Mixins for C++?

Dr. Roland Bock

2014-09-08

Inheritance

- *Inheritance is the base class of evil*
  Sean Parent

Inheritance

We have

- Single inheritance
- Multiple inheritance
- Variadic inheritance

And we could use CRTP with all of these.

Composition

Composition seems much more limited:

- I have to know how many members I want to add in advance. There is no variadic composition.
- I have to decide on the names of those members in advance. There is no compile time way of choosing a name.
- And if I want my class to offer methods of those members, I have to write forwarding functions and need to know the name of those methods in advance.

Variadic CRTP

Example from sqlpp11

```
for (const auto& row : db(select(p.id, p.name).from(p).where(p.id > 42)))
{
  int64_t id = row.id;
  std::string name = row.name;
}
```

Variadic CRTP

The statement template

```
template<typename Db,
  typename... Policies
  >
struct statement_t:
  public expression_operators<statement_t<Db, Policies...>,
                value_type_of<detail::statement_policies_t<Db, Policies...>>>,
  public detail::statement_policies_t<Db, Policies...>::_result_methods_t,
  public Policies::template _member_t<
                detail::statement_policies_t<Db, Policies...>>...,
  public Policies::template _methods_t<
                detail::statement_policies_t<Db, Policies...>>...
{
};
```

Variadic CRTP

The statement template

```
template<typename Database>
using blank_select_t = statement_t<Database,
            select_t,
            no_select_flag_list_t,
            no_select_column_list_t,
            no_from_t,
            no_where_t,
            no_group_by_t,
            no_having_t,
            no_order_by_t,
            no_limit_t,
            no_offset_t>;

template<typename... Columns>
auto select(Columns... columns)
-> decltype(blank_select_t<void>().columns(columns...))
{
    return blank_select_t<void>().columns(columns...);
}
```

Variadic CRTP

The no_where template

```
struct no_where_t
{
  // Additional methods for the statement
  template<typename Policies>
  struct _methods_t
  {
    template<typename... Args>
    auto where(Args... args) const
    -> new_statement<Policies, no_where_t, where_t<void, Args...>>
    {
      return { static_cast<const derived_statement_t<Policies>&>(*this),
               where_data_t<void, Args...>{args...} };
    }
  };
};
```

Variadic CRTP

The no_where mixin

```
struct no_where_t
{
  // Additional methods for the statement
  mixin _methods_t
  {
    template<typename... Args>
    auto where(Args... args) const
    -> _new_statement_t<no_where_t, where_t<void, Args...>>
    {
      return { *this,
               where_data_t<void, Args...>{args...} };
    }
  };
};
```

Idea: Copy and paste the mixin's body

Variadic CRTP

The statement template

```
template<typename Db,
  typename... Policies
  >
struct statement_t:
  public Policies::template _member_t<
                        detail::statement_policies_t<Db, Policies...>>...,
{
  using _value_type = value_type_of<Policies...>;

  using mixin expression_operators<value_type>;
  using mixin result_provider<Policies>::_result_methods;
  using mixin Policies::_methods...;
};
```

How about using names as ‘parameters‘?

Examples

```
template<typename T, name x>
struct field
{
  using type = T;
  using name = x;
  using another_name = `foo`;

  T x; // A member of type T with name x
};

using my_field = field<int, `bar`>;
```

Variadic Members

The statement template

```
template<typename Db,
  typename... Policies
  >
struct statement_t:
{
  using _value_type = value_type_of<Policies...>;

  // methods
  using mixin expression_operators<value_type>;
  using mixin result_provider<Policies>::_result_methods;
  using mixin Policies::_methods...;

  // data members
  policies::member_type policies::member_name...;
};
```

Variadic Members

Example from sqlpp11

```
for (const auto& row : db(select(p.id, p.name).from(p).where(p.id > 42)))
{
  int64_t id = row.id;
  std::string name = row.name;
}
```

Variadic Members

Another annoyance

```
auto match = [product] (const auto& cand)
{
  return true
    and (cand.first.currencyId == product.currencyId or cand.first.currencyId == 0)
    and (cand.first.paymentMethodId == product.paymentMethodId or cand.first.paymentMethodId == 0)
    and (cand.first.subPaymentMethodId == product.subPaymentMethodId or cand.first.subPaymentMethodId == 0)
    and (cand.first.merchantHistoryId == product.merchantHistoryId or cand.first.merchantHistoryId == 0)
    and (cand.first.industryId == product.industryId or cand.first.industryId == 0)
    and (cand.first.countryId == product.countryId or cand.first.countryId == 0)
    ;
};
```

How about this instead?

```
auto match = [product] (const auto& cand)
{
  return true
    and equalOrZero<`currency`>(cand.first, product);
    and equalOrZero<`paymentMethodId`>(cand.first, product);
    and equalOrZero<`subPaymentMethodId`>(cand.first, product);
    and equalOrZero<`merchantHistoryId`>(cand.first, product);
    and equalOrZero<`industryId`>(cand.first, product);
    and equalOrZero<`countryId`>(cand.first, product);
    ;
};
```

Summary

Mixin

- The body of the mixin is copied to its destination
- Template parameters are applied before copying
- Overloads or name clashes are handled as if the code of the mixin had been written at its destination

Name Literals

- Name literals can be used wherever ordinary names can be used
- Name literals can have a name of their own
- Name literals can be used as template parameters
- Name literals can be used as string literals (not vice versa)