any_iterator: Type Erasure for C++ Iterators

Contents

Overview

The class template `any_iterator` is the analog to `boost::function` for iterators. It allows you to have a single variable and assign to it iterators of different types, as long as these iterators have a suitable commonality.

For a more in-depth view of type erasure in general and my `any_iterator` in particular, see my October 2007 article at The C++ Source.

Quick Tutorial

The `any_iterator` Class Template Declaration

The class template `any_iterator` is declared like this:

```
template<
  class Value,
  class CategoryOrTraversal,
  class Reference = Value&,
  class Difference = std::ptrdiff_t
>
class any_iterator;
```

For the second template argument, you may use the old-style STL iterator categories such as `std::random_access_iterator_tag`, or you may use the new traversal categories as set forth in the Boost iterator library.

Assigning Concrete Iterators to `any_iterator` Variables

Here is an iterator type that can hold an `std::vector<double>::const_iterator` as well as an `std::list<double>::const_iterator`:

```
typedef any_iterator<
  double const,
  boost::bidirectional_traversal_tag
>
any_number_iterator;

any_number_iterator number_iter;

std::vector<double> number_vector(42, 43.0);
number_iter = number_vector.begin();
```

```
double d = *number_iter;

std::list<double> number_list(42, 44.0);
number_iter = number_list.begin();
d = *number_iter;
```

Next, suppose you want the variable `number_iter` to hold not only the vector and list iterators, but also a Boost transform iterator which, upon dereferencing, multiplies by 100.0, like this:

```
number_iter = boost::make_transform_iterator(
  number_vector.begin(),
  boost::bind(std::multiplies<double>(), _1, 100)
);
```

The `any_iterator`'s assignment operator is not enabled for this assignment, and you will get an error message such as

```
binary '=': no operator found which takes a right-hand operand of
type...
```

Why is this assignment not allowed? The transform iterator's `operator*` returns a `double`. If we were to pass that through our number iterator, which currently has a reference type of `double const&`, then the number iterator's `operator*` would be returning a reference to a temporary local variable. The solution is to change the `any_iterator`'s reference type so that it is no longer a reference:

```
typedef any_iterator<
  double const, // Value
  boost::bidirectional_traversal_tag,
  double const // Reference
>
any_number_iterator;
```

Now all of the assignments above will compile and work. It should be intuitively clear by now what the rules are for assigning "concrete" iterators to `any_iterator` variables. For example, an `any_iterator` that's a bidirectional iterator will not accept something that's a forward iterator only, like an iterator into a singly linked list. That would not make sense: a variable whose iterator category is bidirectional cannot at runtime hold something that is only a forward iterator. Similarly, our `any_number_iterator` would not accept anything that dereferences to something that does not convert to a `double`. After such an assignment, dereferencing would not make sense anymore.

You can find more examples in the file `any_iterator_demo.hpp` that comes with the `any_iterator` distribution. The exact, formal rules for assigning "concrete" iterators to `any_iterator` variables are stated below.

Important Warning: Type Erasure Erases Interoperability

Suppose you have two concrete iterators that point into the same sequence but are of different type:

```
std::vector<int> int_vector;
std::vector<int>::iterator it = int_vector.begin();
std::vector<int>::const_iterator cit = int_vector.begin();
```

These two iterators are of course interoperable: the comparison

```
it == cit
```

will compile, run, and give the correct answer (true in this case). If, however, you wrap these two iterators into `any_iterator`s, then their interoperability is lost. For example, if you define

```
typedef any_iterator<int, boost::random_access_traversal_tag,  int const &>
  random_access_const_iterator_to_int;

random_access_const_iterator_to_int ait_1 = it;
random_access_const_iterator_to_int ait_2 = cit;
```

then the comparison

```
ait_1 == ait_2; // bad comparison!
```

behaves as if you were comparing iterators into different sequences: the behavior is undefined! (In my implementation, you will get an assertion in debug mode and a null pointer dereferencing in release mode.)

This behavior is most certainly highly undesirable and extremely dangerous. In fact, it is so bad that I had at one point decided to declare the idea of the `any_iterator` infeasible. But then it occurred to me that this pitfall is not entirely specific to the `any_iterator`. It occurs with any type-erasing class that implements binary operators. Therefore, it is perhaps beneficial to put all this out there and alert people to the problem.

Acknowledgment: I believe the first person to spot this problem with interoperability was Thomas Witt. The issue was pointed out to me independently by Sergei Politov.

Performance Overhead

The extra cost of an operation such as dereferencing or incrementing an `any_iterator` is one level of indirection and a virtual function call. Construction and destruction of an `any_iterator` object involve one heap access each.

Relationships between Different `any_iterator` Types

Instantiations of the `any_iterator` class template are Standard conforming iterators. Therefore, it would theoretically be possible to assign an object of one `any_iterator` type to a variable of another `any_iterator` type. However, this would lead to nesting levels greater than 1. Consequently, the overhead of using an `any_iterator` could increase to several level of indirections and several virtual function calls. Therefore, assignments that would lead to nested `any_iterator` objects are not allowed. Instead, there are certain conversions between `any_iterator` types that behave nicely insofar as they do not cause nesting levels deeper than 1. The exact rules for these conversions are stated below.

The Explicit Constructor from Wrapped Iterator

The constructor that creates an `any_iterator` object from a "concrete" iterator is currently explicit. This is an unfortunate limitation that is caused by a technicality having to do with an arcane limitation of the SFINAE principle. This annoyance will go away once concepts will be available in C++.

For now, this means that a "concrete" iterator never converts to an `any_iterator`. There are only two ways to get a "concrete" iterator object into an `any_iterator` variable: either by ordinary assignment, as in

```
std::vector<int> vect;
any_iterator<int, std::forward_iterator_tag> ait;
ait = vect.begin();
```

or using *explicit* construction:

```
std::vector<int>;
any_iterator<
  int,
  std::forward_iterator_tag
> ait_1(vect.begin()); // fine

any_iterator<
  int,
  std::forward_iterator_tag
> ait_2 = vect.begin(); // error, requires non-explicit copy ctor
```

Supported Compilers

The `any_iterator` has been tested under Microsoft VC7.1, Microsoft VC8, Microsoft VC9, gcc 3.2.4, gcc 3.4.2, gcc 4.0.3, and gcc 4.1.2. More recent compilers and compiler versions seem to always work, probably because C++ template support has stabilized by now.

The `make_any_iterator_type` Metafunction

There is a metafunction named `make_any_iterator_type` which takes an iterator type as its argument and produces an instantiation of the `any_iterator` class template with the same iterator traits. In other words, it allows you to create an `any_iterator` type "by example."

```
typedef
make_any_iterator_type<
  std::vector<int>::iterator
>::type ait;
```

This has the same effect as

```
typedef any_iterator<int, std::random_access_iterator_tag> ait;
```

Download and Installation

The `any_iterator` be downloaded from here. The download contains the source code, this HTML documentation, a demo .cpp file, and regression tests.

The directory `any_iterator` in the download contains the file `any_iterator.hpp` and a subdirectory named `detail`. To use the `any_iterator`, you must include `any_iterator.hpp` and make sure the header files in the subdirectory `detail` are found by `any_iterator.hpp`. The `any_iterator` currently lives in the namespace `IteratorTypeErasure`.

The `any_iterator` makes ample use of Boost libraries; however, it uses only header files. Therefore, no Boost binaries are needed. Just make sure that Boost is in your include path.

If you wish to build and run the regression tests, you will find solution files for Microsoft VC7.1 and Microsoft VC8 and a makefile for gcc in the corresponding subdirectories of the directory `regression_tests`. If you use the makefile, you will of course have to fix the path variables to match your development environment. The Microsoft solution files should work out of the box, except for the fact that you have to fix the include directories so that the Boost headers are found.

Feedback and Bug Reports

Click here to send feedback and bug reports concerning the `any_iterator`.

Acknowledgements

I am indebted to Dave Abrahams, Christopher Baus, Fred Bertsch, Don Harriman, and Thomas Witt for their input to my efforts regarding iterator type erasure.

Appendix

Exact Rules for Assignment to `any_iterator` Variables

Suppose that `some_any_iterator` is an instantiation of the `any_iterator` class template with value type, traversal tag, reference type, and difference type equal to `AnyItValue`, `AnyItTraversal`, `AnyItReference`, and `AnyItDifference`, respectively. Assume further that `some_iterator` is an iterator type with value type, traversal tag, reference type, and difference type equal to `ItValue`, `ItTraversal`, `ItReference`, and `ItDifference`, respectively. Then a variable of type `some_any_iterator` will accept an object of type `some_iterator` if and only if the following four conditions are met:

1. `ItValue` converts to `AnyItValue`.

2. `ItTraversal` and `AnyItTraversal` are equal, or the former is derived from the latter. This means that `some_iterator`'s traversal category is equal to or better than that of `some_any_iterator`.

3. The following are all true:

    - `ItReference` converts to `AnyItReference`.

    - If `AnyItReference` is a reference, then so is `ItReference`.

    - If `AnyItReference` and `ItReference` are both references, then the following is true: after stripping const qualifiers and references from `AnyItReference` and `ItReference`, the two are either the same, or the former is a base class of the latter.

   The second and third of the three conditions above ensure that no situation is allowed where `some_any_iterator`'s `operator*` would return a reference to a temporary.

4. If `some_any_iterator` is a random access iterator, then `ItDifference` and `AnyItDifference` are convertible to each other both ways. Here, we need convertibility in both directions because the difference type occurs as an argument type as well as a result type of iterator operators.

Exact Rules for Conversions between `any_iterator` Types

Let `ait_source` and `ait_target` be two different instantiations of the `any_iterator` class template. Then there is a conversion from `ait_source` to `ait_target` if and only if either

- The traversal category of `ait_source` is better than or equal to the traversal category of `ait_target`, and all other iterator traits are exactly the same,

or

- `ait_target` is a const iterator version of `ait_source`.