Abstract or Generic?

Music, topology and abstraction in modern C++

ACCU short talk, September 2017

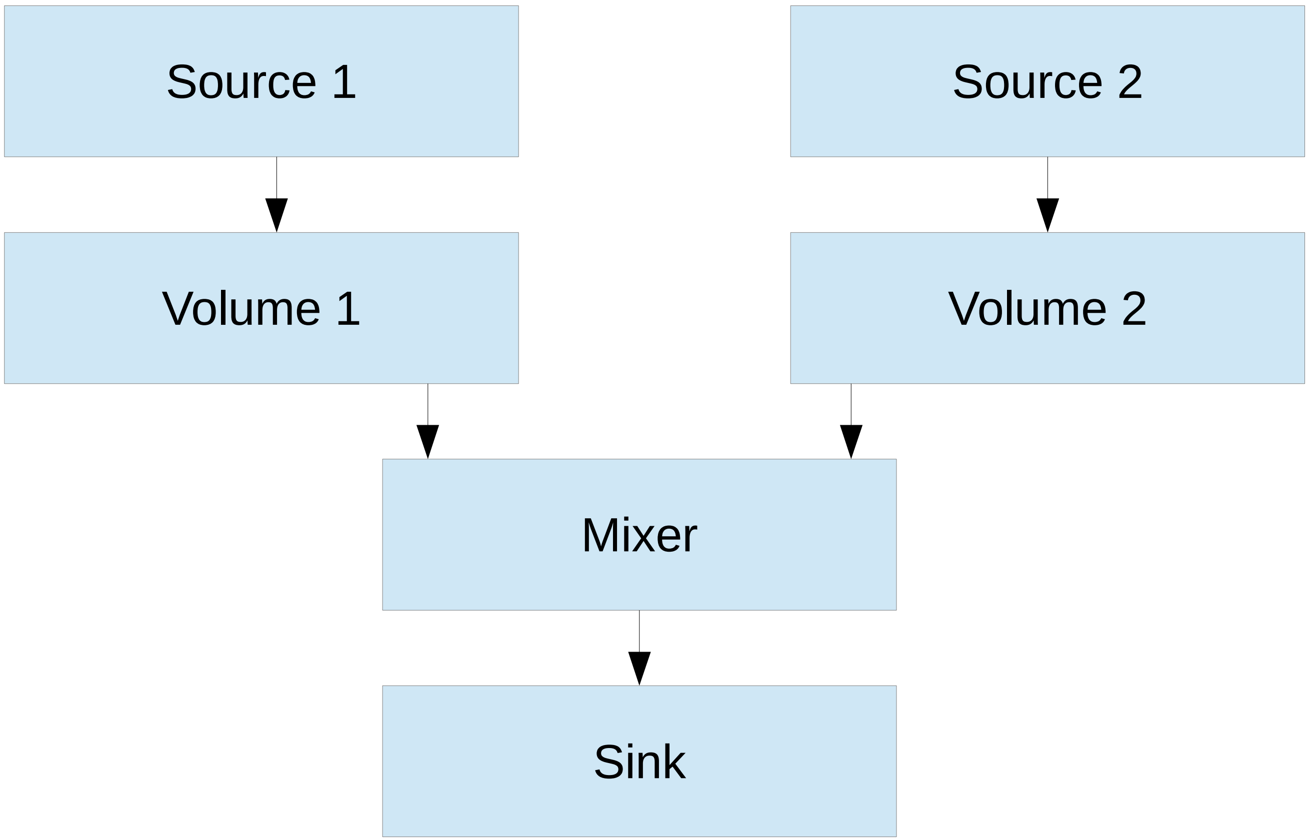
Mike Seymour
mike@mikeseymour.co.uk
github.com/mikeseymour/wocca

Background – Sound Processing

- A lot of knobs, all controlling something

Background – Sound Processing

- Network of components



Goals

- Define components of many types
  - Components have inputs, outputs or both
- Connect to make a network
- Sort into processing order
  - Each before its dependents
  - Sources → Volumes → Mixer → Sink
- Process in order
  - As fast as possible

Approaches

- Abstract (80s style)
  - Common abstract base class for components
  - Virtual function for processing
  - Run-time configuration
  - Pointers and indirect function calls
- Generic (21st century style)
  - Templates deal with concrete component types
  - Compile-time configuration
  - No pointers, direct function calls (can be inlined)
  - Harder to use? Maybe not.

Define components

- Abstract

```
struct component {
    virtual ~component() {}
    virtual void process() = 0;
};
struct mixer final : component {
    input in[2];
    output out;
    void process() override;
};
```

- Generic

```
template <unsigned> struct inputs;
template <unsigned> struct outputs;

struct mixer final : inputs<2>, outputs<1> {
    void process();
};
```

Connect components

- Abstract

```
void connect(output &, input &);
```

```
connect(source1.out, volume1.in);
connect(source2.out, volume2.in);
connect(volume1.out, mixer.in[0]);
connect(volume2.out, mixer.in[1]);
connect(mixer.out, sink.in);
```

- Generic

```
template <class... Connections> struct network;
template <class Output, class Input> struct connect;
```

```
network<
   connect<output<source1>, input<volume1>>,
   connect<output<source1>, input<volume1>>,
   connect<output<source1>, input<volume1>>,
   connect<output<source1>, input<volume1>>,
   connect<output<source1>, input<volume1>>
> network;
```

Sort components and process

- Abstract

```
std::vector<component*> sort(component &);
auto sorted = sort(mixer);

for (component * c : sorted) c->process();
```

- Generic

```
template<class...> struct tuple;

// here be dragons
template<class Tuple, class Function> void visit(Tuple &&, Function &&);
template<class Connections> using sort = tuple<???>;


sort<connections> sorted;
visit(sorted, [](auto & c){c.process();});
```

Dragon taming: tuples

- A simple recursive implementation

```
template <class...> struct tuple {
    template <class Fn> void visit(Fn &&) {}
};
template <class Head, class... Tail> struct tuple<Head, Tail...> {
    Head head;
    tuple<Tail...> tail;
    template <class Fn> void visit(Fn && f) {f(head); tail.visit(f);}
};
template<class Tuple, class Fn> void visit(Tuple && t, Fn && f) {t.visit(f);}
```

- Standard Library
  - Tuples use indexed access rather than recursion
  - Slightly more work to implement, left as an exercise

Dragon taming: basic utilities

- We need some tools to help sort our network

```
// Compare two types. We could use std::is_same.
template <class, class> constexpr bool same = false;
template <class T> constexpr bool same<T,T> = true;

static_assert( same<t1,t1>);
static_assert(!same<t1,t2>);


// Choose a type depending on a condition. We could use std::conditional_t.
template <bool, class T1, class> struct cond_ {using type = T1;};
template <class T1, class T2> struct cond_<false,T1,T2> {using type = T2;};
template <bool C, class T1, class T2> using conditional = typename
cond_<C,T1,T2>::type;

static_assert(same<t1, conditional<true,  t1, t2>>);
static_assert(same<t2, conditional<false, t1, t2>>);
```

Dragon taming: tuple utilities

- We need to manipulate tuples

```
// Check whether a type appears in a tuple
template <class T, class Tuple> constexpr bool contains = false;
template <class T, class H, class... Ts>
    constexpr bool contains<T,tuple<H,Ts...>> =
        same<T,H> || contains<T,tuple<Ts...>>;

// Prepend a type to a tuple
template <class T, class Tuple> struct prepend_;
template <class T, class... Ts> struct prepend_<T,tuple<Ts...>> {
    using type = tuple<T,Ts...>;
};

template <class T, class Tuple> using prepend =
    typename prepend_<T,Tuple>::type;

// Prepend a node to a tuple if it's not already there
template <class T, class Tuple> using prepend_unique =
    conditional<contains<T,Tuple>,Tuple,prepend<T,Tuple>>;
```

Dragon taming: graph theory

- Analyse a graph specified by its edges

```
// An edge of a directed graph. The direction is from Head to Tail.
template <class Head, class Tail> struct edge {};

// Extract the nodes from a set of edges
template <class Edges> struct nodes_ {using type = tuple<>;};
template <class Edges> using nodes = typename nodes_<Edges>::type;
template <class Head, class Tail, class... Edges>
struct nodes_<tuple<edge<Head,Tail>, Edges...>> {
    using add_tail = prepend_unique<Tail,nodes<tuple<Edges...>>>;
    using type = prepend_unique<Head,add_tail>;
};

// The set of nodes adjacent to this one: the tails of edges for which this is the head.
template <class Node, class Edges> struct adj_{using type = tuple<>;};
template <class Node, class Edges> using adjacent =
    typename adj_<Node,Edges>::type;
template <class Node, class Head, class Tail, class... Edges>
struct adj_<Node, tuple<edge<Head,Tail>, Edges...>> {
    using rest = adjacent<Node, tuple<Edges...>>;
    using type = conditional<same<Node,Head>, prepend_unique<Tail,rest>, rest>;
};
```

Dragon taming: sorting the graph

- There are many algorithms to choose from
- A recursive depth-first search is nice and easy

```
// Pseudocode for sorting a graph with a depth-first search.
tuple sorted = {};
sort(nodes)
    for (node in nodes)
        if (!node in sorted)          // ignore if already added
            sort(adjacent(node))    // recursively add all the dependencies of this node
            sorted.prepend(node) // add this node before all its dependencies
```

- Assumes no cycles in the graph
  - can be detected
  - swept under the carpet for simplicity

Dragon taming: sorting the graph

- Implementation as a template

```
template <class Edges, class Nodes = nodes<Edges>, class Sorted = tuple<>>
    struct sort_ {using type = Sorted;};

template <class Edges, class Sorted, class Node, class... Nodes>
struct sort_<Edges, tuple<Node, Nodes...>, Sorted> {
    using add_adj = typename sort_<Edges, adjacent<Node, Edges>, Sorted>::type;
    using add_node =
        conditional<contains<Node, Sorted>, Sorted, prepend<Node, add_adj>>;
    using type = typename sort_<Edges, tuple<Nodes...>, add_node>::type;
};

template <class Edges> using sort = typename sort_<Edges>::type;
```

What did modern C++ do for us?

- Generic lambdas

```
visit(sorted, [](auto & c){c.process();});
```

- Variable templates
  - can be partially specialised

```
template <class, class> constexpr bool same = false;
template <class T> constexpr bool same<T,T> = true;
```

- Alias templates
  - can't be partially specialised: wrap in a class

```
template <bool, class T1, class> struct cond_ {using type = T1;};
template <class T1, class T2> struct cond_<false,T1,T2> {using type = T2;};
```

  - can give the wrapped types a nice name

```
template <bool C, class T1, class T2> using conditional =
   typename cond_<C,T1,T2>::type;
```

Conclusions

- Generics are good...
  - Smaller code (no run-time configuration)
  - Better performance (no indirection)
  - User code is no uglier
- But...
  - More rigid (no run-time configuration)
  - More dragons need hiding from the user
    - Modern C++ has tamed the dragons somewhat