Introduction to template metaprogramming

Hans Dembinski, MPIK Heidelberg

CORSIKA Workshop, Karlsruhe, Germany (17-20 June 2019)

Technical note

This is a HTML talk, you can click on any blue text, it is a link.

About me

- Cosmic ray/HEP physicist now in LHCb
- Trying to solve the Muon Puzzle in air showers
- Active in the Boost C++ and Scikit-HEP Python communities
- My OSS projects
  - Boost::Histogram
  - iminuit (maintainer)
  - pyhepmc

Take-home message

- Template metaprogramming (TMP) is about computing with types
  - Not as scary as it sounds
  - Useful for library writers
  - Used to look like arcane magic in pre-C++11, getting better
- Uses
  - Avoid code repetition (DRY = Don’t Repeat Yourself)
  - Static function/method dispatch
  - Mixins (CRTP = Curriously-Reoccuring-Template-Pattern)
  - Compile-time arithmetic calculations (but use `constexpr`)

C++ is strongly-typed language

- Strongly-typed? 🤔
  - Every variable has distinct type
  - Functions/methods/operators only work on matching types
  - Detect errors at compile-time rather than run-time 😎

```
int add(int a, int b) {
  return a + b;
}

int main() {
  // this is an error, because int add(int, const char*) is not defined
  int x = add(1, "1");
}
```

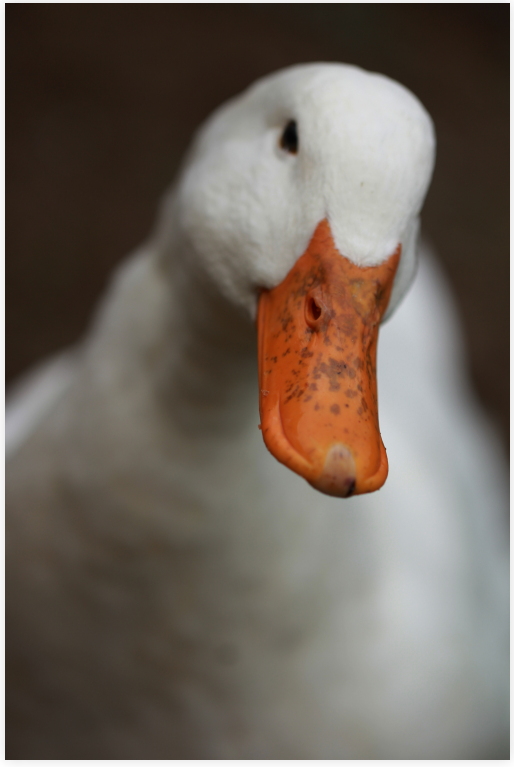
- Some implicit type conversions allowed
  - `float` → `double` 😃, but also `int` → `char` and `int` → `unsigned` 😱
  - ⚠️ Question: What does `static_cast<unsigned>(-1)` do?

- … but type-safety can lead to code repetition
- Sometimes same code, except for the types

```
// urgh, have to write two almost identical functions for `int` and `double`

int add(int a, int b) {
  return a + b;
}

double add(double a, double b) {
  return a + b;
}

int main() {
  int x = add(1, 1);
  double y = add(1.0, 1.0);
}
```

Duck-typing

- Templates originally designed to avoid writing repeated code



```
// cookie-cutter implementation that
  generates
// concrete function (code may be inlined) on
// first instantiation with concrete type
template <class T>
T add(T a, T b) { return a + b; }

using str = std::string

int main() {
  // generates int add(int, int)
  auto x = add(1, 1); // 2
  // generates double add(double, double)
  auto y = add(1.0, 1.0); // 2.0
  // generates str add(str, str)
  auto z = add(str("Hello"), str("World"));
}
```

```cpp
#include <memory>
#include <algorithm>

// classes can be templated as well
template <class T>
struct container {
  container(std::size_t s, T init_value) : size_(s), ptr(new T[size_]) {
    std::fill(begin(), end(), init_value);
  }
  T* begin() { return ptr.get(); }
  T* end() { return ptr.get() + size_; }
  std::size_t size() const { return size_; }
  std::unique_ptr<T[]> ptr;
};

int main() { container<int> c(10, 42); }
```

- Made generic containers possible (same implementation)
  - `std::vector<int>`, `std::vector<float>`, …
  - Can’t write all containers by hand and user types not known
- Beware: `container c(10, 42);` doesn’t work out of the box
  - Compiler does not infer type of *class* from *constructor*
  - In C++17, one can provide *deduction guide* to make it work

Some rules for templated classes/functions

- No code generated when not *instantiated* with concrete type
- Argument types inferred for functions/methods/ctors
- Can specialize template instantiation for particular type

```
template <class T>
T add(T a, T b) { return a + b; }

using str = std::string;

// specialization for std::string
str add(str a, str b) { return a + " " + b; }

int main() {
  auto x = add(1, 1); // 2
  auto y = add(str("Hello"), str("World")); // "Hello World"

  // beware: compiler cannot infer types when situation is ambiguous
  auto z = add(1, 1.0); // ambiguous, should z be int or double?
}
```

```
// specialization for class

template <class T>
struct container { /* generic implementation */ };

template <>
struct container<char> {
  // implementation for a string-like container

  const char* c_str() const { /* ... */}

  // ...
};
```

- Class specialization is distinct new type
  - Interface and implementation completely independent of generic container
  - Use common *base* class to share common interface/implementation

Options to resolve overload ambiguities

- Solution A: cast types at call-site to avoid this
- Solution B: write explicit overload

```
template <class T>
T add(T a, T b) { return a + b; }

// solution B
double add(int a, double b) { return a + b; }

int main() {
  auto x = add(1, static_cast<int>(1.0)); // solution A
  auto y = add(1, 1.0);
}
```

- Solutions A and B have limited use, do not work for library code
- Want to write rule: always cast to type with higher capacity!
- Cannot specialize all combinations (we wanted to avoid repetition!)

- Solution C: write rule with template metaprogramming

```cpp
#include <type_traits>

template <class T, class U>
auto add(T a, U b) {
  using R = std::common_type_t<T, U>; // computes type R from types T, U
  return static_cast<R>(a) + static_cast<R>(b);
}

int main() {
  auto y = add(1, 1.0); // returns 2.0
}
```

- `std::common_type_t` is binary metafunction
  - Computes type based on type arguments
  - Choses the type which has more capacity
- How to write that?

Simple metafunction

- Instead of `std::common_type_t`, let's do something basic
- Unary metafunction: maps one type to another type

```
template <class T>
struct add_pointer {
  using type = T*; // "return value" of metafunction
};

// common boilerplate for easier access of result
template <class T>
using add_pointer_t = typename add_pointer<T>::type;

using A = add_pointer_t<int>; // int*
using B = add_pointer_t<const char>; // const char*
```

```
// simpler alternative here, but less useful in general
template <class T>
using add_pointer_t = T*;
```

- Template specialization and inheritance very useful
- Lot of metacode never called, so *zero runtime cost*

```cpp
#include <type_traits> // get std::true_type, std::false_type

template <class T>
struct is_floating_point : std::false_type {};

template <>
struct is_floating_point<float> : std::true_type {};

template <>
struct is_floating_point<double> : std::true_type {};

// ::value is implemented in base type std::true_type or std::false_type
static_assert(is_floating_point<int>::value == false, "");
static_assert(is_floating_point<float>::value == true, "");
```

- Following doesn't work, one cannot specialize *alias templates*

```cpp
template <class T>
using is_floating_point = std::false_type; // ok

template <>
using is_floating_point<float> = std::true_type; // forbidden, not valid C++
```

- Binary metafunctions work in same way

```cpp
#include <type_traits> // get std::true_type, std::false_type

template <class T, class U>
struct is_same : std::false_type {};

template <class T>
struct is_same<T, T> : std::true_type {};

static_assert(is_same<float, int>::value == false, "");
static_assert(is_same<int, int>::value == true, "");
```

Rules and limitations of metaprogramming

- Metaprogramming is turing-complete, can do any computation
- Programming syntax and rules rather different
  - Only constants, no variables (cannot *change* existing type)
  - No loop constructs, need to use recursion
  - Type collections are based on variadic templates

```
// convention is to call them type lists
template <class ... Ts> struct type_list {};

using A = type_list<int, char, float>;
```

- Recommended: Boost.Mp11 library to manipulate type lists
  - Easier than writing recursions by hand and compiles faster
  - Expressive user code, implementation details hidden in library
  - Must-read articles by Peter Dimov (author of Boost.Mp11)

    Simple C++11 metaprogramming, Simple C++11 metaprogramming 2

Boost.Mp11 in action

- Boost.Mp11 can operate on any kind of type list:
  `boost::mp11::mp_list`, `std::tuple`, `std::variant`, …

```
#include <type_traits> // std::add_pointer_t
#include <boost/mp11.hpp> // mp_append, mp_transform
using namespace boost::mp11; // to save some typing

using A = mp_append<std::tuple<int, char, long>, std::tuple<float, double>>;
// A is std::tuple<int, char, long, float, double>

using B = mp_transform<std::add_pointer_t, A>;
// B is std::tuple<int*, char*, long*, float*, double*>
```

Question: what does this code?

```
struct container {
  container(unsigned size, unsigned init_value); // ctor 1

  template <class Iterator>
  container(Iterator begin, Iterator end); // ctor 2
};

int main() {
  container c(10, 42);
}
```

Metaprogramming in practice: restricting the overload set

- Template substitutions match better than implicit conversions

```
// Example 1: restricting set of overloads
struct container {
  container(unsigned size, unsigned init_value); // ctor 1

  template <class Iterator> // bad! Iterator can match anything
  container(Iterator begin, Iterator end); // ctor 2
};

int main() { container c(10, 42); } // calls ctor 2! 💣💥🔥
```

- Error usually caught at compile-time – if you are lucky
- Why is compiler using **ctor 2** although **ctor 1** would work?
  - **Interface only**: Overload resolution ignores function body
  - Exact match better than match with implicit conversion
  - Template substitution always generates exact match

```cpp
#include <type_traits>
#include <boost/mp11.hpp>

template <class T> // ::iterator_category exists if T is iterator
using iterator_category_t =
    typename std::iterator_traits<T>::iterator_category;
template <class T>
using is_iterator = boost::mp11::mp_valid<iterator_category_t, T>;

struct container {
  container(unsigned size, unsigned init_value); // ctor 1
  template <class Iterator,
            class = std::enable_if_t<is_iterator<Iterator>::value>>
  container(Iterator begin, Iterator end); // ctor 2
};

int main() { container c(10, 42); } // calls ctor 1! 🎉
```

- Now **ctor 2** fails substitution if `Iterator` is `int`
- Doesn't stop compilation, overload just *removed from set*
- **SFINAE**: Substitution-failure-is-not-an-error
- **ctor 1** is now the best (only remaining) match

- Poor-man's alternative (don't use this)

```cpp
#include <utility> // std::declval

struct container {
  container(unsigned size, unsigned init_value); // ctor 1

  // don't do this
  template <class Iterator, class = decltype(*++std::declval<Iterator>())>
  container(Iterator begin, Iterator end); // ctor 2
};
```

- `std::declval<T>()` returns *rvalue reference* for `T`
  - Doesn't call default ctor, which may not exist
  - Often used to test interfaces, may only be used inside `decltype`
- Quick-and-dirty solution, probably works, but bad style

❌ Not full test of iterator interface - still matches non-iterators

❌ Not readable code, `is_iterator` is self-documenting

❌ Better error messages with `std::enable_if_t` in **clang**

```
:X:Y: note: candidate template ignored: requirement 'your requirement'
was not satisfied [with Iterator = your type]

container(Iterator begin, Iterator end); // ctor 2
```

Metaprogramming in practice: mixins

- Inject interface with Curiously recurring template pattern (CRTP)

```
// Example 2: Inject interface into classes
template <class Derived>
struct iterator_mixin {
  Derived& derived() { return static_cast<Derived&>(*this); } // helper
  auto begin() { return &derived()[0]; }
  auto end() { return &derived()[derived.size()]; }
};

template <class T, std::size_t N>
struct array : iterator_mixin<array<T, N>> {
  T& operator[](std::size_t i) { return data[i]; }
  std::size_t size() const { return N; }
  T data[N];
};
```

- `array` inherits `begin()` and `end()` interface from `iterator_mixin`
- CRTP: Methods in *base* can use methods in *templated derived*
  - Normally it is opposite and it only works through templates

- CRTP can be used for *static polymorphism*
- Avoids runtime cost of classic *dynamic polymorphism*
- …but cannot be extended at run-time (e.g. by loading libraries)

```
template <class Derived>
struct base {
  template <class T>
  auto interface(T arg) {
    // do something common here, e.g. input checks
    auto result = static_cast<Derived*>(this)->priv_impl(arg);
    // do something common here, e.g. cleanup
    return result;
  }
};

struct derived : base<derived> {
private:
  template <class T>
  auto priv_impl(T arg) { return 42; }
  friend class base<derived>;
};
```

- Cannot use `protected` here, because *base* needs to access *derived*

Metaprogramming in practice: introspection and static dispatch

- Boost::Histogram uses TMP to be easily extendable by users while generating the fastest possible code for any given configuration

```
// Example 3: generic label access
struct NoLabel {};
struct WithLabel { const char* label() { return "foo"; } };
struct WithIntLabel { int label() { return 42; } };

int main() {
  NoLabel nl; WithLabel wl; WithIntLabel wil;
  auto a = get_label(nl); // ""
  auto b = get_label(wl); // "foo"
  auto c = get_label(wil); // 42
}
```

- Add defaults to reduce boilerplate, e.g. `std::allocator_traits`

```cpp
// Example 3: generic label access
#include <type_traits> // std::true_type, std::false_type
#include <boost/mp11.hpp>
// valid expression only if t.label() exists for instance t of type T
template <class T>
using has_method_label_impl = decltype(std::declval<T>().label());
template <class T>
using has_method_label = boost::mp11::mp_valid<has_method_label_impl, T>;
template <class T>
auto get_label_impl(std::true_type, const T& t) { return t.label(); }
template <class T>
auto get_label_impl(std::false_type, const T& t) { return ""; }
template <class T>
auto get_label(const T& t) {
  return get_label_impl(has_method_label<T>{}, t);
}
```

- Classic solution: library-defined *base* & user-defined *derived*
  - But no base class required here, also works with POD types
  - More flexible, e.g. `label()` can return arbitrary type
- Detect special cases that allow optimizations, e.g. `std::copy`

Summary

- Template metaprogramming is weird
  - Exploits various mechanisms of C++ to get desired effects
  - Idioms not designed but discovered
  - Metafunctions based on template substitution and specialization rules
  - Types are immutable, no loops…
- TMP got less arcane with C++11 and is getting easier
- Highly recommended: Boost.Mp11 for type list manipulation
- Common uses
  - Restricting overload sets
  - Mixins and static polymorphism
  - Feature detection and static dispatch
- Try Godbolt as playground for TMP experiments