Template Magic For Beginners

Dr. Roland Bock

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

https://github.com/rbock/sqlpp11

Dresden, 2017-08-10

Template Magic

What is it?

Template Magic

What is it?

It is not magic.

Template Meta Programming

What is it?

Template Meta Programming

What is it? Wikipedia:

Template metaprogramming (TMP) is a metaprogramming technique in which templates are used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled.

Template Meta Programming

What is it? Wikipedia:
Template metaprogramming (TMP) is a metaprogramming technique in which templates are used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled.
The output of these templates include compile-time constants, data structures, and complete functions.

Template Meta Programming

What is it? Wikipedia:
Template metaprogramming (TMP) is a metaprogramming technique in which templates are used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled.
The output of these templates include compile-time constants, data structures, and complete functions.

Let's take a look.

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)
    {
        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 something else
```

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 something else

        else if (T is noexcept movable)
            do yet another thing
```

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 something else

        else if (T is noexcept movable)
            do yet another thing

        else
            oh, come on!
    }
}
```

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 something else

        else if (T is noexcept movable)
            do yet another thing

        else
            oh, come on!
    }
}
```

Can’t do all of this at runtime.

std::vector::reserve

We'll try to solve this with TMP

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

        else if (T is trivially copyable)
            do something else

        else if (T is noexcept movable)
            do yet another thing

        else
            oh, come on!
    }
}
```

std::vector::reserve

We'll try to solve this with TMP

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

        else if (T is trivially copyable)
            do something else

        else if (T is noexcept movable)
            do yet another thing

        else
            oh, come on!
    }
}
```

First, we will reduce the problem.

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

Partial Specialization

Let's do something about bool

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

template<class 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 something

        else if (T is noexcept movable)
            do another thing

        else
            oh, come on!
    }
}
```

Template Meta Programming

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);
    }
}
```

Template Meta Programming

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_memmove(new_memory);

        else
            copy_or_move_every_item(new_memory);
    }
}
```

How can we evaluate the condition?

Type Traits

From header <type_traits>

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

Type Traits

From header <type_traits>

```
template< class 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 <class T>
struct is_trivially_copyable
{
    using type = ...; // Either true_type or false_type
}
```

Type Traits

From header <type_traits>

```
template <class T>
struct is_trivially_copyable
{
    using type = ...; // Either true_type or false_type
}

template <class T>
using is_trivially_copyable_t = typename is_trivially_copyable<T>::type;
```

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);

        move_mem_or_elements(is_trivially_copyable_t<T>{}, begin(), end(), new_memory);
    }
}
```

Tag Dispatch

Function overloads on tag types

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

Tag Dispatch

Function overloads on tag types

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

template<typename It, typename Data>
auto move_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, class T = void>
struct enable_if
{
};
```

SFINAE

From header <type_traits>

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

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

SFINAE

From header <type_traits>

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

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

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

SFINAE

Turning overloads on/off with enable_if

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;
```

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;

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;
```

Tag Dispatch and SFINAE

That’s tough.

Tag Dispatch and SFINAE

That's tough.

However. . .

C++17

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);
    }
}
```

C++17

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

C++17 rocks!

Also look at `void_t`.

CRTP

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();
}
```

CRTP

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

CRTP

std::enable_shared_from_this

```
template<class _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; };
```

CRTP

std::enable_shared_from_this

```
template<class _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_);}
```

CRTP

std::enable_shared_from_this

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

sqlpp::statement_t

Let mix in variadics. . .

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%"))))
{
  if (row.name.is_null())
    std::cerr << "name will convert to empty string" << std::endl;
  std::string name = row.name;

  bool hasFun = row.hasFun;

}
```

CRTP + Variadic Templates

The statement

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

CRTP + Variadic Templates

The statement

```
struct no_from_t
{
  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);
    }
```

CRTP + Variadic Templates

The statement

```
struct no_from_t
{
  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

Thank you!