Type Erasure in C++: The Glue between Object-Oriented and Generic Programming

Presentation Abstract (Submission for MPOOL 07)

by
Thomas Becker
Zephyr Associates, Inc.
mpool@thbecker.net

Summary

C++ is a multi-paradigm language. The two main paradigms in C++ are object-oriented programming and generic programming. Many real-world C++ software projects use these two paradigms side by side. This creates considerable tension due to the fact that object-oriented programming is largely based on the judicious choice of types and hierarchies, while generic programming tends to cause an abundance of unrelated types. We show how type erasure can reconcile these conflicting tendencies. We present iterator type erasure as a concrete example that we have implemented and that is being used in production code at Zephyr Associates, Inc..

The Tension between Object-Oriented and Generic Programming

The most widely used programming paradigms in C++ are object-oriented programming (classes, objects, and runtime polymorphism come to mind) and generic programming (algorithms, templates, and compile time polymorphism come to mind). A cornerstone of object-oriented programming is the judicious choice of types and hierarchies. Much of what has been written on object-oriented programming is centered around "object-oriented analysis and design." The gist of this approach is that the classes—that is, the types—in an object-oriented piece of software should be such that they model actual entities in the domain of the application.

In generic programming, on the other hand, one tends to see a multitude of types that exist for pure technical reasons and do not express much, if anything, related to the application domain. We will present examples of this phenomenon below.

When object-oriented and generic programming are used side-by-side, this contradiction is no longer just philosophical. It will create very practical annoyances, because the object-oriented side of the code will want to give uniform treatment to objects that represent the same domain entity, e.g., place them in containers, pass them to interfaces that use runtime polymorphism, and so on. If unrelated types abound, none of this can be done easily and naturally.

Using Type Erasure to Resolve the Conflict

From what has been said so far, it is clear how to resolve the tension that occurs when object-oriented and generic programming meet: given a set `T` of unrelated types, one must come up with a unifying type `S` such that

- the interface of `S` expresses the commonality of the types `T`$\in$ `T`, and
- if `s` is a variable of type `S`, then any object of type `T`$\in$ `T` can be assigned to `s`.

This is exactly what type erasure does ([1]). The most radical example of type erasure is `boost::any` ([2]), which can hold objects of any type. Except for the basics

such as construction and assignment, the interface of this class is of course empty, because there is no commonality between the types that it unifies.

`boost::any` also exemplifies the most common implementation technique for type erasure. The idea is to graft runtime polymorphism onto the set of types that are to be unified. To this end, one defines a templatized wrapper class that holds the object whose type is to be erased. This class template derives from a common abstract base class. The actual type-erasing class holds a pointer to this abstract base class.

Another widely used example of type erasure is `boost::function` ([3]). Here, the type erasing class unifies all types that are callable as functions and—simplifying only slightly—have the same function signature.

The Need for C++ Iterator Type Erasure

We feel strongly that type erasure will become more common as C++ continues to evolve as a multi-paradigm language. One situation where the absence of type erasure has caused us grief in commercial software engineering is with C++ iterators.

The problem occurs when pairs of iterators are used to pass sequences of objects between interfaces. Suppose you have a class `X` that internally holds a collection of objects, let us say doubles, in an `std::vector<double>`. The class wishes to expose a method that allows clients to iterate over the collection and retrieve the elements. The standard way of doing this, endorsed by the STL, is to expose a typedef like

```
typedef std::vector<double>::const_iterator XIterator;
```

and a pair of iterators of type `XIterator` that point to the begin and end position of the vector. Purists of object-oriented programming cringe at this, because the type `std::vector<double>`, which is purely an implementation detail of the class `X`, gets exposed in the interface of `X`.

Were this just a philosophical issue, one could dismiss it in the name of pragmatism. However, the practical annoyances caused by this are considerable. Suppose, for example, that the following occurs: there is a change in the implementation of `X` that causes the vector to be filled in the opposite order than before, but clients are not supposed to see this change. There is no doubt as to how the STL wants us to deal with this situation: expose a pair of const reverse iterators instead of ordinary iterators. But alas, ordinary iterators and reverse iterators are of unrelated type. Therefore, the implementation change spills into the interface. Clients of the class `X` will have to recompile.

Another conceivable situation is that the class `X` holds a collection of objects, as before, or perhaps a small number of collections, but it wishes to expose these collections in several variations, say, forward and backward, then with the objects processed in a number of different ways using Boost's transform iterators, then with the sequence filtered in a number of different ways using Boost's filter iterators, and so on and so forth. All that clients ever want to see are sequences of objects of one and the same type. Instead, they have to deal with a multitude of iterator types that express nothing to them, because the differences in type reflect implementation detail only.

A conspicuous and ugly side effect of this abundance of iterator types is the fact that the interface of the class `X` becomes fat. That is because each pair of iterators is retrieved via a different function. Soon clients will ask (and they have done so, rather adamantly, in real life) to get, instead of a multitude of functions, a single function which returns a pair of iterators and takes an `enum` value as an argument to specify which sequence is requested. This cannot be done in a natural way as long as the types of the iterators involved are all different.

Apart from the unwanted multitude of types, there is another problem with exposing iterators such as vector iterators directly to clients. Remember that the intent was to allow clients to retrieve the objects in the exposed sequences via iteration. The interface of an STL vector iterator, however, allows clients to do many other things. For example, someone could look at `end - begin` to determine the length of a sequence. Now if the class `X` internally replaces the vector with a list, which is in perfect keeping with the original intent, clients will see their code break.

It is clear that all these problems could be solved if there were a type-erasing iterator class that could

- hold any one of the different iterators that our class `X` uses internally, and
- cut down the interface to forward iteration, regardless of what other capabilities the internal iterators may have.

The Cost of C++ Iterator Type Erasure

It is clear that type erasure for iterators, like every type erasure, will come at the cost of a level of indirection and a virtual function call. Normally, this is not considered much of an issue because type erasure is meant to be used as a bridge between compile time and runtime polymorphism. However, in the case of iterators, there are often higher expectations in terms of performance. Therefore, the use of iterators with type erasure has to be weighed carefully against performance requirements.

Implementation of C++ Iterator Type Erasure

Motivated by the type of problems described in the previous section, we have written a class template named `any_iterator` ([4]) which provides type erasure for C++ iterators. As with any type-erasing class, the main issue to be addressed is the granularity of the type erasure. The following approach has worked well for us:

- `any_iterator` is a class template. There is one instantiation of this class template for each set of iterator traits. (More precisely, `any_iterator` has the exact same template argument list as `boost::iterator_facade`.)
- Suppose that `iterator` is some concrete iterator type, and furthermore, `any_iterator_inst` is an instantiation of the `any_iterator` class template. Then an object of type `iterator` can be assigned to an variable of type `any_iterator_inst` if and only if the following holds: the iterator traits of `iterator` convert elementwise to the iterator traits of `any_iterator_inst`.

The main ingredients of the implementation of `any_iterator` are

- the standard type erasure implementation technique as exemplified by `boost::any`,
- `boost::iterator_facade`, and
- one of the CRTP's ([5]), where a derived class passes itself as a template argument to its base class.

The `any_iterator` class template is currently used in production code.

Acknowledgements

I am indebted to Don Harriman and Thomas Witt for inspiring conversations on this subject. In particular, it was Don Harriman who first pointed out how the simple act of exposing a pair of vector iterators from a class is a blatant violation of OO principles.

References

[1] In their book on C++ template metaprogramming, Dave Abrahams and Aleksey Gurtovoy define type erasure as "the process of turning a wide variety of types with a common interface into one type *with that same interface*."

[2] http://www.boost.org/doc/html/any.html

[3] http://www.boost.org/doc/html/function.html

[4] The `any_iterator` package is downloadable from here.

[5] James Coplien, *A curiously recurring template pattern*, C++ Report, February 1995.