Type erasure — Part II

Posted on December 6, 2013 by Andrzej Krzemieński

In the previous post, we have seen that there is a number of ways to erase the type of an object while still holding a "handle" to it and being able to make use of it. This can be summarized in the following table.

	any type?	value semantics	useful iface	simple to create
`void*`	yes	no	no	yes
OO interfaces	no	no	yes	yes
`std::function`-like	yes	yes	yes	no

While there are many ways to erase a type, I will use name *type erasure* to denote the usage of `std::function`-like value-semantic interface. This convention appears to be widely accepted in C++ community.

In this post, we will see how to create a custom type-erased interface. This is not easy, as there is no language feature for that.

Using `std::function` is easy, but this is because someöne has made an effort to implement it for us. So let's try to see how much effort it is to write our own type-erased interface. We will be using countdown counters: something that (1) can be decremented and (2) tested if the count reached zero. A simple model for such counter is type `int`: you can assign it an initial value:

```
int counter = 10;
```

decrement it:

```
--counter;
```

and test for reaching zero:

```
if (counter) {}   // still counting
if (!counter) {}  // done counting

if (--counter) {} // decrement and test
```

I realize that this may not be a convincing real-life example of a concept, but this is just to illustrate the techniques. I needed a concept that is (a) small and (b) can be 'modeled'/'implemented' by built-in types that have no member functions. Apart from an `int`, we can imagine other implementations of counters: a counter that logs each decrement, a counter that contains a list of other counters and decrements all of them when it itself is decremented.

If we had Concepts Lite (for description see here, for technical specification draft see here), especially with support for variable templates (as has been indicated in Concepts SG mailing list) we could specify the above requirements formally:

```
// maybe in C++17
template <typename C>
concept bool CountdownCounter = requires (C c) {
  --c;
  (bool(c));      // contextual conversion to bool
  { !c } -> bool; // !c must be convertible to bool
  (bool(--c));    // decrement and test

  requires std::is_copy_constructible<C>::value;
  requires std::is_copy_assignable<C>::value;
};
```

and even test if a type satisfies them:

```
// maybe in C++17
static_assert (CountdownCounter<int>, "not a counter");
```

This is a different (than OO) way of specifying interfaces: we say what expressions (including function calls) are valid on our type — not member functions that it has. This way built in types are also able to comply to interfaces and we can also use free functions in addition to member functions. If type erasure was supported by the language, we could probably bind a model to the interface with a single instruction:

```
// not in C++
ERASED<CountdownCounter> counter = int{3};
```

So, how do we do the same without language support?

Boost Concept Check Library

We may (but don't have to) specify a concept. This is not strictly necessary, but it can improve our diagnostic messages when someöne tries to bind an incorrect type to our interface. For specifying concepts, we could use

Boost Concept Check Library.

```
#include <boost/concept_check.hpp>

template <typename C>
struct CountdownCounter : boost::CopyConstructible<C>
                        , boost::Assignable<C>
{
  BOOST_CONCEPT_USAGE(CountdownCounter)
  {
    --c;
    (bool(c));   // contextual conversion to bool
    bool b = !c; // !c must be convertible to bool
    (bool(--c)); // decrement and test
  }

private:
    C c;
};
```

The library also allows us to test if a given type models (satisfies the requirements of) a given concept:

```
BOOST_CONCEPT_ASSERT((CountdownCounter<int>));          // ok
BOOST_CONCEPT_ASSERT((CountdownCounter<std::string>)); // error
```

Double parentheses are the price to be paid for using a library-based solution to concepts rather than concepts built into the language. There is a lot of macro and template meta-programming magic involved to make this library work.

Adobe.Poly

Now, to type erasure. We will use Adobe.Poly library first. We will need to declare two classes and one class template. First an *internal* interface. The users will never see it:

```
#include <adobe/poly.hpp>

struct CounterIface : adobe::poly_copyable_interface
{
  virtual void decrement() = 0;  // for operator--
  virtual bool done() const = 0; // for operator bool
};
```

This interface corresponds to our concept, but does not have to be identical, and in fact, it cannot be identical in case we are using non-member functions. Instead of names `decrement` and `done`, we could have used names `operator--` and `explicit operator bool`, and this would have even been more sane, but my goal was to stress that the names in the internal interface need not be the same as these in the *external* interface — the one that is really going to be used. Class `adobe::poly_copyable_interface` is a base interface that we extend. It provides definitions of member functions common to all value-semantic interfaces: `assign`, `clone`, `exchange` for copying and swapping, plus some other functions for querying the exact type bound to the outer interface — but this is done for us: we only need to worry about specifying the operations custom to our interface.

Next, we have to provide a class template for creating implementations of our internal interface:

```
template <typename T>
struct CounterImpl
  : adobe::optimized_storage_type<T, CounterIface>::type
{
  BOOST_CONCEPT_ASSERT((CountdownCounter<T>));

  using base_t = typename
    adobe::optimized_storage_type<T, CounterIface>::type;
```

```
9
10    CounterImpl(T x) : base_t(x) {}
11    CounterImpl(adobe::move_from<CounterImpl> x)
12      : base_t(adobe::move_from<base_t>(x.source)) {}
13
14    void decrement() override { --this->get(); }
15    bool done() const override { return bool(this->get()); }
16  };
```

Here, `T` is the to-be-erased type that users want to bind to our external value-semantic interface. `T` has to model our concept `CountdownCounter`. We check that in line 5. This is a compile-time test, and it makes template error messages more readable. But this line is not strictly necessary. If we omit it, we will simply get less readable error messages if we bind a type to the incompatible interface. Line 7 is an alias declaration, it is the new substitute for `typedef`s. `adobe::optimized_storage_type` is a utility for picking the most efficient storage for type `T`. In short, if `T` is small enough, rather than allocating it on a heap, we will use a potentially stack-based aligned_storage. This trick is often called a *small buffer optimization*.

We also need to define two constructors: one allows initialization from `T`, the other is Adobe's way of implementing move constructor. Finally, we implement the internal interface's member functions with `T`'s interface. Expression `this->get()` returns `T&` (or `T const&` respectively). This is where we map `T`'s interface onto our internal interface. Note that we lost the type returned by `T`'s `operator--`. That's fine here. We will restore it in the next class we define. Other member functions that need to be overridden, which deal with copying and swapping, are already defined in `base_t`, the class we derive from: the framework does it for us.

Finally, we define a class that represents the external interface:

```
1  struct Counter : adobe::poly_base<CounterIface, CounterImpl>
2  {
```

```
  using base_t = adobe::poly_base<CounterIface, CounterImpl>;
  using base_t::base_t; // inherit constructors

  Counter(adobe::move_from<Counter> x)
    : base_t(adobe::move_from<base_t>(x.source)) {}

  Counter& operator--()
  {
    interface_ref().decrement();
    return *this;
  }

  explicit operator bool() const
  {
    return interface_ref().done();
  }
};
```

Now, the outer interface has the same interface as our original concept. `operator--` returns a reference to self. We map from the inner interface to the outer interface. But we will not use naked `Counter`. Our users will have to use a derived type:

```
using AnyCounter = adobe::poly<Counter>;
```

Now we can statically test that our interface is at the same time the model of concept `CountdownCounter`:

```
BOOST_CONCEPT_ASSERT((CountdownCounter<AnyCounter>));
```

And we can also test our counters at runtime. Let's invent some other model of `CountdownCounter`:

```
struct LoggingCounter
{
```

```
  int c = 2; // by default start from 2

  explicit operator bool () const { return c; }

  LoggingCounter& operator--()
  {
    --c;
    std::cout << "decremented\n";
    return *this;
  }
};
```

And the test:

```
AnyCounter counter1 {2};  // bind to int (initially 2)
assert (counter1);        // still counting
assert (--counter1);      // still counting (1)
AnyCounter counter2 = counter1;
                          // counter2 (int) counts from 1
--counter1;
assert (!counter1);       // done
assert (counter2);        // counter2 still 1
assert (!--counter2);     // counter2 also done

counter1 = AnyCounter{LoggingCounter{}};
                          // reset with a different type
assert (counter1);        // 2
--counter1;
assert (counter1);        // 1
--counter1;
assert (!counter1);       // 0
```

Well, performing mutating operations in assertions is a bad idea. I just tried to make the example short. Don't do it at home. You can see that `adobe::poly` offers no implicit conversion from `T` to the interface object. I had to use

explicit initialization. Also, you can see tat there is a lot of boiler-plate code involved in creating an interface. One would expect some macro-based automation for this process. For a complete, working program code, see here.

Boost.TypeErasure

Another library serving a similar purpose is Steven Watanabe's Boost.TypeErasure. It follows a slightly different philosophy. Rather than creating one concept composed of a number of requirements, we create a separate concept per each operation. We can later combine them together into bigger concepts, as needed. We have two operations in our concept: counter decrement and the test for reaching zero. We will start with the first. The library already offers concepts for nearly every C++ operator. The concept that we need is `boost::type_erasure::decrementable`:

```
#include <boost/type_erasure/operators.hpp>
using boost::type_erasure::decrementable;
```

Regarding the other operation, we are very unlucky: the library offers convenience tools for specifying operators, named member functions and named free functions, but it does not offer any convenience for conversion functions. Hopefully this will be addressed soon (see this ticket), but for now, we have to do it the long way: customize the framework with a template specialization. First we define our mini-concept:

```
template <class T>
struct testable
{
  static bool apply(const T& arg) { return bool(arg); }
};
```

Its name is `testable`. It checks for the valid expression inside the body, and it offers the external interface (function `apply`) recognized by the framework. Now, the framework needs to be taught about our new concept:

```cpp
#include <boost/type_erasure/any.hpp>

namespace boost { namespace type_erasure {

  template <class T, class Base>
  struct concept_interface<testable<T>, Base, T> : Base
  {
    explicit operator bool () const
    { return call(testable<T>(), *this); }
  };

}}
```

We can see that specializations of `concept_interface` for our concept `testable` expose the explicit conversion to `bool` themselves. Function `call` in the implementation internally calls our function `testable<T>::apply`.

That's quite a lot of boiler palate, but this is an exceptional situation (a conversion function), and likely to be fixed. Also, note that we had to define our mini-concept only once. Now it can be used to build many other composed concepts: not only our counter. Now we have to compose our mini-concepts together.

```cpp
#include <boost/mpl/vector.hpp>
namespace te = boost::type_erasure;

using Counter = boost::mpl::vector<
  te::copy_constructible<>,
  decrementable<>,
  testable<te::_self>,
  te::relaxed
>;
```

This is four requirements rather than two. You already know why we need `copy_constructible`, but what about this `relaxed`? According to the documentation, it allows the interface object to provide a couple of other operations that

we would often want to have: rebind the interface to another implementation of different type, create a null interface, equality comparison, and a few more. Now, with thus defined concept, we can produce the interface type:

```
using AnyCounter = te::any<Counter>;
```

And that's it; you can apply the same compile-time and run-time test as with `adobe::poly` examples. One difference with Boost.TypeErasure is that objects bind to interfaces implicitly. So, you can write:

```
AnyCounter counter1 = 2;
AnyCounter counter2 = counter1;
counter1 = LoggingCounter{};
```

For a complete, working program code, see here.

For another example...

Since this post is supposed to be an introduction to type erasure libraries, let me show you briefly an another example, so that you can see how these libraries work with types having member functions and free functions in their interface. Let's define the following, somewhat silly, concept:

```
template <typename H>
struct HolderConcept : boost::Assignable<H>
                     , boost::CopyConstructible<H>
{
  BOOST_CONCEPT_USAGE(HolderConcept)
  {
    h.store(i);  // member
    i = load(h); // non-member (free)
  }
```

```
10
11  private:
12      H h;
13    int i;
14  };
```

And here is a type that models our concept:

```
1  struct SomeHolder
2  {
3    int val = 0;
4    void store(int i) { val = i; }
5  };
6  int load(SomeHolder& h) { return h.val; }
7
8  BOOST_CONCEPT_ASSERT((HolderConcept<SomeHolder>));
```

First, using Adobe.Poly:

```
1  struct HolderIface : adobe::poly_copyable_interface
2  {
3    virtual void store(int) = 0;
4    virtual int free_load() = 0;
5  };
6
7  template <typename T>
8  struct HolderImpl
9    : adobe::optimized_storage_type<T, HolderIface>::type
10 {
11   using base_t = typename
12     adobe::optimized_storage_type<T, HolderIface>::type;
13
14   BOOST_CONCEPT_ASSERT((HolderConcept<T>));
15   HolderImpl(T x) : base_t(x) {}
```

```
  HolderImpl(adobe::move_from<HolderImpl> x)
    : base_t(adobe::move_from<base_t>(x.source)) {}

  void store(int i) override { this->get().store(i); }
  int free_load() override { return load(this->get()); }
};

struct Holder : adobe::poly_base<HolderIface, HolderImpl>
{
  using base_t = adobe::poly_base<HolderIface, HolderImpl>;
  using base_t::base_t;

  Holder(adobe::move_from<Holder> x)
    : base_t(adobe::move_from<base_t>(x.source)) {}

  void store(int i) { interface_ref().store(i); }

  friend int load(Holder & h) // free function
  { return h.interface_ref().free_load(); }
};

using AnyHolder = adobe::poly<Holder>;
```

It is boringly similar to the previous example. There are two things worth noting. First, look how member function `free_load` is implemented in terms of free function `load`. Second, note how I used `friend` declaration to declare a free function inside class, visible in a namespace enclosing the class.

Now, with Boost.TypeErasure:

```
BOOST_TYPE_ERASURE_MEMBER((has_member_store), store, 1)
BOOST_TYPE_ERASURE_FREE((has_free_load), load, 1)

namespace te = boost::type_erasure;
```

```
5
6  using Holder = boost::mpl::vector<
7    te::copy_constructible<>,
8    has_member_store<void(int)>,
9    has_free_load<int(te::_self&)>,
10   te::relaxed
11 >;
12
13 using AnyHolder = te::any<Holder>;
```

This may require some explanation. First line declares a mini-concept `has_member_store`. It requires that the model has member function `store` taking `1` argument: we do not specify yet what the type of this argument is or if the member function is `const`. similarly, second line defines a mini-concept `has_free_load`, requiring that there exists a free function `load` taking one argument (the model). Next, we compose the requirements. Now, we do specify the types missed previously. The strange `_self&` means that we want to pass argument to function `load` by non-`const` reference. Because we didn't specify otherwise, our member function `store` is non-`const`.

It is much shorter than Adobe.Poly, but it involves a lot of template and macro magic, which is likely to hit you. I spent a lot of time trying to make these examples compile. Here, you can find complete working examples for for Adobe.Poly and for Boost.TypeErasure.

And that's it for part II. Stay tuned for the next one.

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

6 Responses to *Type erasure — Part II*

Szymon Gatner *says:*
December 8, 2013 at 6:43 pm

Very nice follow-up. Again, a joy to read.
A Q: Do you actually use type-erasure frameworks/libs in real life?
P.S.: Tiny issue: LoggingCounter& operator–() has return *this; missing.

Reply

Andrzej Krzemieński *says:*
December 9, 2013 at 8:13 am

Ha! That's a very important question. I wanted to tell about it in part III, but since you have asked. No: I did not use either of these libraries in production code. Typically I rely on already existing components: `std::functions`, `std::shared_ptr`. Only once did I want to provide a type-erased iterator (of different interface than STL iterators), and then I implemented type erasure from scratch. Boost.TypeErasure did not exist at that time, but even if it was I would hesitate to use it, mostly because of the compiler magic involved: the error messages…. Adobe.Poly would be a candidate, but writing my own was faster that incorporating a new third-party library to a my project and learning it. Perhaps the choice would have been different, If I had a tutorial like this one above.

Reply

Szymon Gatner *says:*
December 9, 2013 at 9:11 am

Asking because even tho I don't use external "TE" libs I do erase types in some circumstances. boost::any and std::function are obvious examples (tho any is used much less as its interface does not allow for much). For example, I rolled my own range library based on D'd ranges rather than boost::range, point is I do also have a type InputRange which iterates over Ts hiding actual implementation (might even be stream of random numbers).

I don't see std::shared_ptr as a TE mechanism because of different semantics – every pointer has always reference semantics and the way I see it, proper TE always produces value types (deep copy / assign).

Did you see Sean Parent's Going Na(t)ive 2013 presentation? (Tho he has been presenting this for years really). I call what he does "valualizing polymorphic types". He also erases the type of implementation to hide it behind non-virtual "facade" that provides actual interface to the client and deals with cloning / swapping. He also argues that those types should almost always be immutable (which I must admit I find really hard to disprove). He is from Adobe btw and most likely directly related to Poly library.

To sum up: I do TE too but never found a reason to get a framework for that. Especially because it is also useful to break compilation dependencies and pulling boost to do that… ouch 😉

Reply

Andrzej Krzemieński *says:*
December 9, 2013 at 10:01 am

Yes, have seen Sean Parent's presentations. I consider them the introduction to Adobe.Poly. One motivation for using Adobe.Poly is that it already provides the small buffer optimization, which you do not have to re-implement.

Regarding `std::shared_ptr` I feel I need to add one word, as I do not want to encourage anyone to using them. But they have one interesting thing: type-erased deleter. I intend to write about it in Part III.

Szymon Gatner *says:*
December 9, 2013 at 12:34 pm

Don’t know why but I can’t reply to your response. Anyway, regarding shared_ptr’s deleter: yes, that is imho the biggest advantage over unique_ptr. No need to burden the client with the deletion mechanism. For (similar) reason all my std containers are used with type-erased allocators – vector of T is the important part, rest is detail.

Reply

Pingback: *Problematic parameters | WriteAsync .NET*