Type erasure — Part IV

Posted on January 13, 2014 by Andrzej Krzemieński

Update. The information about `boost::hold_any` was imprecise. This tool does not work just for any type, but requires that the type provides operators `<<` and `>>` for writing into and reading it from IOStreams.

In this post we will be wrapping up the series on type erasure. We will see an another form of value-semantic type erasure: `boost::any`, and try to compare the different methods.

Value-semantic, type-aware “anything”

In C we have an old good way of representing just any type: `void*`, but it is often not good enough in C++ for at least two reasons. You can pass arguments as `void*` to functions, but returning from a function this way is not always possible: if the to-be-erased object has been created inside the function, its life-time will end just before the function is finished and the pointer we return will be dangling. Similarly, you cannot easily make a copy of something pointed to by `void*`. Second, you cannot check what the type of the erased object is. This is why often a `void*` argument is passed along an `enum` that holds this information; but an `enum` will only work if you can enumerate any type you will be erasing in advance.

How can you overcome these difficulties? There exists an "OO" way of solving this problem: just create a dummy OO-interface:

```
struct Object
{
  virtual ~Object() =0;
};

inline Object::~Object() {}
```

A pure virtual member function (destructor) with a body. You do not see it often, but it is legal, and useful. By making the destructor pure virtual, we made the class abstract. But we need to specify the body, because the destructors of derived classes need to call (not via `vtable` look-up) the base class destructor. This interface is silly: it doesn't have any useful member functions; but it offers a number of things. Now we can return handles to type-erased objects from functions with: `std::unique_ptr<Object>`. It is almost like a value-semantic type, except that you cannot make copies. Also, we can now check what the erased type is, using `dynamic_cast` and `typeid`.

Almost good a solution, but it still has some problems. First, it will only work for types that inherit from `Object`. Second, we still cannot make copies of the type-erased objects. Third, we force free-store allocation for every created object, which may hit our performance.

This is where Boost.Any can help solve some of these problems:

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

boost::any value = string{"txt"};  // initialize with string
value = 1;                         // assign/reset with an int
boost::any value2 = value;         // deep copy

```

```cpp
if (int * i = boost::any_cast<int>(&value2))       // an int?
  cout << *i;

if (string * s = boost::any_cast<string>(&value2)) // a string?
  cout << *s;

cout << value.type().name();       // get stored type's name
```

As you can see, we can assign values of different types at different times. If we anticipate a certain type, we can query if it is our type. We can also request for a `typeinfo` struct representing our object: e.g., in order to get the type's name.

There is only one of the above listed problems that `boost::any` (at least today's version) doesn't fix. It will allocate memory for every small type you assign even an `int` or a `char`. It will allocate memory on every copy-construction. It does not perform a small buffer optimisation (SBO).

If you do want an SBO and other performance optimizations, like avoiding RTTI (and I guess you should care about performance) there is a tool for that, also in Boost, but not advertised in the docs: `boost::spirit::hold_any` in Boost.Spirit library. It has almost the same interface as `boost::any`:

```cpp
#include <boost/spirit/home/support/detail/hold_any.hpp>

boost::spirit::hold_any value{ string{"txt"} };
value = 1;
boost::spirit::hold_any val2 = value;

if (int * i = boost::spirit::any_cast<int>(&val2))
  cout << *i;

if (string * s = boost::spirit::any_cast<string>(&val2))
```

```
    cout << *s;

  cout << value.type().name();
```

It is not an ideal substitute though, it requires of the types it erases that they provide operators << and >> for the interaction with IOStreams. After all, Boost.Spirit is about dealing with input and output.

One minor difference in the interface is that `hold_any` does not offer a converting constructor, so you have to direct-initialize the object: I couldn't just use the `=` syntax. Second, mentioned by its author (see here) is that you cannot read into an "empty" (default-constructed) `hold_any`. I have found a nice blog post comparing the efficiency of `boost::any`, `boost::spirit::hold_any` and `void*`.

Interestingly, if you compare the size (with `sizeof`) of `any` and `hold_any`, you will find that the latter is twice bigger (two pointers vs. one pointer size). Yet, using it is faster. The time of copying does not depend on the `sizeof` of the object (or not only), but on the number of operations you have to perform, like memory allocation, RTTI calls, etc.. If you imagine some type `Val` defined as `std::array<std::string, 8>`, you can observe that the size of `std::vector<Val>` storing a thousand elements is smaller than the size of one `Val`, if measured with `sizeof`.

Querying for the erased type

In the examples so far we have seen how you can query for the erased type. This is a necessary feature for these techniques of type erasure where no meaningful interface is exposed: "empty" OO-interfaces like `Object`, or `any`-like libraries, for instance the currently proposed for the C++ Standard Library Any Library.

In the case of Adobe.Poly-like interfaces, because the object can usually be effectively manipulated via the interface, there is much less need for restoring the erased type. Yet, these libraries often offer a way to do it. For

Adobe.Poly, this would look like this:

```cpp
void test (adobe::poly<Counter> counter)
{
  cout << counter.type_info().name(); // erased type's name

  int i;
  if (counter.cast(i)) // an int?
    cout << i;
}
```

For Boost.TypeErasure querying for the erased type looks like this:

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

void test (boost::type_erasure::any<Counter> counter)
{
  cout << boost::type_erasure::typeid_of(counter).name();

  if (int * i = boost::type_erasure::any_cast<int*>(&counter))
    cout << *i;
}
```

For `std::function` a similar test looks like this:

```cpp
struct Val
{
  int i;
  int operator()() const { return i; }
};

void test (std::function<int ()> f)
{
  cout << f.target_type().name();

  if (Val * val = f.target<Val>())
    cout << val->i;
}
```

In general, it is up to the library whether it allows such type queries or not; but as we can see, it is doable.

Boost.Variant

You may have already observed that in the above examples with `boost::any`, we might have as well used `boost::variant`. Wouldn't that have been a better choice? It also allows storing any of a set of unrelated types, and offers full value semantics.

In our small example either library will do, but this is because the example is short, and cannot reflect the difficulty of real life situations. Note that in this series of posts we are discussing type erasure; i.e., how to provide an interface and avoid specifying the types under the interface. We typically do it because the to-be-erased types may not have yet been defined or because we do not want to introduce header dependencies (in order to be able to name the types). For these purposes, `boost::variant` is not a good choice. Creating an object of type:

```
boost::variant<int, std::string, Counter> value;
```

requires of us to provide the definitions of `std::string` and `Counter`; this is unacceptable in certain situations. Note that in the case of `boost::any`, its type does not depend on the types of the elements you store inside.

Therefore we do not consider it "type erasure" in this discussion, even though we cannot tell from the type of the `variant` which type is currently being held. But whether we call `boost::variant` a "type erasure" or not, it is still worth considering using it in cases where its restrictions are acceptable to you. `variant` offers a number of advantages over `any`:

- Compile-time safety — you do not have to rely on run-time casts, but can use StaticVisitors; they verify that you have taken care of all possible types.
- Efficiency — because `variant` knows the sizes of all its types, its own size can be adapted to the largest of the types and avoid heap allocation altogether.
- Clarity of intentions — by specifying a narrow list of allowed types, we disallow any inadvertent assignment of the type from outside of the list, which can be considered a safety feature.

A more detailed comparison of `variant` and `any` is provided in Boost.Variant's documentation here.

Summary

In the below table we summarize the characteristics of different type erasure methods. In order to make it fit into the blog, I had to use very short names. The first column tells whether we can erase just any type, or if special requirements are imposed on the types, like the explicit declaration of being conformant to the given interface. Of course, for type erasure with an interface, the erased types will have to guarantee that certain operations are valid, but this particular restriction on the types is not considered.

The second column says whether the interface objects offer value semantics: if copying the interfaces deep-copies the erased value. The third column says whether the object can be manipulated via the interface or if the casting is necessary. The fourth column says if it is easy to create a meaningful interface for the erased-types. The fifth column says whether it is possible to query what type is currently under the interface. The last column says if the technique can make use of small buffer optimization.

	any type?	value semantic	useful iface	simple to make	type query	SBO
`void*`	yes	no	no	yes	no	no
OO ifaces	no	no	yes	yes	yes	no
`any`-like	yes	yes	no	yes	yes	yes
`poly`-like	yes	yes	yes	no	yes/no	yes

While `void*` gets the most "no" answers, it should be noted that it may still be the most efficient way of passing the erased type as function arguments (provided you know ho to make use of them). While the goal of this series of posts is to promote the fourth, Boost.TypeErasure-like technique, we also acknowledge that using it — creating and maintaining a new interface — is difficult.

Finally, note that the goal of this series of posts was to show you that such thing as 'type erasure' exists, and is a tool available for you *if you need it*. Please, do not consider it an advice that you should always use it, or that you should try to find places where you can stuff it in. Value-semantic type-erasure tools come with certain costs, so one should use them only when one is convinced that in the particular case under consideration the benefits outweigh the costs.

References

If you wish to explore the subject further, the following is a couple of links for a good start.

1. Mat Marcus, Jaakko Järvi, Sean Parent, "Runtime Polymorphic Generic Programming—Mixing Objects and Concepts in ConceptC++".

2. Sean Parent“Value Semantics and Concept Based Polymorphism”.
3. Sean Parent, “Concept-Based Runtime Polymorphism”
4. Thomas Becker, “On the Tension Between Object-Oriented and Generic Programming in C++”.
5. Thomas Becker, “Type Erasure in C++: The Glue between Object-Oriented and Generic Programming”.
6. Sean Parent, “Inheritance Is the Base Class of Evil” (a video).
7. Thomas Becker, “any_iterator: Type Erasure for C++ Iterators”.
8. Steven Watanabe, Boost.TypeErasure.
9. Adobe Systems Incorporated, Poly library.
10. Kevlin Henney, “Valued Conversions”.
11. Kevlin Henney, Boost.Any
12. Beman Dawes, Kevlin Henney, Daniel Krügler, “*Any* Library Proposal (Revision 3)”.
13. felipe’s blog, “Why you shouldn’t use `boost::any` (especially not in time-critical code)”.
14. Eric Friedman, Itay Maman , Boost.Variant.
15. Effective Go -> Interfaces.
16. Thorsten Ottosen, Neil Groves, Boost.Range 2.0.
17. James C. Dehnert, Alexander Stepanov, “Fundamentals of Generic Programming”.
18. ‘jsmith’ at cplusplus.com, “C++ type erasure”.

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

11 Responses to *Type erasure — Part IV*

Barry *says:*
January 13, 2014 at 5:07 pm

Minor typo… when presenting target in std::function, you do f.target<Sqr> when you mean f.target<Val>.

Reply

Andrzej Krzemieński *says:*

January 13, 2014 at 5:13 pm

Thanks! Fixed.

Reply

xenakios *says:*

January 14, 2014 at 1:20 pm

“Now we can return handles to type-erased objects from functions with: std::unique.” You mean std::unique_ptr, right?

Reply

Andrzej Krzemieński *says:*

January 14, 2014 at 1:55 pm

Yes, thanks.

Reply

gnzlbg *says:*

January 14, 2014 at 1:36 pm

Do you mean std::unique_ptr instead of std::unique?

Reply

concerned reader *says:*
February 10, 2014 at 1:52 pm

> A pure virtual member function (destructor) with a body. You do not see it often, but it is legal, and useful.

Hmm, it doesn't work with gcc or clang, unless I move the body outside the class definition.

Reply

Andrzej Krzemieński *says:*
February 10, 2014 at 2:06 pm

Indeed, the Standard [class.mem] forbids combining =0 with function definition. I'll fix this. Thanks!

Reply

Pingback: *Problematic parameters | WriteAsync .NET*

Pingback: *How do you declare an interface in C++? - ExceptionsHub*

Jeremy Murphy *says:*
April 19, 2018 at 12:02 am

Btw, you just have a little typo in Reference #6. 😉

Reply

Andrzej Krzemieński *says:*

April 19, 2018 at 6:23 am

Thanks 🙂

Reply