The View Template Library

Martin Weiser
Konrad-Zuse-Zentrum für
Informationstechnik Berlin
weiser@zib.de

Gary Powell
Sierra On-Line
gary.powell@sierra.com

Abstract. Views are container adaptors providing access to different on the fly generated representations of the data in the container they are applied to. The concept fits nicely into the framework defined by the STL. This paper explains design, usage, and implementation of the View Template Library, the currently most advanced implementation of the views concept.

1 Introduction

Since the STL and its iterator concept became popular in the C++ world, a variety of *smart iterators* [1, 3] have been constructed. These smart iterators encapsulate both an iterator and a modifying or filtering algorithmic part, thus providing a different view onto the data pointed to by the iterator. Smart iterators form a kind of iterator adaptors.

This technique can be extended to construct container adaptors bundling a container and an algorithmic part, providing a different view onto the data elements in the container. In contrast to smart iterators, *views* provide a container interface instead of an iterator interface.

This concept has been developed by Seymour [11] and subsequently been extended and implemented using advanced C++ template features by Powell and Weiser in the *View Template Library (VTL)* [9, 8]. This paper describes the design of the VTL as well as the implementation techniques that have been used to create a highly flexible though easily usable library.

Similar special purpose container adaptors can be found e.g. in the Matrix Template Library [12]. There are container adaptors providing scaled vector and matrix views, sparse vector representations from dense data storage, and strided vector views.

2 Views, Smart Iterators, and all the rest

A *view* is an adaptor for STL type containers that provides access to transformed or a subset of elements of the *underlying container* it is applied to. Transformation and filtering is done on the fly, without affecting the data actually stored in the container.

The interface provided by the views is the well known STL container interface, such that views can usually be used transparently as a drop-in replacement for the corresponding STL containers, if data is only to be read. Additionally, views can be applied to views, permitting a mix and match approach to construct increasingly complex views by layering suitable VTL components on top of each other (see section 4). Nearly all of the container requirements are met, except for occasional technical details and complexity guarantees.

From a read-only user's view, the relation between views and smart iterators is the same as between containers and iterators: views and containers are just factories for smart iterators and iterators, respectively, providing some additional meta information such as the number of elements.

There are essentially three ways to get a different sight onto the data stored in a container. First, one can use a view. Second, one can rely directly on smart iterators. And third, one can use a second container storing the result of an STL algorithm generating the required data.

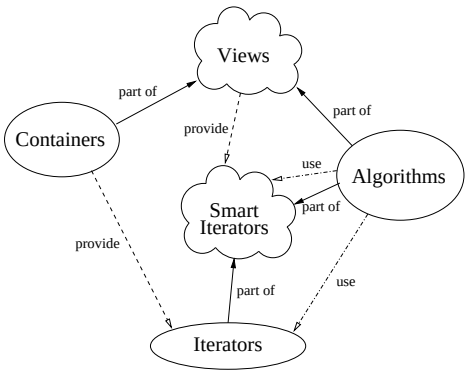

Fig 1. Relation of views to containers, iterators, algorithms, and smart iterators.

Compared to using smart iterators, views

- are a natural extension. Providing the iterator interface but not the container interface is going only half the way.
- are not invalidated if the underlying container is modified. In general, smart iterators will be invalidated together with their underlying iterator.
- can clean up interfaces. Instead of providing several `begin_xy()` and `end_xy()` methods for data sets provided by a class, the class can provide access to views which in turn provide the usual container interface: `xy().begin()` and `xy().end()`.
- can contain the underlying container by value. This permits layering views on top of each other as well as creating and giving around self-contained views.
- provide meta data such as `size()` which is sometimes more convenient or more efficient than obtaining the same information from smart iterators.

On the other hand, smart iterators are sufficient in many situations, e.g. for feeding them into a STL algorithm. In many of those cases, the use of views does not provide any benefit. Note that, since all the hard work is done by

the smart iterators produced by the views, there is no performance difference between using smart iterators and views.

Compared to transforming the data into a second container, views

- have a smaller memory footprint, since the provided elements are generated or accessed on the fly without copying.
- are more efficient if only parts of the data are accessed. The copy approach usually must generate the whole data set, and reproduce it if the original data changes.
- provide data consistency, since the view's elements are rendered on demand. Tracing changes in the original data and updating the second container may be a difficult task, otherwise.
- can lead to worse compiler error messages if used incorrectly.

On the other hand, the generate-on-demand technique of the views is likely to be less efficient if the data elements are accessed very often, and the complete control over the second container enables the use of suitable access patterns.

3 Views Design

What features make up the view concept? Figure 2 shows a feature diagram following Czarnecki and Eisenecker [5]. The most prominent feature is of course the *container*. Every view has an underlying container (that may actually be another view) holding the data elements on which the view operates.

The *ownership* semantics of the container are a point of variation. The view may just reference an external container and rely on the existence of the container during its lifetime, performing no resource management of any kind. Then the view may share the ownership of the underlying container with different views, copy constructed or assigned from each other. The container will be deleted together with the last referencing view. Changes in the underlying container will be visible through all the views that share the container. Finally, the view can own its underlying container exclusively, either by aggregation or by reference. In both cases, the container is copied when the view is copied, and changes in the underlying container are local to the view. Exclusive ownership by reference may be prefered if the container is constructed separately or the view needs to be exception-safe.

The *functionality* of the view is one more point of variation. Since views can be easily stacked to perform increasingly complex tasks, every single view can be restricted to one single task. There are filtering (showing only elements that satisfy a given predicate), transformation (showing the results of a transformation applied to the elements), and so on. Then there are views combining elements from two underlying containers, which then have a second container feature with the same subfeatures as above. Among them are the zip view (showing pairs of corresponding elements), union view (which concatenates two containers head to tail), and the crossproduct view (showing all possible pairs of elements).

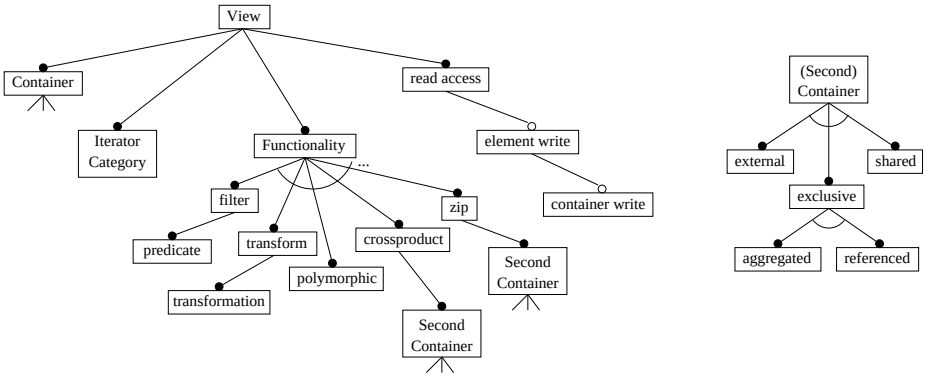

Fig 2. Feature diagram of the view concept (left) and of the container feature (right). See Appendix A.

Every view provides *read access* to its elements, but may or may not provide *write access*. Modification of the view's elements can be inhibited by the user defining a corresponding template parameter, but can also be restricted by functionality. E.g. for the transform view to be able to define element write access in a sensible way, the transformation must be invertible, whereas a crossproduct view cannot offer write access to individual elements at all. If the view is able to modify elements, it may or may not be able to *modify the underlying container* by inserting or erasing elements.

Finally, the *iterator category* of the view's iterators is a feature that affects several algorithms working on the view. The iterator category depends on the iterator category of the underlying container(s), but can be restricted by the functionality. E.g. a filtering view can have at most bidirectional iterators: Advancing an iterator by an arbitrary amount n requires checking the predicate for all the elements in between, and therefore is not more efficient than n increments.

In order to mimic STL containers as closely as possible and not to introduce unnecessary performance hits by this thin abstraction layer, all of the features are bound statically.

Currently, not all possible feature variations are implemented.

4 Views Usage

Consider a container holding pointers to dynamically allocated `Widgets`. Among the widgets are `Buttons`, `ScrollBars` and so on, all derived from `Widget`. For an operation working on buttons, you would like to have a container of `Buttons`.

The procedure is the following: first test every widget pointer whether or not it actually points to a button, then take only the button pointers, and dereference them. The `downcast_view` from VTL that does exactly this is built up from three more fundamental components, each of which performs one of the three

tasks mentioned above.

On the ground layer, there is a

```
typedef transform_view<container_type,
                       downcast<Widget,Button> > downcasted;
```

presenting pointers to `Buttons` that result from a `dynamic_cast<Button>(...)`. To perform this dynamic cast is the task of the *transformation* `downcast<Widget, Button>`. The only non null pointers in this view are the ones that actually point to buttons. Transform views are one of the most useful versatile building blocks of the VTL. E.g. they can be used to extract the values of a pair associative container by specifying `select_2nd<...>` as transformation. This is realized in VTL as the convenience view `map_values`.

On the middle layer, there is a

```
typedef filter_view<downcasted,
                    is_nonzero<Button*>,
                    ...,
                    aggregated_ownership> filtered;
```

presenting only the non null pointers, i.e. exactly the pointers to buttons. The selection is based on the *predicate* `is_nonzero<Button*>`. Of course, the filtered view must own the downcasted view exclusively or shared, otherwise the user would be required to delete both the filtered view and the intermediate downcasted view.

On the top layer, there is a

```
typedef transform_view<filtered,
                       dereference<Button*>,
                       ...,
                       aggregated_ownership> dereferenced;
```

presenting references to the buttons instead of pointers. This transform view with dereferencing transformation is convenient in its own right, and is provided as `polymorphic_view` in the VTL. Again, the dereferenced view must own the filtered view exclusively or shared.

All this is packaged up into a single component, to be used as follows:

```
vector<Widget*> widgets;
...
downcast_view<vector<Widget*>,Button> buttons(widgets);
for_each(buttons.begin(),buttons.end(),use_button);
```

The remaining unspecified template parameters default to read only access, the maximum justified iterator category, and external ownership.

The second example, though of probably little practical use, is interesting because it demonstrates how views can be layered on top of each other to create powerful tools. Given two sorted containers, the usual generalized set operations intersection, union, difference, and symmetric difference can be performed in

linear time, as is done by the corresponding STL algorithms. Following we will show how these operations can be implemented on the fly using stacked views. For simplicity, we will use the containers $A = [1, 2, 2, 4, 4]$ and $B = [2, 3, 4, 4]$ for illustration.

The main insight is that all pairs of corresponding ranges of equal elements can be considered independently of one another when computing the set operations. Thus, a view that presents such pairs of corresponding ranges of equal elements is a basic building block for all the set operation views.

First, we have to identify the equal ranges in each container. This is done by a basic view the iterators of which move two underlying iterators appropriately through the underlying container. Applied to the example data we get $A' = [[1], [2, 2], [4, 4]]$ and $B' = [[2], [3], [4, 4]]$.

Second, corresponding ranges in the two views have to be identified. This is done by another basic view the iterators of which move two iterators one to one through the two underlying containers. The merged example data is then $C = [([1], []), ([2, 2], [2]), ([], [3]), ([4, 4], [4, 4])]$.

On these pairs of ranges, a transformation specific to each set operation creates an appropriate range from the pair. E.g. the intersection transformation selects the shorter of the two ranges, and the difference transformation takes $\max\{0, m - n\}$ elements from the first range with m elements, ignoring the m elements from the second range. Proceeding with the intersection operation for our example, we get $D = [[], [2], [], [4, 4]]$.

Finally, the ranges need to be flattened out into a single view. This is done by a `concatenation_view`. In the example we get $E = [2, 4, 4] = A \cap B$.

Thus, all four set operations share the same stack of views, with a transformation function of a few lines of code that is plugged in to define the actual operation.

5 Views Implementation

In this section we will explain some of the template techniques we use to implement the library. We use traits [7, 2] and template metaprogramming [10, 14] for computing internally needed types and sensible default template parameters. In particular these defaults make the flexibility of the VTL manageable and usable.

The template metaprogramming and traits techniques are used to simplify declaring views by deducing the minimum iterator functionality of two container types. For example to make a union view (concatenation head to tail of two containers) of a linked list and a vector, the iterator category would be bidirectional as the linked list has the lowest common denominator.

In order to make intelligent deductions like this in VTL we first create a template metafunction designed to select the minimum value of two integers by using the intialization of an enum.

```
// Compute the minimum of two
// integers at compile time.
template<int a, int b>
```

```
struct min_traits
{ enum { x = a<b? a: b }; };
```

Now we create a template which given a type sets an enum to a value. The name `x` for the enum value was arbitrary but must be known by the template which does the final value evaluation. Then we assign a value for each STL iterator type in ascending order from least functionality to most functionality by creating specializations of our `iterator_tag_mapping` template.

```
// Mapping iterator tags to integers.
template <class T>
struct iterator_tag_mapping
{ enum { x = 0 }; }; // default

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 }; };
```

Now we define a template class which given a value will return the iterator category associated with it. Notice that the values must be the same as those we used for the template `iterator_tag_mapping`. This is an inverse template question to `iterator_tag_mapping`.

```
// Mapping integers to iterator tags.
template<int x>
struct mapping_iterator_tag
{ typedef void type; };

template<>
struct mapping_iterator_tag<1>
{ typedef std::input_iterator_tag type; };

template<>
struct mapping_iterator_tag<2>
{ typedef std::forward_iterator_tag type; };
```

```
template<>
struct mapping_iterator_tag<3>
{ typedef std::bidirectional_iterator_tag type; };

template<>
struct mapping_iterator_tag<4>
{ typedef std::random_access_iterator_tag type; };
```

Next we define a template class which will take our two internal template mapping classes and do the work of defining a typename based on our lookup critera. Namely, the lesser category type of the two iterator types. This kind of technique was invented in the context of type promotion [4, 15].

```
// Given two types and priority
// mappings to and from integers,
// compute the lower type.
template <class A, class B,
          template<class T> class map,
          template<int x>   class inv>
struct combine_traits {
  typedef typename
      inv<min_traits<map<A>::x,
                     map<B>::x>::x
         >::type
    type;
};
```

In the template `combine_traits`, we used template template parameters instead of specifying the actual classes, `iterator_tag_mapping`, and `mapping_iterator_tag` so that we could reuse the template `combine_traits` for further rank selection tasks. All that would be required is to write new `tag2map` and `map2tag` templates with the appropriate specializations and values.

Now we make an intermediate template in case we only need to compare two categories, and not the iterators.

```
// Given two iterator categories,
// compute the lower one.
template <class cat_a, class cat_b>
struct combine_iterator_categories
  : public combine_traits<
             cat_a,
             cat_b,
             iterator_tag_mapping,
             mapping_iterator_tag
           >
{};
```

Finally we create the template which combines all our internal workings so that only the two iterator types are the required inputs.

```
// Given two iterator types,
// compute the lower category.
template<class A, class B>
struct combine_iterator_tags
  : public combine_iterator_categories<
             iterator_traits<A>::iterator_category,
             iterator_traits<B>::iterator_category,
           >
{};
```

For specifying the ownership and mutability features, VTL uses the tag classes `external_ownership`, `shared_ownership`, `aggregated_ownership`, and `referenced_ownership`, and `const_view_tag`, `mutable_elements_tag`, and `mutable_view_tag`, respectively.

From the container type and the actual tag classes specifying the desired features of the view, type generators deduce the correct smart pointer type for referencing the underlying container as well as correct view constructor argument types.

This is how the actual `transform_view` is defined:

```
template <class container,
          class transformation,
          class const_tag = const_view_tag,
          class ownership_tag = external_ownership,
          class T = typename transformation::result_type>
class transform_view;
```

A user of the `transform_view` with exclusive ownership by aggregation would declare it like this:

```
transform_view<ContainerType,
               TransformationResult (*)(DataType),
               const_tag,
               aggregated_ownership,
               TransformationResult > OwnedTransform;
```

The smart pointer type selection is based on the `ContainerType`, the `const_tag` and the `ownership_tag`.

Of course, the smart pointer could have been defined directly by a template argument, e.g.

```
template <class container,
          class transformation,
          class const_tag = const_view_tag,
          class smart_pointer = container const*,
```

```
          class T = typename transformation::result_type >
class transform_view2;
```

A user could declare a `transform_view2` with shared ownership instead of the default external ownership as

```
transform_view2<ContainerType,
                TransformationResult (*)(DataType),
                const_view_tag,
                boost::shared_ptr<ContainerType const>,
                TransformationResult >
```

This simpler solution exposes implementation details, contains redundant information about the access properties, and does not provide intelligent defaults for the smart pointer.

Duplicate specification information is a potential source of confusion and errors and should be avoided if possible. The use of type generators, while not required to implement ownership attributes, eases their use and decouples implementation from usage.

6 Views and Expression Templates

There are essentially two areas where expression templates [13, 6] and views are related. These are:

- Defining transformations and predicates for views by expression templates.
- Using expression templates directly to provide the views functionality.

It would be nice to be able to write the transformation for a `transform_view`, or the filter predicate for a `filter_view`, using an expression template. Unfortunately, expression templates bind up the information about the expression into the type itself, thereby forcing the transformation template parameter being declared in a `transform_view` to match it. This requires the user of the expression template library to understand the internals of how the expression template will be built. This is nearly as bad as having to know the name mangling method of the compiler. There is work being done in this area but at the moment it appears that when the compiler creates the expression template the type information is used and then lost. So for the moment this is not a satisfactory way of building the functors.

On the contrary, using expression templates directly to provide the functionality instead of using views is possible. But it puts the burden on the user side of the interface: Since expression templates are built locally and thrown away after use, they must be defined at the point of use. Thus the user of an interface must know the details of the underlying container's elements. On the other hand, smart iterators and views provide a clean interface to the user without exposing the internals of the underlying container.

7 Performance and the Abstraction Penalty

The convenience of views does not come for free. One drawback are the worse compiler error messages which are typical for complex template code. The other disadvantage is the higher complexity of the code the compiler has to deal with. Thus, the well-known abstraction penalty can be expected to incur a performance decrease, depending on the optimization abilities of the compiler.

On the other hand, use of views may reduce the amount of data that is copied around in memory, and can therefore be expected to increase the performance, especially if memory bandwidth is low compared to the processor speed.

The following *simplistic* test may provide a first idea. Consider a container of pairs of integers:

```
typedef pair<int,int> value;
vector<value> container;
```

To a client (that happens to be interested in minimal values) we want to present the second components via a standard container interface. Without views or smart iterators, one would probably write something like

```
vector<int> tmp(container.size());
transform(container.begin(),container.end(),
          select2nd<value>(),tmp.begin());
...
return *min_element(tmp.begin(),tmp.end());
```

The corresponding views solution would be

```
transform_view<vector<value>,select2nd<value> >
    tmp(container,select2nd<value>());
...
return *min_element(tmp.begin(),tmp.end());
```

This simple test has been studied for different container sizes on a Sun UltraSparc workstation, using GCC 2.95 with and without optimization. The results are shown in Figure 3. A comparison with a broader set of compilers would be interesting, but the other compilers to which we have access (Sun, SGI, MS) are not yet sufficiently standards compliant to compile the code.

Obviously, there is an abstraction penalty, especially without optimization, as can be expected. With optimization turned on, the abstraction penalty is significantly decreased, and the smaller memory footprint is often more important. Note that the copying solution starts earlier with swapping.

Given that views and smart iterators share the same performance properties by design, since views are just smart iterator factories, a separate comparison of views and smart iterators is not necessary.

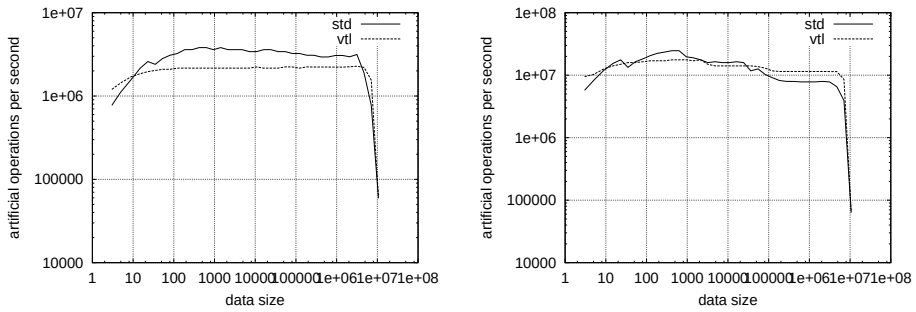

Fig 3. Simple performance test without (left) and with (right) optimization.

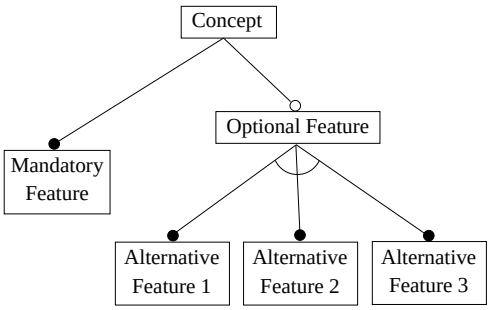

Fig 4. Different flavours of features.

A Feature Diagrams

Here we will give a brief introduction to feature diagrams to ease the understanding of Figure 2. Features characterize the relevant properties of instances of a general concept. In particular, points of variation can be described by features.

The features are represented as the nodes of a tree, with the concept being the root. Features come in several flavours:

Mandatory Features are present in a concept instance iff their parent is present. They are shown by a filled circle at the end of the edge.

Optional Features may be present only if their parent is present. They are shown by an empty circle at the end of the edge.

Alternative Features are a set of features from which one is present if their parent is present. They are shown by an arc grouping their edges.

For a detailed discussion of feature modeling we refer to [5].

B `transform_view` Definition

For illustration we present a nearly complete definition of `transform_view`, with some details stripped off to increase readability (e.g. namespaces).

First we define a template for the smart iterator that actually does the transformation on dereferentiation.

```
template <class transform,
          class iter_t,
          class T>
class transform_iterator
  : public iterator<iterator_traits<iter_t>::iterator_category,
                    T,
                    iterator_traits<iter_t>::difference_type,
                    pointer_traits<T>::type,
                    T> {
public:
  typedef iterator<iterator_traits<iter_t>::iterator_category,
                   T, iterator_traits<iter_t>::difference_type,
                   pointer_traits<T>::type,T> inherited;
  typedef typename const_traits<reference>::type const_reference;
```

Iterators must be default constructible.

```
  explicit transform_iterator() {}

  transform_iterator(transform trans_, iter_t iter_)
    : iter(iter_), trans(trans_) {}
```

Incrementing the transform iterators just increments the underlying iterator.

```
  transform_iterator& operator++()    {
    ++iter;
    return *this;
  }
  transform_iterator  operator++(int) {
    transform_iterator tmp(*this);
    ++iter;
    return *tmp;
  }
```

Decrementing the transform iterators just decrements the underlying iterator. Therefore the decrement operators are only instantiable if the underlying iterator is at least bidirectional.

```
  transform_iterator& operator--()    {
    --iter;
    return *this;
  }
```

```
transform_iterator  operator--(int) {
  transform_iterator tmp(*this);
  --iter;
  return *tmp;
}
```

Dereferencing the iterator does the main work, i.e. it applies the transformation to the value in the underlying container and returns a temporary result object.

```
const_reference operator*() const { return trans(*iter); }
reference       operator*()       { return trans(*iter); }
```

Iterator arithmetics is forwarded to the underlying iterator and since we use `std::advance` will be done as efficiently as possible. Note that because we determine the `transform_iterator` category based on the underlying iterator category, `std::advance(transform_iter,n)` will not call these operators unless the underlying iterator supports them.

```
transform_iterator& operator+=(int n) {
  advance(iter,n);
  return *this;
}
transform_iterator operator+(difference_type n) const {
  transform_iterator tmp(*this);
  return tmp += n;
}
transform_iterator& operator-=(int n) {
  advance(iter,-n);
  return *this;
}
transform_iterator operator-(difference_type n) const {
  transform_iterator tmp(*this);
  return tmp -= n;
}
const_reference operator[](difference_type n) const {
  return *(*this + n);
}

difference_type operator-(const transform_iterator& y ) const {
  if (iter < y.iter) return -std::distance(iter,y.iter);
  else               return std::distance(y.iter,iter);
}
```

Comparison operators are templated to allow comparison between mutable and immutable iterator variants. More exactly, the underlying iterators must be comparable.

```
template <class iter2, class T2>
```

```
  bool operator==(const transform_iterator<transform,
                                           iter2,T2>& rhs) const {
    return iter==rhs.iter;
  }

  template <class iter2, class T2>
  bool operator< (const transform_iterator<transform,
                                           iter2,T2>& rhs) const {
    return iter < rhs.iter;
  }
```

Conversion from mutable to immutable iterator. Instantiation requires the conversion from `iter_t` to `const_iter` and `T` to `const_T`.

```
  template<class const_iter, class const_T>
  operator transform_iterator<transform,const_iter,
                              const_T>() const {
    return transform_iterator<transform,const_iter,
                              const_T>(trans,iter);
  }
```

The smart iterator provides access to the underlying iterator.

```
  iter_t base() const { return iter; }
```

The smart iterator encapsulates an underlying iterator as well as the transformation to be applied.

```
private:
  iter_t iter;
  transform trans;

  template<class trans2, class iter2, class T2>
  friend class transform_iterator;
};
```

Second, the view template itself is defined. It provides type definitions for its iterator types, iterator generation methods and meta data methods.

```
template <class container,
          class transformation,
          class const_tag,
          class ownership_tag,
          class T>
class transform_view {
public:
  typedef one_container_base<container,
                             const_tag,ownership_tag> proxy_t;
  typedef T                                value_type;
```

```
  typedef typename const_traits<T>::type const_value_type;

  typedef typename ctor_arg<container,const_tag,
                            ownership_tag>::type ctor_arg_type;

  typedef view_traits<container,const_tag> vt;
  typedef typename vt::container_type domain_type;
  typedef typename vt::iterator       domain_iterator;
  typedef typename vt::const_iterator domain_const_iterator;

  typedef transformation transform_type;
  typedef typename pointer_traits<T>::type  pointer;
  typedef T                                 reference;
  typedef typename const_traits<T>::type    const_reference;
  typedef typename domain_type::size_type size_type;
  typedef typename domain_type::difference_type difference_type;

  typedef transform_iterator<transform_type,
                             domain_iterator,
                             value_type>          iterator;
  typedef transform_iterator<transform_type,
                             domain_const_iterator,
                             const_value_type>    const_iterator;

  typedef reverse_iterator<iterator>       reverse_iterator;
  typedef reverse_iterator<const_iterator> const_reverse_iterator;
```

Views must be default constructible, although a default constructed views is good for nothing except subsequent assignment.

```
  explicit transform_view() {}

  transform_view(ctor_arg_type& cont, const transform_type& tr)
    : proxy(cont), trans(tr) {}
```

The iterator factory methods follow the standard container interface.

```
  const_iterator begin() const {
    return const_iterator(trans,proxy.cont().begin());
  }
  const_iterator end()   const {
    return const_iterator(trans,proxy.cont().end());
  }
  iterator begin() {
    return iterator(trans,proxy.cont().begin());
  }
  iterator end()   {
```

```
    return iterator(trans,proxy.cont().end());
  }
```

Reverse iterator factory methods require the underlying container to be reversible.

```
  const_reverse_iterator rbegin() const {
    return const_reverse_iterator(end());
  }
  const_reverse_iterator rend()   const {
    return const_reverse_iterator(begin());
  }
  reverse_iterator rbegin() {
    return reverse_iterator(end());
  }
  reverse_iterator rend()   {
    return reverse_iterator(begin());
  }
```

The usual container meta information is provided as well.

```
  size_type size()     const { return proxy.cont().size(); }
  size_type max_size() const { return proxy.cont().max_size(); }
  bool      empty()    const { return proxy.cont().empty(); }
```

If the underlying container is a sequence or back insertion sequence, the `front()` and `back()` methods can be instantiated.

```
  const_reference front() const {
    return trans(proxy.cont().front());
  }
  const_reference back()  const {
    return trans(proxy.cont().back());
  }
  reference       front()       {
    return trans(proxy.cont().front());
  }
  reference       back()        {
    return trans(proxy.cont().back());
  }
```

Subscription operators require the underlying container to be random accessable.

```
  const_reference operator[](size_type n) const {
    return trans(proxy.cont()[n]);
  }
  reference       operator[](size_type n)       {
    return trans(proxy.cont()[n]);
  }
```

```
  const_reference at(size_type n) const {
    range_check(n);
    return (*this)[n];
  }
  reference       at(size_type n)       {
    range_check(n);
    return (*this)[n];
  }
```

Erasing elements from the view is done by erasing them in the underlying container. Inserting elements would require an inverse transformation.

```
  iterator erase(iterator first, iterator last) {
    return iterator(proxy.cont().erase(first.base(),last.base()));
  }
```

Popping elements from the ends of the view requires the underlying container to be a front or back insertion sequence.

```
  void pop_front() { proxy.cont().pop_front(); }
  void pop_back() { proxy.cont().pop_back(); }
```

A swap method easing the definition of the free standing swap function.

```
  void swap(transform_view& b) {
    swap(proxy,b.proxy);
    swap(trans,b.trans);
  }
```

The transform view encapsulates (a reference to) the underlying container and the transformation to be applied.

```
private:
  proxy_t proxy;
  transformation trans;

  void range_check(size_type n) const {
    if (n < 0 || n >= size())
      throw std::range_error("transform_view");
  }
};
```

Finally, views have to be equality and less than comparable. For this purpose we define non-member comparison operators.

```
template <class container_1,class container_2,
          class transformation_1,class transformation_2,
          class const_tag_1,class const_tag_2,
          class ownership_tag_1,
```

```
          class ownership_tag_2,
          class T_1,class T_2>
bool operator==(transform_view<container_1,
                               transformation_1,
                               const_tag_1,
                               ownership_tag_1,T_1> const & lhs,
                transform_view<container_2,
                               transformation_2,
                               const_tag_2,
                               ownership_tag_2,T_2> const & rhs) {
  return lhs.size() == rhs.size() &&
         equal(lhs.begin(), lhs.end(), rhs.begin());
}


template <class container_1,class container_2,
          class transformation_1,class transformation_2,
          class const_tag_1,class const_tag_2,
          class ownership_tag_1,
          class ownership_tag_2,
          class T_1,class T_2>
bool operator<(transform_view<container_1,
                              transformation_1,
                              const_tag_1,
                              ownership_tag_1,T_1> const & lhs,
               transform_view<container_2,
                              transformation_2,
                              const_tag_2,
                              ownership_tag_2,T_2> const & rhs) {
  return lexicographical_compare(lhs.begin(),lhs.end(),
                                 rhs.begin(),rhs.end());
}
```

Views must also be swapable.

```
template <class container,
          class transformation,
          class const_tag,
          class ownership_tag,
          class T>
void swap(transform_view<container,transformation,
                         const_tag,ownership_tag,T>& a,
          transform_view<container,transformation,
                         const_tag,ownership_tag,T>& b) {
  a.swap(b);
}
```

Acknowledgement The authors would like to thank the referees for several comments and suggestions to improve the paper.

References

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

[2] A. Alexandrescu. Mappings between types and values. *C/C++ Users Journal*, 18(10), 2000. online column.

[3] Thomas Becker. Smart iterators. *C/C++ User's Journal*, 16(9), 1998.

[4] K. Czarnecki. *Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* PhD thesis, Technical University of Ilmenau, 1998.

[5] K. Czarnecki and U. W. Eisenecker. *Generative Programming.* Addison-Wesley, 2000.

[6] J. Järvi and G. Powell. The Lambda Library. `http://lambda.cs.utu.fi`, 2000.

[7] N. Myers. Traits: a new and useful template technique. *C++ Report*, 7(5), 1995.

[8] G. Powell and M. Weiser. View Template Library container adaptors. `http://www.zib.de/weiser/vtl/`, 1999.

[9] G. Powell and M. Weiser. Views, a new form of container adaptors. *C/C++ Users Journal*, 18(4):40–51, 2000.

[10] C. Prescio. Template metaprogramming. *C++ Report*, 9(7):22–26, 1997.

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

[12] J. Siek and A. Lumsdaine. A modern framework for portable high-performance numerical linear algebra. In H. P. Langtangen, A. M. Bruaset, and E. Quak, editors, *Advances in Software Tools for Scientific Computing*, volume 10 of *Lecture Notes in Computational Science and Engineering*, chapter 1, pages 1–55. Springer, 2000.

[13] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[14] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, 1995.

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