Boost.Foreach

Eric Niebler

Table of Contents

Introduction

> "Make simple things easy."
> **-- Larry Wall**

What is BOOST_FOREACH?

In C++, writing a loop that iterates over a sequence is tedious. We can either use iterators, which requires a considerable amount of boiler-plate, or we can use the `std::for_each()` algorithm and move our loop body into a predicate, which requires no less boiler-plate and forces us to move our logic far from where it will be used. In contrast, some other languages, like Perl, provide a dedicated "foreach" construct that automates this process. `BOOST_FOREACH` is just such a construct for C++. It iterates over sequences for us, freeing us from having to deal directly with iterators or write predicates.

`BOOST_FOREACH` is designed for ease-of-use and efficiency. It does no dynamic allocations, makes no virtual function calls or calls through function pointers, and makes no calls that are not transparent to the compiler's optimizer. This results in near-optimal code generation; the performance of `BOOST_FOREACH` is usually within a few percent of the equivalent hand-coded loop. And although `BOOST_FOREACH` is a macro, it is a remarkably well-behaved one. It evaluates its arguments exactly once, leading to no nasty surprises.

Hello, world!

Below is a sample program that uses `BOOST_FOREACH` to loop over the contents of a `std::string`.

```
#include <string>
#include <iostream>
#include <boost/foreach.hpp>

int main()
{
    std::string hello( "Hello, world!" );

    BOOST_FOREACH( char ch, hello )
    {
        std::cout << ch;
    }

    return 0;
}
```

This program outputs the following:

```
Hello, world!
```

Supported Sequence Types

`BOOST_FOREACH` iterates over sequences. But what qualifies as a sequence, exactly? Since `BOOST_FOREACH` is built on top of Boost.Range, it automatically supports those types which Boost.Range recognizes as sequences. Specifically, `BOOST_FOREACH` works with types that satisfy the Single Pass Range Concept. For example, we can use `BOOST_FOREACH` with:

- STL containers
- arrays
- Null-terminated strings (`char` and `wchar_t`)
- std::pair of iterators

Note

The support for STL containers is very general; anything that looks like an STL container counts. If it has nested `iterator` and `const_iterator` types and `begin()` and `end()` member functions, `BOOST_FOREACH` will automatically know how to iterate over it. It is in this way that `boost::iterator_range<>` and `boost::sub_range<>` work with `BOOST_FOREACH`.

See the section on Extensibility to find out how to make `BOOST_FOREACH` work with other types.

Examples

Below are some examples that demonstrate all the different ways we can use `BOOST_FOREACH`.

Iterate over an STL container:

```
std::list<int> list_int( /*...*/ );
BOOST_FOREACH( int i, list_int )
{
    // do something with i
}
```

Iterate over an array, with covariance (i.e., the type of the iteration variable is not exactly the same as the element type of the container):

```
short array_short[] = {1,2,3};
BOOST_FOREACH( int i, array_short )
{
    // The short was implicitly converted to an int
}
```

Predeclare the loop variable, and use `break`, `continue`, and `return` in the loop body:

```
std::deque<int> deque_int( /*...*/ );
int i = 0;
BOOST_FOREACH( i, deque_int )
{
    if( i == 0 ) return;
    if( i == 1 ) continue;
    if( i == 2 ) break;
}
```

Iterate over a sequence by reference, and modify the underlying sequence:

```
short array_short[] = { 1, 2, 3 };
BOOST_FOREACH( short & i, array_short )
{
    ++i;
}
// array_short contains {2,3,4} here
```

Iterate over a vector of vectors with nested `BOOST_FOREACH` loops. In this example, notice that braces around the loop body are not necessary:

```
std::vector<std::vector<int> > matrix_int;
BOOST_FOREACH( std::vector<int> & row, matrix_int )
    BOOST_FOREACH( int & i, row )
        ++i;
```

Iterate over an expression that returns a sequence by value (i.e. an rvalue):

```
extern std::vector<float> get_vector_float();
BOOST_FOREACH( float f, get_vector_float() )
{
    // Note: get_vector_float() will be called exactly once
}
```

Iterate in reverse:

```
std::list<int> list_int( /*...*/ );
BOOST_REVERSE_FOREACH( int i, list_int )
{
    // do something with i
}
```

Iterating over rvalues doesn't work on some older compilers. Check the Portability section to see whether your compiler supports this.

Making BOOST_FOREACH Prettier

People have complained about the name `BOOST_FOREACH`. It's too long. `ALL CAPS` can get tiresome to look at. That may be true, but `BOOST_FOREACH` is merely following the Boost Naming Convention. That doesn't mean you're stuck with it, though. If you would like to use a different identifier (`foreach_`, perhaps), you can simply do:

```
#define foreach_         BOOST_FOREACH
#define foreach_r_       BOOST_REVERSE_FOREACH
```

Only do this if you are sure that the identifier you choose will not cause name conflicts in your code.

Note

Do not use `#define foreach_(x,y) BOOST_FOREACH(x,y)`. This can be problematic if the arguments are macros themselves. This would result in an additional expansion of these macros. Instead, use the form shown above.

Lastly, a word of warning. Lots of folks use a `foreach` macro as a short form for `BOOST_FOREACH`. I discourage this. It leads to name conflicts within the `BOOST_FOREACH` macro itself, where `foreach` is the name of a namespace. Besides, `foreach` is a common-eough identifier; even Qt defines it as a macro. If you insist on using `foreach`, you might try something like this:

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

namespace boost
{
    // Suggested work-around for https://svn.boost.org/trac/boost/ticket/6131
    namespace BOOST_FOREACH = foreach;
}

#define foreach   BOOST_FOREACH
```

This will work around *some* of the problem you're likely to encounter, but not all. Prefer using a different identifier.

Extensibility

If we want to use `BOOST_FOREACH` to iterate over some new collection type, we must "teach" `BOOST_FOREACH` how to interact with our type. Since `BOOST_FOREACH` is built on top of Boost.Range, we must extend Boost.Range in order to extend `BOOST_FOREACH`. The section Extending Boost.Range explores this topic in detail.

Below is an example for extending `BOOST_FOREACH` to iterate over a sub-string type, which contains two iterators into a `std::string`.

```
namespace my
{
    // sub_string: part of a string, as delimited by a pair
    // of iterators
    struct sub_string
    {
        std::string::iterator begin;
        std::string::iterator end;

        /* ... implementation ... */
    };

    // Add overloads of range_begin() and range_end() in the
    // same namespace as sub_string, to be found by Argument-Dependent Lookup.

    inline std::string::iterator range_begin( sub_string & x )
    {
        return x.begin;
    }

    inline std::string::iterator range_end( sub_string & x )
    {
        return x.end;
    }

    // Also add overloads for const sub_strings. Note we use the conversion
    // from string::iterator to string::const_iterator here.

    inline std::string::const_iterator range_begin( sub_string const & x )
    {
        return x.begin;
    }

    inline std::string::const_iterator range_end( sub_string const & x )
    {
        return x.end;
    }
}

namespace boost
{
    // specialize range_mutable_iterator and range_const_iterator in namespace boost
    template<>
    struct range_mutable_iterator< my::sub_string >
    {
        typedef std::string::iterator type;
    };

    template<>
    struct range_const_iterator< my::sub_string >
    {
        typedef std::string::const_iterator type;
    };
}
```

Now that we have taught Boost.Range (and hence BOOST_FOREACH) about our type, we can now use BOOST_FOREACH to iterate over our sub_string type.

```
my::sub_string substr;
BOOST_FOREACH( char ch, substr )
{
    // Woo-hoo!
}
```

There are some portability issues we should be aware of when extending BOOST_FOREACH. Be sure to check out the Portability section. In particular, if your compiler does not support Argument-Dependent Lookup, the Boost.Range Portability section offers some suggested work-arounds.

Making BOOST_FOREACH Work with Non-Copyable Sequence Types

For sequence types that are non-copyable, we will need to tell BOOST_FOREACH to not try to make copies. If our type inherits from boost::noncopyable, no further action is required. If not, we must specialize the boost::foreach::is_noncopyable<> template, as follows:

```
class noncopy_vector
{
    // ...
private:
    noncopy_vector( noncopy_vector const & ); // non-copyable!
};

namespace boost { namespace foreach
{
    template<>
    struct is_noncopyable< noncopy_vector >
      : mpl::true_
    {
    };
}}
```

Another way to achieve the same effect is to override the global boost_foreach_is_noncopyable() function. Doing it this way has the advantage of being portable to older compilers.

```
// At global scope...
inline boost::mpl::true_ *
boost_foreach_is_noncopyable( noncopy_vector *&, boost::foreach::tag )
{
    return 0;
}
```

Tip

Even though we have to tell BOOST_FOREACH that our type is non-copyable, that doesn't mean that BOOST_FOREACH always makes a copy of our sequence type. Obviously, doing so would be expensive and even wrong in some cases. BOOST_FOREACH is quite smart about when to make a copy and when not to. The is_noncopyable<> trait is needed to elide the copy, which is on a branch that might never get taken.

Optimizing BOOST_FOREACH for Lightweight Proxy Sequence Types

On some compilers, BOOST_FOREACH must occasionally take a slightly slower code path to guarantee correct handling of sequences stored in temporary objects. It asks itself, "Should I make a copy of this object?" and later, "Did I make a copy or not?" For some

types of sequences, this is overkill. Consider a sequence which is a simple pair of iterators. Jumping through hoops of fire to avoid copying it doesn't make sense because copying it is so cheap.

A pair of iterators is an example of a lightweight proxy. It does not store the values of the sequence; rather, it stores iterators to them. This means that iterating over a copy of the proxy object will give the same results as using the object itself. For such types, `BOOST_FOREACH` provides a hook that lets us tell it not to worry about the expense of making a copy. This can result in slightly faster loop execution. Simply specialize the `boost::foreach::is_lightweight_proxy<>` trait, as follows:

```
struct sub_string
  : boost::iterator_range< std::string::iterator >
{
    // ...
};

namespace boost { namespace foreach
{
    template<>
    struct is_lightweight_proxy< sub_string >
      : mpl::true_
    {
    };
}}
```

Alternately, we could achieve the same effect by overriding the global `boost_foreach_is_lightweight_proxy()` function, as follows:

```
// At global scope...
inline boost::mpl::true_ *
boost_foreach_is_lightweight_proxy( sub_string *&, boost::foreach::tag )
{
    return 0;
}
```

This method is portable to older compilers.

Portability

BOOST_FOREACH uses some fairly sophisticated techniques that not all compilers support. Depending on how compliant your compiler is, you may not be able to use BOOST_FOREACH in some scenarios. Since BOOST_FOREACH uses Boost.Range, it inherits Boost.Range's portability issues. You can read about those issues in the Boost.Range Portability section.

In addition to the demands placed on the compiler by Boost.Range, BOOST_FOREACH places additional demands in order to handle rvalue sequences properly. (Recall that an rvalue is an unnamed object, so an example of an rvalue sequence would be a function that returns a `std::vector<>` by value.) Compilers vary in their handling of rvalues and lvalues. To cope with the situation BOOST_FOREACH defines three levels of compliance, described below:

Table 1. BOOST_FOREACH Compliance Levels

Level	Meaning
Level 0	*Highest level of compliance* BOOST_FOREACH works with lvalues, rvalues and const-qualified rvalues.
Level 1	*Moderate level of compliance* BOOST_FOREACH works with lvalues and plain rvalues, but not const-qualified rvalues. BOOST_FOREACH_NO_CONST_RVALUE_DETECTION is defined in this case.
Level 2	*Lowest level of compliance* BOOST_FOREACH works with lvalues only, not rvalues. BOOST_FOREACH_NO_RVALUE_DETECTION is defined in this case.

Below are the compilers with which BOOST_FOREACH has been tested, and the compliance level BOOST_FOREACH provides for them.

Table 2. Compiler Compliance Level

Compiler	Compliance Level
Visual C++ 8.0	Level 0
Visual C++ 7.1	Level 0
Visual C++ 7.0	Level 2
Visual C++ 6.0	Level 2
gcc 4.0	Level 0
gcc 3.4	Level 0
gcc 3.3	Level 0
mingw 3.4	Level 0
Intel for Linux 9.0	Level 0
Intel for Windows 9.0	Level 0
Intel for Windows 8.0	Level 1
Intel for Windows 7.0	Level 2
Comeau 4.3.3	Level 0
Borland 5.6.4	Level 2
Metrowerks 9.5	Level 1
Metrowerks 9.4	Level 1
SunPro 5.8	Level 2
qcc 3.3	Level 0
tru64cxx 65	Level 2
tru64cxx 71	Level 2

Pitfalls

This section describes some common pitfalls with `BOOST_FOREACH`.

Types With Commas

Since `BOOST_FOREACH` is a macro, it must have exactly two arguments, with exactly one comma separating them. That's not always convenient, especially when the type of the loop variable is a template. Consider trying to iterate over a `std::map`:

```
std::map<int,int> m;

// ERROR! Too many arguments to BOOST_FOREACH macro.
BOOST_FOREACH(std::pair<int,int> p, m) // ...
```

One way to fix this is with a typedef.

```
std::map<int,int> m;
typedef std::pair<int,int> pair_t;

BOOST_FOREACH(pair_t p, m) // ...
```

Another way to fix it is to predeclare the loop variable:

```
std::map<int,int> m;
std::pair<int,int> p;

BOOST_FOREACH(p, m) // ...
```

Hoisting and Iterator Invalidation

Under the covers, `BOOST_FOREACH` uses iterators to traverse the element sequence. Before the loop is executed, the end iterator is cached in a local variable. This is called *hoisting*, and it is an important optimization. It assumes, however, that the end iterator of the sequence is stable. It usually is, but if we modify the sequence by adding or removing elements while we are iterating over it, we may end up hoisting ourselves on our own petard.

Consider the following code:

```
std::vector<int> vect(4, 4);
BOOST_FOREACH(int i, vect)
{
    vect.push_back(i + 1);
}
```

This code will compile, but it has undefined behavior. That is because it is logically equivalent to the following:

```
std::vector<int> vect(4, 4);
for(std::vector<int>::iterator it1 = vect.begin(), it2 = vect.end();
    it1 != it2; ++it1)
{
    int i = *it1;
    vect.push_back(i + 1); // Oops! This invalidates it1 and it2!
}
```

The call to `vect.push_back()` will cause all iterators into `vect` to become invalid, including `it1` and `it2`. The next iteration through the loop will cause the invalid iterators to be used. That's bad news.

The moral of the story is to think twice before adding and removing elements from the sequence over which you are iterating. If doing so could cause iterators to become invalid, don't do it. Use a regular `for` loop instead.

History and Acknowledgements

History

The ideas for `BOOST_FOREACH` began life in the Visual C++ group at Microsoft during the early phases of the design for C++/CLI. Whether to add a dedicated "foreach" looping construct to the language was an open question at the time. As a mental exercise, Anson Tsao sent around some proof-of-concept code which demonstrated that a pure library solution might be possible. The code was written in the proposed C++/CLI dialect of the time, for which there was no compiler as of yet. I was intrigued by the possibility, and I ported his code to Managed C++ and got it working. We worked together to refine the idea and eventually published an article about it in the November 2003 issue of the CUJ.

After leaving Microsoft, I revisited the idea of a looping construct. I reimplemented the macro from scratch in standard C++, corrected some shortcomings of the CUJ version and rechristened it `BOOST_FOREACH`. In October of 2003 I began a discussion about it on the Boost developers list, where it met with a luke-warm reception. I dropped the issue until December 2004, when I reimplemented `BOOST_FOREACH` yet again. The new version only evaluated its sequence expression once and correctly handled both lvalue and rvalue sequence expressions. It was built on top of the recently accepted Boost.Range library, which increased its portability. This was the version that, on Dec. 12 2004, I finally submitted to Boost for review. It was accepted into Boost on May 5, 2005.

Acknowledgements

Thanks go out to Anson Tsao of Microsoft for coming up with the idea and demonstrating its feasibility. I would also like to thank Thorsten Ottosen for the Boost.Range library, on which the current version of `BOOST_FOREACH` is built. Finally, I'd like to thank Russell Hind, Alisdair Meredith and Stefan Slapeta for their help porting to various compilers.

Further Reading

For more information about how `BOOST_FOREACH` works, you may refer to the article “Conditional Love” at The C++ Source.