Policy Adaptors and the Boost Iterator Adaptor Library

David Abrahams[†] and Jeremy Siek[‡*]

[†] Altra Broadband
abrahams@altrabroadband.com

[‡] AT&T Labs - Research
Florham Park, NJ 07932, USA
jsiek@research.att.com

Abstract. The iterator abstraction is one of the most commonly used in programming, but implementing an iterator type can be challenging. The requirements for a standard-conforming iterator are at once tedious and subtle: tedious because much of an iterator's rich interface is "boiler-plate" surrounding a few core operations, and subtle because of the intricate details involved in getting that interface right. This paper presents the generalized iterator template from the Boost Iterator Adaptor Library. In addition to automating the error-prone and redundant job of implementing new iterator types, the library simplifies the creation of iterator types that are variations on other iterators (adapted iterators) and generators of new iterator families (iterator adaptors). The Iterator Adaptor Library is an example of policy-based design and employs template meta-programming. We also present the Policy Adapter implementation pattern, a strategy which can also be used to generate new representatives of other abstract concept families.

1 Introduction

Iterators play an important role in modern C++ programming. The iterator is the central abstraction of the algorithms of the Standard Library, allowing algorithms to be re-used in in a wide variety of contexts.

1.1 Iterators

The power of iterators derives from several key features:

*This work was partially supported by NSF grant ACI-9982205.

- Iterators form a rich *family* of concepts[1] whose functionality varies along several axes: movement, dereferencing, and associated type exposure.
- The iterator concepts of the C++ standard form a refinement hierarchy which allows the same basic interface elements to implement diverse functionality.
- Because built-in pointer types model the RandomAccessIterator concept, iterators can be both efficient and convenient to use.

The C++ Standard Library contains a wide variety of useful iterators. Every one of the standard containers comes with constant and mutable iterators[2], and also `reverse_` versions of those same iterators which traverse the container in the opposite direction. The Standard also supplies `istream_iterator` and `ostream_iterator` for reading from and writing to streams, `insert_iterator`, `front_insert_iterator` and `back_insert_iterator` for inserting elements into containers, and `raw_storage_iterator` for initializing raw memory [7].

Despite the many iterators supplied by the Standard Library, many obvious iterators are missing, and creating new iterator types is still a common task for C++ programmers. The literature documents several of these, for example `line_iterator` [3] `Constant_iterator` [9]. The iterator abstraction is so powerful, however, that we expect programmers will always need to invent new iterator types.

1.2 Adaptors

Because iterators combine traversal, indirection, and associated type exposure, it is common to want to adapt one iterator to form a new one. This strategy allows one to reuse some of original iterator's axes of variation while redefining others. For example, the Standard provides `reverse_iterator`, which adapts any BidirectionalIterator by inverting its direction of traversal.

As with plain iterators, iterator adaptors defined outside the Standard have become commonplace in the literature:

- `Checked_iter` [13] adds bounds-checking to an existing iterator.
- The iterators of the View Template Library [14], which adapts containers, are themselves adaptors over the underlying iterators.
- smart iterators [5] adapt an iterator's dereferencing behavior by applying a function object to the object being referenced and returning the result.
- Custom iterators [4], in which a variety of adaptor types are enumerated.
- compound iterators [1], which access a slice out of a container of containers.

[1] We use the term *concept* to mean a set of requirements that a type must satisfy to be used with a particular template parameter.

[2] The term *mutable iterator* refers to iterators over objects that can be changed by assigning to the dereferenced iterator, while *constant iterator* refers to iterators over objects that cannot be modified.

- Several iterator adaptors from the MTL [12]. The MTL contains a strided iterator, where each call to `operator++()` moves the iterator ahead by some constant factor, and a scaled iterator, which multiplies the dereferenced value by some constant.

2 The Design of the Boost Iterator Adaptor Library

To automate the repetitive work of constructing iterators, one would need a generator of new iterator types that can accommodate all the ways in which iterators vary. One could then make new iterators with relative ease, specifying the parts that matter and letting the library do the rest. To that end, the Boost Iterator Adaptor Library provides a fully-generalized iterator called `iterator_adaptor`. The `iterator_adaptor` class template adapts a `Base` type, (usually an iterator), to produce a new adapted iterator type.[3]

2.1 Core Elements of the Iterator Concept

The first step in designing such a generalized model of the iterator concept is to identify the core elements of its interface. We have identified the following core behaviors for iterators:

- dereferencing
- incrementing
- decrementing
- equality comparison
- random-access motion
- distance measurement

In addition to the behaviors listed above, the core interface elements include the associated types exposed through `iterator_traits`: `value_type`, `reference`, `pointer`, and `iterator_category`. The library supports two ways of specifying these: as traditional template parameters and also as *named* template parameters (described below), and uses a system of smart defaults which in most cases reduces the number of these types that must be specified.

2.2 From Building Models to Building Adaptors

A generalized iterator generator is useful (helping to create new iterator types from scratch), but a generalized iterator *adaptor* is even more useful. An adaptor generator allows one to build whole families of iterator instances based on existing iterators.

[3] The term"`Base`" is not meant to imply the use of inheritance. We have followed the lead of the standard library, which provides a `base()` function to access the underlying iterator object of a `reverse_iterator` adaptor.

In the Boost Iterator Adaptor Library, the `iterator_adaptor` class template plays the roles of both iterator generator and iterator adaptor generator. The behaviors of `iterator_adaptor` instances are supplied through a policies class [2] which allows users to specialize adaptation. Users go beyond generating new iterator types to easily generating new iterator adaptor families.

The library contains several examples of specialized adaptors which were quickly implemented using `iterator_adaptor`:

- Indirect Iterator Adaptor, which iterates over iterators, pointers, or smart pointers and applies an extra level of dereferencing.
- Reverse Iterator Adaptor, which inverts the direction of a `Base` iterator's motion, while allowing adapted constant and mutable iterators to interact in the expected ways. We will discuss this further in Section 5.2.1.
- Transform Iterator Adaptor, which applies a user-defined function object to the underlying values when dereferenced. We will show how this adaptor is implemented in Section 3.1.
- Projection Iterator Adaptor, which is similar to Transform Iterator Adaptor except that when dereferenced it returns by-reference instead of by-value.
- Filter Iterator Adaptor, which provides a view of an iterator range in which some elements of the underlying range are skipped.
- Counting Iterator Adaptor, which adapts any incrementable type (e.g. integers, iterators) so that incrementing/decrementing the adapted iterator and dereferencing it produces successive values of the `Base` type.
- Function Output Iterator Adaptor, which makes it easier to create custom output iterators.

Based on the examples in the library, users have generated many new adaptors, among them a permutation adaptor which applies some permutation to a RandomAccessIterator, and a strided adaptor, which adapts a RandomAccessIterator by multiplying its unit of motion by a constant factor. In addition, the Boost Graph Library (BGL) uses iterator adaptors to adapt other graph libraries, such as LEDA [10] and Stanford GraphBase [8], to the BGL interface (which requires C++ Standard compliant iterators).

3 The Boost `iterator_adaptor` Class Template

The `iterator_adaptor` class template simplifies the creation of iterators by automating the implementation of redundant operators and delegating functions and by taking care of the complex details of iterator implementation.

The central design feature of `iterator_adaptor` is parameterization by a policies class. The policies class is the primary communication mechanism between the iterator implementer and the `iterator_adaptor`; it specifies how the new iterator type

behaves. Unlike the policy classes in [2], we group several policies into a single class as this proved more convenient for iterator implementation.

3.1 Iterator Policies Class

The following example shows how to implement the policies class for a transform iterator adaptor: an iterator that applies some function to the value returned by dereferencing the base iterator. The `transform_iterator_policies` class is templated on the function object type, and a function object is stored as a data member of the policies class.

When adapting an underlying iterator, it is easiest to store any extra state needed by the resulting iterator in the policies class. The alternative is to wrap the underlying iterator in another class that contains the state, thereby incorporating the state into the `Base` type. This approach is much more work since the wrapping class will have to delegate many operations (instead of allowing the `iterator_adaptor` to implement the delegations).

The policies class inherits from `default_iterator_policies`, which delegates all other operations to the base iterator. The main event of the `transform_iterator_policies` class is the `dereference()` member function, which simply applies the function to the dereferenced value. The base iterator object is the second argument to the `dereference()` function. Because the iterator type is a template parameter of the `dereference()` function, the same concrete policies class can be used with any base iterator type, which greatly simplifies adaptation.

```
template <class AdaptableUnaryFunction>
struct transform_iterator_policies : public default_iterator_policies
{
  transform_iterator_policies() { }
  transform_iterator_policies(const AdaptableUnaryFunction& f)
    : m_f(f) { }

  template <class Reference, class Base>
  Reference dereference(type<Reference>, const Base& x) const
    { return m_f(*x); } // apply the function and return the result

  AdaptableUnaryFunction m_f;
};
```

Notes on the policies class implementation:

- Because `iterator_adaptor` stores an instance of the policies class as a data member, and all iterators are required to have default constructors, policies classes to be used with `iterator_adaptor` are also required to have default constructors.
- The `type<Reference>` parameter is an empty class used only to convey the appropriate return type to the `dereference()` function. Although it might have been more elegant to rely on the caller for explicit specification of the `Reference` template argument as in `policies.dereference<reference>(base_iterator)`, that approach proved not to to be portable to all of the targeted compilers.

With the policies class complete, the iterator implementer is almost finished, and only eleven lines of code have been written. The code consists of little more than the main idea of the transform iterator, applying a function to the result of dereferencing the base iterator. Next we will take a closer look at the `default_iterator_policies` class and then in §3.3 we will show how the transform iterator type is constructed using `iterator_adaptor`.

3.2 Default Iterator Policies Class

The `default_iterator_policies` class is the mechanism that automatically delegates operator implementations to the base iterator, freeing the iterator implementer from the tedious task of writing delegating functions. As above, an iterator policies class inherits from this class and overrides any functions that should not be delegated. The `default_iterator_policies` class also serves as an example of the iterator policies interface. There are six member functions corresponding to the core iterator operations and an `initialize()` function which provides a hook for customized iterator construction.

```
namespace boost {
  struct default_iterator_policies
  {
    template <class Base>
    void initialize(Base&) { }

    template <class Reference, class Base>
    Reference dereference(type<Reference>, const Base& x) const
      { return *x; }

    template <class Base>
    void increment(Base& x) { ++x; }

    template <class Base>
    void decrement(Base& x) { --x; }

    template <class Base, class Difference>
    void advance(Base& x, Difference n) { x += n; }

    template <class Difference, class Base1, class Base2>
    Difference distance(type<Difference>, const Base1& x,
                        const Base2& y) const { return y - x; }

    template <class Base1, class Base2>
    bool equal(const Base1& x, const Base2& y) const
      { return x == y; }
  };
} // namespace boost
```

3.3 Iterator Type Generator

In Section 3.1 we showed how to create the policy class for the transform iterator; the next step is to use the `iterator_adaptor` template to construct the actual iterator type. The best way to package the construction of the transform iterator is to create a

type generator: a class template whose sole purpose is to simplify the instantiation of some other complicated class template. It fulfills the same need as a template typedef would, if that were part of the C++ language. The first template parameter to the type generator is the type of the function object and the second is the base iterator type. The following code shows the type generator for the transform iterator.

```
template <class AdaptableUnaryFunction, class BaseIterator>
struct transform_iterator_generator
{
  typedef typename AdaptableUnaryFunction::result_type val_t;
public:
  typedef iterator_adaptor<BaseIterator,
    transform_iterator_policies<AdaptableUnaryFunction>,
    iterator_category_is<std::input_iterator_tag>,
    value_type_is<val_t>, reference_is<val_t> > type;
};
```

We use `iterator_adaptor` to define the transform iterator type as a nested `typedef` inside the `transform_iterator_generator` class. The first parameter to `iterator_adaptor` is the base iterator type and the second is the policies class. The remaining parameters specify the iterator's associated types and are given as *named parameters*. We will discuss this technique in §5.1.2.

The `iterator_category` is set to `std::input_iterator_tag` because the function object may return by-value. For the same reason the `reference` type (which will be the return type of `operator*`) is set to `val_t` (and not `val_t&`). There are two parameters that are left out: the `pointer` type defaults to `value_type*` and the `difference_type` defaults to the `difference_type` of the base iterator.

It is tempting to create a `transform_iterator` class template which is derived from `iterator_adaptor` instead of using the type generator. This approach does not work, for example, because the return type of `operator++` of an iterator is required to be the same iterator type, while in this case the return type would be `iterator_adaptor` and not `transform_iterator`.

3.4 Iterator Object Generator

Even though we now have a way to easily express the type of our transform iterator, writing the type down at all is often more trouble than it is worth. The AdaptableUnaryFunction's type alone could be quite complex if it were generated using standard library facilities such as `std::bind1st`, `std::bind2nd`, or `std::ptr_fun`. Declaring the entire transform iterator type can be much worse. For example, suppose we wanted to multiply the elements of a `set` by 2, and append them to a `list`. The transform iterator type might be declared as follows:

```
typedef transform_iterator_generator<
      std::binder2nd<std::multiplies<int> >,
      std::set<int, std::greater<int> >::const_iterator>::type
  int_set_doubler;
// to be continued...
```

Continuing our example, we find that the "adapting constructor" of `iterator_adaptor` is not always well-suited to easy construction of specialized adaptor types. That constructor is declared as follows:

```
iterator_adaptor(const Base& it, const Policies& p = Policies())
```

Because we stored the AdaptableUnaryFunction object (which may have state) inside the Policies class of our transform iterator, the user can't rely on the default constructor argument to generate the correct policies object. Instead the policies object must be explicitly constructed and passed to the adaptor's constructor:

```
// ...example (continued)
typedef transform_iterator_policies<
    std::binder2nd<std::multiplies<int> > > policies;

policies p(std::bind2nd(std::multiplies<int>(),2));

std::copy(
  int_set_doubler(my_set.begin(), p),
  int_set_doubler(my_set.end(), p),
  std::back_inserter(my_list));
```

If every use of transform iterator required this much code, users would quickly give up on it and use handwritten loops instead. Fortunately, the C++ standard library provides a useful precedent.

In the example above, `std::back_inserter` is a type of function called an *object generator* which returns an object of type `std::back_insert_iterator<my_list>`. An object generator allows the user to build adapted types "on the fly" and pass them directly to functions, so that no declaration is needed. This idiom is especially convenient in the case of iterators, since so many algorithms are implemented in terms of iterator ranges. We therefore recommend that iterator implementers create an object generator for their iterators. The object generator function for the transform iterator adaptor, `make_transform_iterator`, is shown below.[4]

```
template <class AdaptableUnaryFunction, class BaseIterator>
typename transform_iterator_generator<
    AdaptableUnaryFunction, BaseIterator>::type
make_transform_iterator(BaseIterator base,
    const AdaptableUnaryFunction& f = AdaptableUnaryFunction())
{
  typedef typename transform_iterator_generator<AdaptableUnaryFunction,
    BaseIterator>::type result_t;

  transform_iterator_policies<AdaptableUnaryFunction> policies(f);

  return result_t(base, policies);
}
```

[4] Although there is precedent in the standard for calling such an object generator, simply "`transform_iterator()`" (e.g. `std::back_inserter`), the standard also uses the more explicit "`make_`" prefix (e.g. `std::make_pair()`) and occasionally also reserves the simple name for the iterator type itself (e.g. `std::reverse_iterator`). In the end, the authors felt that explicit was better than implicit and decided to use the "`make_`" prefix for object generators.

With the object generator in place, we can considerably simplify the code for our previous example:

```
std::copy(
  make_transform_iterator(my_set.begin(),
      std::bind2nd(std::multiplies<int>(),2)),
  make_transform_iterator(my_set.end(),
      std::bind2nd(std::multiplies<int>(),2)),
  std::back_inserter(my_list));
```

3.5 Example Use of the Transform Iterator Adaptor

This example shows how a transform iterator can be used to negate the numbers over which it iterates.

```
#include <functional>
#include <algorithm>
#include <iostream>
#include <boost/iterator_adaptors.hpp>
int main(int, char*[])
{
  int x[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
  const int N = sizeof(x)/sizeof(int);
  std::cout << "negating the elements of the array:" << std::endl;
  std::copy(
      boost::make_transform_iterator(x, std::negate<int>()),
      boost::make_transform_iterator(x + N, std::negate<int>()),
      std::ostream_iterator<int>(std::cout, " "));
  std::cout << std::endl;
  return 0;
}
```

This output is:

```
-1 -2 -3 -4 -5 -6 -7 -8
```

4 The Policy Adaptor Design Pattern

The Iterator Adaptor Library illustrates how a generalized Model (`iterator_adaptor`) of a concept family (iterators) combined with default policy delegation allows users to easily build new Models and behavioral adaptors for existing Models. We can capture this strategy in the Policy Adaptor design pattern:[5]

1. Identify the core elements of the public interface of the concept family to be modeled. In our case, the Adaptor will model one of the iterator concepts: InputIterator, ForwardIterator, BidirectionalIterator, or RandomAccessIterator (this depends on the base iterator type and the parameters of the Adaptor).

2. Encapsulate core elements of the concept family in a Policies concept.

[5] This is not quite the same as the Policy Class pattern which has been discussed previously in the literature [2]. The construction of an adaptor which can easily transform existing Models into new ones is the key difference

3. Write a default policies class which delegates behavior to the public interface of the Adaptor's concept. This is the mechanism that supplies default adaptation behavior.

4. Build an Adaptor class template parameterized on Policies. The Adaptor should be a generalized model of the Adaptor Concept, providing the public interface, but delegating functionality to the policies class.

5. Store a member of the Policies parameter in the Adaptor template so that users can maintain additional state while taking advantage of default behavior delegation.

We believe this design pattern is a powerful new tool for modeling any concept which varies along several axes and contains significant redundancy.

5 The Implementation of `iterator_adaptor`

The outline for the implementation of the `iterator_adaptor` class template is as follows. In the next few sections we will discuss aspects of the implementation in more depth, including how the problems discussed in the introduction were solved.

```
namespace boost {
  template <class Base, class Policies,
            class Value = default_argument,
            class Reference = default_argument,
            class Pointer = default_argument,
            class Category = default_argument,
            class Distance = default_argument>
  struct iterator_adaptor {
    // Deduce iterator associated types (value_type, etc.) from the
    // named template parameters, and resolve any defaults.
  public:
    // Core operators, delegate to policies class.
    // Redundant operators, implemented in terms of the core operators.
  private:
    // If the policies class is empty, compressed_pair applies the
    // empty-base class optimization to conserve space. The base is
    // ''first'' and the policies are ''second''.
    compressed_pair<Base, Policies> m_iter_p;

    Policies& policies() { return m_iter_p.second(); }
    Base& base() { return m_iter_p.first(); }
    // and similarly for const...
  };
  // Core binary operators.
  // Redundant binary operators.
} // namespace boost
```

5.1 Deducing the Associated Types

Iterators have five associated types: `value_type`, `reference`, `pointer`, `iterator_category`, and `difference_type`. Each of these types must either be supplied by the

user, using the named parameter technique described below in §5.1.2, or a default must be computed for the type.

5.1.1 Defaults for the Associated Types

Because an iterator has so many type parameters, the order and semantics of the associated type parameters was carefully chosen so that users would be able to use as many defaults as possible. The list of associated types begins with the most fundamental element, the iterator's `value_type`. If no `Value` parameter is supplied, the `Base` type is assumed to be an iterator, and the adapted iterator takes its `value_type` from the `Base` iterator's `iterator_traits`. However, if the `Value` parameter *is* supplied, an adjustment is made which allows the user to more easily create a constant iterator: if the `Value` parameter is `const T`, the `value_type` will just be `T`. Perhaps strangely, a constant iterator's `value_type` should never be `const`, because it would prevent algorithms from declaring modifiable temporary objects which are copied from dereferenced iterators. For example:

```
template <class ForwardIterator>
typename iterator_traits<ForwardIterator>::value_type
sum(ForwardIterator start, ForwardIterator finish)
{
    typedef typename
        iterator_traits<ForwardIterator>::value_type value;
    if (start == finish)
        return value();
    value x = *start;
    while (++start != finish)
        x += *start; // error?
    return x;
}
```

The defaults for the `pointer` and `reference` types cooperate with the `Value` parameter: if the `Value` parameter is supplied, the `pointer` and `reference` types default to simply `Value*` and `Value&` respectively (without the `const`-ness stripped). Otherwise, as above the `Base` type is assumed to be an iterator and the `pointer` and `reference` types are taken from its `iterator_traits`.

Since these defaults correspond to the required relationships between the `reference`, `pointer`, `value_type` for all constant and mutable ForwardIterators, it is often sufficient to supply just the `Value` parameter when there is no `Base` iterator with appropriate `iterator_traits`.[6]

The defaults for the `iterator_category` and `difference_type` are straightforward: they are the respective types from the `Base` iterator. These work out well as the final parameters, because one usually wants all of the capabilities supplied by the iterator being adapted, and it is difficult to provide more capabilities.

The code used to select the appropriate defaults for the iterator's associated types used to look something like this:

[6] The `Reference` parameter precedes the `Pointer` parameter because it must be often customized for OutputIterators and other iterator types (e.g. `std::vector<bool>::iterator`, which uses a proxy `reference`).

```
// compute default pointer and reference types.
template <class Iterator,class Value>
struct iterator_defaults : iterator_traits<Iterator>
{
   // If the Value parameter is not the same as its default, the
   // user supplied it.
   static const bool value_type_supplied
    = !is_same<Value,typename iterator_traits<Iterator>::value_type>::value;

   typedef typename type_if<value_type_supplied,
           Value*,
      // else
            typename iterator_traits<Iterator>::pointer
   >::type pointer;

   typedef typename type_if<value_type_supplied,
           Value*,
      // else
            typename iterator_traits<Iterator>::reference
   >::type reference;
};

template <class Base, class Policies,
    class Value = typename std::iterator_traits<Base>::value_type,
    class Reference = typename iterator_defaults<Base,Value>::reference,
    class Pointer = typename iterator_defaults<Base,Value>::pointer,
    class Category = typename std::iterator_traits<Base>::iterator_category,
    class Distance = typename std::iterator_traits<Base>::difference_type
         >
class iterator_adaptor
{
 public:
    typedef Distance difference_type;
    typedef remove_const<Value>::type value_type;
    typedef Pointer pointer;
    typedef Reference reference;
    typedef Category iterator_category;
```

Unfortunately, this strategy can only take us so far. It turns out that there are plenty of iterators which don't fit neatly into the system for ordering defaults. For example, the specialized Transform Iterator Adaptor described in Section 3.1 limits the category of its `Base` iterator to InputIterator, so we'd only need to supply the `value_type`, `reference`, and `iterator_category` if the `Category` parameter didn't appear last. Iterators where the `Base` type is not itself an iterator also act this way, since there are no appropriate `iterator_traits` from which to derive the `Pointer` and `Reference` parameters.

5.1.2 Named Template Parameters

Instead of matching arguments to parameters based on order, the assignment of arguments to parameters can be made explicitly by name, so that the order no longer matters [6]. The Iterator Adaptors library supplies an appropriately-named wrapper class for each parameter. For example:

```
template <class Value> struct value_type_is {
  typedef value_type_tag tag;
  typedef Value type;
};
```

Instead of passing the argument `Value` directly to `iterator_adaptor` the user can pass `value_type_is<Value>`. The `iterator_adaptor` has five arguments for the associated types, each of which could be used to specify any of the actual parameters. The `iterator_adaptor` must deduce which argument is for which parameter based on the `tag` inside the wrapper.

First we take all of the parameters and place them in a lisp-style list, using `std::pair` for `cons`. Each parameter wrapper has a key/value pair (the `tag` and `type` respectively), so we can treat this as an associative list.

```
typedef pair<Param1, pair<Param2, pair<Param3, pair<Param4,
          pair<Param5, list_end_type> > > > > NamedParamList;
```

For each parameter we perform a look-up in the associative list using a template meta-program utility class named `find_param`.

```
template <class AssociativeList, class Key>
struct find_param {
  typedef ... type;
};
```

For example, to retrieve the argument for the `value_type` we write the following:

```
typedef typename find_param<NamedParamList, value_type_tag>::type Value;
```

The result of this look-up will either be the argument specified by the user, or if there is none, the `default_argument` type. If it is the default, then a further step is taken to resolve what the default should be for the parameter. The defaults for some of the parameters depend on other parameters, so the order in which defaults are resolved is tailored to respect these dependencies.

5.2 Core Operators

The core operators of the `iterator_adaptor` are implemented by delegating the work to the policies class. Each core operator passes the base object to the appropriate policy function. Sometimes extra type information is also passed in, as is the case with the `reference` type in the implementation of `operator*`.

```
reference operator*() const {
  return policies().dereference(type<reference>(), base());
}
```

The binary operators of the iterator are implemented as free functions (not member functions) to allow both the left and right hand operands to be treated symmetrically, and to implement constant and mutable iterator interactions (more about this in the following Subsection). The implementation of `operator==()` is shown below. We use separate template parameters for the two `iterator_adaptor` arguments. This allows

a single operator to implement all of the combinations of constant/mutable iterator interactions, avoiding the combinatorial explosion discussed in §5.2.1. Note that we only use a single `Policies` template parameter: this restricts iterator interaction to those iterators with the same policies class. This is not as restrictive as it probably should be, but most iterator interaction errors will be caught anyway, when the policies are applied. The disadvantage of not being restrictive enough is in the kind of error message the user will see when misusing two unrelated iterators. Instead of an "operator not found" message they will see an error message from inside the iterator adaptor.

```
template <class Base1, class Base2, class Policies, class V1, class V2,
    class R1, class R2, class P1, class P2, class C1, class C2,
    class D1, class D2>
bool operator==(
  const iterator_adaptor<Base1,Policies,V1,R1,P1,C1,D1>& x,
  const iterator_adaptor<Base2,Policies,V2,R2,P2,C2,D2>& y)
{
  return x.policies().equal(x.base(), y.base());
}
```

5.2.1 Constant/Mutable Iterator Interactions

Iterators over containers and other sequences of stored objects usually come in pairs: a constant iterator type and a mutable iterator type. It is desirable to allow the constant and mutable iterators to interoperate through comparison and subtraction. For example, suppose that you are implementing a container type `C`. Then you ought to define the following four versions of `operator==`, along with corresponding versions of `operator!=`, and (for RandomAccessIterator), operators `<`, `>`, `<=`, `>=`, and `-`.

```
bool operator==(const C::iterator& x, const C::iterator& y);
bool operator==(const C::const_iterator& x, const C::iterator& y);
bool operator==(const C::iterator& x, const C::const_iterator& y);
bool operator==(const C::const_iterator& x, const C::const_iterator& y);
```

Implementers often forget to define the operators for constant/mutable iterator interaction. In addition, iterator adaptors applied to these kinds of iterators should propagate the ability to interact. For example, a reverse iterator adaptor applied to `C::iterator` and `C::const_iterator` should result in mutable and constant reverse iterator types that have the same ability to interact as the `Base` iterators do. The `reverse_iterator` adaptor supplied by the Iterator Adaptor Library have this ability, although those supplied by the C++ standard library do not.

The iterator adaptor binary operators are implemented using function templates (as shown in the previous Subsection). This allows the same function template to provide the implementation of all the combinations of constant and mutable iterator interaction.

Many of the specialized adaptors in the Boost Iterator Adaptor Library supply an additional type generator (as described in §3.3) for matched pairs of constant and mutable iterators. For example, an "indirect container" using a matched pair of indirect iterators might be declared as follows:

```
template <class BaseContainer>
struct indirect
```

```
{
    typedef typename BaseContainer::iterator base_iterator;
    typedef indirect_iterator_pair_generator<base_iterator> iterator_generator;

    typedef typename iterator_generator::iterator iterator;
    typedef typename iterator_generator::const_iterator const_iterator;
    ...
};
```

5.3 Redundant Operators

Most of the redundant operators are implemented in a straightforward way based on the core operators. For example, the `operator+` is implemented in terms of `operator+=`. There are a total of 6 core operators and 11 redundant operators.

```
template <class B, class P, class V, class R, class Ptr,
    class C, class D, class Distance>
iterator_adaptor<B,P,V,R,Ptr,C,D>
operator+(iterator_adaptor<B,P,V,R,Ptr,C,D> p, Distance x)
{
  return p += x;
}
```

The implementation of `operator->` and `operator[]`are not straightforward. We discuss them in the following two sections.

5.3.1 Implementing `operator->` for Input Iterators

When creating an iterator adaptor that produces an InputIterator some extra care must be taken in the implementation of `operator->`. Remember that an input iterator need not iterate over stored objects: it can manufacture new objects when it is dereferenced as is the case for `std::istream_iterator`. If the iterator's `value_type` is of class type, we need to support `operator->`. Since the result of using `operator->` must produce a true pointer even when dereferencing the iterator does not yield a true reference type, we need a `const` lvalue to which a pointer can be formed.

Fortunately, the Standard makes a workaround possible: section 13.3.1.2 paragraph 8 describes a seemingly quirky rule that the `->` operator will be applied to the *result* of any call to `operator->`. This is a convenient way to describe the semantics of ordinary `operator->`, which returns a pointer: it just uses the pointer to perform the usual member dereferencing. It also turns out to be what we need to make a conforming InputIterator. By making the return type of `operator->` a proxy containing an instance of the iterator's `value_type`, we can eventually form a `const` pointer to the returned temporary:

```
template <class T>
struct operator_arrow_proxy
{
  operator_arrow_proxy(const T& x) : m_value(x) {}
  const T* operator->() const { return &m_value; }
  T m_value;
};
```

The iterator adaptor library uses a small meta-program to select the appropriate type for the result of an iterator's `operator->`:

```
  template <class Category, class Value, class Pointer>
  struct operator_arrow_result_generator
  {
    // Is it an input iterator, or something more?
    static bool const is_input_iter
      = is_convertible<Category*, std::input_iterator_tag*>::value
         && !is_convertible<Category*,std::forward_iterator_tag*>::value;

    typedef typename type_if<is_input_iter,
           operator_arrow_proxy<Value>,
      // else
           Pointer
      >::type type;
  };
```

The Boost Type Traits library is used to check whether the iterator's category is no more refined than InputIterator. If so, the appropriate `operator_arrow_proxy` is selected. Convertibility is used as a criterion to allow for user-defined iterator categories derived from the standard ones.

5.3.2 Implementation of `operator[]`

The implementation of `operator[]` would be trivial except for the question of whether it should return a reference or a value. Although it would be more useful to return a reference, this can cause run-time errors when adapting a certain class of legal base iterators. Suppose the base iterator is reading in elements from a file and caching each element as a data member of the iterator.

```
class file_iter {
  T x;
  int pos;
public:
  file_iter(int pos = 0) { x = read_from_file(pos); }
  T& operator*() const { return x; }
  file_iter operator+(int n) const { return file_iter(pos + n); }
  file_iter& operator++() { x = read_from_file(++pos); return *this; }
  // ...
};
```

The `operator*` of this iterator returns a reference to the data member. Now consider what happens inside the `operator[]` of the adaptor:

```
template <class BaseIterator> class my_iterator_adaptor {
  BaseIterator iter;
public:
  reference operator[](difference_type n) const {
    return *(iter + n);
  }
 // ...
};
```

The iterator addition creates a temporary iterator and the dereference returns a reference to a data member of this temporary, which is destroyed before `operator[]` returns. The result is a dangling reference.

The C++ Standard specifies that the return type of `operator[]` of a random access iterator must be "convertible to `T`". This opens up the possibility of returning by-value from `operator[]` instead of by-reference, thereby avoiding the above problem. This approach, though safer, has the disadvantage of being unintuitive since `operator*` is required to return the exact type `T&` and one might expect that `operator[]` and `operator*` would be same in this respect. The C++ Standards Committee is currently debating the question of whether the random access iterator requirements should be changed.

Boost's `iterator_adaptor` takes the safe route and returns the result by-value. This meets the random access iterator requirements of the Standard, which only says that the return type must be "convertible to T",

```
value_type operator[](difference_type n) const
  { return *(*this + n); }
```

Under the current C++ Standard, you cannot assign into the result of `operator[]` applied to a generic random access iterator, but instead must write `*(i + n) = x`.

It would be nice to return by-reference for iterators that can support it, and by-value for the rest. However, the current `iterator_traits` does not provide enough information to make the choice. The proposal in [11] would solve this problem, but of course that will take some time to gain acceptance.

6 Conclusion

Constructing iterators and iterator adaptors is a common task for modern C++ programming. Despite the conceptual simplicity of most iterators, implementing C++ Standard conforming iterators requires a non-trivial amount of code, some of which is challenging to get right and a lot of which is tedious. The `iterator_adaptor` class that we've presented solves these problem by providing a mechanism by which the user provides a minimal specification (by way of the policies class) for the iterator, and then the `iterator_adaptor` takes care of most of the implementation details.

Taking a step back, the Policy Adaptor design pattern allowed us to easily produce both new models of the iterator concept, and new iterator adaptors. This strategy can be applied in situations where there is large family of components that share the same interface. For example, we plan on applying this design approach to containers and algebraic types.

References

[1] A. Alexandrescu. Compound iterators of STL. *C/C++ Users Journal*, 16(10):79–82, 1998.

[2] A. Alexandrescu. *Modern C++ Design*. Addison Wesley, 2001.

[3] M. H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley, 1999.

[4] C. Baus and T. Becker. Custom iterators for the STL. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[5] T. Becker. Smart iterators and stl. *C/C++ Users Journal*, 16(9), September 1998.

[6] K. Czarnecki and U. Eisenecker. Named parameters for configuration generators. http://www.generative-programming.org/namedparams/, 2000.

[7] I. O. for Standardization (ISO). *ISO/IEC Final Draft International Standard 14882: Programming Language C++*. 1 rue de Varembé, Case postale 56, CH-1211 Genève 20, Switzerland, 1998.

[8] D. E. Knuth. *Stanford GraphBase: a platform for combinatorial computing*. ACM Press, 1994.

[9] A. Koenig and B. Moo. *Ruminations on C++*. Addison Wesley, 1997.

[10] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[11] J. Siek. Improved iterator categories and requirements. Technical Report N1297, ISO IEC JTC1/SC22/WG21 - C++, 2001.

[12] J. G. Siek and A. Lumsdaine. *Modern Software Tools for Scientific Computing*, chapter A Modern Framework for Portable High Performance Numerical Linear Algebra. Birkhauser, 1999.

[13] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.

[14] M. Weiser and G. Powell. The View Template Library. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.