C++ Template Metaprogramming
Practical Approach

code::dive 2016

Szymon Gutaj

In brief

- Introduction
- Metaprogramming techniques
- Questions / Discussion

Introduction

(My Humble) Definition

Shifting complexity from the user/client side
to the library/compiler side

Augmenting the compiler

Aims

- **Correctness (reducing human factor)**
  - By generating code according to (tested) rules
  - By introspecting and checking code at compile time
  - By creating libraries smarter than their user
  - By avoiding repetition

Aims

- **Performance (self-optimisation)**
  - By performing things at compile time
  - By generating fine-tuned code in particular situation
  - By performing optimisations cumbersome to handcraft

Aims

- **Portability (self-adaptability)**
  - By automatically adapting to current hardware
  - By intelligently picking optimal code blocks
  - By keeping all code in the C++ realm

Aims

- **Expressive interfaces**
  - By automatically deducing details for the user
  - By creating domain-specific languages

Consequences (wink)

- You'll keep your job forever, having written metaprogrammed code (*if your colleagues like you*)
- You'll be happy to have trained karate for 15 years (*if they don't like you*)
- Nothing will happen (*are you sure, that anyone has seen your commit?*)

Consequences

- Development time of your (metaprogrammed) library depends on skill and tools used
- Maintenance of your library requires some skill
- Your user code becomes higher level
- Your user code becomes smaller and more declarative
- Changes to your library (bugfixes and new features) propagate to all places of usage

Approaches

- Handcrafted
- With aid of C++ Standard Library (type_traits)
- With aid of Boost (Boost Metaprogramming Library, Boost Hana)
- With aid of higher level, domain-specific libraries

How it works

- C++ Metaprogramming is purely functional
- Mutation is replaced with copying (with changes applied)
- Iteration is replaced with recursion
- Computational complexity of a metaprogram is measured with the amount of template instantiations (copies)

Golden rules

- Keep your code orthogonal
- Keep your conventions consistent
- Solve things by adding one more level of indirection

Handcrafted metaprogramming

Basics and conventions

Disclaimer

Our examples are meant to be fun, easy and demonstrational

Hello world!

```
// This is our metaprogram (actually quite useful!)
template<typename T>
struct return_as_is
{
    typedef T type;
};

// This is our processed data
class HelloWorld
{
};

// What is the type of ‘helloWorld’? Yes, it’s HelloWorld!!!
return_as_is<HelloWorld>::type helloWorld;
```

Operating on types

- Template name is the operation name
- Template parameters are the arguments
- Nested definition named “type” is the result
- The “struct” keyword is used to keep the syntax clean
- We've just described a metafunction!

Operating on types

```
// Adds a pointer to given type
template<typename T>
struct add_pointer
{
    typedef T * type;
};

// 'pointerToInt' is of type 'int *'
add_pointer<int>::type pointerToInt {nullptr};
```

Operating on values

- Template name is the operation name
- Template parameters are the arguments
- Nested member named “value” is the result
- Nested member is static const / constepxr

Operating on values

```cpp
// Increments given value by one
template<int V>
struct increment
{
    static constexpr int value = V + 1;
};

// Variable ‘four’ equals 4
// ‘3 + 1’ happened during compilation
int four = increment<3>::value;
```

Operating on values

```cpp
// Wraps an integer in a type
template<
    typename T,
    T Value
>
struct integral_constant
{
    typedef integral_constant type;

    static constexpr T value = Value;
};

// Dressing values as types can be useful
typedef integral_constant<int, 1> one;
typedef integral_constant<int, 2> two;
```

Simple processing

- Pattern matching is used for “flow control”
- Patterns are defined with partial template spacialisations

Simple processing

```
template< // Primary template: “true” case
    bool Condition,
    typename TrueType,
    typename FalseType
>
struct if_
{
    typedef TrueType type; // TrueType is the result
};

template< // Partially specialised: “false” case
    typename TrueType,
    typename FalseType
>
struct if_<false, TrueType, FalseType>
{
    typedef FalseType type; // FalseType is the result
};
```

Simple processing

```
#include <cstdint> // For INT8_MAX, int8_t and int_least16_t

#include "wild_wild_world/constants.h" // SOME_CONSTANT comes from here

typedef if_<
    (SOME_CONSTANT < INT8_MAX),
    int8_t,
    int_least16_t
>::type smallest_type;

// 'myConstant' has the smallest type, that fits the value
smallest_type myConstant = SOME_CONSTANT;
```

Simple processing

```
#include "configuration.h" // For WANT_SPEED
#include "algorithm_speedy.h" // For algorithm_speedy
#include "algorithm_tiny.h" // For algorithm_tiny

typedef if_<
    WANT_SPEED,
    algorithm_speedy,
    algorithm_tiny
>::type algorithm;

algorithm a;

a.run(); // Runs the selected algorithm
```

Simple recursion

- Pattern matching with a “terminal case” allow for recursion
- Patterns are defined with partial template spacialisations

Simple recursion

```cpp
template<typename T>
struct rank // Terminal case: T is not a table anymore
{
    static constexpr int value = 0;
};
template<typename T, int N>
struct rank<T[N]> // Case: T is a table
{
    static constexpr int value = rank<T>::value + 1;
};
template<typename T>
struct rank<T[]> // Case: T is a table (of unknown size)
{
    static constexpr int value = rank<T>::value + 1;
};
auto rank0 = rank<std::string>::value; // Equals 0
auto rank1 = rank<int[2][4]>::value;   // Equals 2
auto rank2 = rank<int[4][3][]>::value; // Equals 3
```

Metafunction forwarding

- Inheritance is a neat way to compose metaprograms
- Code and conventions are automatically propagated

Metafunction forwarding

```
// ‘false_type’ is an alias for ‘bool’ equal ‘false’
struct false_type :
    integral_constant<
        bool,
        false,
    > {};

// ‘true_type’ is an alias for ‘bool’ equal ‘true’
struct true_type :
    integral_constant<
        bool,
        true,
    > {};
```

Metafunction forwarding

```cpp
template<typename T>
struct is_byte_sized :
    if_<
        (sizeof(T) == 1),
        true_type,
        false_type
    > {};

bool byteSized = is_byte_sized<char>::value; // true

bool notByteSized = is_byte_sized<int>::value; // false
```

Lazy evaluation

- Lazy evaluation gives more flexibility
- Invalid, but unneeded branches of code still compile
- Lazy evaluation leads to faster code
- Stay lazy, whenever you can

Lazy evaluation

```
// Eager!!!
template<typename T>
struct my_favourite :
    if_<
        is_good_enough<T>::value,
        take_it<T>::type, // 'take_it' always evaluated
        fix_it<T>::type // 'fix_it' always evaluated
    > {};

// Lazy...
template<typename T>
struct my_favourite :
    if_<
        is_good_enough<T>::value,
        take_it<T>, // 'take_it' evaluated for ‘true’
        fix_it<T> // 'fix_it' evaluated for ‘false’
    >::type {};
```

Toolset

- **Compile-time operators are the low-level means for code introspection**
  - sizeof, decltype, alignof, ternary operator, template template parameters, …
- **Language rules are your worst enemy and best friend**
  - resolution/conversion rules, SFINAE, variadic templates, …

Toolset

```cpp
struct NeighbourRomek // A well behaved guy
{
    void howdy();
};

struct ManInBlack // Well, who knows...
{
};

// Let's check contents of some types
bool doesnt = has_howdy<int>::value; // false

bool does = has_howdy<NeighbourRomek>::value; // true

bool afraidToCheck = has_howdy<ManInBlack>::value; // ?!
```

Toolset

```
template<typename T, void(T::*)() = &T::howdy>
char hasHowdy(int); // Helper: SFINAE sensitive

template<typename T>
long hasHowdy(...); // Helper: less sensitive

// Returns 'true' if 'T' has a member function 'howdy'
template<typename T>
struct has_howdy :
    integral_constant<
        bool,
        sizeof(char) == sizeof(hasHowdy<T>(0))
    > {};
```

Toolset

```
template<typename T> // Helper: matches a pointer
true_type hasHowdy(decltype(&T::howdy));

template<typename T>
false_type hasHowdy(...); // Helper: matches everything

// Returns 'true' if 'T' has a member function 'howdy'
template<typename T>
struct has_howdy :
    decltype(hasHowdy<T>(nullptr))
{};
```

Homework

- Metafunction classes (first step to passing metafunctions around)
- Higher order metafunctions (composed metafunctions)
- Tag based dispatch (adding a second level of abstraction)
- Variadic template parameter pack access (very useful)

Asertive code

- Assertive code makes error messages better
- Assertive code prevents abuse of your metaprogram
- Use “static_assert”
- Use “std::is_same” or alike
- Wait for a standardised C++ feature: concepts!

Testing

- Can be done in a regular cpp file
- Compile that file with your code for early testing

Testing

```
#include <type_traits>
#include "pass_optimally.h" // Code under test

static_assert( // Test case 1
    std::is_same< // More on ‘std::is_same’ in a minute
        pass_optimally<VeryBigClass>::type,
        VeryBigClass const &
    >::value,
    "Expected passing large arguments by reference"
);
static_assert( // Test case 2
    std::is_same<
        pass_optimally<int>::type,
        int
    >::value,
    "Expected passing small arguments by value"
);
```

Asertive code

```
template<class Agent, class Handler, class ...Args>
void Base::call(Handler Agent::*handler, Args&&... args)
{
    static_assert( // Check if calculated ‘handler’ and ‘args’ match
        std::is_constructible< // More on this in a second
            void(Agent::*)(Args...),
            decltype(handler)
        >::value,
        "Calling with bad arguments."
        "Args need to match the signature of Handler"
    );
    call(std::bind( // If assertion passed, make the call
        handler,
        static_cast<Agent * const>(this),
        std::forward<Args>(args)...)
    );
}
```

Standard Template Library

type_traits

Features

- **Basic type instrospection:**
  - is_integral, is_pointer, is_function, is_class, ...
  - is_same, is_base_of, is_move_constructible, is_convertible, has_virtual_destructor, ...
- **Basic type processing:**
  - add_cv, remove_reference, make_signed, ...
- **Basic metaprogramming facilities:**
  - conditional, integral_constant, enable_if, result_of, ...

Example – type_traits

```
#include <type_traits>
// Return 'as is' for unsigned types
template<typename T>
typename std::enable_if<
     std::is_unsigned<T>::value, // The condition
     T                           // The return type
>::type alwaysPositive(T t) { return t; }

// Return absolute and unsigned
template<typename T>
typename std::enable_if<
     !std::is_unsigned<T>::value, // The condition
     typename std::make_unsigned<T>::type // The return type
>::type alwaysPositive(T t)
{
   return (t < 0) ? -t : t;
}
```

Boost Metaprogramming Library

Boost MPL

Features

- **Mimicks C++ STL features at compile time**
- **Containers and views:**
  - vector, list, map, set, string, ...
- **Algorithms:**
  - fold, transform, sort, copy, find, count, partition, ...
- **Functional processing (including higher order):**
  - lambda, quote, bind, apply, if_, ...
- **Basic metafunctions:**
  - and_, or_, not_, bitand_, less, min, ...
- **Basic runtime interfacing:**
  - for_each, c_str, ...

Example – Boost MPL

```
tuple<int, char const*, bool> myTuple; // Poor man's tuple
get<int>(myTuple) = 123; // Access elements by type
get<char const*>(myTuple) = "hello";

// (Implementation) Holds a tuple element of type T
template<typename T>
struct tuple_field
{
    T field;
};

// (Implementation) Accesses a tuple element by type
template<typename T>
T & get(tuple_field<T> & t)
{
    return t.field;
}
```

Example – Boost MPL

```cpp
#include <boost/mpl/empty_base.hpp>
#include <boost/mpl/fold.hpp>
#include <boost/mpl/inherit.hpp>
#include <boost/mpl/placeholders.hpp>
#include <boost/mpl/vector.hpp>

using namespace boost;
using namespace boost::mpl::placeholders;

// Composes a tuple, inheriting tuple_field's
template<typename ...Fields>
struct tuple :
    mpl::fold< // The algoritm: fold/accumulate
        mpl::vector<Fields...>,                // The data
        mpl::empty_base,                       // The initial value
        mpl::inherit<_1, tuple_field<_2>> // The operation
    >::type {};
```

Boost Hana

Features

- Can do compile-time computation (like Boost MPL)
- Can do runtime computation (like STL)
- Can do heterogeneous computation (like Boost Fusion)
- Requires a (decent) C++14 compiler

Example – Boost Hana

```cpp
#include <boost/hana.hpp>
using namespace boost;
struct Fish { std::string name; };
struct Cat  { std::string name; };
struct Dog  { std::string name; };

auto animals = hana::make_tuple(
    Fish{"Nemo"}, Cat{"Garfield"}, Dog{"Snoopy"}
);
// Applies a (heterogenous) algorithm in runtime
auto names = hana::transform(animals, [](auto a) {
    return a.name;
});

assert(hana::reverse(names) ==
    hana::make_tuple("Snoopy", "Garfield", "Nemo")
);
```

Example – Boost Hana

```cpp
auto animal_types = hana::make_tuple(
    hana::type_c<Fish*>,
    hana::type_c<Cat&>,
    hana::type_c<Dog*>
);
// Applies an algorithm in compile time
auto animal_ptrs = hana::filter(animal_types, [](auto a) {
     return hana::traits::is_pointer(a);
});

static_assert(animal_ptrs == hana::make_tuple(
     hana::type_c<Fish*>, hana::type_c<Dog*>), ""
);
```

Thank you!

Questions?