Type erasure — Part I

Posted on November 18, 2013 by Andrzej Krzemieński

Have you ever came across term *type erasure* in C++? This "pattern" or "technique" is growing more and more popular. In this post I will try to describe what it is. Note that it is something different than a similar term in Java.

What does type encode?

Let's start with describing the opposite of type erasure. We could call such situation "type-full-ness". Forget the term though, let me illustrate what I mean with an example.

```
1  void print (string s, int i)
2  {
3    cout << s;
4    cout << i;
5  }
```

The two instructions in function `print` invoke two different functions; even though `operator<<` spells the same in both cases. Compiler selects the right function overload based on variable's static type. So the type alone (that is not erased) contains information essential for compiler. Note that `operator<<` here has a "polymorphic behaviour": it spells the same but does something different for different objects. And there is no run-time dispatch involved: no run-time overhead. *No run-time overhead* — this is often an important requirement and goal when

programming in C++, therefore techniques like virtual function table look-up or indirect function calls may be unacceptable.

Substituting function overloading for indirect function calls is what makes `std::sort` outperform old C `qsort`, and `std::equal_range` outperform old C `bsearch`. Just consider:

```
int c_bigger (const void * a, const void * b)
{
  return *(int*)b - *(int*)a;
}

bool search_in_C (std::vector<int> & vec, int val)
{
  auto ans = bsearch (&val, vec.data(), vec.size(),
                      sizeof(int), c_bigger);
  return ans;
}

bool search_in_CPP (std::vector<int> & vec, int val)
{
  auto cpp_bigger = [](int a, int b){ return a > b; };

  auto rng = std::equal_range (vec.begin(), vec.end(), val,
                               cpp_bigger);
  return rng.first != rng.second;
}
```

If you try to compare them (with compiler optimizations enabled), the `std::equal_range` version is significantly faster, even though it performs more comparisons and does more things! The reason for it is that function `bsearch` takes a pointer to function as the sorting predicate. Compiler cannot know what this sorting predicate will be, so it can do nothing but put two pointers (function arguments) on the call stack and make an indirect function call —

for each comparison. In contrast, `std::equal_range` is not even a function: it is a *function template*: a tool for generating function overloads on request. `cpp_bigger` is not a function whose pointer we could pass around: it is an object of a class type. You might not have noticed that we have defined a new class here; this is because we used a shorthand notation: a lambda expression. In fact, line

```
auto cpp_bigger = [](int a, int b){ return a > b; };
```

is more-less equivalent to the following:

```
class _CompilerInventedName
{
public:
  bool operator() (int a, int b) const { return a > b; }
};
_CompilerInventedName cpp_bigger;
```

The class has member function `operator()`. We can see the in-line definition of this operator. When instantiating a template, we give it the name of the class as template argument. We cannot see the name here, but compiler can. Given this type, compiler uses the template to instantiate (to create) a function. This function calls inside `operator()`, which can now be inlined. Compiler knows what `operator()` to inline because it knows the type of object `cpp_bigger`, and this type is part of the signature of the function overload generated from function template `std::equal_range`. Thus, again we "encoded" some information in the type.

Why we don't like templates

Even though they can improve run-time performance, templates have also their downsides. First, the size of the program increases, because every new instantiation of a template creates a separate function overload, which

needs to be stored. Second, we get a compile-time slow-down because template instantiation lasts and lasts. Third, unless you can enumerate all instantiations of your template in advance, you have to include the body of each function template in the header file, you cannot separate the declaration from the implementation. If you expose the implementation, you also expose all its dependencies: headers. This causes an otherwise unnecessary inclusion of files, which renders the compile times longer, sometimes amazingly long. And there are these template instantiation error messages that scare many programmers off…

This is a trade-off you have to make: do you want a fast program or a fast build?

`void*`-based type erasure

So, suppose you do not like the fact that `std::equal_range` generates too many overloads, affects your program's size and compilation time, forces you to expose its implementation to the users: you do not want that. Since the searches through your arrays of `int`s never take long, you would rather sacrifice run-time efficiency. What should you do? It is simple; we have seen the answer above: use `bsearch`!

`bsearch` is in fact an example of basic type erasure. Let's look at its interface:

```
1  void* bsearch (const void *key,
2                 const void *base,
3                 size_t nmemb,
4                 size_t size,
5                 int (*compar)(const void*, const void*));
```

It is one function, with one interface that works for arrays of any element type. `base` is a pointer to the first element of the array. The pointee (the first array element) is an object of a concrete type, but we will not be able to figure out from the type of the pointer what that type is. The type of the pointer is "pointer to just anything." We can say

that the type of the element has been *erased*. Passing arguments to functions as `void*` is often used in C in situations when we need to be able to pass just any type that we cannot predict in advance. C++ also allows that, although in C++ this is hardly ever a good idea. There is not much you can do with such a pointer to anything; in fact, about the only thing you can do with it is to pass it to someone else. Figuring out the type of the object back, or making use of it, needs to be implemented manually by the programmer. In case of `bsearch` it is implemented by other function's arguments: `size` tells us by how much bytes we should increase the pointer in order to get to the next element. Similarly, `compar` provides the knowledge on how to compare two pointers to anything: `bsearch` doesn't have to know. Extracting the value back from type-erased `void*` is a burden, but it also offers a gain: by this erasure we 'merged' a family of types (nearly all types) into one type: `void*`; now we only need to implement one function (overload).

But `bsearch` may not be satisfactory for a number of reasons. First, it is not *type-safe*. The user could mistakenly pass an incorrect ordering function to `bsearch`. Consider:

```
1  int i_bigger (const void * a, const void * b)
2  {
3    return *(int*)b - *(int*)a;
4  }
5
6  int c_bigger (const void * a, const void * b)
7  {
8    return -strcmp ((const char *)a, (const char *)b);
9  }
10
11 bool search_in_C (std::vector<int> & vec, int val)
12 {
13   bsearch(&val, vec.data(), vec.size(), sizeof(int), c_bigger);
14   // BUG! It should have been i_bigger()
15 }
```

If you mistakenly pass the wrong function, compiler will not complain, there will be no run-time exception: the function will do something, but likely not what you have intended. This is an *undefined behaviour*.

Second, `bsearch` only works for pointers to arrays. We were lucky that `std::vector` is guaranteed to provide an array layout. But we won't be that lucky if we want to search in anything else, e.g. `std::set`.

OO-based type erasure

We could attempt to solve some of these problems by introducing an OO-style type hierarchy. We can enforce every user of our code/library to derive their types from our super base class. This super base class we would probably call `Object` (as it is usually the case for OO-style designs) and require that all derived classes implement a member function `lessThan(Object*)` and probably a couple of other functions. Our sorting function could then be implemented as something like this:

```
1  class MyInteger : public Object
2  {
3    // ...
4    public: int lessThan (const Object* rhs) const override
5    {
6      auto irhs = dynamic_cast<MyInteger const*>(rhs);
7
8      if (irhs == nullptr) {
9        throw SomeException{};
10     }
11
12     return irhs->getInt() - this->getInt();
13   }
14 };
15
16 Object* OO_search (Object* val, Object** base, size_t size);
```

Now we have solved one problem. In case a compared object has an incompatible type, an exception will be thrown at run-time.

This is another "basic" type of type erasure known probably for a long time to most C++ programmers: inheriting from a common base class and passing around a pointer/reference to the interface. Unlike with `void*` we are now guaranteed that whatever our type really is (`MyInt` or `MyString`, etc.) certain operations on our pointer (like `lessThan`) will always work; that is, never render a UB. Our type has been erased, but our type's interface has not! We no longer have to worry how we retrieve the original type of our object, because nearly anything we will ever need of it can be achieved via the interface.

But now our type is tied to the sorting function. We can fix that by passing sorting function as a separate argument:

```
1  int i_bigger (const Object* a, const Object* b)
2  {
3    auto ia = dynamic_cast<MyInteger const*>(a);
4    auto ib = dynamic_cast<MyInteger const*>(b);
5
6    if (ia == nullptr || ib == nullptr) {
7      throw SomeException{};
8    }
9
10   return ib->getInt() - ia->getInt();
11 }
12
13 Object* OO_search (Object** base,
14                    size_t size,
15                    int (*comp)(const Object*, const Object*));
```

and if we are really keen on OO, instead of a function pointer we could introduce a yet another interface:

```
Object* OO_search (Object* val, Object** base, size_t size,
                   Ordering & compar);
```

There are a couple of problems with this approach though. First, we can no longer sort arrays of `int`s: `int`s can never inherit from `Object`. As you can see we had to wrap them in a class that would have otherwise been unnecessary. Although it only stores an `int`, its size is now greater than than of an `int`, because wee need a room for a pointer to `vtable`. In fact, we now impose a broader additional requirement: all user's types must now inherit from `Object` or else they won't work with our component. This puts a burden on users. What if the user wanted to use a comparison function from a third party vendor? Such comparator couldn't possibly derive from an interface we invented. And we may not be able to change third party code. Also, imagine that this is a library that you are writing and you require that all users' types derive from `MyLib::Object`; but the user also wants to use another library with a similar philosophy, which in turn forces him to also derive all his types from `OtherLib::Object`. Also, there are a lot of function objects out there, taking two `int`s and returning a `bool` (e.g., `std::greater<int>`) which we now will not be able to use due to our OO constraints. If we choose to use OO interface `Ordering` we will not even be able to pass closures as predicate.

Second, note that I changed the type of the first argument to `Object**` (a pointer to pointer). The outer pointer serves as the iterator; the inner pointer is necessary because now we no longer can store our objects directly as values in containers, because we do not know the size of the object under the interface. We would not know by how much to increase the pointer. On the other hand, if we store pointers, any pointer has always the same size. To fix that, we could go back to passing `nmemb` argument to our function, or add member function `size` to the "minimum" interface or `Object`.

Third, we only partially solved the problem of applying a comparator functor to a mismatching type. While we are now avoiding UB, we turned it into a run-time exception. While this is an improvement, it is still poor a solution,

compared with the original example with `std::equal_range`, where such mismatch is detected by the compiler. Just because we throw an exception, it doesn't mean we handled the situation properly. What will the user of a GUI application see when he tries to pick a widget? "Bad comparison function passed to function `OO_search`"? "Internal error"? Bugs should be detected at compilation time rather than at program execution time. This may not be always doable; but in our case it was — we missed the opportunity. This problem could be fixed, though, by turning our searching function back into a template and making the interface `Ordering` also a template, as explained in detail down below.

Fourth, note that if we choose to use a custom comparator (not tie it to the element type) and figure out the size of the structures by other means, `Object` does not contain any useful (for us) member. We inherit only for the sake of being able to pass a pointer to an empty base class and apply the inheritance test with `dynamic_cast`. Not a very useful interface: almost like `void*`.

Finally, our function only works with pointers as iterators. We cannot search in a `map` or anything but a raw array. We could introduce a yet another OO interface: `Iterator` (and it would also know by how much bytes to advance to the next element). But STL containers do not provide iterators derived from `MyLib::Iterator`. You will have to add a wrapper for each iterator you use. Then, you will have to answer next questions: Do you want to pass these `Iterator`s by reference? You want to avoid slicing, don't you? By reference to const or non-const object? (If non-const, where will you create them before passing them to function. If const, you will not be able to increment them easily.) How will you return the iterator to the found object? By value (risking slicing)? By reference (risking a dangling reference)? Also, we would now have created another iterator, incompatible with STL — because STL algorithms pass iterators by value and expect no slicing.

OO-style interfaces do not play well with *value semantics*. It somehow forces us to pass references/pointers to our objects. This is especially difficult when returning a type-erased object from a function: you cannot return by value, because you are risking slicing. You often cannot return by reference, because, you would be returning a reference to an automatic object that would have died before you try to access it. You could return by a smart pointer, but this only opens a new set of problems: which smart pointer? `unique_ptr`? `shared_ptr`? — But neither will work if you need to (deeply) copy the returned object.

Value-semantic type erasure

I will disappoint you. I do not have a perfect solution for the problem I was complaining about above. In order to enforce at compile-time that the comparison function matches the element type, I think you need to mention the type explicitly — I cannot see any other way. In order to provide a reasonable solution we will have to go back to using a template. You can consider it cheating. But this will be a different template; it will require much less instantiations.

Using templates (apart from all disadvantages) offers two advantages: faster programs and more type safety. By abandoning templates we abandoned both the advantages; whereas our original trade-off was to sacrifice faster program for faster build — but not sacrifice type-safe program. So, let me put the template back, but make is a bit more “constrained”. Our original function template is parametrized over three things. Well, technically it is two things, but in practise you can think of it as three:

1. Type of the comparison function.
2. Type of the element in the collection.
3. Type of the iterator.

You cannot see the dependence on element type directly, because you obtain the `T` from the type of the iterator. Nonetheless it is there, and this is the only parameter that matters to us (in order to guarantee type safety), so we will extract it as a template parameter, and erase the type of the iterator and the type of the comparison function. How? Let's start with the comparison. You probably know the tool for this already: `std::function`. It is well documented Boost.Function documentation. In short, it is a holder for any kind of function, function object or closure of an appropriate type. The "type" of function appropriate for us is something taking two `int`s and returning a `bool`:

```
1 | std::function<bool(int, int)> predicate;
```

We can assign to it any function pointer or function object with a matching function signature:

```
1 | bool bigger (int a, int b) { return a > b; }
2 |
3 | predicate = &bigger;
4 | predicate = [](int a, int b) { return a > b; };
5 | predicate = std::greater<int>{};
```

Not only is `predicate` able to "hold" any function-like entity (of an appropriate function signature), but is is also a value: it can be copied or assigned to. It can be passed by value with no risk of slicing, because it guarantees to make a deep copy of the underlying concrete object. And we use it just as any other function:

```
1 | bool ans = predicate (2, 1);
```

our `predicate` works well with STL algorithms: it is still a function object. We give it anything that has `operator()(int, int)` and we get something that has `operator()(int, int)`, but with erased type. Note that `std::function`'s

requirements on the types it can be assigned/initialized with are non-intrusive (or *duck-typing*-like): if it has `operator()` it is a good candidate, nothing else is required: no inheritance.

And note one more thing: we did not use any particular language feature for this; `std::function` is a library component. Isn't that amazing? If you are wondering how it is even possible to implement such a thing, you can have a look at this publication.

You might be puzzled about one thing, though: if `std::function` is used for type erasure, why is it a template itself? Well, it is a template for generating type-erased interfaces. It is like an "interface template". Above, we were only interested in one interface: one instantiation of the template, capable of erasing a lot of types. But to solve our main problem from this post, we will need more than a sorting predicate for `int`s, we will need a predicate for any `T`. We can express this with an *alias template*:

```
1  template <typename T>
2  using AnyBinaryPredicate =
3    std::function<bool(T const&, T const&)>;
```

This defines a new "typedef" or "alias" that can be used with one template parameter, e.g., `AnyBinaryPredicate<int>`, which means "any binary predicate capable of comparing `int`s." Note that it is only a "`typedef`": it does not introduce new types or functions.

With `std::function` we (1) erase the type of the underlying function/function-like object, (2) preserve the interface (`operator()`), (3) we are able to pass it by value, (4) we require of the erased types no declaration of conformance to an interface (no inheritance).

But this is a special case for functions, because they are popular in C++. How do we erase the type of the iterator? Iterators are also popular in C++: we will use another library already waiting for us. We could use Thomas Becker's `any_iterator`.

It comes with a class template `IteratorTypeErasure::any_iterator`, another "interface template" similar to `std::function`. It is parametrized by two parameters:

1. Value type of the underlying sequence (container).
2. Iterator category (forward iterator, random access iterator, etc.).

We will fix the second parameter to "forward iterator" tag. This is the minimum that we require for searching functions. Iterator category is something we do not want to erase:

```
template <typename T>
using AnyForwardIter = IteratorTypeErasure::any_iterator<
  T,                          // variable parameter
  std::forward_iterator_tag   // fixed parameter
>;
```

Now we have an another value-semantic interface capable of managing anything that is a forward iterator:

```
std::vector<int> vec {1, 2, 3};
std::list<int> list {2, 4, 6};

AnyForwardIter<int> it { vec.begin() }; // initialize
it = list.begin();                      // rebind
AnyForwardIter<int> it2 = it;           // copy (deep)
```

And it is an iterator itself:

```
++it;
int i = *it;
it == it2;
```

But let's go one step further, rather than using two iterators, let's use a range — a type-erased range: `boost::any_range`. Again, like `std::function` and `IteratorTypeErasure::any_iterator`, it is an "interface template", so we will only pick some specializations. Template `boost::any_range` requires at least 4 parameters:

1. Value type of the container.
2. Iterator category.
3. Type of the reference returned by dereferencing an iterator.
4. Type of the iterator difference.

We will fix the three latter parameters and only leave the value type as the parameter. We will use an alias template again:

```
template <typename T>
using AnyForwardRange = boost::any_range<
  T,                                // real param
  boost::forward_traversal_tag, // fixed
  T&,                               // repeated param
  std::ptrdiff_t                // fixed
>;
```

This means "any forward range capable of iterating over `int`s." This is how we can use it:

```
std::vector<int> vec {9, 8, 5, 4, 2, 1, 1, 0};
std::set<int> set {1, 2, 3, 5, 7, 9};

AnyForwardRange<int> rng = vec; // initialize interface
```

```
std::distance (boost::begin(rng), boost::end(rng));

rng = set;                          // rebind interface
std::distance (boost::begin(rng), boost::end(rng));
```

Thus, we have two value-semantic type-erased interfaces: `AnyForwardRange<T>` and `AnyBinaryPredicate<T>`. Using them we can define our (partially) type-erased searching function:

```
template <typename T>
AnyForwardRange<T> Search (AnyForwardRange<T> rng, T const& v,
                           AnyBinaryPredicate<T> pred)
{
  auto ans = std::equal_range (rng.begin(), rng.end(), v, pred);
  return {ans.first, ans.second};
}
```

We still use `std::equal_range` inside: it is a good algorithm. But because it is wrapped in our new interface, we will control its instantiations: only one per element type:

```
std::equal_range <
  typename boost::range_iterator<AnyForwardRange<T>>::type,
  T
>
```

Our `Search` is still a template, but it will only generate new instantiations when we want to sort different types of elements. It will not generate different instantiations for different iterator types or different predicates: their type will be erased. This is how we can use our `Search`:

```
std::vector<int> vec {9, 8, 5, 4, 2, 1, 1, 0};
auto greater = [](int a, int b) { return a > b; };
AnyForwardRange<int> ans = Search<int> (vec, 1, greater);
```

```
4  assert (distance(ans) == 2);
5
6  std::set<int> set {1, 2, 3, 5, 7, 9};
7  ans = Search<int> (set, 4, std::less<int>{});
8  assert (distance(ans) == 0);
```

It comes with one inconvenience: you have to specify which instantiation you choose. I must admit, I do not know how to overcome this without introducing more templates. This type erasure works at the expense of reduced run-time performance. The implementations of `boost::any_range` and `std::function` internally use indirect function call techniques, like OO interfaces.

The implementation of `std::function` and `boost::any_range` uses templates and template instantiations too, so you might conclude that in order to avoid templates we introduced even more templates. This is true, but only to certain extent. We have now new template instantiations, indeed; but they are localized to places where you bind objects to interfaces. Once you do that other functions/algorithms that use the type-erased interfaces do not have to be templates, or as was the case for our `Search`, they do not have to generate this many instantiations.

If we erased the types completely, you could hide the function's implementation to a "cpp" file and compile separately: you do not need to expose it to the users: you still reduce compilation times (although less than with `void*`). This is not an ideal solution but an alternative when making important trade-offs in your projects: an attractive alternative.

To be continued...

And that's it for today. You may still feel that I have cheated you. `std::function` and `boost::any_range` may be a solution if anything you ever wanted to do in your program was to invoke functions and advance iterators. But what if you want to use a custom interface with custom member functions? I will try to cover it in the next post

(but I already confess, I do not have a perfect solution for this). I will also try to explain why all the names of the interfaces in the examples above start with "`Any`". I will also try to cover some practical examples of type-safe type erasure.

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

21 Responses to *Type erasure — Part I*

kou4307 *says:*
November 18, 2013 at 5:18 pm

have been waiting a long time for this..thanks a lot. looking forward for more.

Reply

Krzysiek *says:*
November 19, 2013 at 12:52 pm

Andrzej, I suppose in the AnyForwardRange alias the second T should in fact be T&, as dereferencing an iterator should return a reference.

As always, a great article and a great read!

Reply

Andrzej Krzemieński *says:*
November 19, 2013 at 1:14 pm

Indeed. Fixed. Thanks!

Reply

Maxim Yegorushkin *says:*
November 19, 2013 at 2:52 pm

Good article. I think one of the oldest examples of type erasure is boost::any. Its source shows what type erasure is and how it is implemented in only 313 lines)

Reply

slimshader *says:*
November 19, 2013 at 7:47 pm

Nice post, you should definitely cover boost.type_erasure too.

Reply

Andrzej Krzemieński *says:*
November 20, 2013 at 7:55 am

Thank you. I intend to; and hope to cover one more.

Reply

JCAB *says:*
November 19, 2013 at 10:36 pm

"It comes with one inconvenience: you have to specify which instantiation you choose" – It occurs to me that a way to avoid that would be to use an adapter parameterized on the actual container/range and comparison predicate. That would be like

going back full circle, I suppose, but it would achieve the lofty goal of “pretending” like we’re using std::lower_bound, but separating the type-erased implementation from the interface, and allowing the compiler the decision of whether to generate more of functions (inlining the implementation) or not (inlining the adapter).

Unfortunately, I don’t think a smart compiler would be able to recover the performance if it decides to inline the implementation. The problem is that std::function (and, I suppose the anyXX adapters) separates interface from implementation by utilizing (in the general case) heap allocations, and then dereferencing them when calling into the underlying implementation. That might preclude even the smart sort of optimizations that can allow calls into function pointers to be inlined. Probably. I think, but frankly I’m not sure.

Maybe this is the sort of stuff you plan to cover in part 2? I’m not sure from you “teaser” 🙂

Reply

Pingback: *Problematic parameters | WriteAsync .NET*

Richard Hodges *says:*
July 13, 2014 at 11:13 am

I’m not sure I agree with the section “Why we don’t like templates”
To address the code size argument; if you want type safety of any kind there must be code to enforce it. Templates offer compile time type and/or concept safety. Polymorphic code must often check for type correctness and therefore incur the penalties of runtime performance, runtime logic errors plus reliance on programmers to behave themselves. Template services have no runtime overhead, check logic at compile time and refuse to compile if programmers misbehave. void * type erasure is simply off the table. It’s an archaic throwback, totally un-necessary and utterly dangerous.
Re compile times: Compile time ought not to be considered a constraint when the payoff is type-safety and compile-time checking of logical correctness. The alternative (runtime checking) leads to very long debugging cycles and unhappy users.
Suggestion: pay the (tiny) cost now to save yourself a huge bill in the future.
If it’s a real problem, for example your compile is taking longer than it takes to make some coffee, there are solutions – break up .cpp files, use parallel builds and of course precompiled headers.

Error Messages: These are improving in modern c++ compilers. Upcoming compiler support for concepts will remove the cryptic error messages.

Reply

Andrzej Krzemieński *says:*
July 14, 2014 at 8:42 am

It doesn't look like we disagree here. Templates offer certain advantages and also come with some cost; so by using them you decide on a certain trade-off. You say that you certainly prefer to have the correctness compile-time checks and you are willing to pay the cost of longer builds and cryptic error messages.

But even if you do so, you have to admit that it is not something desired that templates compile long and that the error messages they give are not obvious. It is a problem. The solution is not necessarily switching to run-time checks. There are alternatives, like adding concepts and modules to C++. But we have to acknowledge that templates have problems.

Reply

Programator *says:*
September 22, 2014 at 12:52 pm

The opposite of type erasure is called "reification". Pozdrawiam.

Reply

Pingback: *How to: How do you declare an interface in C++? | SevenNet*

Pingback: *Fixed How do you declare an interface in C++? #dev #it #asnwer | Good Answer*

Pingback: *Quando usar void* e auto*? | CL-UAT*

Pingback: *Avoiding The Performance Hazzards of std::function | The blog at the bottom of the sea*

Maxx On *says:*
June 26, 2017 at 4:10 pm

> We give it anything that has operator()(int, int) and we get something that has operator()(int, int), but with erased type.

This is unclear. What type is erased?

Reply

Andrzej Krzemieński *says:*
June 26, 2017 at 7:07 pm

Let me explain. First, three definitions of "something that has operator()(int, int)":

```
1 bool bigger1 (int a, int b) { return a > b; } // function
2 auto bigger2 = [](int a, int b) { return a > b; }; // lambda
3 std::greater<int> bigger3; // class object
```

They are *not* type-erased, because their type as declared is preserved:

```
1 decltype(bigger1) ==  bool (&)(int, int);
2 decltype(bigger2) == some-unique-class-type;
3 decltype(bigger3) == std::greater<int>;
```

Now when I declare this:

```
1 std::function<bool(int, int)> predicate;
```

I can store, say, `bigger3` in it:

```
1 predicate = bigger3;
```

Now, after this assignment, `predicate` refers to some copy of `bigger3`. This copy has type `std::greater<int>`, but there is no (easy) way to get to this type anymore. The only useful thing that can be done with an object of type `std::greater<int>` referred to by object `predicate` is to call `predicate(1, 2)`. So, *the type of the referred-to object has been erased*. The only type that we can inspect is `decltype(predicate)`, which is not the type of the object that `predicate` refers to.

Does that make sense?

Reply

ss *says:*
September 27, 2018 at 1:12 pm

small nitpick. can you reduce the font size of your code in the text boxes so that more code can be viewed on a single line. Thanks

Reply

Andrzej Krzemieński *says:*
September 27, 2018 at 4:36 pm

I sympathize with the pain. But I do not think I can do anything about it. This is what WordPress gives me, and I (and you) have to live with it.

Reply

Pingback: *c++ – When to use void* and auto*? - YeahEXP*

Pingback: *c++ – When to use void * and auto *? - YeahEXP*

This site uses Akismet to reduce spam. Learn how your comment data is processed.
