GARY POWELL, MARTIN WEISER

Container Adaptors

Preprint SC 99-41 (December 1999)

Container Adaptors

Gary Powell
Sierra On-Line
3380 146th Pl SE ♯300 Bellevue, WA 98007, USA
e-mail: gary.powell@sierra.com

Martin Weiser
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustr. 7, 14195 Berlin, Germany.
e-mail: weiser@zib.de

Abstract

The C++ standard template library has many useful containers for data. The standard library includes two adpators, queue, and stack. The authors have extended this model along the lines of relational database semantics. Sometimes the analogy is striking, and we will point it out occasionally. An adaptor allows the standard algorithms to be used on a subset or modification of the data without having to copy the data elements into a new container. The authors provide many useful adaptors which can be used together to produce interesting views of data in a container.

CR Classification: D.3.3, E.2

Keywords: C++, STL, views

1 Introduction

When manipulating large containers of data, it would be handy to be able to use the algorithms in the standard template library. By using an adaptor a container can be made to appear to the algorithm as if it contained only the elements of interest.

Although all the standard algorithms work on ranges instead of containers, there may be algorithms that rely on begin() and end(), perhaps rbegin() and rend() as well as the typedefs for iterators and perhaps something like size() or swap(). In these cases, smart iterators are not powerful enough. Smart iterators (see [1]) are adpators of iterators instead of adpators of containers.

Views are a natural way to write smart iterator factories. In fact, most of the algorithmic intelligence of views is encapsulated in their iterators.

From a more theoretical point of view, the views provide a layer of abstraction that is compatible with the STL. These container adaptors are thus a natural way of extending the STL container concept in the same way as smart iterators do with the iterator concept.

Views can simplify writing class interfaces. Assume you have a class that contains nodes, edges, and patches (perhaps a graph package or a finite element mesh – the latter is something we use views for). If you want to provide some restricted access to the graph data, how do you do this? You could make the node, edge, and patch containers public (shock!). Maybe the node container is a polymorphic container of interior nodes and boundary nodes. Users of the mesh would then have to deal with a container of pointers. Alternatively you could provide `beginNodes()`, `endNodes()`, `beginEdges()`, ... methods. And don't forget the `numNodes()`, `numEdges()`, ... and perhaps the `node()`, `edge()`, ... random access methods. Still like it? We didn't. Our solution: Present apropriately adapted views that reference the internal containers. Writing `nodes.begin()` or `edges.size()` is better and easier to remember than `numEdges` (or was ist `num_edges`? or `size_edges`? or ...) Views present the standard interface people are used to when dealing with STL containers. [1]

2 Usage Examples

As an example, if we have a container of employees, and we want to find the oldest one, we can create a functor with a `operator()(Employee&)` which returns the age of the employee. Then we can create a transformed view of the container and call `max_element` and get the employee record which we desire.

```
// A really minimal Employee Record
struct Employee {
Date dateOfBirth;
Height currHeight;
Weight currWeight;
Salary currSalary;
Name   currName;
};

typedef container<Employee> EmployeeContainerType;

EmployeeContainerType AllEmployees;

Time getAge(Employee const& e) const
{ return today() - e.dateOfBirth; }

typedef transform_view< EmployeeContainerType,
                        Time (*)(Employee const&) >
        DateOfBirthView;
```

[1]Before we go too far, we will be using algorithms from the standard template library, hereafter known as STL. The STL code is in the namespace `std`, rather than clutter up our examples with `std::this` and `std::that`, we have ommitted specifing the namespace. In actual code you will have to either open the namespace `std`, or list the algorithms with using statements, or specify the namespace at the actual call. Also we will be creating classes whose only method is the `operator()()`. These classes are known as functors.

```
// Search global Employee Container.
Employee& getOldestEmployee()
{
DateOfBirthView dobv(AllEmployees, getAge):

DateOfBirthView::iterator OldestAge = max_element(dobv.begin(),
                                                  dobv.end() );

Employee& OldestEmployee = *(OldestAge.get());

return OldestEmployee;
}
```

Here we use the type `container` to represent any STL compliant container, which is not a collection of pairs like `map` or `set`. We have also provided a `get()` function in the transform iterator class to return the underlying container iterator. You could use a `map` or `set` but that would require another transformation so we'll get to that later.

The `DateOfBirthView` resembles a projection in relational algebra followed by a simple transformation, as realized by the SQL statement

```
SELECT age(DateOfBirth) FROM AllEmployees.
```

Most of the work done by the view is in the `transform_view` class which maintains a reference to the original container, and a functor (which actually may or may not be a function pointer). The functor is passed to the `transform_iterator` class and then is applied to the data the iterator points to whenever `operator*()` is called. This transformation iterator makes the data appear as if it were something else. In this case it makes an `Employee` reference appear as a `Date` to the algorithm `max_element`. We then retrieve an iterator in the original domain container by calling `get()`.

Now if we only wanted to print a list of employees who were born after a certain date, we could apply a filtering view to the `EmployeeContainer`. First we need to specify the functor to tell us which employees we want.

```
struct GetOldEmployees {
Date ofInterest;
   GetOldEmployee(Date const& rhs) : ofInterest(rhs) {}
bool operator()(Employee const& e) const
   { return e.dateOfBirth < ofInterest; }
};

// Create a global ostream operator<< for Employee Records.
// (Left as an exercise to the reader.)

// Create the filter_view type specific to our need.
typedef filter_view< EmployeeContainerType, GetOldEmployees >
        OldEmployeeViewType;

// Create the view.
OldEmployeeViewType oev(AllEmployees,
                        GetOldEmployees(ThirtyYearsAgoToday));

// Print the employees who were born before Thirty Years Ago.
// (Our date of interest.)
copy(oev.begin(), oev.end(), ostream_iterator<int>(cout,"\n" ));
```

The `filter_view` implements the analogy of the relational algebra select operation, as realized by the SQL `WHERE` clause:

```
SELECT * FROM AllEmployees
    WHERE dateOfBirth < ThirtyYearsAgoToday
```

If we wanted to find the maximum salaried employee who is older than thirty years we can combine the filter view with the transform view to create a view of just those employees we are interested in and make `max_element` only see their salary using the transform function.

```
// Create a global Salary comparison operator.
bool operator>(Salary const& lhs, Salary const& rhs)
{
   // do some real salary comparison.
    return lhs.salary_data > rhs.salary_data;
}

// Create a function to retrieve the Salary data.
Salary& getSalary()(Employee const& e) const
{ return e.currSalary; }

typedef transform_view<OldEmployeeViewType,
                       Salary (*)(Employee const&)>
        OldSalaryViewType;

OldSalaryViewType osvt(oev, getSalary );

OldSalaryViewType::iterator iter;
iter = max_element(osvt.begin(), osvt.end() );
*iter == the maximum salary data.

Employee& DesiredEmployee = *(*(iter.get() ).get();
```

Notice how we can build increasingly complex views of our data by layering these adaptors together[2]. The corresponding SQL statement to the `osvt` view would be:

```
SELECT currSalary FROM AllEmployees
    WHERE dateOfBirth < ThirtyYearsAgoToday
```

[2]Our views concept is based on the ideas of Jon Seymour [4] from 1996. He constructed a view that contained a transformation as well as a filter and thus resembled the SQL statements even closer than our current approach. But since transformation and filtering are useful in their own right, we decided to decouple these tasks and to create a lean, orthogonal interface. Of course, the functionality offered by both approaches is the same.

Now suppose we have sorted our collection of employees by some criteria say salary. And now we'd like to print out a list of the middle of the collection. We could make a `range_view` of this collection.

```
EmployeeContainerType::iterator start = AllEmployees.begin();
EmployeeContainerType::iterator end = start;

advance(start, AllEmployees.size()/4);
advance(end, AllEmployees.size() * 3/4);

typedef range_view<EmployeeContainterType::iterator>
        RangeViewEmployeesType;

RangeViewEmployeesType rve(start, end);

copy(rve.begin(),rve.end(),
     ostream_iterator<Employee>(cout,"\n"));
```

Now you may ask why did we bother creating another container from two iterators? When we could have just as easily written the for loop with the original iterators. Good question, in this case no good reason at all. However if you had written a template which relied on a container having the standard functions, begin(), and end() (e.g. most of the views rely on referencing a container instead of a range) it would not have worked to pass in two iterators. In this way we make the sub range look like the whole container.

If you would like to view this data in reverse order you could apply a `reverse_view`. The `reverse_view` does what you would expect by having `begin()` return the `rbegin()` iterator. Thus unlike the standard algorithm it requires a bidirectional container. The advantage to using a view is that a copy of the data is not required.

```
typedef reverse_view<RangeViewEmployeesType> ReverseEmployeeView;

ReverseEmployeeView rrve(rve);

copy(rrve.begin(), rrve.end(),
     ostream_iterator<Employee>(cout, "\n"));
```

Again we could use the filter view and/or the transform view, to make this sub range appear as a container of a different type, or set of elements. We'll leave this as an exercise to the reader.

3 Views Survey

In this section we will give a short survey of most of the views contained in the library.

First, is `map_keys`. This view uses the transform `select_1st` to make a map appear as a collection of keys. We also have coded its counterpart `map_values`, where the transform function is `select_2nd`. These views work on all pair associative containers. This may be more than maps. `map_values` is useful when you want to view a map as an unordered container, perhaps to apply a transform or filter to search for a subset of the data. An example would be a map of football players stored by their NFL ID, to find the rookie running back of the year, filter for rookies, then transform for yards rushed, then call `max_element` on the resulting view.

Secondly, `downcast_view`. This view calls `dynamic_cast` on a container of pointers, it then filters out the elements which return zero from the call to `dynamic_cast`, and then converts the pointers to references. With this view you can take a container of heterogenous objects, all inheriting from some base class and view them as a collection of derived objects.

This sounds way more complicated than it is. We are using two `transform_views` and one `filter_view`. The first transform does the `dynamic_cast`, then we use the `filter_view` to skip the elements that return a zero. Then we use a `transform_view` to promote the pointers to references. In fact we realized that for containers of pointers to heterogenious objects promotion of a pointer to a reference would be useful on its own. We therefore created the `polymorphic_view` and used it as part of the the `downcast_view`.

The friction between generic programming and object oriented programming shows up in the problems you have to face when representing polymorphic collections with STL containers. The only possibility is to use a container of pointers to the objects. But then it is unsatisfactory to present a pointer to base class interface when conceptually we have a container of base class objects. The polymorphic view just puts a dereferencing layer on top of the pointer to base class interface and presents an interface that matches the concept.

The authors have a number of other useful views which we will discuss in brief. We have a `union_view` which concatinates two containers head to tail. A `merge_view` which uses a sorting predicate to select the element of two containers to use next. A `set_intersection_view` which given two sorted containers will return the elements which are in both containers. A `set_difference_view` which returns the elements which are in one set and not the other. A `set_symmetrical_difference_view` which returns the elements which are in one or the other set but not in both. A `set_union_view` which returns the elements in two containers less the elements which overlap. A `unique_view`, which for a sorted container returns the elements which are not duplicates. We also have a `intersection_view`, `difference_view` and `symmetric_difference_view` which do not require the elements to be sorted. These views will use the `find()` member function of either container if it exists. While this is not efficient if you had to do it anyway, your alternative is to copy the data into a sorted container, or do this comparison manually. The view at least provides an efficent implementation of an inefficent process.

How were we able to make all these views without writing code for the last few years? We applied the age old technique of divide and conquer. For the sorted views we created a group of utility views which could be applied in layers. The lowest layer is the `equal_range_view` applied to each of the two containers. `equal_range_view` returns a `range_view` for each group of elements in a container marking the beginning and end of runs of equal elements. Next we use a `pair_merge_view` which merges a pair of `equal_equal_range` iterators, where each pair element has either a element from the `equal_range_view`, or a `range_view` of `end()` from each container. Then the pairs are choosen from the elements which satisfy a conditional. We default to using the standard functor `equal_to`. To create the `set_*_views` we then applied a transform to select the elements from the pair we wanted. For instance to make a `set_intersection_view` we only return the elements from a `pair_merge_view` which have elements in both the `pair.first` and `pair.second`. We then apply a `concatenation_view` which returns the elements one by one from the transformed view. We have extracted a common base which is really just a template of typedefs for the sorted views and then wrote 5 transform functors. Since each layer could be tested independently it was remarkably easy to do. Ok, it took us a while to decypher the error messages but once we were done we could be confident that the other views would work as well and in fact that was the case[3].

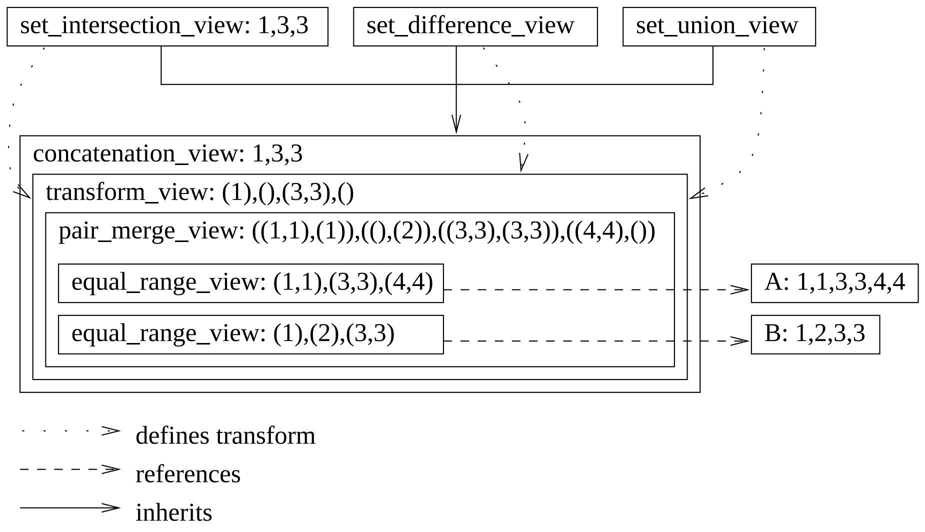
set_intersection_view: 1,3,3
set_difference_view
set_union_view
concatenation_view: 1,3,3
transform_view: (1),(),(3,3),()
pair_merge_view: ((1,1),(1)),((),(2)),((3,3),(3,3)),((4,4),())
equal_range_view: (1,1),(3,3),(4,4)
equal_range_view: (1),(2),(3,3)
A: 1,1,3,3,4,4
B: 1,2,3,3
defines transform
references
inherits

Figure 1: Sorted views composition. Values are examples for the intersection of containers A and B.

We have also created the `filtered_map_keys`, the `transformed_filtered_map_keys` and their cousins for `map_values`. These views are templates with the proper inheritence applied of `filter_view` and `transform_view` to the basic `map_keys`, and `map_values`.

Another interesting view which demonstrates the power of views over copying the elements is the `crossproduct_view`. For two random access containers we apply a operation and return the result for each [i,j] of the containers. The default operation is a pair of references to the elements, however any functor which takes two elements is allowed. This allows for delayed evaluation and sparse storage of a matrix made by (a op b). We did create a proxy template so that `crossproduct_view[i][j]` returns the correct (i op j) result. Thus, the `crossproduct_view` is the analogon of the cross join operation in relational algebra, as realized by the SQL statement `SELECT * FROM Table1, Table2`. In

[3]We didn't start out with a layered approach to the sorted container views. We first tried the "each container does all the work" approach and ended up with a very complex set of code which was difficult to debug and may still, not have all the bugs removed. We have abandoned this code. The curious can find it on our web site under the abandoned directory.

general this is only interesting in combination with a `filter_view` eliminating nonmatching rows, and of course the views performance will not match that of sophisticated database engines in this case.

4 Flexible Code

The advanced template mechanisms of C++ like template template parameters and default template parameters facilitate the construction of highly configurable yet easy to use code. In this section we will explain some of the aspects that make the views code that flexible.

Ownership. Views can own the underlying container or merely reference it. We did this by specifing an argument to the view's template. The tag `view_ref` references the container, and `view_own` has ownership properties. This allows you to nest views together. The innermost view can be specified by the user of the view, the next layers require ownership.

Mutability. Views can be constant or mutable. We provided two tag classes `const_view_tag`, and `mutable_view_tag` to allow specification. We default to `const_view_tag` because in our practice we have found that to be the most useful. A `const_view_tag` uses only the `const_iterator` from the underlying container.

Sensible defaults. One of the techniques we use extensively to increase the flexibility of the code while keeping ease of use and a relatively small code base are traits classes [2]. Traits are used to add or remove reference/pointer indirection and const specifiers to or from types, and to select sensible iterator category defaults for the view's iterators. The latter we will describe in some detail.

Some views limit their iterator categories. A `filter_view` e.g. is at most bidirectional. Views combining two containers, e.g. `union_view`, are limited by the less powerful of the containers involved. The common denominator in both cases is the less powerful iterator category. To be easy to use while keeping flexibility, the views compute the apropriate iterator category as a default template parameter at compile time, thus adapting themselves to the given container(s).

In order to identify the maximum iterator category for a view, we implemented a very interesting set of trait templates. The algorithm is simple: Map the two iterator category tags to integers, choose the smaller one and map it back to the corresponding iterator category tag. Note that this has to be done at compile time. This kind of technique was invented by Veldhuizen [5].

First we set up a template to give us the minimum value of two integers. Since this is a compile time determination we could not use the standard library `min<>` template, instead we use the `?:` operator in an enumeration. Next we created the `combine_traits` template that encapsulates the whole algorithm. Notice that two of the template arguments to `combine_traits` are templates, `map` mapping categories to integers, and `inv` it's inverse. Accordingly, the `map` template provides an `enum x`, and `inv` provides a type named `type`. Then we created the integer mapping for iterator category tags, specializing each type with a different value in ascending order of their inclusiveness.

The random_access_iterator_tag's has the highest value, and input_iterator_tag's, the lowest reflecting their priority for our purpose. Now when we call combine_iterator_categories with two iterator category tags, we end up with the two specializations for those categories, the combine_traits specializes on the minimum value, and sets the typedef type to the correct type. This is a very powerful use of the traits technique. It allows us to at compile time select the correct minimum iterator_trait_tag from the container or view specified. For example, if a view is at best bidirectional, but the underlying container is random access, we will set the view's iterator catagory to bidirectional. We have used this technique to simplify the encoding of the views templates in a lot of places.

```
//
// Combine traits computing the common denominator of two types.
// This works via a mapping to integers, taking the minimum and
// mapping back to corresponding types.
//

template<int a, int b>
struct min_traits { enum { x = a<b? a: b }; };

template <class A, class B,
          template<class T> class map,
          template<class T, int x> class inv>
struct combine_traits {
  typedef typename
    inv<A,min_traits<map<A>::x,map<B>::x>::x>::type type;
};

// Combine traits for iterator tags.
template <class T>
struct iterator_tag_mapping
{ enum { x = 0 }; };
template<>
struct iterator_tag_mapping<std::input_iterator_tag>
{ enum { x = 1 }; };
template<>
struct iterator_tag_mapping<std::forward_iterator_tag>
{  enum { x = 2 }; };
template<>
struct iterator_tag_mapping<std::bidirectional_iterator_tag>
{ enum { x = 3 }; };
template<>
struct iterator_tag_mapping<std::random_access_iterator_tag>
{ enum { x = 4 }; };

template<class T, int x> struct mapping_iterator_tag
{ typedef void type; };
template<class T> struct mapping_iterator_tag<T,1>
{ typedef std::input_iterator_tag type; };
```

```
template<class T> struct mapping_iterator_tag<T,2>
{ typedef std::forward_iterator_tag type; };
template<class T> struct mapping_iterator_tag<T,3>
{ typedef std::bidirectional_iterator_tag type; };
template<class T> struct mapping_iterator_tag<T,4>
{ typedef std::random_access_iterator_tag type; };

template <class cat_a, class cat_b>
struct combine_iterator_categories
  : public combine_traits<cat_a,cat_b,iterator_tag_mapping,
                                      mapping_iterator_tag> {};

template<class A, class B>
struct combine_iterator_tags
  : public combine_traits<
               std::iterator_traits<A>::iterator_category,
               std::iterator_traits<B>::iterator_category,
               std::iterator_tag_mapping,
               mapping_iterator_tag> {};
```

Comparisons. All of the views provide the two basic comparison functions, `operator==()`, and `operator<()`, we also provided the generic relational comparison functions `!=`, `>`, `<=`, and `>=` which can be expressed by combining the first two comparisons. We also provided the conversion operators so that assignment to a `const_view` from a `mutable_view` is possible, as is comparison. The swap specializations were also provided so that swaping of two containers would be the most efficient. These comparison operators are also provided for the view iterators.

Views with these interfaces appear to act just like any other STL container and can be used with STL algorithms. Views are a thin layer on top of their conatiners therefore the overhead of using them is minimal.

A note about the code. We use the ISO C++ template features excessively. In general this can lead to code bloat since for every set of template parameters a corresponding variant of the code is instantiated. This can become a major headache especially for many template paramters. In the views library, this is a non-issue because it is an extremely thin software layer. Very few methods are more than six lines long. Thus, nearly all the methods will be inlined by a good optimizing compiler.

In our implementation we have chosen to use the most advanced template features of their compiler. This code will not compile on many compilers. The reason we did this was simplicity of code and clean design. An early implementation was requrired to duplicate every view for both const and non const domain containers. The amount of code was growing out of control. The authors have this old code, and will be glad to share it but we are not going to do any more maintaince on it. We anticipate that more complier vendors will be able to handle our code soon. The compiler we used was GCC 2.95 which is available at the GNU web page for many systems.

The current code base is still in development and available free of charge at [3]. The design architecture appears to have become stable in late '99. New views seem to occur occasionally as our work projects expand. Contributions and suggestions for the library are welcome.

5 Conclusion

The standard template library containers can be easily adapted to act as sub containers of their data using these adaptors. The adaptors have been used to help simplify the coding of several projects by the authors and we consider them very useful tools. With the adaptors provided, programmers can build interesting views of their own data, and can easily extend and/or combine these adaptors to view any STL compliant container in other more interesting ways. These views are not meant to replace the standard algorithms which generate copies of the containers, but rather as an alternative which may be appropriate for the problem at hand.

The code is available at the VTL homepage [3]

Acknowledgements. The authors would like to thank Jon Seymour for providing them with an initial set of code and the idea of creating views which would work with the STL containers. And we would also like to thank Jon for putting us in touch with each other. This is truely an example of the power of the internet. We would be glad to hear from any users of this code and suggestions for future improvements.

References

[1] Thomas Becker. Smart iterators. *C/C++ User's Journal*, September 1998.

[2] Nathan Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[3] Gary Powell and Martin Weiser. View Template Library container adaptors. `http://www.zib.de/weiser/vtl/`, 1999.

[4] Jon Seymour. Views – a C++ Standard Template Library extension. `http://www.zeta.org.au/~jon/STL/views/doc/views.html`, 1996.

[5] Todd Veldhuizen. Techniques for scientific C++. `http://extreme.indiana.edu/~tveldhui/papers/techniques/techniques.ps`, 1999.