Template Trees for Modern C++ Interfaces

Modern C++ increasingly uses "Do What I Mean" (DWIM) interfaces that depend on interface selection based on secondary properties of types. Often, these interfaces make use of SFINAE techniques and/or static assertions that require the ability to compose complex compile time queries about types.

This installment of Once, Weakly describes the implementation and use of a simple facility for compile time composition and evaluation of abstract syntax trees (ASTs) of template predicates. Code for the facility accompanies this installment.

Modern C++ Interfaces

Increasing complexity in the C++ language and in the problems we solve with it has changed the way we design interfaces. An implementation like this would be considered unacceptably dangerous in modern C++:

```
// DO NOT EVEN THINK OF PASSING AN ARRAY OF COMPLEX
// TYPES TO THIS FUNCTION!!!
template <typename T>
T *copy_it(T const *src, size_t n) {
    ~~~
}
```

Instead of leaving ourselves open to an inevitable improper argument and runtime error, we'd leverage the static type system to ensure the constraints were met:

```
template <typename T>
T *copy_it(T const *src, size_t n) {
    static_assert(is_trivially_copyable<T>::value,
                  "array must be memcopyable");
    ~~~
}
```

We can also use static type queries to make our code (to some extent) self-maintaining:

```
template <typename T>
T *copy_array(T const *s, size_t n) {
    size_t const amt = sizeof(T) * n;
    T *d = static_cast<T *>(::operator new(amt));
    if constexpr (is_trivially_copyable<T>::value)
        d = static_cast<T *>(memcpy(d, s, amt));
    else if constexpr (has_nothrow_copy_constructor<T>::value)
        for (size_t i = 0; i != n; ++i) {
            new (&d[i]) T (s[i]);
        }
    else ...
```

More generally, we increasingly use static properties of types to preprocess overload resolution with SFINAE to construct DWIM interfaces. For example, this function wants to ensure that its argument is some sort of Shape:

```
template <
    typename T,
    typename = enable_if_t<is_base_of<Shape, T>::value>
>
void munge(T const &a) { ~~~ }
```

The syntax of the SFINAE constraint is challenging, but we can do an *ad hoc* improvement:

```
template <typename T>
using IsShape
    = typename enable_if<is_base_of<Shape, T>::value>::type;
```

The constraint on the argument type is now quite readable:

```
template <typename T, typename = IsShape<T>>
void munge(T const &a);
```

As a final example of DWIM interfaces, let's look at the problem of greedy universal members.

```
template <typename T>
class X {
public:
    void operation(T const &);  // #1: lvalue version
    void operation(T &&);       // #2: rvalue version
    template <typename S>
    void operation(S &&);       // #3: universal version
    ~~~
};
```

This interface is problematic because the universal reference overload often provides surprisingly better matches for arguments that a user of the interface would not expect. For example, a non-const lvalue T argument will match function #3, not #1. It's not a DWIM interface.

What *did* we mean? Chances are, we want to call the universal overload only if the deduced argument S differs significantly from the type T used to specialize X. As always, it's best to be explicit, and it's best to express ourselves in code. We can say what we mean in code, but it's not a trivial statement:

```
template <typename S, typename T>
using similar = std::is_same<std::decay_t<S>, std::decay_t<T>>;

template <typename S, typename T>
using NotSimilar = enable_if_t<!similar<S, T>::value>;
~~~
```

```
template <typename T>
class X {
public:
   void operation(T const &);  // #1: lvalue version
   void operation(T &&);       // #2: rvalue version
   template <typename S, typename = NotSimilar<S, T>>
   void operation(S &&);      // #3: universal version
                              // Do What I Mean:
};                            // if S isn't "similar" to T
```

Template ASTs

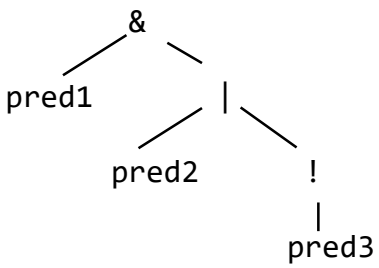
The type queries we use for constraint checking and in the implementation of DWIM interfaces are increasingly complex. Rather than address each situation in an *ad hoc* manner, let's develop a toolkit that will allow the creation of arbitrarily complex compile time type queries in a straightforward and simple manner.

The <type_traits> standard header file defines a number of class templates used as predicates to uncover properties of types at compile time. For example, we used the is_trivially_copyable type trait above in the "self-maintaining" implementation of copy_array. Unary class templates like these will be the leaves of our AST. We'll generate more complex predicates by composing simpler predicates with &, |, ^, and ! operators.

For example, a type predicate expression like

```
pred1 & (pred2 | !pred3)
```

would generate an AST like



The leaves of our AST are class templates that can take a single type argument and produce a Boolean result. For example,

```
template <typename> class Pred;
template <typename, typename = void> class BPred;
```

As AST leaves, we'll wrap these predicates in a trivial object that can be evaluated at compile time with a type argument:

```
template <template <typename...> class Pred>
struct Id : PAST {
   template <typename T>
```

```
    static constexpr bool eval() { return Pred<T>::value; }
};
template <template <typename...> class Pred>
constexpr auto pred() { return Id<Pred>(); }
```

For example, we can provide ready-made leaves from the predicates in <type_traits>:

```
constexpr auto isPod = pred<std::is_pod>(); // an object
using IsPod = decltype(isPod);              // a type
```

We'll implement the binary operators like this:

```
template <typename P1, typename P2>
struct And : PAST {
    template <typename T>
    static constexpr bool eval()
         { return P1::template eval<T>() & P2::template eval<T>(); }
};
```

For clarity and convenience, we'll use an overloaded infix operator to generate the type.

```
template <typename P1, typename P2, typename = all_PAST<P1, P2>>
constexpr auto operator &(P1, P2) { return And<P1, P2>(); }
```

The PAST (Predicate AST) base class is used to implement an SFINAE check to ensure that an overloaded operator doesn't interfere with other overloads of the operator. The all_Past constraint ensures that all arguments to the overloaded operator are derived from PAST.

Note that we're interested entirely in the (compile time) return type of the function rather than the (runtime) return value. Essentially, an AST object and the type of an AST object hold substantially the same information; we'll see that sometimes one is more convenient to use than the other.

The implementation of our unary operator is as you'd expect:

```
template <typename P>
struct Not : PAST {
    template <typename T>
    static constexpr bool eval()
         { return !P::template eval<T>(); }
};
template <typename P, typename = all_PAST<P>>
constexpr auto operator !(P) { return Not<P>(); }
```

Note that not all predicates on types are unary. The <type_traits> header has a number of type predicates that examine the relationship between two or more types, and it's easy to imagine predicates that examine the relationship between a type and an integral or other non-type value. In order to accommodate predicates like this, we'll provide binders (similar to the deprecated STL function object binders).

For example, bind1st converts a binary type predicate into a unary predicate identical in behavior to an Id:

```
template <template <typename, typename> class BPred, typename First>
   struct BindT1st : PAST {
   template <typename Second>
   static constexpr bool eval()
        { return BPred<First, Second>::value; }
};
template <template <typename, typename> class BPred, typename First>
constexpr auto bind1st() { return BindT1st<BPred, First>(); }
```

For instance, we can use bind1st to create an "is derived from Shape" unary predicate from the standard is_base_of binary predicate:

```
constexpr auto isShape = bind1st<is_base_of, Shape>();
```

The code that accompanies this installment implements a number of binders for type and size_t arguments.

Generating and Evaluating ASTs

With this mechanism in place, we can easily generate complex predicates[1]:

```
constexpr auto my_needs             // build an AST
    = isClass & (isPod | !isPolymorphic) ^ isShape;

constexpr auto your_needs           // build another
    = isClass & hasVirtualDestructor
              ^ !isNothrowCopyAssignable;

constexpr auto too_complex         // really go overboard
   = (isClass & (isPod | !isPolymorphic)
              ^ isUnsigned & !isTrivial)
              | (isPtrSize & !isFloatingPoint)
              | bind2nd<is_of_size, 8>();
```

We can evaluate an AST for a type by invoking its eval member template:

```
constexpr bool result = your_needs.eval<T>();
```

We can get additional flexibility and a somewhat simpler syntax by invoking eval indirectly:

```
template <typename... Ts, typename AST, typename = all_PAST<AST>>
constexpr bool satisfies(AST)
   { return AllTrue<AST().template eval<Ts>()...>::value; }
~~~
constexpr bool you_ok = satisfies<T>(your_needs);
```

[1] No, none of these make any sense at all.

```
constexpr bool everyone_ok = satisfies<T, S, R>(your_needs);
```

The guaranteed compile time result provided by satisfies is ideal for constraint checking, as with a static assertion:

```
static_assert(satisfies<You>(your_needs), "Your needs are unmet");
```

However, a guaranteed result is less useful in an SFINAE context. For that situation we'll provide a type result that is available only if the predicate is true:

```
template <typename AST, typename... Ts>
using Constraint
   = typename std::enable_if<satisfies<Ts...>(AST())>::type;
```

This is more appropriate for writing DWIM interfaces:

```
using MyNeeds = decltype(my_needs);
using YourNeeds = decltype(your_needs);
~~~
template <typename Me, typename You,
          typename = Constraint<MyNeeds, Me>,
          typename = Constraint<YourNeeds, You>>
void oy_vey(Me &&me, You &&you) { ~~~ }
```

Or, if we have the same needs:

```
template <typename Me, typename You,
          typename = Constraint<YourNeeds, You, Me>>
void oy_vey(Me &&me, You &&you) { ~~~ }
```

Note that Constraint requires the type of an AST rather than an AST object. This is never a practical issue (since it's easy to generate an AST object from a type or a type from an object) but it can be more convenient to have an SFINAE condition that uses an AST object instead:

```
template <typename... Ts, typename AST,
          bool test = AllTrue<AST().template eval<Ts>()...>::value>
constexpr bool constraint(AST, char (*)[test == true] = nullptr)
   { return true; }
```

This would allow us to use SFINAE without having to extract the type of the predicate from the object:[2]

```
template <typename Me, typename You,
          bool = constraint<Me, You>(my_needs ^ !your_needs)>
void oy_vey(Me &&me, You &&you) { ~~~ }
```

As a final exercise for the reader, consider the following interface:

```
class X {
```

[2] This code is corrected from the original appearance of this installment on 15 February 2017.

```
public:
    ~~~
    template <typename S> void doit(S &&); // #1
    template <typename S> void doit(S &&); // #2
    template <typename S> void doit(S &&); // #3
};
```

The task is to provide appropriate SFINAE disambiguation to allow a user of this interface to call member template #1 if S satisfies your_needs, #2 if S satisfies my_needs, and #3 if S satisfies some other requirement of your choosing.

Code for the AST generator/evaluator, an AST-enabled adaptation of <type_traits>, a simplified set of predicates for STL iterator categories, and sample code that shows how to apply this facility in various contexts (including in the solution to the above task) are available with this installment.

stevedewhurst.com