance works for organisation-wide polymorphism

but not

ecosystem-wide

All of those :

implement ad-hoc runtime OO

because

C++ does not have reflection

yet many systems in which we want to use C++ for

need that info

Motivation

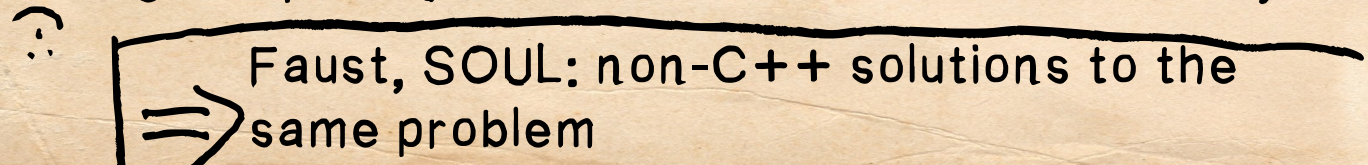
RAGE AGAINST THE GLUE

- Reusability: Max/MSP objects, audio and video processors, "OOP" types...

  → MxN problem

- Portability: C++ : SUPER portable.
  Frameworks? No.
  ↳ Separate algo from binding

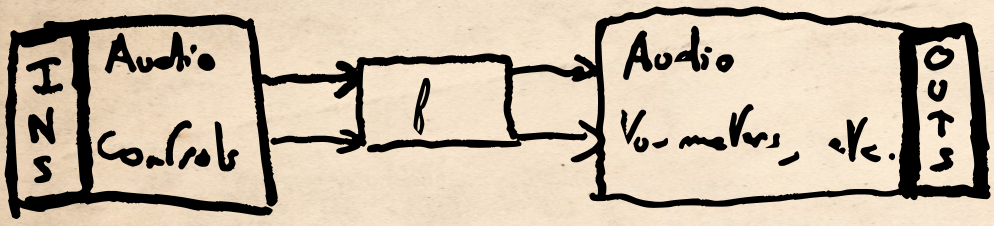
- Ideological purity: Canonical definition of an "object"



Motivation (2)

→ Problem !

- What is an audio processor ?
  - ↳ Simple definition for this talk :



- "Easy" : defining f
- Hard : enumerating, declaring INS & OUTS

C++ historically bad (at that) ⇒ People do it at run-time

→ Not zero-cost

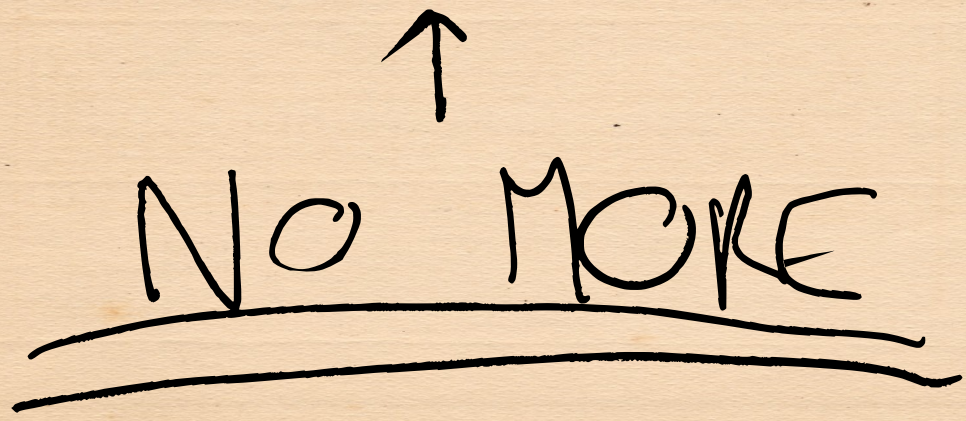
→ Framework-ization

Motivation (3)

- Most existing solutions:
  enum-based or array access-based

```
enum Parameters {
  kGain,
  kCutoff,
  kResonance,
  ...
};
```

```
switch (paramId) {
  case kGain:
    this->gain = param;
    break;
  ...
}
```



Motivation(4)

- The processor should entirely be defined through elements that the compiler can type-check.
  - Removes an entire class of bugs
  - Reduces duplication, improves legibility

```
struct Filter {
  float gain;
  void process(
   double** in, double** out, int N) {
      for(each sample)
        out = in * gain;
  }
};
```

This ought to be enough !!!!

Motivation (5)

- In 2021, C++ compilers are not advanced enough to allow defining everything with only the above

Motivation(5)

there are limits ! the point of this talk is taht you can send kindly

- In 2021, C++ compilers are not advanced enough to allow defining everything with only the above
  - ↳ No reflection on member names :(

BUT we can get very close !

- ↳ Thanks to a structured-binding technique
  - ⇒ Implemented in Boost.PFR
- ↳ One limitation: aggregate types
  - ⇒ In practice: very OK


⚠TRADE OFFER⚠
i receive:
a bare C++ struct
you receive:
vst3 plug-in
osc api
python binding
autogenerated UI
puredata object
max/msp object

Demo

github.com/celtera/avendish

avendish

GPLv3 / Commercial

Verrryyy experimental

```cpp
struct Addition
{
  static consteval auto name() { return "Addition"; }
  static consteval auto c_name() { return "avnd_addition"; }
  static consteval auto uuid()
  {
    return "36427eb1-b5f4-4735-a383-6164cb9b2572";
  }

  struct
  {
    struct
    {
      float value;
    } a;
    struct
    {
      float value;
    } b;
  } inputs;

  struct
  {
    struct
    {
      float value;
    } out;
  } outputs;

  void operator()() { outputs.out.value = inputs.a.value + inputs.b.value; }
};
```

you can't do ANYTHING against your

```cpp
#pragma once
#include <cmath>
#include <algorithm>
#include <numeric>

struct MiniPerSampleProcessor
{
  static consteval auto name() { return "Per-sample processor"; }
  static consteval auto c_name() { return "avnd_persample_1"; }
  static consteval auto uuid() { return "c0ece845-2df7-486d-b096-6d507cbe23d1"; }

  struct outputs { };
  struct inputs {
    struct {
      static consteval auto name() { return "Gain"; }
      static consteval auto control() {
        struct
        {
          const float min = 0.001;
          const float max = 500.0;
          const float init = 1;
        } c;
        return c;
      }

      float value;
    } gain;
  };

  float internal_state{};
  float operator()(float input, const inputs& ins, outputs& outs)
  {
    float out = std::clamp(
        std::fmod(input * ins.gain.value, internal_state + 1.0f), -0.5f, 0.5f);
    internal_state = std::exp(out) + 0.1;
    return out;
  }
```

```
  // A second control
  struct
  {
    static consteval auto name() { return "Volume"; }
    float value{1.0};
  } volume;
} inputs;

// We define the type of our programs, like in the other cases
// it ends up being introspected automatically.
struct program
{
  std::string_view name;
  decltype(Presets::inputs) parameters;
};

// Note: it's an array instead of a function because
// it's apparently hard to deduce N in array<..., N>, unlike in C arrays.
static constexpr const program programs[]{
    {.name{"Low gain"}, .parameters{.preamp = {0.3}, .volume = {0.6}}},
    {.name{"Hi gain"}, .parameters{.preamp = {1.0}, .volume = {1.0}}},
};

void operator()(double** in, double** out, int frames)
{
  const double preamp = 100. * inputs.preamp.value;
  const double volume = inputs.volume.value;

  for (int c = 0; c < channels; c++)
    for (int i = 0; i < frames; i++)
      out[c][i] = volume * std::tanh(in[c][i] * preamp);
}
};
}
```

```
class Lowpass
{
public:
  $(name, "Lowpass (helpers)")
  $(c_name, "avnd_helpers_lowpass")
  $(uuid, "82bdb9b5-9cf8-440e-8675-c0caf4fc59b9")

  using setup = avnd::setup;
  using tick = avnd::tick;

  struct
  {
    avnd::dynamic_audio_bus<"Input", double> audio;
    avnd::hslider_f32<"Weight", avnd::range{.min = 0., .max = 1., .init = 0.5}> weight;
  } inputs;

  struct
  {
    avnd::dynamic_audio_bus<"Output", double> audio;
  } outputs;

  void prepare(avnd::setup info)
  {
    previous_values.resize(info.input_channels);
  }

  // Do our processing for N samples
  void operator()(avnd::tick t)
  {
    // Process the input buffer
    for (int i = 0; i < inputs.audio.channels; i++)
    {
```

```
    previous_values.resize(info.input_channels);
  }

  // Do our processing for N samples
  void operator()(avnd::tick t)
  {
    // Process the input buffer
    for (int i = 0; i < inputs.audio.channels; i++)
    {
      auto* in = inputs.audio[i];
      auto* out = outputs.audio[i];

      float& prev = this->previous_values[i];

      for (int j = 0; j < t.frames; j++)
      {
        out[j] = inputs.weight * in[j] + (1.0 - inputs.weight) * prev;
        prev = out[j];
      }
    }
  }

private:
  // Here we have some state which depends on the host configuration (number of ch
  std::vector<float> previous_values{};
};
```

-> buried: inheritance for cod

Demo

Getting there

- First tentative :
  ossia score's Fx API

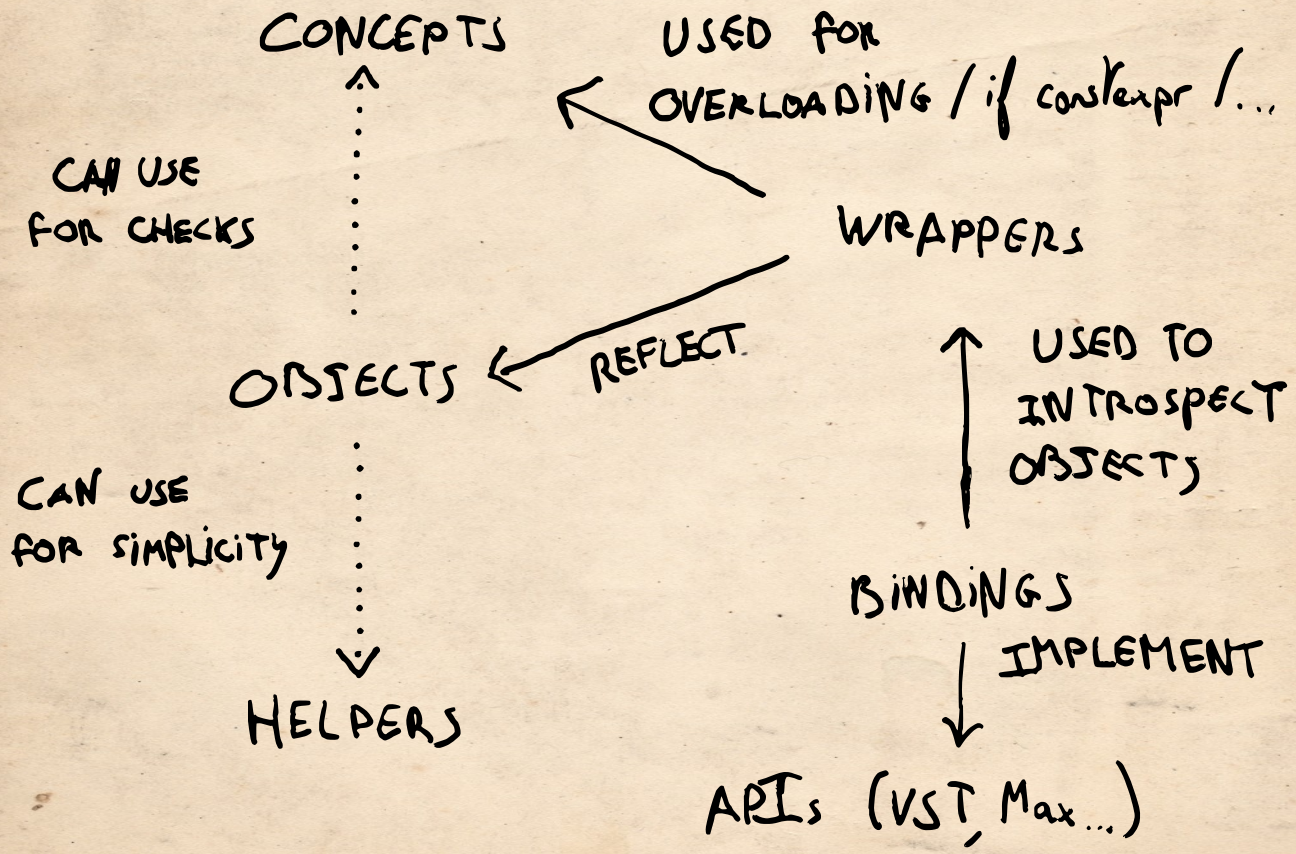
- Second tentative :
  github.com/jcelerier/vintage

- Third tentative :
  github.com/jcelerier/score-simple-api-2

- Fourth (final?)
  github.com/celtera/avendish

Architecture

Consuming the "wrapper



DEEP DIVE

Deep dive — Concepts

- require-expression : "is this code valid ?"

```
constexpr bool ok =
    requires { 2 + 2; };
```

```
requires { foo.bar(123); };
```

```
if constexpr (requires { obj(N) ; })
    obj(N);
else if constexpr(requires {
    obj(in, out, N);
})
    obj(in, out, N);
```

Deep dive Concepts

- concept — an application from the type domain to the boolean domain

```
template<typename T>
concept HasBar = requires (T t) {
  t.bar();
};

template<typename T>
concept FloatBar = requires (T t) {
 { t.bar() } -> std::floating_point;
};
```

Deep dive

Concepts

- Usage : ① `if constexpr(HasFoo<T>) …`

②
```
template<typename T> struct X;

template<HasFoo T> struct X<T>;

template<typename T>
requires (!HasFoo<T>)
struct X<T>;
```

③
```
void f(HasFoo auto x);
template<HasFoo T>
void f(T x);
```

For us : is it…

⇒ … A per-sample audio processor ?

… Per-block ?

… A synth voice ?

```cpp
// struct { float sample; };
template <typename FP, typename T>
concept audio_sample_port = std::same_as<decltype(T::sample), FP>;
template <typename T>
concept generic_audio_sample_port = std::floating_point<decltype(T::sample)>;

// struct { float sample; };
template <typename FP, typename T>
concept poly_array_sample_port = std::same_as<decltype(T::samples), FP**>;

template <typename T>
concept has_programs = requires(T t)
{
  std::size(T::programs);
};

// MIDI messages
template <typename T>
concept midi_message = requires(T t)
{
  t.bytes;
  t.timestamp;
};
```

Let's come up as a community and define the '

Resumable functions

```
generator<int> counter() {
  int i = 0 ;
  while(1)
    co_yield i++;
}

generator<int> gen = counter();

print(gen()); // 0
print(gen()); // 1
print(gen()); // 2
```

Coroutines: why ?

- Allows for generic collection iteration. With ranges: bee's knees.

- In our case: we do not want to pay the cost of a container for the "single object" case.

- Also useful for async work (HTTP...), less relevant for this talk.

```
 */
template <typename T>
struct effect_container
{
  using type = T;

  T effect;

  void init_channels(int input, int output)
  {
    // TODO maybe a runtime check
  }

  auto& inputs() noexcept { return effect.inputs; }
  auto& inputs() const noexcept { return effect.inputs; }
  auto& outputs() noexcept { return effect.outputs; }
  auto& outputs() const noexcept { return effect.outputs; }

  member_iterator<T> effects() { co_yield effect; }
};
```

```cpp
template <avnd::monophonic_audio_processor T>
requires avnd::inputs_is_type<T> && avnd::outputs_is_type<T>
struct effect_container<T>
{
  using type = T;

  typename T::inputs inputs_storage;

  struct state
  {
    T effect;
    typename T::outputs outputs_storage;
  };

  std::vector<state> effect;

  member_iterator<T> effects()
  {
    for (auto& e : effect)
      co_yield e.effect;
  }

  void init_channels(int input, int output)
  {
    effect.resize(std::max(input, output));
  }
```

Remember Timur's talk yesterday.Here we rely o

Deep dive — Template lambdas

```
void print_nth (Tuple & t, int i){

  [i] <size_t... N> (index_sequence <N...>){

      (
        (i==N and (print ( get<N>(t)), true))

          or ...

      );
  }(make_index_sequence <tuple_size_v <Tuple >>( ));

}
```

Deep dive — Template lambdas

```
void print_nth (Tuple & t, int i){

  [i] <size_t... N> (index_sequence<N...>){
   (
    (i==N and (print(get<N>(t)), true))
    or ...
   );
  }(make_index_sequence<tuple_size_v<Tuple>>());

}
```

- `[i]` — capture
- `<size_t... N>` — template args
- `(index_sequence<N...>)` — deduction from args
- `i==N and (print(get<N>(t)), true)` — clang turns that into a switch
- `get<N>(t)` — std::tuple
- `true` — for `and`
- `or ...` — C++ 17 fold-expression
- `}(make_index_sequence<tuple_size_v<Tuple>>());` — immediately-evaluated

https://gcc.godbolt.org/z/PvjzETzP8

Deep dive

Template lambdas: what for ?

- Removes the need for many helper functions (which could improve codegen!).

- Simplify generic lambdas: no more

```
using T = std::decay_t<decltype(value)>;
```

- Easier for doing type-level operations.

Deep dive

Generalized NTTP

LIB

```
struct Domain {float min, max;};

template <Domain dom>
struct Control {
    consteval auto domain() {return dom;}
};
```

Like

```
std::array<int, 42>;
```

C++20 : designated initializers

App

```
Control<Domain{.min = -1, .max = 1}> ctrl;
```

Parametrization on values

```
sizeof(ctrl) == 1;
```

(for now !)

Deep dive — Generalized NTTP

LIB

```
template<std::size_t N>
struct static_string {
  char str[N];
  consteval static_string(const char (&s)[N]) {
    std::copy_n(s, N, str);
  }
};
```

⇒ GUARANTEED AT COMPILE-TIME

<N> is deduced!

APP

```
template<static_string s>
struct Control;

Control<"foo"> my_control;
```

Works for any LITERAL TYPE

Deep dive

Generalized NTTP : why ?

- Optimize cache for "parameter" types ; we want

```
sizeof(my_parameter) == sizeof(float)
```

- In existing parameter systems, effective value interspesed with irrelevant metadata

- But CPU caches will still be wasted loading them upon access...

Deep dive

CONSTEVAL

Also works in λ !
`[]() consteval {};`

Same for constexpr

- C++ 11 :

```
constexpr int add(int a, int b) {
  return a + b;
}

int x = add(2, 3);                // RUNTIME
constexpr int y = add(2, 3); // COMPTIME
std::array<float, y> arr;     // OK
```

- C++ 20 :

```
consteval int add(int a, int b)...
int x = add(2, 3);          // COMPTIME :-)
int y = add(rand(), 3); // ERROR :-)
```

https://gcc.godbolt.org/z/na3q4K5P9

Deep dive

Reflection !!

```
struct Foo {
  int x;
  float y;
} foo;
```

Must be a Simple Aggregate

```
boost::pfr::for_each_field(
  foo,
  []<typename F>(F field) {
    fmt::print("{}", field);
  }
);
```

compile-time performance: log(numbe

O Fortuna theme playing

```cpp
void process_inlet_control(t_symbol* s, int argc, t_atom* argv)
{
  switch (argv[0].a_type)
  {
    case A_FLOAT:
    {
      float res = argv[0].a_w.w_float;
      boost::pfr::for_each_field(
          implementation.inputs(),
          [this, s, res]<typename C>(C& ctl)
          {
            if constexpr (requires { ctl.value = float{}; })
            {
              if (std::string_view{C::name()} == s->s_name)
              {
                avnd::apply_control(ctl, res);
              }
            }
          });
      break;
    }
```

Order in which we do things ma

Boost.PFR : what for ?

- Combined with concepts: filter the fields of a struct depending on a predicate.

"Dear compiler, can you give me all the MIDI inputs of my struct ?"

- Automatically map controls to atomics.

Closing Words

- ⟹ struct-oriented declarative programming
- ⟹ consteval methods + NTTP ≃ [[custom attributes]]
- ⟹ concepts + structured binding ≃ Reflection

  ↳ Code which
  - Does not depend on frameworks
  - Will still compile in 20 years
  - Won't have "runtime errors" due to enum mismatch

https://gcc.godbolt.org/z/s5P74cvPf

Closing words (2)

C++ (somewhat deservedly)
has a reputation for feature-bloatitis.

But, when writing the library, it
constantly felt like when something
was needed, the language provided it in
a way that made sense.
So thanks committee :-)

C++20 is neat, use it !
(tested w/ GCC 11, Clang 13, VS2022)

Future work

- Hot reload with LLVM JIT) We can restore IRO for live-reload!
- Combining objects → Graph system
- UI story ?
- Concepts for VFX → abstracting shader pipelines

Promo !

Digital
Art
Workstation

free & OSS !
linux, mac & win !

ossia score

- Sequencer for interactivity
- Modular & timeline in a single UI
- Sequencer for media art :
  OSC, DMX, GLSL, NDI, PureData, Faust...

https://ossia.io

Demo: VST2 in bi

- Thanks for your attention !

(and sorry for the handwriting)

- Questions ? ♡

- Find this work useful ?

https://github.com/sponsors/jcelerier