Fermilab U.S. DEPARTMENT OF ENERGY Office of Science

Summary of “Template Metaprogramming”

Type Traits (part 1 of 2)

Marc Paterno

4 November 2020

Section 1

My summary

What is metaprogramming?

- Jody's definition is programming in which your data is types.
- Other languages have powerful metaprogramming facilities:
  - Lisp macros allow one to manipulate all code, not just data (Lisp code *is* data).
  - Ruby metaprogramming allows one to modify a program at runtime, *e.g.* to inject new methods into a class, or into an object.
- C++ metaprogramming is a compile-time thing.

Metafunctions

- A *metafunction* is a function evaluated at compile time.
  - implemented as a *class* or *struct*.
- To return a value from a normal function, use `return`.
- To return a type from a metafunction, use a nested type.
  - note: the data (value) is a *type*.
- To return a value from a metafunction, use a nested value.
- Use is not enforced by the language, but rather conventions.
  - use the conventions!
- Constrained templates in C++20 will help regularize this.

Return *types* and return *values*: public members

```
template <typename T>
struct TheAnswer {
  static constexpr int value = 42;
};
template <typename T> using sum_v = sum<T>::value;
int answer = TheAnswer<double>::value;  // usage
```

```
template <typename T>
struct add_pointer {
  using underlying_type = std::remove_reference<T>::type;
  using type = underlying_type*;
};
template <typename T> using add_pointer_t = typename add_pointer<T>::type;
add_pointer<double const> p = nullptr;   // usage
```

Always compile-time

```
template <typename T, T X, T Y>
struct sum {
  static constexpr T value = X + Y;
};

int main() {
  return sum<int, 2, 3>::value;
}
```

- Value is determined at compile-time, regardless of optimization level:
  - https://godbolt.org/z/xsq4v8

Aside: in C++20, more non-type template parameter types

- Current C++ allows:
  (https://en.cppreference.com/w/cpp/language/template_parameters)
  - lvalue reference type (to object or to function)
  - an integral type
  - a pointer type (to object or to function)
  - a pointer to member type (to member object or to member function)
  - an enumeration type
  - `std::nullptr_t`
- C++20 will allow
  - floating-point types
  - some *literal classes*
  - example: https://godbolt.org/z/E5x8eM

Don’t mistake pedagogical examples for production-ready code

- Use the simplest tool that gets the job done efficiently.

```
// Prefer constexpr function to metafunction.
// Only requires C++17, not C++20.
template <typename T> T constexpr sum(T x, T y) { return x+y; };

int main() {
    return sum(2.5, 3.5);
}
```

Why in the world is `std::type_identity` a thing (C++20)?

```
// Part of namespace std in C++20.
template <typename T> struct type_identity { using type = T; };

template <typename T> void f(T a, T b) { };
template <typename T> void g(T a, typename type_identity<T>::type b) { };

f(2.5, 0); // fails to compile, unable to deduce unique T
g(2.5, 0); // OK, deduces T = double, and converts 0 to 0.0
```

- Call to `f` fails because template type deduction happens at an earlier translation phase than does argument conversion.

How to think about dependent names

- In `thing<T>::x`, `x` could name several thing: a value, a function, a template, an enumeration, a type...

```
T::x * p;           // multiply the value T::x by the value p?
typename T::x * p; // p is a pointer to an object of type T::x
typename T::x*  p; // and this is nicer formatting
```

- You need to write `typename` to force the compiler to know that the nested name is a type.
- Dependent names are involved with *two phase name lookup*. That is its own complicated issue.

The most useful metafunctions?

```
template <class T, T v>
struct integral_constant {
  static constexpr T value = v;
  using value_type = T;
  using type       = integral_constant; // simplifying Jody's version
  // implicit conversion of integral_constant to T.
  constexpr operator value_type() const noexcept { return value; }
  // function call operator to return the value v.
  constexpr value_type operator()() const noexcept { return value; }
};
```

```
template <bool b> using bool_constant = integral_constant<bool, b>
using true_type = bool_constant<true>;
using false_type = bool_constant<false>;
```

Using `std::integral_constant`

- Goal: write a function template that yields a *compile-time integer-counter for loop*, which the compiler is guaranteed to unroll. (Adapted from Artificial Mind.)

```
// Start, End, Inc are all compile-time constant integral types
for (auto i = Start; i < End; i += Inc) f(i);
```

- We need to make sure `i` is a compile-time constant, and to invoke `f` on each value.

```
template <auto Start, auto End, auto Inc, class F>
constexpr void constexpr_for(F&& f) {
  if constexpr (Start < End) {
      f(std::integral_constant<decltype(Start), Start>());
      constexpr_for<Start + Inc, End, Inc>(f);
    }
}
```

- See https://godbolt.org/z/1WW3z7.

Unary and binary template metafunctions in standard library

- Unary -> one template type argument
- Binary -> two template type arguments
- Default and copy constructable
- Publicly and unambiguously specialization from integral_constant
- Always have a type member named type

Undefined behavior

- Do not specialize standard type traits
- Be very careful with incomplete types

Sample metafunction

- `std::is_void_v`: is a value of type `bool_constant<b>`

```
// General case: applies when no matching explicit specialization is found.
template <typename T> struct is_void       : std::false_type {};
// Explicit specialization (note empty angle brackets!)
template <> struct is_void<void> : std::true_type  {};
template <> struct is_void<void const> : std::true_type {}
template <> struct is_void<void const volatile> : std::true_type {}
```

- Usage:

```
// Note the braces, to create an object of type is_void<int>.
// Rely on implicit conversion to bool.
static_assert(not std::is_void<int> {});
```

`remove_const` and top-level qualifiers

- Transformation traits to remove qualifiers (`const`, `volatile`) remove only *top-level* qualifiers.

```
remove_const<int>         -> int
remove_const<int const>   -> int
remove_const<int*>        -> int*
remove_const<int const*>  -> int const*
remove_const<int* const>  -> int*
```

- This demonstrates why I prefer *east const*.

Partial specialization *vs.* full specialization

- A full specialization contains no more template parameters.
- `template <>` are still required to tell the compiler you are specializing a template.

```
template <> struct is_void<void> : std::true_type {};
```

- A partial specialization contains one or more template parameters.

```
template <typename T> remove_const<T const> : type_identity<T> {};
```

- It can have even more template parameters than the base template.

```
template <typename T> struct function;
template <typename R, typename... A> struct function<R (A...)> { ... };
```

- The compiler will choose the *unambiguous most specialized match* for an instantiation.
  - If there is ambiguity, a compilation error occurs.

`std::conditional`: a beautiful usage from Boost `static_string`

```
// Find the smallest unsigned integral type that can hold a value as large
// as N
using smallest_width =
  typename conditional<(N <= (numeric_limits<unsigned char>::max)()),
                       unsigned char,
  typename conditional<(N <= (numeric_limits<unsigned short>::max)()),
                       unsigned short,
  typename conditional<(N <= (numeric_limits<unsigned int>::max)()),
                       unsigned int,
  typename conditional<(N <= (numeric_limits<unsigned long>::max)()),
                       unsigned long,
  typename conditional<(N <= (numeric_limits<unsigned long long>::max)()),
                       unsigned long long,
  size_t>::type>::type>::type>::type>::type;
```

What standard type traits must be implemented through compiler intrinsics?

- I'm willing to trust Arthur O'Dwyer's list.
- There are 15 of them (quite a few more than Jody guessed).
- See https://github.com/Quuxplusone/from-scratch/blob/master/include/scratch/bits/type-traits/compiler-magic.md.