

Good Code & Good Advice



Introduction

*How do you iterate over the seconds of a list of pairs? I recently answered that question on the mailing list for Boost¹ users² regarding the library Boost.Iterator. Actually, there were several posts on the topic, and it turned out that **the best solution to the problem wasn't necessarily the best answer**. How can that be? The short answer: You have to know your audience in order to determine what constitutes good code and good advice. (Part of the audience for this article has no idea what the first question means. Hang in there; all will be revealed.)*

The Problem

You are writing a class, `some_class`, which contains a `std::list`. The elements of the list are pairs of doubles and floats, like this: `std::pair<double, float>`. Now you need to let users of `some_class` iterate over the `float` elements (accessed through the member `std::pair::second`) of the pairs. You want to expose iterators for doing that and someone told you that there's a powerful iterator adaptor in Boost.Iterator called `transform_iterator` that can help. You give it your best shot. You fail. And now you ask for help.

Starting from the End

Here is an elegant solution to the problem.

```
class some_class
{
public:
    typedef std::list<std::pair<double, float> > container_type;
    typedef boost::transform_iterator<
        boost::function<float& (container_type::reference)>, container_type::iterator>
        iterator;
```

¹ The most important set of C++ libraries in the world and also a community of skilled programmers.

² A mailing list where users of the fantastic Boost libraries ask and answer questions on how to use the libraries.

```

iterator begin()
{
    return boost::make_transform_iterator(
        container_.begin(),
        boost::bind<float&>(&container_type::value_type::second, _1));
}

iterator end()
{
    return boost::make_transform_iterator(
        container_.end(),
        boost::bind<float&>(&container_type::value_type::second, _1));
}
};

```

Say What?

If you understand the solution, that's great. It means you already know a lot about the C++ Standard Library³ containers⁴, how to use Boost.Bind⁵ and Boost.Function⁶, and are therefore able to deduce how the iterator adaptor `boost::transform_iterator`⁷ is used to solve the problem. But not everyone knows about these auxiliary topics, and to make matters worse, the use of templates makes the code scary and hard to understand for a lot of programmers⁸. If you are one of those programmers, how can you possibly learn something from this "answer"? And if you came across code like this in the Real World™ and needed to maintain it, could you do it?

Start from the Beginning!

So someone wants to know how to iterate over the seconds of a list of pairs. To do that with the help of Boost, they will need to know only about the following (since they've come this far we can safely conclude that they can handle the C++ Standard Library containers):

- Function objects. Is it safe to assume they do? Let's answer yes to this one.
- How to use `transform_iterator`. Here's where we need to focus our efforts.

Now it's clear that the solution I presented in the beginning requires much more knowledge than is actually necessary to answer the question. Therefore the solution might be too complex and could be simplified. Here's an approximation of the modified answer I offered on the mailing list:

1. Write a function object that extracts the second element of the pair.
2. Use the `transform_iterator` adaptor to provide you with the iterators for your class.
3. Use the helper function `make_transform_iterator()` in your `begin()` and `end()` functions to create the iterators.

And given the above it was straightforward to provide a simple, complete, and hopefully helpful example:

³ Professional C++ programmers will have a hard time understanding that lots of other professional C++ programmers are oblivious to the fact that there's even something called `std::pair` in the header `<utility>`.

⁴ It's true. For example, not everyone knows about the typedefs in the container types of the C++ Standard Library.

⁵ A generalization of `bind1st` and `bind2nd` from the C++ Standard Library. It's a great way of composing function objects.

⁶ A generalized callback, i.e., a deferred call to a function or function object.

⁷ An adaptor that applies a function object to the dereferenced iterator and returns the result.

⁸ Don't look so surprised! You can do a lot of powerful things in C++ without ever touching templates.

```

#include <iostream>
#include <list>
#include <utility>

#include "boost/iterator/transform_iterator.hpp"

template <typename T> class pair_second
{
public:
    typedef T& result_type;

    template <typename U> result_type operator()(std::pair<U,T>& element) const
    {
        return element.second;
    }
};

class some_class
{
public:
    typedef std::list<std::pair<double,float> > container_type;
    typedef boost::transform_iterator<
        pair_second<float>, container_type::iterator> iterator;

    some_class()
    {
        container_.push_back(std::make_pair(2.123, 3.14f));
        container_.push_back(std::make_pair(4.123, 4.14f));
        container_.push_back(std::make_pair(7.123, 5.14f));
        container_.push_back(std::make_pair(8.123, 6.14f));
    }

    iterator begin()
    {
        return boost::make_transform_iterator(
            container_.begin(),
            pair_second<float>());
    }

    iterator end()
    {
        return boost::make_transform_iterator(
            container_.end(),
            pair_second<float>());
    }

private:
    container_type container_;
};

int main(int argc, char** argv)
{
    some_class sc;

    some_class::iterator it, end;

    it = sc.begin();
    end = sc.end();

    while (it != end)
    {
        std::cout << *it++ << '\n';
    }

    return 0;
}

```

A simple question and a simple answer using a simple example. Only some annotations to the code remains (see Appendix A: Annotated example). No need to involve other libraries for the terse and more elegant solution.

But I'm not just saying that in order for you to marvel at my pedagogic devices...**perhaps something was lost on the way towards "simple"**? (And if you still have a puzzled look on your face after reading the sample code it's because you haven't read the Boost.Iterator documentation available at www.boost.org.)

Coding in the Real World

What if this wasn't an example, but rather new code in the system we're working on. Should I (and you) write the sweet and short version – or the simplified version that more people are likely to understand? Someone asked that exact question on the mailing list and noted that since he typically went for the best solutions⁹, he was sometimes criticized for writing too complex code, and by extension for not being a team player. Is that really a fair criticism? Or is it the other way around; is it in fact condescending and patronizing to (over)simplify solutions for your peers who may or may not understand the tools you're using?

Here's what I responded to that question, and the answer holds an important truth when writing code that others need to understand.

"When my peers think I write overly complex code they are always right. There are two probable reasons:

- 1. I'm writing overly complex code.*
- 2. I'm wielding tools (in the language, libraries, or in the domain model) that are currently beyond them.*

The great opportunity here is to teach my peers about the tools I'm using. Or, if I find that there are no such tools, realize that I've written overly complex code and simplify it."

Of course, this is what I *said* – not necessarily the way I actually think. But I would love to be a person who works that way. And I would love for you to do the same; because it seems to actually be possible to write the code you want (using every power tool you can think of) **and** to also make those tools available to your peers. If we aim for that, everybody wins. Oh, I've put my money where my mouth is in "Appendix B: Short and Sweet".

Know Your Audience

I conclude that it's certainly true that you need to know your audience in order to write code – or answer questions – that they will understand, use and appreciate. But it's also true that **you have the power to influence your audience**. You can learn from them and give them the opportunity to learn from you. Rather than go looking for that boring lowest common denominator, why not stretch your mind – and the minds of others' – and try to go beyond? Want my advice on where to start doing that when it comes to C++? www.boost.org.

Thank you for reading,
Bjorn Karlsson

⁹ Clearly a matter of both taste and competence, but here it means that sweet, terse, generic, and locally defined implementation that you saw in the beginning of the text.

Acknowledgements

- Robert Jones provided both important questions and answers for the article.
- Zachary Turner posted the original question on Boost users' mailing list.
- David Abrahams, Jeremy Siek, and Thomas Witt, wrote Boost.Iterator.
- Peter Dimov wrote Boost.Bind.
- Douglas Gregor wrote Boost.Function.

References and Resources

Online

- Boost, www.boost.org.
- Boost.Iterator, http://www.boost.org/doc/libs/1_39_0/libs/iterator.
- Boost.Bind, http://www.boost.org/doc/libs/1_39_0/libs/bind.
- Boost.Function, http://www.boost.org/doc/libs/1_39_0/libs/function.

Boost Books

- Jeremy Siek, Lie-Quan Lee and Andrew Lumsdaine, *The Boost Graph Library*. Addison-Wesley, 2002.
- David Abrahams and Aleksey Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, November, 2004.
- Björn Karlsson, *Beyond the C++ Standard Library: An Introduction to Boost*. Addison-Wesley, August, 2005.

Appendix A: Annotated example

```
#include <iostream>
#include <list>
#include <utility>

#include "boost/iterator/transform_iterator.hpp"

#include "boost/function.hpp"
#include "boost/bind.hpp"

// The class pair_second has only one purpose in life:
// Providing a function call operator that accepts a reference to a pair,
// and to return the second element of that pair. The template accepts
// a type T that is the type of that second element.
template <typename T> class pair_second
{
public:
    // The typedef result_type is needed if you want to use
    // this function object together with Boost.Bind and
    // play nicely with the C++ Standard Library. We also
    // need it so that transform_iterator can know the
    // result type of our function call operator!
    typedef T& result_type;

    // The type U is automatically deduced when the function call
    // operator is invoked.
    template <typename U> result_type operator()(std::pair<U,T>& element) const
    {
        return element.second;
    }
};

// The class some_class needs to expose iterators that transform
// existing iterators (that yield std::pair<double,float>&)
// to iterators that yield float&.
class some_class
{
public:
    // The typedef for the container, a list with elements that are pairs
    // (the first part is a double, the second part is a float).
    typedef std::list<std::pair<double,float> > container_type;

    // The typedef for our iterator. The template arguments to boost::transform_iterator are:
    // * The type of the function object
    // * The container's iterator type (the one to be transformed)
    // The resulting type (here: called iterator) is a standard-conforming iterator,
    // very hard to write by hand
    typedef boost::transform_iterator<pair_second<float>, container_type::iterator> iterator;

    some_class()
    {
        // Let's add some sample elements to our container.
        // The helper function make_pair() deduces the types of its
        // arguments and in our case creates a std::pair<double,float>.
        container_.push_back(std::make_pair(2.123, 3.14f));
        container_.push_back(std::make_pair(4.123, 4.14f));
        container_.push_back(std::make_pair(7.123, 5.14f));
        container_.push_back(std::make_pair(8.123, 6.14f));
    }

    // Our member function begin() will use a helper function called
    // make_transform_iterator -- it deduces the types of the arguments
    // and gives us an iterator of the type that we defined earlier.
    // * The first argument is the container_'s begin() iterator.

```

A function call operator is a member function that lets your classes be invoked using the same syntax as functions.

```

// * The second argument is an instance of our pair_second function object.
iterator begin()
{
    return boost::make_transform_iterator(container_.begin(), pair_second<float>());
}

iterator end()
{
    return boost::make_transform_iterator(container_.end(), pair_second<float>());
}

private:
// Here is the internal list that holds pairs of doubles and floats.
container_type container_;
};

int main(int argc, char* argv[])
{
    // Create an instance of some_class.
    some_class sc;

    // Create iterators
    some_class::iterator it, end;

    it = sc.begin();
    end = sc.end();

    // When dereferencing the iterator,
    // we get a reference to the float in the corresponding pair.
    while (it != end)
    {
        std::cout << *it++ << '\n';
    }

    return 0;
}

```

Appendix B: Short and Sweet

It's time to get a bit more creative than with that long-winded function object `pair_second`. First we are going to generalize the function object that we use for our `transform_iterator`. You'll remember that we previously encoded our function object type in the `transform_iterator` like so:

```
typedef boost::transform_iterator<pair_second<float>, container_type::iterator> iterator;
```

But let's say that we didn't have the time to write our nice function object `pair_second`. Instead, we had to use a function that someone had already written. Here it is:

```
float& get_second(std::pair<double, float>& pair)
{
    return pair.second;
}
```

I know what you're thinking – that's not a function *you* would have written. However! That's what we are forced to work with. Your boss has written it, she is very proud of it, and forcefully insists that you go ahead and use it. Here's our first stab at the new `transform_iterator`:

```
typedef boost::transform_iterator<
    float& (container_type::reference),
    container_type::iterator> iterator;
```

The first template argument is the type of the function, i.e., `float& (container_type::reference)`. But wait, what about the required `result_type`? There's just no way to put that information inside our boss' little function, and the compiler will complain and say something like:

```
transform_iterator.hpp(43) : error C2825: 'UnaryFunc': must be a class or namespace when followed by ':'
```

To get around this problem and use the nasty `get_second()` function together with the iterator adaptor we need a function object (as you recall, a function object is any class that provides the function call operator) that can wrap the function and provide:

- 1) The required typedef `result_type` that lets `transform_iterator` know about the return type of the function call operator.
- 2) A function call operator that invokes the wrapped function.

Such a wrapper would in effect be a *generalized callback*. It could potentially store any compatible function or function object, and it would be possible to invoke it as many times as we'd like. Fortunately, we don't have to worry about writing that wrapper – it already exists and is called `Boost.Function`. Here's how we would create an instance of `boost::function` that's compatible with `get_second()`:

```
boost::function<float& (std::pair<double, float>&)> func;
```

The syntax of the template argument is similar to how a function is declared. Just start with the return type and enclose the function parameters in parenthesis. To assign `get_second()` to `func`, we do this:

```
func = &get_second;
```

Now, to invoke this function wrapper we need a `std::pair` and then we're ready to give it a go:

```
std::pair<double, float> pair(0.01, 0.02f);
float f = func(pair);
```

Armed with this basic knowledge about Boost.Function we are now in a position to change our code to become compatible with `get_second()`.

```
typedef boost::transform_iterator<
    boost::function<float& (container_type::reference)>,
    container_type::iterator> iterator;
```

And now we can change our implementation of `begin()` and `end()` to use `get_second()` – but note that it also will work with our function object `pair_second!`

```
iterator begin()
{
    return boost::make_transform_iterator(container_.begin(), &get_second);
}
```

So far we've done little more than make our boss happy¹⁰ and our solution a little more generic. But we're not quite done yet. **It's time to make our boss angry**¹¹. I'm sure you agree that the implementation of `get_second()` is trivial. Its only job is to extract the second element and return it. Wouldn't it be nice if we could express that directly when creating the `transform_iterator`, rather than having to keep the logic in a separate place, namely the implementation of `get_second()`?

It turns out that we can express that with the help of another library. Boost.Bind is intended for functional composition and it works with functions, function pointer, member function pointers, and function objects. We can use it to *bind* to the `std::pair` member `second` and thusly create a function object compatible with our `boost::function`:

```
boost::function<float& (std::pair<double, float>&)> func;
func = boost::bind<float&>(&container_type::value_type::second, _1);
```

The first argument to `bind()` passes the pointer to `std::pair<double, float>::second` (in the line above spelled `&container_type::value_type::second`) and the second argument is the *placeholder* `_1` used to tell `bind()` that "this argument is to be provided when the function object you create for me is invoked". As a result we get a unary function object with a `const` function call operator that returns a reference to `float` and expects a reference to `std::pair<double, float>` as its argument. And of course, such a function object can be used to implicitly create an instance of `boost::function`.

Note that we tell `bind()` about the fact that it's supposed to return a *reference* to `float` rather than a plain `float` by explicitly providing the return type as a template argument (when we say `bind<float&>(...)`). If we didn't need to modify the return value we could omit the return type and have it automatically deduced:

```
func = boost::bind(&container_type::value_type::second, _1);
```

In order for us to use `boost::function` and `boost::bind` in our code we need to include their headers.

```
#include "boost/function.hpp"
#include "boost/bind.hpp"
```

¹⁰ That's still worth something!

¹¹ Oh. Forget about the previous footnote.

And now we have all the tools and knowledge needed to create that short and sweet version of `some_class`. Here is the code again; read through it carefully, and now you should be able to understand the magic that takes place with the help of these useful Boost libraries.

```
class some_class
{
public:
    typedef std::list<std::pair<double, float> > container_type;
    typedef boost::transform_iterator<
        boost::function<float& (container_type::reference)>, container_type::iterator>
        iterator;

    iterator begin()
    {
        return boost::make_transform_iterator(
            container_.begin(),
            boost::bind<float&>(&container_type::value_type::second, _1));
    }

    iterator end()
    {
        return boost::make_transform_iterator(
            container_.end(),
            boost::bind<float&>(&container_type::value_type::second, _1));
    }
};
```

As you can see, when calling `make_transform_iterator()` we create the function object on the fly, which keeps our simple logic right at the call site. It's short, it's sweet, and it's also quite cool. Does coolness have a place in contemporary C++ code? Of course it does, perhaps more than ever.

You will also be happy to know that `Boost.Bind` and `Boost.Function` are part of TR1 (the first Technical Report on C++ Library Extensions), which means that they are **extremely** likely to be part of the next revision of the C++ Standard, and that your compiler will soon be shipping with these great libraries. Of course, then you will need to spell them `std::bind` and `std::function`!