Item 4: Extensible Templates: Via Inheritance or Traits?

.

ITEM 4: EXTENSIBLE TEMPLATES: VIA INHERITANCE OR TRAITS? DIFFICULTY: 7

This Item reviews traits templates and demonstrates some cool traits techniques. Even without traits, what can a template figure out about its type—and what can it do about it? The answers are nifty and illuminating, and not just for people who write C++ libraries.

1. What is a traits class?
2. Demonstrate how to detect and make use of template parameters' members, using the following motivating case: You want to write a class template `C` that can be instantiated only on types that have a `const` member function named `Clone()` that takes no parameters and returns a pointer to the same kind of object.

```
// T must provide T* T::Clone() const
template<typename T>
class C
{
  // ...
};
```

Note: It's obvious that if `C` writes code that just tries to invoke `T::Clone()` without parameters, then such code will fail to compile if there isn't a `T::Clone()` that can be called without parameters. But that's not enough to answer this question, because just trying to call `T::Clone()` without parameters would also succeed in calling a `Clone()` that has default parameters and/or does not return a `T*`. The goal

here is to specifically enforce that `T` provide a function that looks exactly like this: `T* T::Clone() const`.

3. A programmer wants to write a template that can require (or just detect) whether the type on which it is instantiated has a `Clone()` member function. The programmer decides to do this by requiring that classes offering such a `Clone()` must derive from a predetermined `Cloneable` base class. Demonstrate how to write the following template:

```
template<typename T>
class X
{
  // ...
};
```

   a) to require that `T` be derived from `Cloneable`; and
   b) to provide an alternative implementation if `T` is derived from `Cloneable`, and work in some default mode otherwise.

4. Is the approach in #3 the best way to require/detect the availability of a `Clone()`? Describe alternatives.
5. How useful is it for a template to know that its parameter type `T` is derived from some other type? Does knowing about such derivation give any benefit that couldn't also be achieved without the inheritance relationship?

SOLUTION

1. What is a traits class?

Quoting 17.1.18 in the C++ standard [C++98], a traits class is

> *a class that encapsulates a set of types and functions necessary for template classes and template functions to manipulate objects of types for which they are instantiated.*[2]

The idea is that traits classes are instances of templates, and are used to carry extra information—especially information that other templates can use—about the types on which the traits template is instantiated. The nice thing is that the traits class `T<C>` lets us record such extra information about a class `C`, without requiring any change at all to `C`.

2. Here *encapsulate*s is used in the sense of bringing together in one place, rather than hiding behind a shell. It's common for everything in traits classes to be public, and indeed they are typically implemented as `struct` templates.

For example, in the standard itself `std::char_traits<T>` gives information about the character-like type `T`, particularly how to compare and manipulate such `T` objects. This information is used in such templates as `std::basic_string` and `std::basic_ostream` to allow them to work with character types that are not necessarily `char` or `wchar_t`, including working with user-defined types for which you provide a suitable specialization of `std::char_traits`. Similarly, `std::iterator_traits` provides information about iterators that other templates, particularly algorithms and containers, can put to good use. Even `std::numeric_limits` gets into the traits act, providing information about the capabilities and behavior of various kinds of numeric types as they're implemented on your particular platform and compiler.

For more examples, see:

- Items 30 and 31 on smart pointer members
- *Exceptional C++* [Sutter00] Items 2 and 3, which show how to customize `std::char_traits` to customize the behavior of `std::basic_string`
- The April, May, and June 2000 issues of *C++ Report*, which contained several excellent columns about traits

Requiring Member Functions

2. Demonstrate how to detect and make use of template parameters' members, using the following motivating case: You want to write a class template `C` that can be instantiated only on types that have a member function named `Clone()` that takes no parameters and returns a pointer to the same kind of object.

```
// Example 4-2
//
// T must provide T* T::Clone() const
template<typename T>
class C
{
  // ...
};
```

Note: It's obvious that if `C` writes code that just tries to invoke `T::Clone()` without parameters, then such code will fail to compile if there isn't a `T::Clone()` that can be called without parameters.

For an example to illustrate that last note, consider the following:

```
// Example 4-2(a): Initial attempt,
// sort of requires Clone()
//
// T must provide /*...*/ T::Clone( /*...*/ )
template<typename T>
class C
{
```

```
public:
  void SomeFunc( const T* t )
  {
    // ...
    t->Clone();
    // ...
  }
};
```

The first problem with Example 4-2(a) is that it doesn't necessarily require anything at all. In a template, only the member functions that are actually used will be instantiated.[3] If `SomeFunc()` is never used, it will never be instantiated, so `C` can easily be instantiated with types `T` that don't have anything resembling `Clone()`.

The solution is to put the code that enforces the requirement into a function that's sure to be instantiated. Many people put it in the constructor, because it's impossible to use `C` without invoking its constructor somewhere, right? (This approach is mentioned, for example, in [Stroustrup94].) True enough, but there could be multiple constructors. Then, to be safe, we'd have to put the requirement-enforcing code into every constructor. An easier solution is to put it in the destructor. After all, there's only one destructor, and it's unlikely that `C` will be used without invoking its destructor (by creating a `C` object dynamically and never deleting it). So, perhaps, the destructor is a somewhat simpler place for the requirement-enforcing code to live:

```
// Example 4-2(b): Revised attempt, requires Clone()
//
// T must provide /*...*/ T::Clone( /*...*/ )
template<typename T>
class C
{
public:
  ~C()
  {
    // ...
    const T t; // kind of wasteful, plus also requires
               // that T have a default constructor
    t.Clone();
    // ...
  }
};
```

That's still not entirely satisfactory. Let's set this aside for now, but we'll soon come back and improve this enforcement further, after we've done a bit more thinking about it.

3. Eventually, all compilers will get this rule right. Yours might still instantiate all functions, not just the ones that are used.

This leaves us with the second problem: Both Examples 4-2(a) and 4-2(b) don't so much test the constraint as simply rely on it. (In the case of Example 4-2(b), it's even worse because 4(b) does it in a wasteful way that adds unnecessary runtime code just to try to enforce a constraint.)

But that's not enough to answer this question, because just trying to call `T::Clone()` without parameters would also succeed in calling a `Clone()` that has defaulted parameters and/or does not return a `T*`.

The code in Examples 4-2(a) and 4-2(b) will indeed work most swimmingly if there is a function that looks like `T* T::Clone()`. The problem is that it will also work most swimmingly if there is a function `void T::Clone()`, or `T* T::Clone( int = 42 )`, or with some other oddball variant signature, like `T* T::Clone( const char* = "xyzzy" )`, just as long as it can be called without parameters. (For that matter, it will work even if there isn't a `Clone()` member function at all, as long as there's a macro that changes the name `Clone` to something else, but there's little we can do about that.)

All that may be fine in some applications, but it's not what the question asked for. What we want to achieve is stronger:

The goal here is to specifically enforce that `T` provide a function that looks exactly like this: `T* T::Clone() const`.

So here's one way we can do it:

```
// Example 4-2(c): Better, requires
// exactly T* T::Clone() const
//
// T must provide T* T::Clone() const
template<typename T>
class C
{
public:
  // in C's destructor (easier than putting it
  // in every C constructor):
  ~C()
  {
    T* (T::*test)() const = &T::Clone;
    test; // suppress warnings about unused variables
    // this unused variable is likely to be optimized
    // away entirely

    // ...
  }

  // ...
};
```

Or, a little more cleanly and extensibly:

```
// Example 4-2(d): Alternative way of requiring
// exactly T* T::Clone() const
//
// T must provide T* T::Clone() const
template<typename T>
class C
{
  bool ValidateRequirements() const
  {
    T* (T::*test)() const = &T::Clone;
    test; // suppress warnings about unused variables
    // ...
    return true;
  }

public:
  // in C's destructor (easier than putting it
  // in every C constructor):
 ~C()
  {
    assert( ValidateRequirements() );
  }

  // ...
};
```

Having a `ValidateRequirements()` function is extensible, for it gives us a nice clean place to add any future requirements checks. Calling it within an `assert()` further ensures that all traces of the requirements machinery will disappear from release builds.

Constraints Classes

There's an even cleaner way to do it, though. The following technique is publicized by Bjarne Stroustrup in his *C++ Style and Technique FAQ* [StroustrupFAQ], crediting Alex Stepanov and Jeremy Siek for the use of pointer to function.[4]

Suppose we write the following `HasClone` constraints class:

```
// Example 4-2(e): Using constraint inheritance
// to require exactly T* T::Clone() const
//

// HasClone requires that T must provide
// T* T::Clone() const
template<typename T>
class HasClone
{
```

4. See *http://www.gotw.ca/publications/mxc++/bs_constraints.htm*.

```
public:
  static void Constraints()
  {
    T* (T::*test)() const = &T::Clone;
    test; // suppress warnings about unused variables
  }
  HasClone() { void (*p)() = Constraints; }
};
```

Now we have an elegant—dare I say "cool"?—way to enforce the constraint at compile time:

```
template<typename T>
class C : HasClone<T>
{
  // ...
};
```

The idea is simple: Every `C` constructor must invoke the `HasClone<T>` default constructor, which does nothing but test the constraint. If the constraint test fails, most compilers will emit a fairly readable error message. The `HasClone<T>` derivation amounts to an assertion about a characteristic of `T` in a way that's easy to diagnose.

Requiring Inheritance, Take 1: `IsDerivedFrom1` Value Helper

3. A programmer wants to write a template that can require (or just detect) whether the type on which it is instantiated has a `Clone()` member function. The programmer decides to do this by requiring that classes offering such a `Clone()` must derive from a predetermined `Cloneable` base class. Demonstrate how to write the following template:

```
template<typename T>
class X
{
  // ...
};
```

a) to require that `T` be derived from `Cloneable`

We'll take a look at two approaches. Both work. The first approach is a bit tricky and complex; the second is simple and elegant. It's valuable to consider both approaches because they both demonstrate interesting techniques that are good to know about, even though one technique happens to be more applicable here.

The first approach is based on Andrei Alexandrescu's ideas in "Mappings Between Types and Values" [Alexandrescu00a]. First, we define a helper template that tests whether a candidate type `D` is derived from `B`. It determines this by determining whether a pointer to `D` can be converted to a pointer to `B`. Here's one way to do it, similar to Alexandrescu's approach:

```
// Example 4-3(a): An IsDerivedFrom1 value helper
// (                )  (  )
// Advantages: Can be used for compile-time value test
// Drawbacks:  Pretty complex
//
template<typename D, typename B>
class IsDerivedFrom1
{
  class No { };
  class Yes { No no[2]; };

  static Yes Test( B* ); // declared, but not defined
  static No Test( ... ); // declared, but not defined

public:
  enum { Is = sizeof(Test(static_cast<D*>(0))) == sizeof(Yes) };
};
```

Get it? Think about this code for a moment before reading on.

* * * * *

The above trick relies on three things:

1. `Yes` and `No` have different sizes. This is guaranteed by having a `Yes` contain an array of more than one `No` object. (And anyway, two negatives sometimes do make a positive. This time it's not really a `No` no.)
2. Overload resolution and determining the value of sizeof are both performed at compile time, not runtime.
3. Enum values are evaluated, and can be used, at compile time.

Let's analyze the `enum` definition in more detail. First, the innermost part is:

```
Test(static_cast<D*>(0))
```

All this does is mention a function named `Test` and pretend to pass it a `D*`—in this case, a suitably cast null pointer will do. Note that nothing is actually being done here, and no code is being generated, so the pointer is never dereferenced or, for that matter, even ever actually created. All we're doing is creating a typed expression. Now, the compiler knows what `D` is, and will apply overload resolution at compile time to decide which of the two overloads of `Test()` ought to be chosen: If a `D*` can be converted to a `B*`, then `Test( B* )`, which returns a `Yes`, would get selected; otherwise, `Test( ... )`, which returns a `No`, would get selected.

The obvious next step, then, is to check which overload would get selected:

```
sizeof(Test(static_cast<D*>(0))) == sizeof(Yes)
```

This expression, still evaluated entirely at compile time, will yield `1` if a `D*` can be converted to a `B*`, and `0` otherwise. That's pretty much all we want to know, because a `D*` can be converted to a `B*` if and only if `D` is derived from `B`, or `D` is the same as `B`.[5]

So now that we've calculated what we need to know, we just need to store the result someplace. The said "someplace" has to be a place that can be set and the value used, all at compile time. Fortunately, an `enum` fits the bill nicely:

```
enum { Is = sizeof(Test(static_cast<D*>(0))) == sizeof(Yes) };
```

In this case, for our `Cloneable` purposes, we don't care if `D` and `B` are the same type. We just want to know whether a `D` can be used polymorphically as a `B`, and that's what's being tested in `IsDerivedFrom1`. It's trivially true that a `B` can be used as a `B`.

That's it. We can now use this facility to help build an answer to the question, to wit:

```
// Example 4-3(a), continued: Using IsDerivedFrom1
// helper to enforce derivation from Cloneable
//
template<typename T>
class X
{
  bool ValidateRequirements() const
  {
    // typedef needed because otherwise the , will be
    // interpreted as delimiting macro parameters to assert
    typedef IsDerivedFrom1<T, Cloneable> Y;

    // a runtime check, but one that can be turned
    // into a compile-time check without much work
    assert( Y::Is );

    return true;
  }

public:
  // in X's destructor (easier than putting it
  // in every X constructor):
  ~X()
  {
  assert( ValidateRequirements() );
  }

  // ...
};
```

5. Or `B` happens to be `void`.

Requiring Inheritance, Take 2: `IsDerivedFrom2` Constraints Base Class

By now you've probably noticed that we could use Stroustrup's approach to create a functionally equivalent version, which is syntactically nicer:

```
// Example 4-3(b): An IsDerivedFrom2 constraints base class
//
// Advantages: Compile-time evaluation
//             Simpler to use directly
// Drawbacks:  Not directly usable for compile-time value test
//
template<typename D, typename B>
class IsDerivedFrom2
{
  static void Constraints(D* p)
  {
    B* pb = p;
    pb = p; // suppress warnings about unused variables
  }

protected:
  IsDerivedFrom2() { void(*p)(D*) = Constraints; }
};

// Force it to fail in the case where B is void
template<typename D>
class IsDerivedFrom2<D, void>
{
  IsDerivedFrom2() { char* p = (int*)0; /* error */ }
};
```

Now the check is much simpler:

```
// Example 4-3(b), continued: Using IsDerivedFrom2
// constraints base to enforce derivation from Cloneable
//
template<typename T>
class X : IsDerivedFrom2<T,Cloneable>
{
  // ...
};
```

Requiring Inheritance, Take 3: A Merged `IsDerivedFrom`

The main advantage of `IsDerivedFrom1` over `IsDerivedFrom2` is that `IsDerived From1` provides an `enum` value that's generated, and can be tested, at compile time. This isn't important to the `class X` example shown here, but it will be important in

the following section, when we want to switch on just such a value to select different traits implementations at compile time. On the other hand, `IsDerivedFrom2` provides significant ease of use for the common case, when we just need to place a requirement on a template parameter to ensure some facility will exist, but without doing anything fancy, such as selecting from among alternative implementations. We could just provide both versions, but the duplicated and similar functionality is a problem, especially for naming. We can't do much better to distinguish them than we have done, namely by tacking on some wart to make the names different, so that users would always have to remember whether they wanted `IsDerivedFrom1` or `IsDerivedFrom2`. That's ugly.

Why not have our cake and eat it, too? Let's just merge the two approaches:

```
// Example 4-3(c): An IsDerivedFrom constraints base
// with testable value
//
template<typename D, typename B>
class IsDerivedFrom
{
  class No { };
  class Yes { No no[2]; };

  static Yes Test( B* ); // not defined
  static No Test( ... ); // not defined

  static void Constraints(D* p) { B* pb = p; pb = p; }

public:
  enum { Is = sizeof(Test(static_cast<D*>(0))) == sizeof(Yes) };

  IsDerivedFrom() { void(*p)(D*) = Constraints; }
};
```

Selecting Alternative Implementations

The solutions in 3(a) are nice and all, and they'll make sure `T` must be a `Cloneable`. But what if `T` isn't a `Cloneable`? What if there were some alternative action we could take? Perhaps we could make things even more flexible—which brings us to the second part of the question.

b) […] provide an alternative implementation if `T` is derived from `Cloneable`, and work in some default mode otherwise.

To do this, we introduce the proverbial "extra level of indirection" that solves many computing problems. In this case, the extra level of indirection takes the form of a helper template: `X` will use `IsDerivedFrom` from Example 4-3(c), and use partial specialization of the helper to switch between "is-`Cloneable`" and "isn't-`Cloneable`" implementations. (Note that this requires the compile-time testable value from

IsDerivedFrom1, also incorporated into IsDerivedFrom, so that we have something we can test in order to switch among different implementations.)

```
// Example 4-3(d): Using IsDerivedFrom to make use of
// derivation from Cloneable if available, and do
// something else otherwise.
//
template<typename T, int>
class XImpl
{
  // general case: T is not derived from Cloneable
};

template<typename T>
class XImpl<T, 1>
{
  // T is derived from Cloneable
};

template<typename T>
class X
{
  XImpl<T, IsDerivedFrom<T, Cloneable>::Is> impl_;
  // ... delegates to impl_ ...
};
```

Do you see how this works? Let's work through it with a quick example:

```
class MyCloneable : public Cloneable { /*...*/ };

X<MyCloneable> x1;
```

X<T>'s impl_ has type:

```
XImpl<T, IsDerivedFrom<T, Cloneable>::Is>
```

In this case, T is MyCloneable, and so X<MyCloneable>'s impl_ has type:

```
XImpl<MyCloneable, IsDerivedFrom<MyCloneable, Cloneable>::Is>
```

which evaluates to

```
XImpl<MyCloneable, 1>
```

which uses the specialization of XImpl that makes use of the fact that MyCloneable is derived from Cloneable. But what if we instantiate X with some other type? Consider:

```
X<int> x2;
```

Now T is int, and so X<int>'s impl_ has type

```
XImpl<MyCloneable, IsDerivedFrom<int, Cloneable>::Is>
```

which evaluates to

```
XImpl<MyCloneable, 0>
```

which uses the unspecialized `XImpl`. Nifty, isn't it? It's not even hard to use, once written. From the user's point of view, the complexity is hidden inside `X`. From the point of view of `X`'s author, it's just a matter of directly reusing the machinery already encapsulated in `IsDerivedFrom`, without needing to understand how the magic works.

Note that we're not proliferating template instantiations. Exactly one `XImpl<T,...>` will ever be instantiated for any given `T`, either `XImpl<T,0>` or `XImpl<T,1>`. Although `XImpl`'s second parameter could theoretically take any integer value, we've set things up here so that the integer can only ever be `0` or `1`. (In that case, why not use a `bool` instead of an `int`? The answer is, for extensibility: It doesn't hurt to use an `int`, and doing so allows additional alternative implementations to be added easily in the future as needed—for example, if we later want to add support for another hierarchy that has a `Cloneable`-like base class with a different interface.)

Requirements versus Traits

4. Is the approach in #3 the best way to require/detect the availability of a `Clone()`? Describe alternatives.

The approach in Question #3 is nifty, but I tend to like traits better in many cases—they're about as simple (except when they have to be specialized for every class in a hierarchy), and they're more extensible as shown in Items 30 and 31.

The idea is to create a traits template whose sole purpose in life, in this case, is to implement a `Clone()` operation. The traits template looks a lot like `XImpl`, in that there'll be a general-purpose unspecialized version that does something general-purpose, as well as possibly multiple specialized versions that deal with classes that provide better or just different ways of cloning.

```
// Example 4-4: Using traits instead of IsDerivedFrom
// to make use of Cloneability if available, and do
// something else otherwise. Requires writing a
// specialization for each Cloneable class.
//
template<typename T>
class XTraits
{
public:
  // general case: use copy constructor
  static T* Clone( const T* p ) { return new T( *p ); }
};

template<>
class XTraits<MyCloneable>
{
```

```
public:
  // MyCloneable is derived from Cloneable, so use Clone()
  static MyCloneable* Clone( const MyCloneable* p )
  {
    return p->Clone();
  }
};

// ... etc. for every class derived from Cloneable
```

X<T> then simply calls XTraits<T>::Clone() where appropriate, and it will do the right thing.

The main difference between traits and the plain old XImpl shown in Example 4-3(b) is that, with traits, when the user defines some new type, the most work that has to be done to use it with X is external to X —just specialize the traits template to "do the right thing" for the new type. That's more extensible than the relatively hard-wired approach in #3 above, which does all the selection inside the implementation of XImpl instead of opening it up for extensibility. It also allows for other cloning methods, not just a function specifically named Clone() that is inherited from a specifically named base class, and this too provides extra flexibility.

For more details, including a longer sample implementation of traits for a very similar example, see Item 31, Examples 31-2(d) and 31-2(e).

Note: The main drawback of the traits approach above is that it requires individual specializations for every class in a hierarchy. There are ways to provide traits for a whole hierarchy of classes at a time, instead of tediously writing lots of specializations. See [Alexandrescu00b], which describes a nifty technique to do just this. His technique requires minor surgery on the base class of the outside class hierarchy—in this case, Cloneable.

Inheritance versus Traits

5. How useful is it for a template to know that its parameter type T is derived from some other type? Does knowing about such derivation give any benefit that couldn't also be achieved without the inheritance relationship?

There is little extra benefit a template can gain from knowing that one of its template parameters is derived from some given base class that it couldn't gain more extensibly via traits. The only real drawback to using traits is that traits can require writing lots of specializations to handle many classes in a big hierarchy, but there are techniques that mitigate or eliminate this drawback.

A principal motivator for this Item was to demonstrate that "using inheritance for categorization in templates" is perhaps not as necessary a reason to use inheritance as some have thought. Traits provide a more general mechanism that's more extensible when it comes time to instantiate an existing template on new types—such as types that come from a third-party library—that may not be easy to derive from a foreordained base class