GENERIC<PROGRAMMING>

Traits: The `else-if-then` of Types

Andrei Alexandrescu

Andrei Alexandrescu is a Development Manager at RealNetworks Inc., based in Seattle, WA. He may be contacted at

What are traits, and why do people keep referring to them as an important technique for generic programming in C++?

In short, traits are important because they allow you to make compile-time decisions based on types, much as you would make runtime decisions based on values. Better still, by adding the proverbial "extra level of indirection" that solves many software engineering problems, traits let you take the type decisions out of the immediate context where they are made. This makes the resulting code cleaner, more readable, and easier to maintain. If you apply traits correctly, you get these advantages without paying the cost in performance, safety, or coupling that other solutions might exact.

AN EXAMPLE Traits are a tool that's not just for hard-core generic programming. I hope the following example will convince you that they can be of great help in providing better solutions to very specific problems.

Suppose you're writing a relational database application. For database access, it's likely you will use the database vendor's native API library. Of course, it's not long before you feel compelled to write some wrapper functions around the primitive API, both to reduce its clumsiness and to better fit it to the problem at hand. This is where life gets interesting.

Typically, such an API will provide some fundamental means of transferring raw data from a cursor (a.k.a. rowset or query result) to memory. So let's try to write a higher-level function that extracts a value from a column without exposing all the low-level details. It might look something like Example 1: (I'll gloss over some details of the hypothetical database API. The API-defined names start with "`db_`" and "`DB_`".)

```
// Example 1: Wrapping a raw cursor int fetch
// operation.
// Fetch an integer from the
//     cursor "cr"
//     at column "col"
//     in the value "val"
void FetchIntField(db_cursor& cr,
    unsigned int col, int& val)
{
    // Verify type match
    if (cr.column_type[col] != DB_INTEGER)
        throw std::runtime_error(
        "Column type mismatch");
    // Do the fetch
    db_integer temp;
    if (!db_access_column(&cr, col))
        throw std::runtime_error(
        "Cannot transfer data");
    memcpy(&temp, cr.column_data[col],
        sizeof(temp));
    // Required by the DB API for cleanup
    db_release_column(&cr, col);
    // Convert from the database native type to int
    val = static_cast<int>(temp);
}
```

This is the kind of interface function that all of us have had to write at one time or another: messy, highly imperative, dealing with lots of low-level details—and this is only a simplified example. `FetchIntField` is an abstraction that creates higher-level functionality—a caller can fetch an integer from a cursor at a column without worrying about many details.

Because this function is so useful, we would like to reuse it as much as possible. But how? Well, one important generalization would be to make it work with types other than `int`. To do that, we need to understand the `int`-specific parts of the code. But first, what do `DB_INTEGER` and `db_integer` mean, and where do they come from? As part of their APIs, relational database vendors typically provide some type-mapping helpers; they define a symbolic constant for each type they support, and some `typedefs` or simple

structures that map database types to C/C++ types.

Looking into our fictitious database API header, we might find something like this:

```
#define DB_INTEGER 1
#define DB_STRING 2
#define DB_CURRENCY 3
...
typedef long int db_integer;
typedef char     db_string[255];
typedef struct {
    int integral_part;
    unsigned char fractionary_part;
} db_currency;
...
```

As our first step to generalization, let's try to write a `FetchDoubleField`, a function that fetches a `double` value from a cursor. The type mapping provided by the database is `db_currency`, but we'd like to operate with it in the form of a `double`. `FetchDoubleField` looks much like `FetchIntField`, with a couple of twists. In Example 2, the edits needed to transform `FetchIntField` into `FetchDoubleField` are shown in bold:

```
// Example 2: Wrapping a raw cursor double fetch operation.
//
void FetchDoubleField(db_cursor& cr, unsigned int col, double& val)
{
    if (cr.column_type[col] != DB_CURRENCY)
        throw std::runtime_error("Column type mismatch");
    if (!db_access_column(&cr, col))
        throw std::runtime_error("Cannot transfer data");
    db_currency temp;
    memcpy(&temp, cr.column_data[col], sizeof(temp));
    db_release_column(&cr, col);
    val = temp.integral_part + temp.fractionary_part / 100.;
}
```

The function looks very similar to `FetchIntField`. Because we don't want to maintain such a function for each type the database supports, it would be nice to consolidate `FetchIntField`, `FetchDoubleField`, and any other fetching functions, in one place.

Let's enumerate the differences between the two pieces of code:

- The input type: `double` as opposed to `int`.
- Another type, used internally: `db_currency` versus `db_integer`.
- A constant value: `DB_CURRENCY` versus `DB_INTEGER`.
- An algorithm: An expression versus the `static_cast`.

The mapping between the input types (`int/double`) and the other entities doesn't seem to follow any obvious rule. It's quite arbitrary, and depends on what conventions and definitions the database vendor happens to have provided. Templates by themselves won't help, because they don't offer such sophisticated type inference out of the box. We can't just associate various types using inheritance because we're dealing with primitive types. Given the API constraints and the low-level character of the problem, it looks like we simply cannot solve the fetch problem generically, yet we can.

ENTER TRAITS Traits are a technique intended to solve problems like the aforementioned: to consolidate pieces of code that, depending upon a type (in our case, `int` or `double`), sport slight variations in terms of structure and/or behavior.

To achieve this end, traits rely on *explicit template specialization.* This feature allows you to provide a separate implementation of a class template for a specific type, as shown in Example 3:

```
// Example 3: A traits example
//
template <class T>
class SomeTemplate
{
```

```
    // generic implementation (1)
    ...
};
template <>
class SomeTemplate<char>
{
    // implementation tuned for char (2)
    ...
};
...
SomeTemplate<int> a;      // will use (1)
SomeTemplate<char*> b;    // will use (1)
SomeTemplate<char> c;     // will use (2)
```

If you instantiate the `SomeTemplate` class template with `char`, the compiler will use the explicitly specialized version of the template. For any other type, the compiler will instantiate the generic template. This looks like an `if` statement, only it's driven by types. Because usually the most general template (the `else` part) is defined first, the `if` statement is a bit backwards. You may even decide not to provide the general template at all; then, only the specialized instantiations will be usable, and all others will yield compile-time errors.

Now, let's link this language feature with the problem at hand. We want to implement a template function `FetchField`, parameterized with the type fetched. Inside that function, we need to be able to reason: "I will use a symbolic integral constant named, say, `TypeId`. Its value must be `DB_INTEGER` if the type to `fetch` is `int`, and `DB_CURRENCY` if the type to fetch is a `double`. Otherwise, a compile-time error must occur." Similarly, depending on the type being fetched, we also need to manipulate different types (`db_integer/db_currency`) and different conversion algorithms.

Let's solve the problem by leveraging explicit template specialization. We can have `FetchField` defer the aforementioned variations to a template class, and explicitly specialize that template class for `int` and `double`. Each specialization provides uniform names for these variations.

```
// Example 4: Defining DbTraits
//
// Most general case not implemented
template <typename T> struct DbTraits;
// Specialization for int
template <>
struct DbTraits<int>
{
    enum { TypeId = DB_INTEGER };
    typedef db_integer DbNativeType;
    static void Convert(DbNativeType from, int& to)
    {
        to = static_cast<int>(from);
    }
};
// Specialization for double
template <>
struct DbTraits<double>
{
    enum { TypeId = DB_CURRENCY };
    typedef db_currency DbNativeType;
    static void Convert(const DbNativeType& from, double& to)
    {
        to = from.integral_part + from.fractionary_part / 100.;
    }
};
```

Now if you write `DbTraits<int>::TypeId`, you get `DB_INTEGER`. For `DbTraits<double>::TypeId`, you get `DB_CURRENCY`. For `DbTraits<`**`anything else`**`>::TypeId`, you get a compile-time error, because the template class itself hasn't been defined (only declared).

Does all this ring a bell yet? Let's see how to implement a generic `FetchField` function by leveraging `DbTraits`. We defer all the variations—the enumerated type, the database native type, and the conversion algorithm—to `DbTraits`. Our function now contains only the common part of `FetchIntField` and `FetchDoubleField`.

```
// Example 5: A generic, extensible FetchField using DbTraits
//
template <class T>
void FetchField(db_cursor& cr, unsigned int col, T& val)
{
    // Define the traits type
    typedef DbTraits<T> Traits;
    if (cr.column_type[col] != Traits::TypeId)
        throw std::runtime_error("Column type mismatch");
    if (!db_access_column(&cr, col))
        throw std::runtime_error("Cannot transfer data");
    typename Traits::DbNativeType temp;
    memcpy(&temp, cr.column_data[col], sizeof(temp));
    Traits::Convert(temp, val);
    db_release_column(&cr, col);
}
```

That's it—we just implemented and used a traits class template.

Traits rely on explicit template specialization to pull out type-related variations from code, and to wrap them under a uniform interface. The interface can contain anything a C++ class can: nested types, member functions, member variables. Templated client code will indirect through the interface advertised by the traits class template.

Such a traits interface is usually implicit—a mutual convention between the traits class template and the code that uses it. Implicit interfaces are more relaxed than function signatures. For instance, although `DbTraits<int>::Convert` and `DbTraits<double>::Convert` have quite different signatures, both will work because they honor the calling code.

A traits template class establishes a uniform interface over a type-driven collection of design choices that make sense at a high level, but differ in their implementation details (types, values, algorithms). Because traits capture a concept, a set of coherent decisions, they are likely to be reused in similar contexts. In this example, we'll be able to reuse `DbTraits` in code that writes back to a cursor or otherwise manipulates database values.

*Definition: A **traits template** is a template class, possibly explicitly specialized, that provides a uniform symbolic interface over a coherent set of design choices that vary from one type to another.*

TRAITS AS ADAPTERS Enough database stuff. Let's use a more popular example—smart pointers.

Let's assume you are developing a `SmartPtr` template class. The nice thing about smart pointers is that they can automate memory management while otherwise behaving much like regular pointers. The not-so-nice thing about them is that they are highly nontrivial pieces of code. (There are C++ techniques related to smart pointers really indistinguishable from black magic.) This fact has an important practical effect: You'd prefer, if at all possible, to develop a single good, industrial-strength implementation for most of your smart pointer needs. In addition, you often cannot modify a class to suit your smart pointer's conventions, so your `SmartPtr` has to be flexible.

Many class hierarchies use reference counting, and provide appropriate functions, for managing object lifetime. However, because there is no standardized way of implementing or exposing reference counting, each C++ library vendor implements it with slight differences in syntax and/or semantics. Say, for instance, that in your application you have these two interfaces:

- Most of your own classes implement the `RefCounted` interface:

```
class RefCounted
{
public:
    void IncRef() = 0;
    bool DecRef() = 0; // if you DecRef() to zero
        // references, the object is destroyed
        // automatically and DecRef() returns true
    virtual ~RefCounted() {}
};
```

- A `Widget` class, provided by a third-party vendor, uses a slightly different interface:

```
class Widget
{
```

```
    public:
        void AddReference();
        int RemoveReference(); // returns the remaining
            // number of references; it's the client's
            // responsibility to destroy the object
        ...
    };
```

Because you don't want to maintain two smart pointer classes, you'd like to share a unique `SmartPtr` back end for your own hierarchies as well as for `Widget`-based hierarchies. A traits-based solution collects the two slightly different interfaces under a unified syntactic and semantic interface. We can define the general template to support the `RefCounted` interface, and we specialize it for `Widget`, like so:

```
// Example 6: Reference counting traits
//
template <class T>
class RefCountingTraits
{
    static void Refer(T* p)
    {
        p->IncRef(); // assume RefCounted interface
    }
    static void Unrefer(T* p)
    {
        p->DecRef(); // assume RefCounted interface
    }
};
template <>
class RefCountingTraits<Widget>
{
    static void Refer(Widget* p)
    {
        p->AddReference(); // use Widget interface
    }
    static void Unrefer(Widget* p)
    {
        // use Widget interface
        if (p->RemoveReference() == 0)
            delete p;
    }
};
```

Now, inside `SmartPtr`, we use `RefCountingTraits` like this:

```
template <class T>
class SmartPtr
{
private:
    typedef RefCountingTraits<T> RCTraits;
    T* pointee_;
public:
    ...
    ~SmartPtr()
    {
        RCTraits::Unrefer(pointee_);
    }
};
```

In the preceding example, however, you might argue that you could directly specialize `SmartPtr`'s destructor and constructors for class `Widget`. Instead of using template specialization for the trait, you can use template specialization for `SmartPtr` *itself*, and get rid of an extra class. Although this is true for this case specifically, specializing (parts of) `SmartPtr` itself has some drawbacks you should be aware of:

- The technique doesn't scale. Add a second template parameter to `SmartPtr`, and everything goes south. You will not be able to

partially specialize `SmartPtr<T, U>` member functions for `Widget` and any `U`. Incidentally, smart pointers are good candidates for lots of template parameters (refer to Van Horn's Smart Pointers[1] for a copious example).

- The resulting code might be less clear. A trait has a name, and nicely collects related things. Traits therefore lead to code that's much easier to understand. By comparison, the direct specialization of some `SmartPtr` member functions looks more like a hack.
- You cannot have multiple traits for the same type.

The same problems cripple a solution based on inheritance, to say nothing of inheritance's own caveats.[2] Inheritance is just too heavy a tool to express this kind of variation. Furthermore, the classic alternative to inheritance, namely containment, is unnecessarily clumsy and verbose. By contrast, the traits-based solution is simple, clear, and concise. It just does what's needed: no more, no less.

*An important use of traits is as **"interface glue"**—universal non-intrusive adapters. If various classes implement a given concept in slightly different ways, traits can fit those implementations to a common interface.*

MULTIPLE TRAITS FOR A GIVEN TYPE Now, let's say that everybody likes and uses your `SmartPtr` class template, until mysterious bugs start occurring in your multithreaded application. You discover the culprit is `Widget`, whose reference-counting functions are not thread-safe. You must then serialize calls to `Widget::AddReference` and `Widget::RemoveReference` yourself, and the best place to do this is in its `RefCountingTraits` specialization. You would patch your code like so:

```
// Example 7: Patching Widget's traits for thread safety
//
template <>
class RefCountingTraits<Widget>
{
    static void Refer(Widget* p)
    {
        Sentry s(lock_); // serialize access
        p->AddReference();
    }
    static void Unrefer(Widget* p)
    {
        Sentry s(lock_); // serialize access
        if (p->RemoveReference() == 0)
            delete p;
    }
private:
    static Lock lock_;
};
```

Unfortunately, as you recompile and retest the application, it runs correctly, but at a snail's pace. By profiling your code you notice that you just introduced a major bottleneck for the whole program. Only a few `Widget` objects are accessed from multiple threads. The rest of them are used—and rather heavily—only by single-threaded code.

What you need is to tell the compiler to use the single-threaded traits or the multithreaded traits as your needs dictate. The majority of your code will use the single-threaded traits.

How can you tell the compiler which traits to use? By passing the traits as an *additional template parameter* to `SmartPtr`. The traits parameter defaults to the old traits template, instantiated with the subject type:

```
template <class T, class RCTraits = RefCountingTraits<T> >
class SmartPtr
{
    ...
};
```

You leave the `RefCountingTraits<Widget>` specialization with the old, single-threaded code. You move the multithreaded reference-counting management to a separate class:

```
class MtRefCountingTraits
{
```

```
    static void Refer(Widget* p)
    {
        Sentry s(lock_); // serialize access
        p->AddReference();
    }
    static void Unrefer(Widget* p)
    {
        Sentry s(lock_); // serialize access
        if (p->RemoveReference() == 0)
            delete p;
    }
private:
    static Lock lock_;
};
```

Then you use `SmartPtr<Widget>` for single-threaded purposes and `SmartPtr<Widget, MtRefCountingTraits>` for multithreaded purposes. That's about it. As Scott Meyers might have put it, "If you're not having fun yet, you just don't know how to party."

When only a trait per type was needed, a specialized template was just enough. Now a type can have multiple traits, depending on some constraints. Consequently, the traits must be injected from the outside instead of being "computed" from a single traits template. A common idiom is to provide a traits class as the last parameter of a template. The trait defaults to the traits calculated from a traits template, as shown.

*Definition: A **traits class** (as opposed to a traits template) is either an instantiation of a traits template, or a separate class that exposes the same interface as a traits template instantiation.*

TEASER Our discussion of traits has just begun. In future installments I'll discuss traits with state, general-purpose traits, and hierarchy-wide traits—traits that you can define in one shot not just for a class, but for a whole hierarchy or subhierarchy.

But first, a homework question. There is still a source of inefficiencies in our multithreaded code. Where is it, and how can we get rid of it?

Acknowledgments Scott Meyers came up with the very idea and the title of this month's column. Sorin Jianu and Herb Sutter helped with excellent reviews.

References

1. Van Horn, K. S. Kevin S. *Van Horn's Smart Pointers,* http://www.xmission.com/~ksvhsoft/code/smart_ptrs.html.
2. Sutter, H. "Uses and Abuses of Inheritance, Part 2," *C++ Report,* 11(1): 58–61, Jan. 1999.