The Detection Idiom - a better way to SFINAE

Marshall Clow

Qualcomm Technology, Inc.

May 19, 2017

The detection idiom

1. What is it?
2. What problem does it solve?
3. What does it look like?
4. How does it work?
5. How do I use it?

What is it?

1. Introduced in the Library Fundamentals TS 2
2. Mentioned in the `experimental/type_traits` synopsis
3. A couple of examples in the text of the TS
4. Two structs: `void_t` and `nonesuch`
5. An 'exposition-only' struct named `DETECTOR`
6. Two type-traits like structs: `is_detected` and `is_detected_t` and variations of them.

Where did it come from?

Walter Brown developed it. He wrote it up in several papers for the C++ standard committee (N3911, N4436 and N4502) proposing (and refining) the idea.

Walter gave a pair of talks on this topic at CPPCon in 2014 (videos on YouTube).

What problem does it solve?

The standard library provides a set of pre-defined type-traits that let you find out information about the types that you are dealing with. If one of those does what you need, then you're golden. However, those traits don't cover all possible eventualities.

> *When* `T` *is an array type, this constructor shall not participate in overload resolution unless* `is_move_constructible_v<D>` *is true, the expression* `d(p)` *is well-formed, and either* `T` *is* `U[N]` *and* `Y(*)[N]` *is convertible to* `T*`*, or* `T` *is* `U[]` *and* `Y(*)[]` *is convertible to T*.*

The detection idiom is a way of asking *"is this code (with these types) well-formed?"* at compile time - w/o generating a compiler error when the answer is 'no'.

A simple example

```
struct Yes { typedef void type; };
class  Huh { typedef void type; }; // private
struct No  { /* no 'type' member typedef */ };

template <typename T>
    using has_type = typename T::type;

static_assert(
       is_detected<has_type, Yes>::value );
static_assert(
      !is_detected<has_type, Huh>::value );
static_assert(
      !is_detected<has_type,  No>::value );
```

A slightly more practical example

```
struct Yes { typedef size_t size_type; };
struct No  { /* no 'size_type' */ };

template <typename T>
    using has_size = typename T::size_type;

template <typename T>
void doSomething( const T& t )
{
    detected_or_t<short, has_size, T> v = 123;
//  more with v
}
```

What does it look like?

void_t and nonesuch

```
template <class...>
    using void_t = void;

struct nonesuch {
    nonesuch() = delete;
    ~nonesuch() = delete;
    nonesuch(nonesuch const&) = delete;
    void operator=(nonesuch const&) = delete;
};
```

The exposition-only class DETECTOR

```
template <class Default, class AlwaysVoid,
    template<class...> class Op, class... Args>
struct DETECTOR
{
    using value_t = false_type;
    using type = Default;
};

template <class Default, template<class...>
          class Op, class... Args>
struct DETECTOR<Default, void_t<Op<Args...>>,
                        Op, Args...>
{
    using value_t = true_type;
    using type = Op<Args...>;
};
```

The bits that you do use

```
template <template<class...> class Op,
          class... Args>
  using is_detected =
    typename DETECTOR<nonesuch, void,
                      Op, Args...>::value_t;

template <template<class...> class Op,
          class... Args>
  using detected_t =
    typename DETECTOR<nonesuch, void,
                      Op, Args...>::type;
```

How does it work?

It is surprisingly simple. You provide a 'type function'; a template that takes a type (or series of types) and returns a type.

Then you use `is_detected` to apply that type function to the types that you're interested in, and it returns to you: (a) whether or not this worked, and (b) if it did, what the result (type) was.

That's not what I meant: How does it work?

It attempt to instantiate `DETECTOR` with the types and parameters that are passed. The first specialization always successfully instantiates. The second specialization can be instantiated if the expression `Op<Args...>` can be instantiated, and if so, it is a more-specialized version, and as such, is chosen during overload-resolution.

All of this happens in an "unevaluated context", so no code is generated. We're leveraging the compiler's overload resolution mechanism to make decisions for us.

Variations on a theme

```
template <class Default, template<class...>
          class Op, class... Args>
  using detected_or =
    DETECTOR<Default, void, Op, Args...>;

template <class Expected, template<class...>
          class Op, class... Args>
  using is_detected_exact =
    is_same<Expected, detected_t<Op, Args...>>;

template <class To, template<class...>
          class Op, class... Args>
  using is_detected_convertible =
    is_convertible<detected_t<Op, Args...>, To>;
```

`decltype` and `declval`

These techniques are not just limited to type members. You can test the validity of expressions as well; you just need to express them in terms of a type.

Fortunately, we have `decltype` and `declval` which make that easy.

Detecting a member variable

```
struct Yes { int foo; };
class  Huh { int foo; }; // private
struct No  { /* no member foo */ };

template <typename T>
    using has_foo = decltype(T::foo);

static_assert(
       is_detected<has_foo, Yes>::value );
static_assert(
      !is_detected<has_foo, Huh>::value );
static_assert(
      !is_detected<has_foo,  No>::value );
```

Detecting a member function

```
struct Yes { int foo(); };
class  Huh { int foo(); }; // private
struct No  { /* no member foo */ };

template <typename T>
    using has_foo =
        decltype(std::declval<T&>().foo());

static_assert(
        is_detected<has_foo, Yes>::value );
static_assert(
      !is_detected<has_foo, Huh>::value );
static_assert(
      !is_detected<has_foo,  No>::value );
```

Detecting a free function

```
int                 foo(int);
string              foo(string);

template <typename T>
  using foo_res = decltype(foo(std::declval<T>()));

static_assert(
    is_detected<foo_res, long>::value );
static_assert(
    is_detected<foo_res, string>::value );
static_assert(
   !is_detected<foo_res, complex<double>>::value );
```

We can also test for operators

Copy-assignment example (1)

```
struct Yes { Yes &operator=(const Yes&)
                              { return *this; }};
struct Huh { void operator=(const Huh&) {}};
struct No  { No  &operator=(const No&) = delete; };

template <class T>
  using copy_assign =
    decltype(std::declval<T&>() =
                std::declval<T const &>());

static_assert( is_detected_v<copy_assign, Yes>);
static_assert( is_detected_v<copy_assign, Huh>);
static_assert(!is_detected_v<copy_assign, No> );
```

Copy-assignment example (2)

```
struct Yes { Yes &operator=(const Yes&)
                          { return *this; }};
struct Huh { void operator=(const Huh&) {}};
struct No  { No  &operator=(const No&) = delete; };

template <class T>
  using copy_assign =
    decltype(std::declval<T&>() =
                std::declval<T const &>());

static_assert(
     is_detected_exact_v<Yes&, copy_assign, Yes>);
static_assert(
    !is_detected_exact_v<Huh&, copy_assign, Huh>);
static_assert(
    !is_detected_exact_v<No&,  copy_assign, No> );
```

Questions?

Thank you

References

1. Library Fundamentals 2
2. N3911: TransformationTrait Alias `void_t`
3. N4436: Proposing Standard Library Support for the C++ Detection Idiom
4. N4502: Proposing Standard Library Support for the C++ Detection Idiom, v2
5. Modern Template Metaprogramming: A Compendium, Part I: https://www.youtube.com/watch?v=Am2is2QCvxY
6. Modern Template Metaprogramming: A Compendium, Part II: https://www.youtube.com/watch?v=a0FliKwcwXE