Template Magic For Beginners

Dr. Roland Bock

http://ppro.com
rbock at eudoxos dot de

https://github.com/rbock/sqlpp11

Munich, 2017-08-31

Template Meta Programming For Beginners

std::vector::reserve

```
auto v = std::vector<T>{};
...
v.reserve(n); // What's happening here?
```

std::vector::reserve

```
auto reserve(size_type n) -> void
{
    if (capacity < n)
    {
```

std::vector::reserve

```
auto reserve(size_type n) -> void
{
    if (capacity < n)
    {


        if (T is bool)
            do something
```

std::vector::reserve

```
auto reserve(size_type n) -> void
{
    if (capacity < n)
    {

        if (T is bool)
            do something

        else if (T is trivially copyable)
            do_memcpy
```

std::vector::reserve

```
auto reserve(size_type n) -> void
{
    if (capacity < n)
    {

        if (T is bool)
            do something

        else if (T is trivially copyable)
            do_memcpy

        else if (T is noexcept movable)
           move_every_item
```

std::vector::reserve

```
auto reserve(size_type n) -> void
{
    if (capacity < n)
    {

        if (T is bool)
            do something

        else if (T is trivially copyable)
            do_memcpy

        else if (T is noexcept movable)
           move_every_item

        else
           copy_every_item
    }
}
```

Template Meta Programming

Reduced problem

```
auto reserve(size_type n) -> void
{
    if (capacity < n)
    {
        if (T is bool)
            do something
        else
            do something else
    }
}
```

Partial Specialization

Let's do something about bool

```
template<typename T, typename Allocator = std::allocator<T>>
class vector;
```

Partial Specialization

Let's do something about bool

```
template<typename T, typename Allocator = std::allocator<T>>
class vector;

template <typename Allocator>
class vector<bool, Allocator>
{...};
```

Template Meta programming

Bool is out of the way now

```
auto reserve(size_type n) -> void
{
    if (capacity < n)
    {
        if (T is trivially copyable)
            do_memcopy

        else if (T is noexcept movable)
            move_every_item

        else
            copy_every_item
    }
}
```

Factor out common stuff

```
auto reserve(size_type n) -> void
{
    if (capacity < n)
    {
        auto new_memory = allocate_new_memory(n);

        if (T is trivially copyable)
            do_memcpy(new_memory);

        else if (T is noexcept movable)
            move_every_item(new_memory);

        else
            copy_every_item(new_memory);
    }
}
```

Make it a binary problem again

```
auto reserve(size_type n) -> void
{
    if (capacity < n)
    {
        auto new_memory = allocate_new_memory(n);

        if (T is trivially copyable)
            do_memcopy(new_memory);

        else
            copy_or_move_every_item(new_memory);
    }
}
```

Type Traits

From header <type_traits>

```
template <typename T, T v>
struct integral_constant
{
    static constexpr T value = v;
};
```

Type Traits

From header <type_traits>

```
template <typename T, T v>
struct integral_constant
{
    static constexpr T value = v;
};

using true_type = std::integral_constant<bool, true>;
using false_type = std::integral_constant<bool, false>;
```

Type Traits

From header <type_traits>

```
template <typename T>
struct is_trivially_copyable: public // either true_type or false_type
{
    //...
}
```

Note: Even though there actually is an embedded type which is either true_type or false_type, there is no 'is_trivially_copyable_t'. Tag dispatch as shown on the next slide relies on 'is_trivially_copyable' publicly inheriting from true_type or false_type.

Type Traits

Still in pseudo code

```
auto reserve(size_type n) -> void
{
    if (capacity < n)
    {
        auto new_memory = allocate_new_memory(n);

        if (is_trivially_copyable<T>::value)
            do_memcpy(new_memory);

        else
            copy_or_move_every_item(new_memory);
    }
}
```

Tag Dispatch

Using type traits as function call arguments

```
auto reserve(size_type n) -> void
{
    if (capacity < n)
    {
        auto new_memory = allocate_new_memory(n);

        copy_mem_or_elements(is_trivially_copyable<T>{}, begin(), end(), new_memory);
    }
}
```

Function overloads on tag types

```
template<typename It, typename Data>
auto copy_mem_or_elements(true_type, It begin, It end, Data& new_memory) -> void
{
    // Do memcpy
}
```

Tag Dispatch

Function overloads on tag types

```
template<typename It, typename Data>
auto copy_mem_or_elements(true_type, It begin, It end, Data& new_memory) -> void
{
    // Do memcpy
}

template<typename It, typename Data>
auto copy_mem_or_elements(false_type, It begin, It end, Data& new_memory) -> void
{
    copy_or_move(begin, end, new_memory);
}
```

SFINAE

SFINAE

From header <type_traits>

```
template <bool B, typename T = void>
struct enable_if
{
};
```

SFINAE

From header <type_traits>

```
template <bool B, typename T = void>
struct enable_if
{
};

template <typename T>
struct enable_if<true, T>
{
    using type = T;
};
```

SFINAE

From header <type_traits>

```
template <bool B, typename T = void>
struct enable_if
{
};

template <typename T>
struct enable_if<true, T>
{
    using type = T;
};

template <bool B, typename T = void>
using enable_if_t = typename enable_if<B, T>::type;
```

SFINAE

Turning overloads on/off with enable_if

```
template <typename Iterator, typename Data>
auto copy_or_move(Iterator begin, Iterator end, Data& new_memory) -> void
{ /* do copy */ }

template <typename T, typename Data,
          typename = enable_if_t<is_nothrow_move_constructible<T>::value>>
auto copy_or_move(wrap_iter<T*> begin, wrap_iter<T*> end, Data& new_memory) -> void
{ /* do move */ }
```

Note: Gah! I tricked myself into telling you utter nonsense here. Please ignore what I said about why the second overload is chosen in case the type is noexcept move-constructible. I tried to shortcut the technique used in libc++ and failed. The full technique from the library uses a two-stage combination of SFINAE and is beyond the level of this talk. The "beginners" version is shown on the next (added) slide.

Turning overloads on/off with enable_if

```
template <typename Iterator>
using value_type = typename Iterator::value_type;

template <typename Iterator, typename Data>
auto copy_or_move(Iterator begin, Iterator end, Data& new_memory)
     -> enable_if_t< ! is_nothrow_move_constructible<value_type<Iterator>>::value>
{ /* do copy */ }

template <typename Iterator, typename Data>
auto copy_or_move(Iterator begin, Iterator end, Data& new_memory)
     -> enable_if_t<is_nothrow_move_constructible<value_type<Iterator>>::value>
{ /* do move */ }
```

Note: This is what I probably should have shown: Assuming a template alias ‘value_type‘ for the value type of the iterator and depending on whether the that value type is no-except move-constructible or not, one version is fine, while the other is dropped because the ‘enable_if_t‘ specialization produces a substitution failure.

Tag Dispatch and SFINAE

That's tough. *No kidding, see errata on previous slides.*

Tag Dispatch and SFINAE

That's tough. *No kidding, see errata on previous slides.*

However. . .

We started with this pseudo code

```
auto reserve(size_type n) -> void
{
    if (capacity < n)
    {
        auto new_memory = allocate_new_memory(n);

        if (T is trivially copyable)
            do_memcpy(new_memory);

        else if (T is noexcept movable)
            move_every_item(new_memory);

        else
            copy_every_item(new_memory);
    }
}
```

if constexpr makes the pseudo code become reality

```
auto reserve(size_type n) -> void
{
    if (capacity < n)
    {
        auto new_memory = allocate_new_memory(n);

        if constexpr (is_trivially_copyable_v<T>)
            do_memcpy(new_memory);

        else if constexpr (is_nothrow_move_constructible_v<T>)
            move_every_item(new_memory);

        else
            copy_every_item(new_memory);
    }
}
```

C++17 rocks!

Also look at `void_t`.

CRTP

How do you make this work?

```
#include <memory>

struct Foo;

void do_something(std::shared_ptr<Foo> f);

struct Foo
{
    auto do_it()
    {
        do_something(???);
    }
};

int main() {
    auto foo = std::make_shared<Foo>();
    foo->do_it();
}
```

std::enable_shared_from_this

```
#include <memory>

struct Foo;

void do_something(std::shared_ptr<Foo> f);

struct Foo : public std::enable_shared_from_this<Foo>
{
    auto do_it()
    {
        do_something(shared_from_this());
    }
};

int main() {
    auto foo = std::make_shared<Foo>();
    foo->do_it();
}
```

CRTP

std::enable_shared_from_this

```
template <typename _Tp>
class enable_shared_from_this
{
    mutable weak_ptr<_Tp> __weak_this_;
```

CRTP

std::enable_shared_from_this

```
template <typename _Tp>
class enable_shared_from_this
{
    mutable weak_ptr<_Tp> __weak_this_;

protected:
    constexpr enable_shared_from_this() noexcept {}
    enable_shared_from_this(enable_shared_from_this const&) noexcept {}
    enable_shared_from_this& operator=(enable_shared_from_this const&) noexcept{ return *this; };
```

std::enable_shared_from_this

```
template <typename _Tp>
class enable_shared_from_this
{
    mutable weak_ptr<_Tp> __weak_this_;

protected:
    constexpr enable_shared_from_this() noexcept {}
    enable_shared_from_this(enable_shared_from_this const&) noexcept {}
    enable_shared_from_this& operator=(enable_shared_from_this const&) noexcept{ return *this; };

public:
    shared_ptr<_Tp> shared_from_this()
        {return shared_ptr<_Tp>(__weak_this_);}

    shared_ptr<_Tp const> shared_from_this() const
        {return shared_ptr<const _Tp>(__weak_this_);}
```

std::enable_shared_from_this

```
template <typename _Tp>
class enable_shared_from_this
{
    mutable weak_ptr<_Tp> __weak_this_;

protected:
    constexpr enable_shared_from_this() noexcept {}
    enable_shared_from_this(enable_shared_from_this const&) noexcept {}
    enable_shared_from_this& operator=(enable_shared_from_this const&) noexcept{ return *this; };

public:
    shared_ptr<_Tp> shared_from_this()
        {return shared_ptr<_Tp>(__weak_this_);}

    shared_ptr<_Tp const> shared_from_this() const
        {return shared_ptr<const _Tp>(__weak_this_);}

    template <typename _Up> friend class shared_ptr;
};
```

Deferred instantiation

CRTP

Deferred instantiation

```
template <typename T>
struct Base
{
    auto foo() -> int
    {
        return static_cast<T*>(this)->fooImpl();
    }
};
```

Deferred instantiation

```
template <typename T>
struct Base
{
    auto foo() -> int
    {
        return static_cast<T*>(this)->fooImpl();
    }
};

struct Derived : Base<Derived>
{
    auto fooImpl() -> int;
};
```

Derived is incomplete when Base is instantiated

```
template<typename T>
struct Base
{
  auto foo() -> typename T::fooType
  {
    return static_cast<T*>(this)->fooImpl();
  }
};
```

CRTP

Derived is incomplete when Base is instantiated

```
template<typename T>
struct Base
{
  auto foo() -> typename T::fooType
  {
    return static_cast<T*>(this)->fooImpl();
  }
};

struct Derived : public Base<Derived>
{
  using fooType = int;

  auto fooImpl() -> fooType
  {
    return 42;
  }
};
```

Deferred instantiation

```
template<typename T>
struct Base
{
  template<typename X = T>
  auto foo() -> typename X::fooType
  {
    return static_cast<T*>(this)->fooImpl();
  }
};

struct Derived : public Base<Derived>
{
  using fooType = int;

  auto fooImpl() -> fooType
  {
    return 42;
  }
};
```

Let's see look at a combination.

CRTP + Variadic Templates

sqlpp11: SQL expressions in C++

```
for (const auto& row : db.run(
                        select(foo.name, foo.hasFun)
                        .from(foo)
                        .where(foo.id > 17 and foo.name.like("%bar%"))))
{
  std::cout << row.name << ' ' << row.hasFun << '\n';
}
```

CRTP + Variadic Templates

The statement

```
template <typename... Clauses>
struct statement_t : public Clauses::template _base_t<statement_t<Clauses...>>...
{
    //...
};
```

Tag dispatch

The FROM clause

```
template <typename Statement>
struct _base_t
{
  template <typename Table>
  auto from(Table table) const -> _new_statement_t<check_from_t<Table>, from_t<from_table_t<Table>>>
  {
    return _from_impl(check_from_t<Table>{}, table);
  }
```

Tag dispatch

The FROM clause

```
template <typename Statement>
struct _base_t
{
  template <typename Table>
  auto from(Table table) const -> _new_statement_t<check_from_t<Table>, from_t<from_table_t<Table>>>
  {
    return _from_impl(check_from_t<Table>{}, table);
  }

private:
  template <typename Check, typename Table>
    auto _from_impl(Check, Table table) const -> inconsistent<Check>
  template <typename Table>
  auto _from_impl(consistent_t, Table table) const
      -> _new_statement_t<consistent_t, from_t<Database, from_table_t<Table>>>;
};
```

Template Magic?

No

Template Magic?

No

Template Perseverance

We looked at

- Partial specialization

We looked at

- Partial specialization
- Type traits

We looked at

- Partial specialization
- Type traits
- Tag dispatch

We looked at

- Partial specialization
- Type traits
- Tag dispatch
- SFINAE

We looked at

- Partial specialization
- Type traits
- Tag dispatch
- SFINAE
- CRTP

Template Meta Programming Basics

We looked at

- Partial specialization
- Type traits
- Tag dispatch
- SFINAE
- CRTP
- Deferred instantiation

Use it, play with it.

Use it, play with it.

It becomes easier over time and with every new standard.

Thank you!