Type erasure — Part III

Posted on December 11, 2013 by Andrzej Krzemieński

Update. My advice about using Boost.Pointer Container library was based on the false understanding of the library. It could even cause bugs. This has now been corrected.

This is the third part of the series of posts on type erasure. We will focus on the motivation for choosing this technique and see some practical use cases.

Who needs type erasure?

This is a very important question. Is there any (pun not intended) point in using this type erasure? After all, one of the main motivations for choosing C++ is uncompromised performance and, as we already said, type erasure compromises performance by performing indirect function calls. Plus the non-trivial effort to create a new interface...

But because the question is important — to you — you will have to make the choice whether and when to use type erasure or not. This series of posts is just to provide a list of choices.

Here is what I learned from my experience. There are a couple of trade-offs to be made when using or not type erasure: run-time efficiency vs compilation time, run-time efficiency vs binary size. The choice is not obvious.

Even in programs that need to be fast, not every part of the program needs to be fast. Some portions, like the interaction with the user, can be slow, and for these parts you can apply different trade-offs.

Even if you decide to go with type erasure, there is a cost of implementing and maintaining the interface. Using type erasure will more likely pay off if this is a common abstraction used in a lot of places in your program. In my projects I often use `std::function` because the cost of maintaining the interface is zero. It is a good replacement for all these OO-like interfaces with one virtual function: `Receiver` with sole member `receive()`, `Processor` with `process()`, `Executor` with `execute()` — I do not have to define these interfaces anymore. It saves my time.

It is only once in my professional experience that I had to build my own type erased interface. My project used custom iterators all over the the place. Well, they were iterators aware of having reached the end of the collection, so they were more like ranges, but more efficient than Boost ranges, because they weren't a pair of STL iterators. My interface was re-used in many many places in the code, so the effort payed off, but I used neither of the type erasure libraries. I wrote it from scratch. You can find a tutorial for doing it for instance in Sean Parent's presentation. But the libraries may produce faster interfaces. For instance, they will make use of *small buffer optimization* (for small erased types) and therefore will cause no free-store (heap) allocations. This is likely to make type erased interfaces faster than OO-style interfaces, in case you have to return the erased types. Returning by value is often faster than returning by pointer, in case the latter requires heap allocation of the pointee.

Type-erased deleter

The example given in Part I was not a real life case (the goal was only to compare different techniques). So, in case you feel type erasure is something useless, let me give you some practical examples. In general, I do not like `std::shared_ptr`. There are certain rare situations where it is just the right tool for the job, but far more often I have seen people overuse it for things like bringing garbage collection to C++, because they lost track of their memory,

because they are not familiar with the capabilities of the destructor. But `std::shared_ptr` has one very useful feature, not connected to reference counting. I was once surprised when compiling the following code:

```
#include <memory>
class Toy; // only forward declaration

std::shared_ptr<Toy> fwd (std::shared_ptr<Toy> p)
{
  if (!p) throw int{};
  return p;
}
```

Do you know what surprised me? That it compiles! But why? If you try the same with `std::unique_ptr` it will not work. It will tell you something like "default deleter cannot delete an incomplete type." It makes sense: smart pointer's destructor may need to call `Toy`'s destructor, so we have to see the declaration of the destructor. So how come `shared_ptr` works? It may also need to delete its pointee. Well, you probably know the answer already: `shared_ptr`'s deleter is type-erased. Its type is something like `std::function<void(Toy*)>`. `shared_ptr` just needs to call it, and it does not care what the deleter does. Of course, upon creation of the `shared_ptr`, you have to tell it exactly how the deleter should delete the object, but once the construction is done, the type of the deleter is erased. And in our short example there is no construction involved; except move-construction, but here we only need to move the erased deleter.

This also means that different `shared_ptr`'s can have different deleters (of different type), but it does not affect the type of `shared_ptr`. this is different than the signature of `std::vector` which does depend on the type of the allocator.

So, why does `unique_ptr` not work in the same way? For performance reasons. `unique_ptr` needs to be (almost) as fast as a raw pointer in order to be an attractive alternative for raw pointers.

Our exercise is to implement a yet another smart pointer that semantically works like `unique_ptr` (has unique ownership semantics), but with a type-erased deleter. This will be done at the expense of run-time performance. This is not difficult, when we know that `unique_ptr` allows us to customize the deleter:

```
template <typename T>
using ErasedPtr = std::unique_ptr<T, std::function<void(T*)>>;
```

This solution has one problem though: when constructing these pointers, if we forget to pass the allocator, it will use the default-constructed `std::function`, which will throw when called. Not only will we throw, but also we will implement a [throwing destructor](). So, we have to create our pointers with an ugly declaration:

```
ErasedPtr<Toy> p { new Toy, [](Toy * p){delete p;} };
```

In order to fix this we will use a technique similar to what Matthew Wilson calls a *veneer*:

```
template <typename T>
struct ErasedDeleter : std::function<void(T*)>
{
  ErasedDeleter()
    : std::function<void(T*)> { [](T * p){delete p;} } {}
};
```

We derive form a Standard Library component! Many people call it unsafe: “you shouldn’t derive from classes that do not have virtual functions,” “you risk slicing.” But our use case is safe. We derive only to change the default constructor, so that it sets a different default value. We have added no member data, no virtual functions, so the memory layout of our derived type is exactly same as that of the base class. Even if we pass by value and silently cast up from the derived class to the base class, there is no slicing.

Now, we can make the second attempt at the new pointer type:

```
template <typename T>
using ErasedPtr = std::unique_ptr<T, ErasedDeleter<T>>;
```

And that's it! Note one thing: we just used `std::unique_ptr`'s second template parameter (not everyone knows of its existence) and derived from `std::function`. If I were not using term "type erasure" all over the place, one might have never noticed that we were erasing a type. "Type erasure" is more of an 'idea' or an 'idiom' or a 'pattern' rather than a language feature.

Note one other thing. If you use OO-like type erasure and use `unique_ptr` rather than raw pointers or references, you get something that is almost a value-semantic type erasure. You cannot copy them and relational operators do not work as you would expect (they compare only addresses), but you can return them by value.

Heterogeneous collections

Another use case for value-semantic type erasure is a container of objects of different types (but with the same interface). Suppose you want to have a vector of people, but you want to put types like `Contributor`, `Manager`, `Director`. How often do you need such heterogeneous containers?

One way to solve this is to first enforce inheritance hierarchy and then use a collection of smart pointers:

```
std::vector<std::unique_ptr<Person>> container;
container.push_back( std::unique_ptr<Person>{new Director} );
```

This takes care of memory management but has some negative effects. Iterators iterate over pointers: you have to dereference twice to get to the value. You really store pointers, so the sorting/searching functions compare addresses rather than values; you may really get surprised. Constness is not propagated from container to objects anymore. Also, the container is now non-copyable. Don't even think about using `shared_ptr` instead. This would

make the container depart from value semantics: modifications in one copy would affect other copies. By preventing copying (but allowing moving) `unique_ptr` at least guarantees that value semantics are not violated.

Actually there already exists a solution to all the above problems in Boost: Pointer Container Library.

```
#include <boost/ptr_container/ptr_vector.hpp>

boost::ptr_vector<Person> container;
container.push_back(new Director);
```

The philosophy behind the library is that you insert pointers to heap allocated memory (and the container takes the ownership), but you use them as though they were values.

This almost looks like *the* solution, but there is one potential problem with it. When copying this `ptr_vector` it attempts to deep-copy the elements, but it is likely to fail or to do the wrong thing. If `Person` is an abstract class, you will get a compile-time error when trying to make a copy of the vector. Otherwise, it is even worse: it will be copying assuming that it stores objects of type `Person` (rather than the type derived from it), and it will slice the objects. What you get as a copy is a vector of sliced base classes rather than the original.

There is a way to fix it though, as explained here.

The only aspect where value-semantic type erasure can beat Pointer Container Library is performance. The library requires that each pointer element is allocated on the free store (heap), whereas implementations of type erasure at least for small erased objects can do without the free store (by utilizing small buffer optimization). The usage would be probably:

```
std::vector<AnyPerson> container;
```

```
container.emplace_back(Director{});
```

And that's it for Part III. In the next part we will see a yet another type erasure technique, and try to compare all of them.

This entry was posted in programming and tagged C++11, type erasure, value semantics. Bookmark the permalink.

15 Responses to *Type erasure — Part III*

J *says:*
December 13, 2013 at 11:05 am

As a side note, regarding the safety of veneers. Here's one case (using your class, though silly in this context) which yields undefined behaviour according to the C++ standard (§5.3.5:3).

```
std::function<void(T*)> * pFunction = new ErasedDeleter;
delete pFunction;
```

Since the "veneer" is the same size as the base class I had always thought it was safe. However, I've been looking for a good explanation, do you know of one?

Reply

Andrzej Krzemieński *says:*
December 13, 2013 at 12:23 pm

Indeed, you are right. My guess would be that no-one anticipated the (perhaps too clever) trick like veneers.

Reply

J *says:*
April 10, 2014 at 9:14 am

Sorry for the late reply, but I just had an idea. If you declare a base class that is friend to shared_ptr and has the new operators as private, it should be safe in this respect to use a veneer class that derives from it, right? It means that you prevent the new operators from being called directly, but you can only call std::make_shared, which will call the right destructor without it being virtual. With C++14, I guess it can also be used with unique_ptr and make_unique. Well, of course the STL classes don't have this friendship, but you can use this technique for you own classes. What do you think?

Reply

Andrzej Krzemieński *says:*
April 10, 2014 at 10:44 am

I am not sure if it is a problem worth solving. You do not write a veneer to pass it around by pointer.

Gábor Márton *says:*
December 13, 2013 at 11:51 am

Really nice series of articles about Type Erasure!
Let's assume:
using VariantPerson = boost::variant<Contributor, Manager, Director>;
To have an even wider scope of Heterogeneous collections, It would be nice to see a comparison about std::vector<AnyPerson> and std::vector<VariantPerson>.

Reply

Andrzej Krzemieński *says:*

December 13, 2013 at 12:25 pm

Yes, that's another alternative. In fact, I intended to mention it in the next part, along with boost::any.

Reply

Evgeny Panasyuk *says:*
December 23, 2013 at 2:37 pm

If order of elements is not important, then another option is to have several vectors – one vector for each type. It is possible to automate boilerplate using Boost.Fusion:

```
poly_sequence<Contributor, Manager, Director>  seq;
push_back(seq, Contributor{1});
push_back(seq, Contributor{2});
push_back(seq, Director{});
for_each(seq, Print{});
```

Key difference from boost::variant and type-erasure is that there is no runtime dispatch per each element (at the cost of partial loss of elements ordering). In addition, elements are packed more tightly.

Here is proof-of-concept live demo: http://coliru.stacked-crooked.com/a/5535e51d210d59ec

Reply

Gustavo Totoy *says:*
December 16, 2013 at 4:32 pm

Excellent article, I have a question:
Why are you using a pointer to int in the line
ErasedPtr p { new Toy, [](int * p){delete p;} };
instead of a pointer to Toy like this

ErasedPtr p { new Toy, [](Toy * p){delete p;} }; ?
And the same question for ErasedDeleter constructor.

Reply

Andrzej Krzemieński *says:*
December 16, 2013 at 4:44 pm

That's just an oversight, which is now fixed. Thanks for pointing this out.

Reply

Oliver *says:*
January 3, 2014 at 10:04 am

Great series. Can't wait for the next one!
Hope to see some implementation details in the following articles.

Reply

Pingback: *Problematic parameters | WriteAsync .NET*

Andy Prowl *says:*
September 28, 2014 at 9:37 pm

Thank you for the great series 🙂

If you spare some time and/or are interested, I would appreciate to hear your opinion on this idea I had about making type erasure simpler at a language level: http://bit.ly/ZgruFY. There is also a discussion on std-proposals, under the thread "Virtual concepts (non-intrusive dynamic polymorphism)".

Reply

kalx *says:*
June 26, 2015 at 10:18 pm

Regarding containers that know their end, I would love to hear your take on https://github.com/keithalewis/fms/blob/master/iter/README.md.

Reply

Andrzej Krzemieński *says:*
June 27, 2015 at 7:30 am

Is it not what Ranges handle?

Reply

kalx *says:*
June 27, 2015 at 8:45 am

I'm well aware of Eric's heroic efforts on that. This is a much more modest endeavor. I thought you had written something similar and wanted to get your feedback.

Reply