CSE-491
Techniques for type introspection in C++

cgnitash@msu.edu

January 30, 2019

1 Introduction

Programming with types has tons of applications, e.g. algorithm specialization based on the properties of a type. My personal favourite is implementing forms of run-time polymorphism using this technique. Unfortunately, C++ introspection tools have been rather verbose, cumbersome and not very natural to work with. So unnatural in fact, that easier techniques and tools were often discovered long after the language features that enabled them were introduced. This article discusses the approach first discovered about 5 years ago, (even though the technique itself has been doable for 20 years). The content is based largely on the talks given by Walter Brown and this standards proposal. Note that much of this is already in C++17, and even more tools have been added, but this is where I first understood how this stuff works so I'm using this as a reference. Also, I think this is a much cleaner way to use SFINAE than the one we saw in class last week, or the multiple approaches that are available through an online search.

2 Motivation

We are going to use what is very much a toy example, but one that I think is exemplary enough of the problem that we're trying to solve.

In general, let's say, we would like to write a function that takes in a value v of some type T, and use that value in some particular computation. However, if the type T doesn't satisfy the requirements of that computation, we might still want to do something reasonable, instead of just getting a compiler error.

Concretely, we have a bunch of types in our system that behave like Animals. These Animals have a greeting, but not all of them! When they have a greeting, we just use it, but if they don't, we growl!! Here's the code

```
struct Dog {
    std::string greeting(){
        return "Woof!";
    }
};
```

```
struct Cat {
    std::string greeting(){
        return "Purr!";
    }
};

void print_greeting(Dog d) {
    std::cout << "Hi! " << d.greeting() << std::endl;
}

void print_greeting(Cat c) {
    std::cout << "Hi! " << c.greeting() << std::endl;
}

int main() {
    Dog d;
    Cat c;
    print_greeting(d);
    print_greeting(c);
}
```

Great. That compiles and runs and does what one would expect. Straight away, it looks like there's repeated code, which is just a potential for inconsistencies, so let's make the function generic.

```
template<typename Animal>
void print_greeting(Animal a) {
    std::cout << "Hi! " << a.greeting() << std::endl;
}
```

That's fine, but now for the interesting bit. What happens if we try it with some type that doesn't have a greeting?

```
struct Snake {
};

int main() {
    Dog d;
    Cat c;
    Snake s;
    print_greeting(d);
    print_greeting(c);
    print_greeting(s);
}
```

Ok, nothing surprising here. As expected, we get the hard error

```
main.cpp: In instantiation of 'void print_greeting(Animal) [with Animal = Snake]':
```

```
main.cpp:71:21:   required from here
main.cpp:48:30: error: 'struct Snake' has no member named 'greeting'
     std::cout << "Hi! " << a.greeting() << std::endl;
                              ~~~~~~~~~~
```

How we can get around this? We can of course add an overload for print_greeting that accepts an argument of Snake Type, like this

```
void print_greeting(Snake ) {
    std::cout << "Grrr :(" << std::endl;
}
```

Now the code compiles and prints

```
Hi! Woof!
Hi! Purr!
Grrr :(
```

While this technically solves our problem, this is not a generic solution for the case where the Type doesn't have a member function greeting(); we will have to add an overload for each one of those types. More importantly, this is not a general solution; if we want to check for other members, or combinations of members, etc, we can't use this technique.

3 Solution

Ok, so we can kind of state what we want to happen, but don't know how to say it in C++. So let's go ahead and write a comment to start off.

```
template<typename Animal>
void print_greeting(Animal a) {
    if constexpr (/* Animal Type has a method called greeting() */)
        std::cout << "Hi! " << a.greeting() << std::endl;
    else
        std::cout << "Grrr :(" << std::endl;
}
```

If you haven't seen 'constexpr if' before, it's basically a compile time if, i.e. the condition must be something that can be figured out at compile time. Well, that seems fine, since what we want to check for is the existence of a struct method, which is definitely known at compile time.

Now, the next step is to figure out how to write the condition itself. Breaking it down, we have 2 problems here. 1.) How to express a particular expression (i.e. describe the expression without actually computing the expression) and 2.) How to make a choice based on whether that expression is well-formed. Let's tackle them separately.

3.1 Naming expressions

Conceptually, all we want to do is give a name to a particular piece of syntax. This happens to take a bit of work in C++.The trick is, we are going to use a Type (hah!) to represent an expression. To take an example,

```
x += 5 * y;
```

is an expression (informally, a piece of syntax). Now if we simply say

```
auto expression = decltype(x+=5*y);
```

conceptually, we have a name for that expression. Ok, this is a **Type**, so we make it an alias template.

```
template<typename T>
using expression = decltype(x+=5*y);
```

But unfortunately, this won't work because decltype only works in non-evaluated contexts. Ok, that's a lot of words, so let's just dive into our example.

We want to ask, for an object v of type T, is the expression

```
v.greeting();
```

well-formed? Now as we saw above we can't just do

```
decltype(v.greeting())
```

because v is not in a non-evaluated context. But it's not too hard to figure out what v is. It's actually just an lvalue reference of type T. Now we need to pretend that we have on object of this type. Fortunately, there is a utility named

```
std::declval<>()
```

for exactly this. Saying

```
std::declval<T&>()
```

is basically saying, 'pretend' that I have an lvalue reference of type T. So now the evaluated context expression v.greeting() can be written in an unevaluated context as

```
decltype(std::declval<T&>().greeting())
```

Simple enough transformations, though it ends up looking rather ugly. Let's give it a name

```
template<typename T>
  using has_greeting_t =
    decltype(std::declval<T&>().greeting());
```

3.2 Choosing types

Now for the 2nd part, viz, make a choice based on whether a type is well-formed or not. Conceptually, what we need is a type that can be used as a 'marker' for all well-formed expressions. It's actually completely irrelevant what this type is, just that we have a name for it. The C++ standard has chosen void (why not?), and defined the type void_t as

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

That's it. In fact, this is in C++17, so we can just use std::void_t. But what does it do? Very simple, it's a template, and whatever template arguments you give it, it returns void. However, and here's the really beautiful bit, it returns void only if all the template arguments are well-formed. If even a single argument is ill-formed, the entire expression is ill-formed. For illustration,

```
std::void_t<int> // this is just void
std::void_t<abc> // if abc is not a type,
                 // and a substitution is forced
                 // this will be a hard error.
```

But, we don't have to force the substitution of this template argument if we're careful and then we can use the fact that *substitution failure is not an error* (yay, sfinae) . And that can be done by partially specializing class templates.

3.2.1 Partial Specialization

Partial template specialization rules are a lot of fun. 12.6.5 is a fun section (it does describe a Turing-complete mechanism after all). But we don't need anything that powerful; we just want to make a true/false choice. And for that, all we need is

12.6.5.1(1.1) - If exactly one matching specialization is found, the instantiation is generated from that specialization.

So what happens when we write

```
// primary template
template<typename T, typename = std::void_t<>>
  struct has_greeting : std::false_type{};
// partial template (partial specialization)
template<typename T>
  struct has_greeting<T, std::void_t<E>> : std::true_type{};
```

Well, the partial specialization is chosen. Always. It must be. The rule says so. And we get std::true_type. (std::true_type and std::false_type are 2 convenience types, that basically represent compile time types, true and false (the names are not an accident)). So we always get compile-time-true. Except, and here's the wonderful bit, that template is only valid if E is well-formed. Otherwise, the partial specialization **cannot** be chosen, and instead the primary template is chosen. (which is just compile-time-false). Done. We're all set.

4 Bringing it home

All right, so how do we put all this stuff together to get something working? Here goes. First we need to substitue for E. Well, that's just the expression whose well-formedness we want to test for. So the partial specialization becomes,

```
template<typename T>
  struct has_greeting<T, std::void_t<has_greeting_t<T>>>
  : std::true_type{};
```

Now let's go through what's happening again. There are 2 possibilities (i,e 2 possible templates that can be instantiated). Now, either Type T has a method called greeting() or it doesn't. If it does, then

```
std::void_t<has_greeting<T>>
```

is well formed. This means that the partially specialized template is a better match, and it's chosen, yielding std::true_type. Importantly, the primary template could also match, but the rule says it won't get chosen :)

On the other hand, if the method doesn't exist, then

```
std::void_t<has_greeting<T>>
```

is ill-formed. But, as we have seen above, , phew! we don't have a hard compile error (thank you SFINAE), because there is another template that matches; the primary template that has 2 template parameters, T and void (the void is spelled funnily of course). And that one yields std::false_type. And that's it, now we can actually implement our function

```
template<typename Animal>
void print_greeting(Animal a) {
 if constexpr (has_greeting<Animal>{})
     std::cout << "Hi! " << a.greeting() << std::endl;
 else
    std::cout << "Grrr :(" << std::endl;
}
```

Yay, that compiles and prints

```
Hi! Woof!
Hi! Purr!
Grrr :(
```

5 Generalization

So we achieved what we set out to do. However, we can make this more general, so we can use this technique multiple times. Note that we still need to write the pair of full and partial specializations for each new expression that we want to test for, and that's annoying. We'd be repeating code, and the syntax is non-trivial, making it error prone. So let's go ahead

and abstract away the expression that we are testing for. We can look at the one example we have already, viz has_greeting_t. This is just an alias template, so the abstract expression will just be a template. Ok, now what does that look like?

```
template <typename, template <typename> typename,
          typename = std::void_t<>>
  struct is_detected : std::false_type {};

template <typename T, template <typename> typename E>
  struct is_detected<T, E, std::void_t<E<T>>> : std::true_type {};
```

All that we've done here is inserted a new template argument E, into the second position (and changed the name of the structs). This E is just a type, and this type is a template that depends upon T. Note that we don't have to specify that in the nested template of the second argument, since the user is going to specialize it later. If you look closely at how E is used in the 3rd argument of the partial specialization, it's exactly how we used it when we explicitly named E. Now we've just abstracted that away.

Now, if we move these structs (and it's literally 4 lines of code), we have a powerful technique that can be used in a variety of ways.

To clarify, the complete solution (apart from the 4 lines above) is

```
template<typename T>
  using has_greeting_t =
    decltype(std::declval<T&>().greeting());

template<typename Animal>
void print_greeting(Animal a) {
 if constexpr (is_detected<Animal, has_greeting_t>{})
     std::cout << "Hi! " << a.greeting() << std::endl;
 else
    std::cout << "Grrr :(" << std::endl;
}
```

That's pretty nice. The only sticking point is the syntax for writing an expression in an unevaluated context. There's no way around that, but within a few short years, concepts will be here, and I think they'll help a lot. Also, I would like to declare the alias template inside the function (assuming I don't want to use it anywhere else), but C++ grammar doesn't allow template declarations at function scope, and we don't want to lose that.

Note that we have barely scratched the surface here. We can do a lot more with these techniques. Here is an example of an application that's super super cool.

6 Bonus

Here's a random example. I want to write a function called is_push_back_able(), that returns e.g. true for std::vector and false for int. Here's my solution,

```
template<typename T>
```

```
    using has_push_back_t =
      decltype(std::declval<T &>().push_back(
                std::declval<const typename T::value_type &>()));

template<typename T>
bool is_push_back_able(T){
    if constexpr (is_detected<T, has_push_back_t>{})
        return true;
    else
        return false;
}

int main() {
  std::vector<int> v;
  int i;
  std::cout << is_push_back_able(v) << std::endl; // prints 1
  std::cout << is_push_back_able(i) << std::endl; // prints 0
}
```

For additional credit (note, this is meaningless, I have no authority to give credit :P, so please do this only if you enjoy doing this stuff in your spare time), clean up the alias template has_push_back_t. There are several utilities that are already in C++, e.g. is_const, that you can use to do this.

For additional, additional credit, write a function that accepts an iterator, and figures out if the iterator models a ForwardIterator or BidirectionalIterator. Note that these iterator categories will have the Legacy prefix from C++20, since all of that is being revised for the new age of Concepts, but the ideas are still pretty similar.