Refactoring the CPH STL:
Designing an independent and generic iterator

Bo Simonsen
Department of Computing, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
bosim@diku.dk

Abstract. The iterator design pattern is widely used in the STL. Iterators act as mediators for the generic algorithms to access the data stored in the containers (`set`, `priority_queue`, etc.). Every container in the STL must provide iterators (`const_iterator` and `iterator`). Therefore, it is important that there is a solid iterator infrastructure. In this paper the current design for the iterator infrastructure of the CPH STL is described; the design and its implementation were jointly done with Jyrki Katajainen.

1. Foreword

At the beginning of this project, every component of the CPH STL was an independent package, which could be used without any shared code (some improvements were done, but it was very little). One goal for the new design was to eliminate the amount of duplicate code within the library, i.e. to maximize the amount of shared code.

The STL is a complete package; therefore components may depend on each other. The main shared components are bridge classes and iterator classes. These will be described in the following sections. The code for the implementation can be found in the appendix.

2. Bridge classes

One goal in the CPH STL project is to design an alternative version of individual STL components. This means that there can be several different data structures, which implement the same kind of containers. An example is the AVL tree data structure and the red-black tree data structure, both data structures implement the containers `set` and `multiset`. To handle this, there is a demand to design a unified interface for these containers (which is compliant to the standard [7]).

This interface is implemented as a *bridge class*, the design of which is based on the bridge design pattern [5] (therefore the name). The intention of this design pattern is to decouple the abstraction from the implementation, which means it should be easy to change the implementation (e.g. from AVL

CPH STL Report 2008-6, June 2008. Revised September 2008.

tree to red-black tree), because a new layer in the middle between the client and the implementation is inserted. The addition of the bridge layer results in a looser coupling between the components; another advantage is that some methods can be implemented within the bridge class. An example is `clear()`; it will simply call `erase(begin(), end())` in order to clean the container.

Gamma et al. [5] use inheritance to implement the bridge pattern, but in this design the implementation is done with templates [15]. The major difference is that a template parameter is used to define the realization class. The template method has these advantages:

- It is possible to add a default implementation (e.g. for set the red-black tree container is default), which means if the bridge class is used without a realization parameter the default realization will be used. If the same principle should be implemented using inheritance, this could be done by either: an instantiation of the super class, or an instantiation of a subclass called "default" with the name of the bridge class prefixed. This would require very clear documentation of which classes to use [10, §6.3].
- The interface is final, no realization can provide additional methods which could be accessible from the client. This is a nice property since the only methods and typedefs which should be accessible are the methods and typedefs defined in the C++ standard. This means the user of the library can get a clear overview of which methods are accessible. (It is not possible rely on the library developers ability to carefully select which methods are public, private, and protected; this can confuse the user.)

The usage of the bridge pattern yields a layered design with the elements: **Client**, **Bridge** *(abstraction [5])*, and **Realization** *(implementation [5])*. The layered design is described and recommended in [12], and we will see later in this paper that the layered design has several advantages. An example of the interface to a bridge class is:

```
template <typename V, typename C, typename A, typename R>
class set {
  ...
};
```

where `V` is the value type, `C` the comparator, `A` the allocator, and `R` the *realization*.

2.1 Bridge classes in the CPH STL

In Table 1 a list of bridge classes and realizations is shown (this list is far from complete). As the reader can verify the names of the bridge classes are given in the C++ standard, and the names of the realization classes are arbitrary selected, but of course descriptive.

Bridge	Realization
set, multiset	red_black_tree
	avl_tree
	aa_tree
meldable_priority_queue	binomial_heap
	binary_heap
	leftist_tree
	2-3_heap
list	doubly_linked_list
vector	dynamic_array
	levelwise_allocated_pile

Table 1. List of bridge classes and realizations in the CPH STL.

3. Iterator

One of the advantages by using bridge classes is code reusability. The same advantages are desired for the iterator infrastructure, but also some additional desirable properties are:

- **Safety**: The user should not be able to corrupt the data structure. If the position variable (the pointer to an element the iterator points to) is accessible by the client, it can corrupt the internal structure of the container.
- **Bridge-only**: Only the bridge should know the iterators. This means that the realization will not instantiate or use iterators. This is a natural property, since realizations usually work with nodes or in general pointers. What happens to iterator related operations like `erase` are that the client calls the bridge with an iterator as parameter, and then the bridge calls the realization with the position variable (a pointer) as parameter. This property confirms the layered approach, the bridge classes know iterators and the realization classes know nodes. This relationship is shown in Figure 1.
- **Independence**: When a container implementation is adapted to this design, there should not be made any changes to the existing iterator implementation. A small well-defined individual interface is desired (explained later, see Tables 2, 3) such that it is an easy and fast task for the implementor to add iterator support. This requirement implies that const and non-const iterators are implemented with the same class, which is possible according to [1].

With the goals presented above in the mind, it can be determined what kind of iterators there need to be in this design. As earlier mentioned there are tree-like containers in the CPH STL, so there will definitely be an iterator construction which suits this type of containers. This iterator is a so-called *bidirectional iterator*, which means that it can either move one step forward or one step backward with one iterator operation (`++`, `--`). There are con-

Figure 1. The relationship between client, bridge, and realization.

tainers (e.g. dynamic array) where it is possible to move the iterator an arbitrary number of positions, no matter where the iterator is pointing to. That kind of iterators are called *random access iterators*. In our design, the bidirectional iterator is named *the node iterator*, since it is always used for containers that are based on nodes (trees, heaps, etc.) and the random access iterator is named *the entry iterator*, since it is using a customization class called *a entry*, to define the behavior of this iterator type.

4. The node iterator

The node iterator is the most simple of the two iterator types described in the previous section. In this section we will study the design and usage of this iterator type.

4.1 The Design

A class diagram of the classes involved and their relationships is shown in Figure 2; only the relevant methods are shown. The design is far from trivial, so in this section we will look into details of what happens in each class.

4.1.1 The bridge class

The involved bridge class is in this example the *set* class. This class defines the types `iterator` and `const_iterator`, and provides the `begin()` and `end()` methods. These types and methods are required by the ISO C++ standard. The bridge class obtains the types of the iterator classes from the realization class, which is given as template parameter `R`. One may ask why the iterator types are not defined in the bridge class. There are three reasons for that: First all realization classes may not use the same iterator type. Second the bridge class does not know the node class. At last legacy components, which do not use this iterator scheme, may not work.

4.1.2 The realization class and node class

The realization class, which is in this example the AVL-tree implementation `avl_tree`, has a class for representing the nodes which is called `avl_node`. To ensure loose coupling between the realization class and the node class, the node class is given to the realization as a template parameter.

The realization should only work with nodes, so `begin()` and `end()` return a pointer to a node. The methods (e.g. `successor()` and

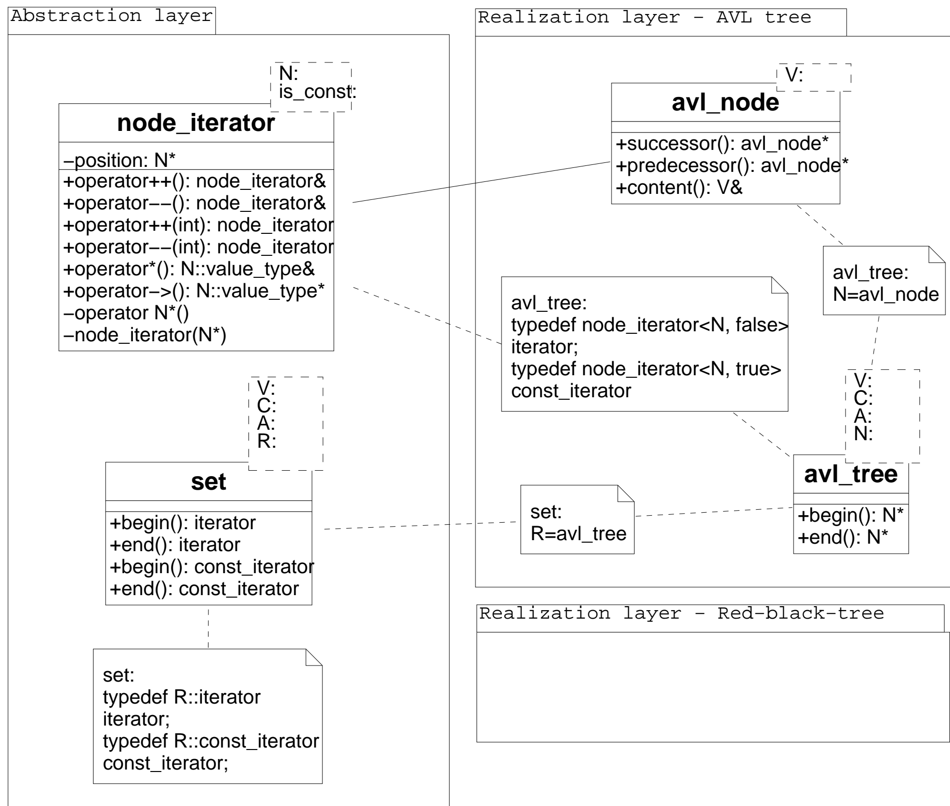

Figure 2. An overview of the design.

`predecessor()`) defined in the node class are interesting; these are used by the iterator to implement the increment (`++`) and decrement (`--`) operators. An overview of the methods is shown in Table 2. The complexity requirements are specific requirements for the CPH STL [8], these are not part of the ISO C++ standard.

4.1.3 The node iterator class

The `node_iterator` class takes two template parameters `N` and `is_const`. `N` is the node class/type, which is the only type the iterator should know, and `is_const` is a boolean value, which makes it possible to use the same template class to implement both iterator types (`const_iterator` and `iterator`) as described in [1].

There exists only one variable in this class, namely the position variable. This variable holds a pointer to the current position and it is kept private, in order to satisfy the safety property.

The class implements all methods defined in the ISO C++ standard. Below is a detailed description of the most important methods:

Method	Description	Return type	Complexity
`successor()`	Returns the next element such that the iterator can implement the increment operator	`node_type*`	constant in the worst case
`predecessor()`	Returns the previous element such that the iterator can implement the decrement operator	`node_type*`	constant in the worst case
`content()`	Returns a reference to the value stored inside the node	`value_type&`	constant in the worst case

Table 2. The methods which the node class (e.g. avl_node) should implement.

- `operator++()` : This method calls `position.successor()` and stores the return value (the pointer to the next node) as the new position.
- `operator--()`: The same as `operator++()` but calls `position.predecessor()` instead.
- `operator++(int)` and `operator--(int)`: instantiates a new `node_iterator` object and increase/decrease this iterator and returns the new object.
- `operator N*()`: This method makes the design shown in Figure 1 possible. The operator is a so-called *conversion operator* [15, 9]: when a type cast to type `N*` is requested, this method is invoked. This makes typecast from `iterator/const_iterator` to `node_type` transparent. The typecast occurs when e.g. the `erase` function is called in the bridge class. It is called with an iterator, but since the realization expects an object of type `node_type*`, the conversion operator is invoked to do the cast. This method is essential, since the bridge class does not know objects of the type `node_type*`, such that the conversion cannot be done by copy constructing the node.
- `node_iterator(node_type*)`: This copy constructor has the opposite effect as the casting operator. It can construct an iterator from a node. This is done when a method's return type should be an iterator but a node is returned instead, i.e. when the realization returns a object of the type `node_type`, which will be casted to an iterator inside the bridge class, using this copy constructor.
- `operator*()` and `operator->()`: These operators make it possible to dereference the iterator such that the value stored inside the node is returned (using the position variable). This is implemented by calling `position.content()`. The `*` operator returns a reference and the `->` operator returns a pointer.

A last detail is important for the `node_iterator` class to work as a bidirectional iterator:

```
typedef std::bidirectional_iterator_tag iterator_category;
```

This is important for several generic algorithms, since they use this tag for branching, so there are one version for bidirectional iterators and one version for random access iterators. An example is `binary_search` which uses `std::advance` to move the iterator n positions. In libstdc++ [6] a loop is used to move the bidirectional iterator n positions, but for random access iterators it simply adds n to the iterator (since there is a `+` operator).

4.2 Usage

Below is a recipe of how an implementor can adapt this iterator design during a *refactoring process* [4] of an existing implementation:

1. Add `successor()`, `predecessor()`, and `content()` to the node class, these should be public. See Table 2 for details.
2. Add these typedefs to the realization class:

   ```
   #include "node_iterator.h++"
   ...
   template <typename V, typename A, typename C, typename N>
   class realization_class {
   public:
     ...
     typedef N node_type;
     typedef cphstl::node_iterator<N, true> const_iterator;
     typedef cphstl::node_iterator<N, false> iterator;
     ...
   };
   ```

3. Make sure that `begin()`, `end()`, and other methods which should return an iterator return a object of the type `node_type*` instead.
4. If the bridge classes are not prepared for this iterator design, some changes to the iterator class are required; see the next section.

5. Solving the encapsulation problem

As mentioned the position variable in the `node_iterator` class is kept private in order to satisfy the safety property. If it was public, the user could easily corrupt the data structure. By keeping the position variable private, it is not possible for the client to get a pointer to the node which the iterator points to (which the client will need to corrupt the container). The client can only get the data which is contained inside the node the iterator points to (using the `*` and `->` operators). We define the *encapsulation problem* by denying the user direct access to the position variable (a diagram of the encapsulation problem is shown in Figure 3). Encapsulation is usually recommended [10, §6.2] and the description of the iterator pattern says that the iterator should provide access to elements without exposing the underlying representation [5], so this problem (the encapsulation problem) is a very important problem in this context.

We will now look into our solution to the encapsulation problem, namely to keep the position variable private. There is one major problem with this

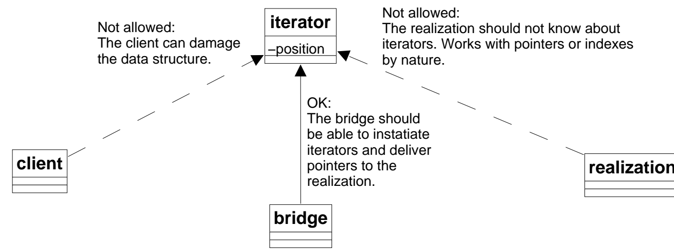

Figure 3. An overview of the encapsulation problem.

approach: The bridge class needs access to the position variable (according to the bridge-only property). This is done using the conversion operator and the copy constructor (This was shown in §4.1.3) . If these methods were kept public, the user can still make damage to the data structure. One could simply do this to access the internal structure of the container and thereby damage it:

```
typedef cphstl::avl_tree<int> myrel;
typedef cphstl::set<int, std::allocator, std::less, myrel> mycont;
mycont c;
c.insert(5);
typename myrel::node_type* tmp = c.begin();
// do something with 'tmp' to damage the container
```

So these methods need to be private too. To give a class access to private variables/methods in another class, one can use the friend directive. We will use this directive in the `node_iterator` class to allow bridge classes access to the private variables/methods. It is desired that a class given as template parameter can be defined as friend such that one could add an extra template parameter called `B` to `node_iterator` class:

```
template<typename N, typename B, bool is_const = false>
class node_iterator {
   ...
   friend class B; // not allowed
}
```

This construction is not allowed in the current C++ standard [7, 15] (it is proposed for the C++0x standard [11]), and it would further more require changes to the current realization scheme, since the realization necessarily do not know their bridges, and a realization can have several bridges, an example: `red-black-tree` has both set and multiset (map and multimap too) as possible bridges. The number of possible bridge classes is a limited number; the number of realization classes is not. Therefore the final solution is to add all bridge classes as friends inside the `node_iterator` class (only the bridge classes which have realizations that use the node iterator). So the `node_iterator` class should be changed to:

```
/* Forward declarations */
template <typename V, typename C, typename A, typename R>
class set;
template <typename V, typename C, typename A, typename R>
```

```
class meldable_priority_queue;

/* Implementation */
template <typename N, bool is_const = false>
class node_iterator {
  template <typename V, typename C, typename A, typename R>
  friend class cphstl::meldable_priority_queue;

  template <typename V, typename C, typename A, typename R>
  friend class cphstl::set;

  ...
};
```

Because the bridge classes are declared using forward declaration, the node iterator file does not need to include the files where the bridge classes are defined, so the node iterator file is still independent. Declaring every individual class (especially bridge classes) as friends yields the following benefits:

- The layered design is preserved, and it confirms the bridge-only property defined in §3.
- It is possible to control access to the iterator's private methods and variables, which means only trusted classes are given access (there might be other classes).
- The interconnection between the components becomes explicit.

Friend classes are usually not recommended [10, §6.2], since it increases the code complexity and raises the coupling between the classes, but this is the only place in the entire infrastructure where it is needed. Another argument is that this code is final, when all bridge classes are implemented, the code is rarely modified.

Unfortunately there is still one minor problem left. It occurs in the `insert` method in set. The prototype for the realization is:

```
std::pair<node_type*, bool> insert(value_type const&);
```

And the prototype for the method in the bridge is:

```
std::pair<iterator, bool> insert(value_type const&);
```

When the realization returns the pair `<node_type*, bool>` to the bridge, it will try to copy construct this pair in order to make a pair with `<iterator, bool>`. The copy constructor of `node_iterator` will now be invoked in the copy constructor of pair, but it should take place in the bridge (recall the bridge is a friend of the iterator). This gives an error.

An attempt to solve this problem was to split the pair inside the bridge, but it requires knowledge of the type `node_type*`, and the bridge does not have that knowledge. It only knows the realization type, and it can not assume that the realization defines the type `node_type`, since it is not a part of the C++ standard. The best solution right now is to let `std::pair` be friend with the iterator class, until the new C++0x standard is accepted which provides the **auto** keyword [14]. This keyword acts as a placeholder for a type until it is deduced i.e. it gives access to types which are not known.

6. The entry iterator

There is a demand for a random access iterator construction in the CPH STL, for containers which provide a data structure where it is possible to move the iterator more than one step at one time. An example of a data structure with this property is the dynamic array, which dynamically allocates new space, when the array becomes full. This iterator is named the entry iterator and it is more complicated than the node iterator, since there is not a class (i.e. the node class for the node iterator) associated with each item, where the methods required by the iterator operations can be defined. This yields a complicated design problem, and in the following section one solution to this problem is shown.

6.1 Design

The entry class is a customization class, which provides an `iterator_entry` class, where the methods for modifying the iterator are defined. It is similar to the node class, but it does not have any data associated. The entry class is sent to the realization class using a template parameter. It should provide other kinds of customization, but the only application right now is the iterator construction. This is why it is defined as a class (`iterator_entry`) inside a class (`dynamic_array_entry`). If we did not have this construction (a class inside a class), we could not obtain the `is_const` template parameter when it is needed. The problem is the template parameters are given by the realization, but the iterator needs to give an extra template parameter namely `is_const`. This is equivalent to how the allocator is used within containers, first it is bounded to the type `value_type`, but it should allocate objects of type `node_type`, so a rebinding occur from type `value_type` to type `node_type`.

In Figure 4 an overview of the design is shown; it is not a complete class diagram, only the relevant class methods are shown. It is based on the same principles as the node iterator. The fundamental ideas from the design are still the same:

– The iterators are not used in the realization class, here a *concrete iterator* [3] is used. This "iterator" is defined in the `iterator_entry` class, and in this specific example (Figure 4) it is a pair but it could be a self-defined data type. Like the `node_iterator` class the `entry_iterator` class holds methods for detaching/attaching a concrete iterator from/to an abstract iterator.

– The same iterator class is used for const and non-const iterators.

– Methods for `advance`, `distance`, etc. are located in the entry class such that the iterator class calls these methods when needed. The same principle is the same as for nodes in the `node_iterator` class where `successor`, `predecessor`, and `content` are implemented.

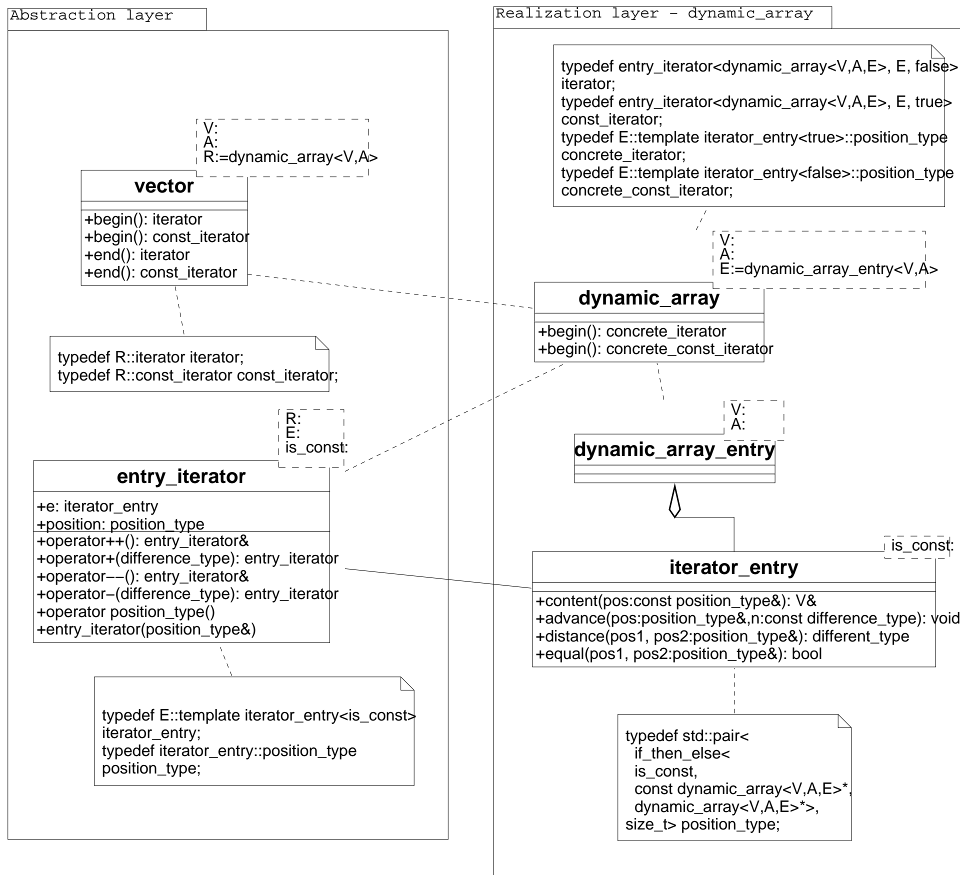

Figure 4. An overview of the entry iterator design.

6.1.1 Bridge class

The behavior of the bridge class is identical to that in the node iterator design. The iterator types (`iterator` and `const_iterator`) are obtained from the realization class; the only difference is that the `end()` method is implemented directly in the bridge class.

6.1.2 Realization and entry class

The entry class is given to the realization using a template parameter `E`. This template parameter defaults to the entry class associated with each realization class (there should be one). Since the realization class should not know anything about iterators, it only defines that the `entry_iterator` class is used as iterator class. The coupling is tight between the entry class and the realization class, since there is no need for sharing them, but it could easily be changed by adding an extra template parameter to the entry class. The `iterator_entry` class should provide methods for moving and comparing

concrete iterators (like `successor`, `predecessor` for the `node_iterator`). These methods are shown in Table 3. The `iterator_entry` class should

Method	Description	Return type	Complexity
`content(pos)`	Should return the value located at `pos` given as parameter	`V&`	constant in the worst case
`advance (pos, n)`	Should advance the concrete iterator `pos`, `n` positions.	`void`	constant in the worst case
`distance (pos, pos2)`	Should return the distance between two positions (`pos, pos2`)	`difference_type`	constant in the worst case
`equal (pos, pos2)`	If `pos` and `pos2` are at the same position return true otherwise false	`bool`	constant in the worst case

Table 3. The methods which iterator_entry should implement.

only provide one type definition:

```
typedef typename if_then_else<is_const, std::pair<const container_type
    *, size_type>, std::pair<container_type*, size_type> >::type
    position_type;
```

in order to inform the realization class and the `entry_iterator` class the type of the concrete iterator in use.

6.1.3 The entry iterator class

Below the implementation details of the `entry_iterator` class are described. The first detail is how to locate the `iterator_entry` class. This is done with the following typedef (The template parameter `is_const` makes it possible for the `iterator_entry` class to set the correct concrete iterator):

```
typedef typename E::template iterator_entry<is_const> iterator_entry;
```

The template parameter `is_const` is given to the `entry_iterator` class in the same way as in the `node_iterator` class.

An instance (named `e`) of the `iterator_entry` class is held inside the `entry_iterator` class in order to invoke methods. Also an instance of the `position_type` is held (named `position`), which is defined as:

```
typedef typename iterator_entry::position_type position_type;
```

Below some selected methods are described in details:

– `operator*()` and `operator->()`: These operators are used to get the data which is held by the position variable. This means that `e.content(position)` is called, and a reference (`*` operator) or a pointer (`->` operator) is returned.

- `operator++()`: This method calls `e.advance(position, 1)` in order to advance the iterator just one position.
- `operator+=(difference_type n)`: Similar to the `++` operator, but advance is called with `n` instead of 1.
- `operator--()` and `operator-=(difference_type n)`: Like the `+=` and `++` operator but the second argument of `e.advance` is negated (`-1` and `-n` respectively).
- `operator-(entry_iterator&)`: Returns the distance between this iterator and the iterator given as parameter. To do that `e.distance(position, pos)` is called, where `position` is the position variable of this iterator, and `pos` is the position variable of the other iterator.
- `operator==(entry_iterator&)` : Checks if two iterators are equal. To check this a call to `e.equal(position, pos)` is made, where `position` is the position variable of this iterator, and `pos` is the position of the other iterator. The reason why the `distance` method is not used, is that `distance` will normally fail (due to an assertion) if a comparison between two iterators from two different containers takes place, but with `operator==` it should be possible.

In order to declare this iterator as a random access iterator, the following typedef is needed inside the `entry_iterator` class (this is used to branch between generic algorithms):

```
typedef std::random_access_iterator_tag iterator_category;
```

The encapsulation problem is solved in the exactly same way for the entry iterator as for the node iterator.

6.2 Usage

The entry iterator can be used by an implementor during refactoring in the following way:

1. Create an *entry class* for your container; a skeleton for the implementation is shown below (our naming convention is `<container>_entry`):

```
template <typename V, typename A>
class my_container_entry {
public:
  template <bool is_const>
  class iterator_entry {
  public:
    typedef V value_type;
    typedef dynamic_array<V, A> container_type;
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;

    typedef typename if_then_else<is_const, std::pair<const
        container_type*, size_type>, std::pair<container_type*,
        size_type> >::type position_type;
    ...
  };
};
```

2. Implement the methods `advance, distance, content, equal` as described in Table 3.
3. Add the template parameter `E` to your container interface such that the class definition looks like this:

```
template <
  typename V,
  typename A = std::allocator<V>,
  typename E = my_container_entry<V, A>
>
class my_container {
...
};
```

4. Add these typedefs to your container:

```
typedef cphstl::entry_iterator<dynamic_array<V, A, E>, E, false>
    iterator;
typedef cphstl::entry_iterator<dynamic_array<V, A, E>, E, true>
    const_iterator;
typedef typename E::template iterator_entry<false>::position_type
     concrete_iterator;
typedef typename E::template iterator_entry<true>::position_type
    concrete_const_iterator;
```

5. Modify `insert, erase, begin,` ... such that they use the concrete iterators instead of abstract iterators.
6. If the bridge class to be used is not prepared for this iterator scheme, use the same approach as described in §5.

7. Final Design

The iterator construction is now complete; it is now possible to provide bidirectional and random access iterators. In Figure 5 an overview of this design is shown: We obtained a design with an *abstraction layer* to where the bridge classes and the iterator classes belong, and a realization layer. Note that the realization layer always contains a container class and a customization class. The customization classes are respectively entries and nodes, depending on the realization type.

8. Design patterns

During the definition and implementation of this iterator design, some decisions of using or not using design patterns have been made:

- **The bridge pattern**: This pattern is already discussed in the section about the bridge classes in the CPH STL, but this pattern can be identified other places in the design. This pattern could be applied to the `entry_iterator` and `node_iterator` classes, but it has been avoided because of the following reasons:
  - With a template construction it will result in too much code overhead, and an unclear interface. An example: it should not be possible to use `operator+` for bidirectional iterators, but the interface

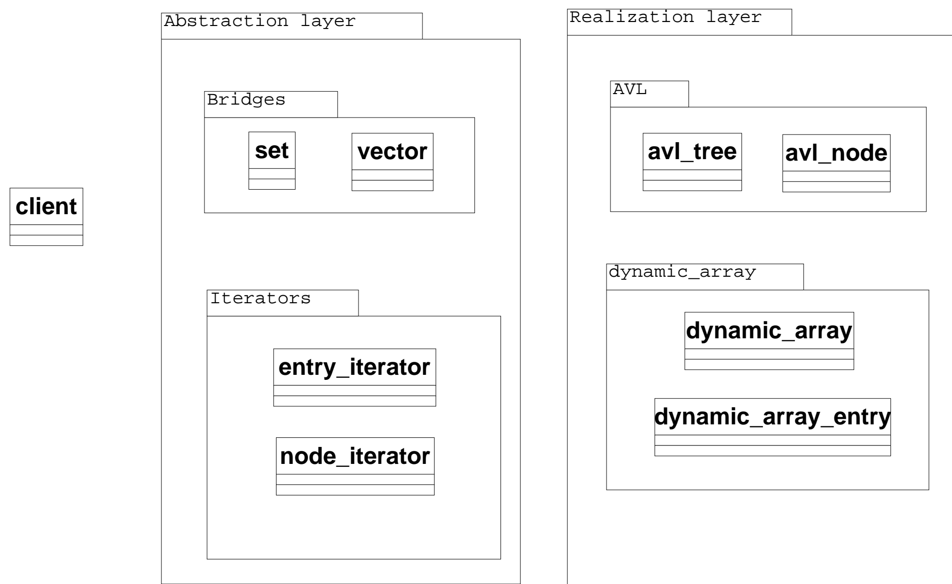

Figure 5. An overview of the final design.

should provide this operator because the random access iterator supports this operation.

- ◦ With inheritance, the interface would still be unclear, since some methods are implemented in the abstract class and some are in the subclasses. Finally the number of methods which can be implemented in the abstract class are few.
- ◦ More iterator classes than the existing are not expected.

– **The visitor pattern**: The entry class is an application of the visitor pattern, since the visitor is used to modify the `position_type`. Normally the object itself is sent to the visitor, but here an object of the type `position_type` is sent, since it is the only attribute inside the iterator object that should be modified. In the node iterator the node class is used as the visiting class.

– **The abstract factory pattern**: This pattern is applied when the instantiation of the iterator takes place. The iterators are instantiated in the bridge class, but the bridge class does not decide which iterator to instantiate; this is done by the realization class. The implementation almost complies to the design described in [3].

– **The proxy pattern**: This pattern is used when there is a need to control access to an object with restricted methods/variables. In the iterator design, it would be useful in order to add restrictions to the iterator's position variable (i.e. to solve the encapsulation problem). In Figure 6 this pattern is shown within the context of the encapsulation problem. The proxy is located between the bridge and the iterator

such that all friend classes are defined in this class, and the only class the iterator needs to be friend with is the proxy class. This is a nice design, since there is an access controlling class, but it results in one major problem. The problem is that it is not possible to use the casting operator, to cast between concrete iterators and abstract iterators. It is possible to cast, but then the bridge should know the concrete iterators, but it should only know abstract/real iterators. This is a nice property, so the full implementation of this pattern is avoided, and instead the proxy facility is built in to the iterator class.

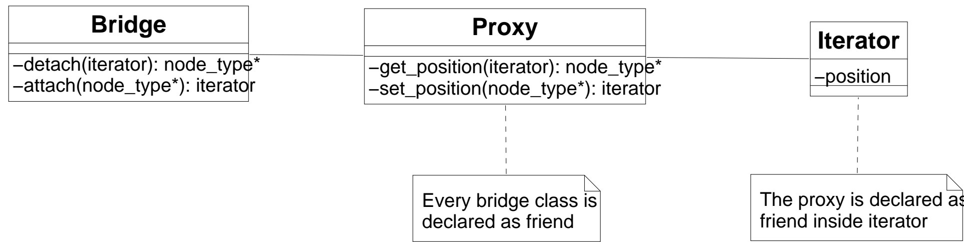

Figure 6. A proposed use of the proxy pattern.

9. Comparing implementations

In this section, a brief comparison is made between our iterator construction and the constructions provided by other libraries and implementations of the STL.

9.1 BOOST

The BOOST library [2] provides a mechanism for constructing iterators with less code than creating entire iterators. BOOST does not provide an STL but they provide add-ons for generic C++ programming, therefore their iterator construction does not rely on the underlying container like the node iterator (i.e. the node iterator assumes that the underlying container uses a node-like class for storing values). The BOOST iterator construction is more similar to the entry iterator. The BOOST iterator construction is divided into two parts:

- **iterator_facade**: This class provides the iterator class which is used by the client, but does not implement the operators, so it is similar to the node iterator and the entry iterator.
- **iterator_adaptor**: This class inherits from the `iterator_facade` class, and the specialization should inherit from this class. The specialization is the class that should be implemented in order to make it possible to iterate over the data stored in our container (i.e. the entry class).

An example of this construction is shown in Figure 7. This example implements a forward iterator (which means `operator++` only) for nodes. Since it is a forward iterator, the `increment` method is the only method that is needed. In the CPH STL construction the adaptor layer does not exist, simply because the iterator construction in the CPH STL is more bounded to the underlying containers. With the BOOST iterator it is possible to iterate over elements which are not in a container (e.g. it is possible to iterate over the sequence $1, ..., n$ with `counting_iterator`, which is a predefined iterator included in the BOOST library). Notice that the iterator type (forward iterator, random access iterator, bidirectional iterator) is set using the `CategoryOrTraversal` template parameter. The CPH STL iterator

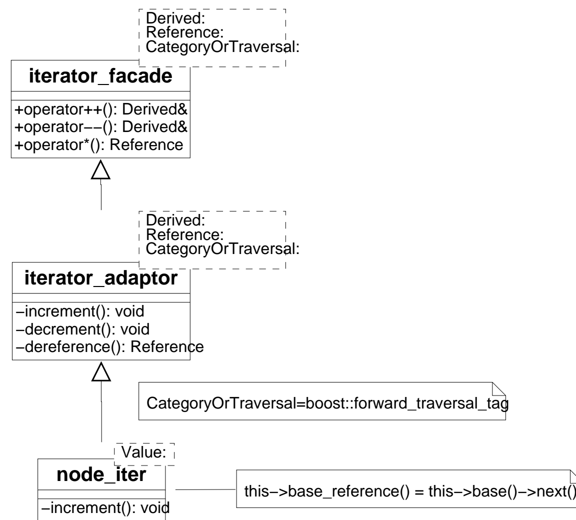

Figure 7. An example iterator construction with BOOST.

construction and the BOOST iterator construction have a lot in common, but to reduce redundant code, the CPH STL construction is more bounded to be used with containers.

9.2 GNU libstdc++

The STL from libstdc++ [6] is usually included in GNU C/C++ compiler (GCC). This examination of the iterator construction for vector and set in libstdc++ is made with version 4.2.3.

9.2.1 vector

The iterator implementation for vector is very simple. It uses the class `normal_iterator` (defined in `bits/stl_iterator.h`), which provides a very simple encapsulation for a pointer-like type. Pointer-like types are types which provide the operators `++`, `--`, `*`, etc. themself. The iterator

object is constructed using an instance of the pointer-like type, and the operators simply apply the same operator to the type, e.g. the `operator++()` will simply use `++` on the position variable. This construction is only usable for arrays, unless the type provides all the operators needed by the iterator (which is quite many). With the entry iterator in the CPH STL at least one advantage is gained and it is: All operators do not need to be implemented for each type, since the number of methods which is needed is reduced.

9.2.2 set

The set implementation is based on the ideas from the bridge class, since it defines the iterators from the `Rep_type` (realization), but it is hardcoded to `red_black_tree`. The reason for that might be that the class is prepared for other realization classes. The iterator implementation for red-black trees (`bits/stl_tree.h`) is very simple, since it is defined explicitly for the red-black trees, which means both const iterator and non-const iterator are defined explicitly. The iterators for list and deque are also defined explicitly. These are defined as structs so the encapsulation problem is not solved.

9.3 SGI STL

The SGI STL [13] is included in the SGI C++ compiler environment. It is freely available and some code in the CPH STL is taken from the SGI STL.

9.3.1 vector

The iterator implementation for the vector class, is even simpler than the implementation in libstdc++. Actually it does not provide an iterator, since the iterator is defined to the pointer type itself:

```
template <class _Tp, class _Alloc = __STL_DEFAULT_ALLOCATOR(_Tp) >
class vector : protected _Vector_base<_Tp, _Alloc>
{
  ...
  typedef value_type* iterator;
  typedef const value_type* const_iterator;
  ...
}
```

In the code is it not clear, how they handle iterator validity [7], since e.g. the `reserve()` call will deallocate the currently used space, and then the iterators given out will become invalid. In the CPH STL this problem is solved using the `[]` operator every time the `content()` method is called in the `iterator_entry` class.

This implementation provides a compliant iterator (a pointer is an iterator by definition), but if any extensions have to be made, they need to design an entire new iterator class. The reason why the iterator is implemented as a pointer may be because of performance considerations (it should be faster to increment a pointer instead of call a function which does the same). Our hope with the CPH STL implementation is it should be possible to inline

the relevant operators at the compiler's optimization stage such that the final assembler code is identical.

9.3.2 set

The iterators for set (red-black trees) are implemented as one class, which can be given two types as template parameters, such that const and non-const iterators can be implemented with one class [1]. The iterators are defined as follows:

```
typedef _Rb_tree_iterator<value_type, reference, pointer> iterator;
typedef _Rb_tree_iterator<value_type, const_reference, const_pointer>
    const_iterator;
```

The iterator is implemented as a struct so every member variable is public, so the encapsulation problem is not solved.

10. Conclusion

From the survey of iterator implementations in other STL implementations and libraries the following observation have been made:

- The iterator construction in the CPH STL reduces the amount of redundant code, since we use one class for implementing const and non-const iterators, and we use two classes for creating iterators for all containers, with a little amount of component-individual code.
- The SGI STL and libstdc++ STL does not provide any safety with respect to encapsulation, the position variables is kept public.
- SGI STL and libstdc++ have a very tight coupling between the realization classes and bridge classes (if they exists), which means it is hard to extend with new realizations.

In conclusion we have created a robust and safe iterator construction for the CPH STL. One can wonder why the code reusability is not better in the other STL implementations (there are too much redundant code), when these STL implementations are widely used on almost every machine with a C++ compiler.

References

[1] M. Austern, Defining iterators and const iterators, *C/C++ Users J.* **19**, 1 (2001), 74–79.
[2] BOOST Community, C++ libraries, Website accessible at `http://www.boost.org` (1999–2008).
[3] A. Duret-Lutz, T. Géraud, and A. Demaille, Design patterns for generic programming in C++, *Proceedings of the 6th Conference on USENIX Conference on Object-Oriented Technologies and Systems*, USENIX Association (2001), 189–202.
[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional (1999).
[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley Professional (1995).
[6] GNU, libstdc++, Website accessible at `http://gcc.gnu.org/onlinedocs/libstdc++/` (1999-2008).
[7] International Organization for Standardization, *ISO/IEC 14882:2003: Programming languages — C++*, International Organization for Standardization (2003).
[8] J. Katajainen, Project proposal: Associative containers with strong guarantees, CPH STL Report 2007-4, Department of Computing, University of Copenhagen (2007).
[9] R. Lafore, *Object–Oriented Programming in C++*, 4th Edition, SAMS (2002).
[10] S. McConnell, *Code Complete*, 2nd Edition, Microsoft Press (2004).
[11] W. M. Miller, Extended friend declarations (rev. 3), Document number WG21 N1791, JTC1/SC22/WG21 - The C++ Standards Committee (2005).
[12] R. S. Pressman, *Software Engineering — A Practitioner's Approach*, 5th Edition, McGraw-Hill (2000).
[13] Silicon Graphics, Inc., Standard template library programmer's guide, Website accessible at `http://www.sgi.com/tech/stl/` (1993–2004).
[14] B. Stroustrup, J. Järvi, and G. D. Reis, Deducing the type of variable from its initializer expression (revision 4), Document number N1984=06-0054, The C++ Standards Committee (2006).
[15] D. Vandevoorde and N. M. Josuttis, *C++ Templates*, Pearson Education, Inc. (2003).

Appendix A. Source code

The source code of the entire iterator construction is provided in this appendix, page numbers are given in the table below:

node_iterator.c++ 22

node_iterator.h++ 24

entry_iterator.c++ 26

entry_iterator.h++ 30

We do also provide the source code for some example implementations. The table of contents is shown below:

avl.h++ 31

avl_node.h++ 35

dynamic_array.h++ 37

dynamic_array_entry.h++ 39

A bridge class example can be found in CPH STL report 2007-4 [8].

Appendix B. Source code - Iterator

Appendix B.1 node_iterator.c++

```
/*
  Implementation of cphstl::node_iterator

  Author: Jyrki Katajainen, Bo Simonsen, April 2008

  General design idea is proposed by J. Katajainen and B. Simonsen
*/

#include <sstream> // defines std::stringstream

namespace cphstl {

  // default constructor

  template <typename N, bool is_const>
  node_iterator<N, is_const>::node_iterator()
    : link(0) {
  }

  // copy constructor

  template <typename N, bool is_const>
  node_iterator<N, is_const>::node_iterator(node_iterator<N, false>
      const& a)
    : link(a.link) {
  }

  // assignment

  template <typename N, bool is_const>
  node_iterator<N, is_const>&
  node_iterator<N, is_const>::operator=(node_iterator<N, is_const>
      const& a) {
    link = a.link;
    return *this;
  }

  // destructor

  template <typename N, bool is_const>
  node_iterator<N, is_const>::~node_iterator() {
  }

  // operator*

  template <typename N, bool is_const>
  typename node_iterator<N, is_const>::reference
  node_iterator<N, is_const>::operator*() const {
    return reference((*link).content());
  }

  // operator->

  template <typename N, bool is_const>
  typename node_iterator<N, is_const>::pointer
  node_iterator<N, is_const>::operator->() const {
    return pointer(&(*link).content());
  }

  // operator++; pre-increment
```

```
template <typename N, bool is_const>
node_iterator<N, is_const>&
node_iterator<N, is_const>::operator++() {
  link = (*link).successor();
  return *this;
}

 // operator++; post-increment

template <typename N, bool is_const>
node_iterator<N, is_const>
node_iterator<N, is_const>::operator++(int) {
  node_iterator<N, is_const> temporary(*this);
  ++(*this);
  return temporary;
}

// operator--; pre-increment

template <typename N, bool is_const>
node_iterator<N, is_const>&
node_iterator<N, is_const>::operator--() {
  link = (*link).predecessor();
  return *this;
}

// operator--; post-increment

template <typename N, bool is_const>
node_iterator<N, is_const>
node_iterator<N, is_const>::operator--(int) {
  node_iterator<N, is_const> temporary(*this);
  --(*this);
  return temporary;
}

// operator==

template <typename N, bool is_const>
template <bool both>
bool
node_iterator<N, is_const>::operator==(node_iterator<N, both> const&
     a) const {
  return link == a.link;
}

// operator!=

template <typename N, bool is_const>
template <bool both>
bool
node_iterator<N, is_const>::operator!=(node_iterator<N, both> const&
     a) const {
  return link != a.link;
}

// parametrized constructor (node -> iterator)

template <typename N, bool both>
node_iterator<N, both>::node_iterator(node_pointer p)
  : link(p) {
}

// conversion operators (iterator -> node)
```

```
  template <typename N, bool is_const>
  node_iterator<N, is_const>::operator node_pointer() const {
    return link;
  }

  // conversion operator (makes it possible to print out an iterator)

  template <typename N, bool is_const>
  node_iterator<N, is_const>::operator std::string() const {
    std::stringstream ss;
    std::string address;
    ss << (int)(char*)((*this).link);
    ss >> address;

    if (is_const == false) {
      return std::string("iterator:␣node␣at␣") + address;
    }
    else {
      return std::string("const_iterator:␣node␣at␣") + address;
    }
  }

  // representation

  template <typename N, bool both>
  std::ostream&
  operator<<(std::ostream& s, node_iterator<N, both> const& i) {
    s  << std::string(i);
    return s;
  }

}
```

Appendix B.2 node_iterator.h++

```
/*
  The idea of combining iterators and const iterators into the same
  class is taken from [Matt Austern. Defining iterators and const
  iterators. C/C++ User's Journal 19,1 (2001), 74-79].

  Author: Jyrki Katajainen, Bo Simonsen, May 2006, April 2008

  General design idea is proposed by J. Katajainen and B. Simonsen
*/

#ifndef __CPHSTL_NODE_ITERATOR__
#define __CPHSTL_NODE_ITERATOR__

#include <iterator> // defines std::bidirectional_iterator_tag
#include <cstddef> // defines std::ptrdiff_t
#include <iostream> // defines std::ostream
#include <string> // defines std::string

#include "type.h++" // defines cphstl::if_then_else

namespace cphstl {

  /* Forward declarations of bridge classes */

  template <typename V, typename A, typename R>
  class list;

  template <typename V, typename C, typename A, typename R>
  class meldable_priority_queue;
```

```
template <typename V, typename C, typename A, typename R>
class multiset;

template <typename V, typename C, typename A, typename R>
class set;

template <typename N, bool is_const = false>
class node_iterator {

public:

  // types

  typedef std::bidirectional_iterator_tag iterator_category;
  typedef typename N::value_type value_type;
  typedef std::ptrdiff_t difference_type;

  typedef typename if_then_else<is_const, value_type const*,
      value_type*>::type pointer;
  typedef typename if_then_else<is_const, value_type const&,
      value_type&>::type reference;

private:

  // types

  typedef typename if_then_else<is_const, N const*, N*>::type
      node_pointer;

public:

  // friends
  friend class node_iterator<N, !is_const>;

  template <typename M, bool both>
  friend std::ostream& operator<<(std::ostream&, node_iterator<M,
      both> const&);

  template <typename V, typename A, typename R>
  friend class cphstl::list;

  template <typename V, typename C, typename A, typename R>
  friend class cphstl::meldable_priority_queue;

  template <typename V, typename C, typename A, typename R>
  friend class cphstl::set;

  template <typename V, typename C, typename A, typename R>
  friend class cphstl::multiset;

  template <typename T1, typename T2>
  friend class std::pair;

  // structors

  node_iterator();
  node_iterator(node_iterator<N, false> const&);
  node_iterator& operator=(node_iterator const&);
  ~node_iterator();

  // operators

  reference operator*() const;
  pointer operator->() const;
```

```
    node_iterator& operator++();
    node_iterator operator++(int);
    node_iterator& operator--();
    node_iterator operator--(int);

    template <bool both>
    bool operator==(node_iterator<N, both> const&) const;

    template <bool both>
    bool operator!=(node_iterator<N, both> const&) const;

  private:
    // converters to be used by the friends

    node_iterator(node_pointer); // node_pointer --> iterator
    operator node_pointer() const; // iterator --> node_pointer
    operator std::string() const; // iterator --> string

    node_pointer link;

  };

}

#include "node_iterator.c++" // implements cphstl::node_iterator

#endif
```

Appendix B.3 entry_iterator.c++

```
/*
  Implementation of cphstl::entry_iterator

  Author: Jyrki Katajainen, Bo Simonsen, April 2008

  General design idea is proposed by J. Katajainen and B. Simonsen
*/

#include <cassert> // defines assert macro
#include <iostream>

namespace cphstl {

  // default constructor

  template <typename R, typename E, bool is_const>
  entry_iterator<R, E, is_const>::entry_iterator()
    : position(typename iterator_entry::position_type())
  {
  }

  // copy constructor

  template <typename R, typename E, bool is_const>
  entry_iterator<R, E, is_const>::entry_iterator(entry_iterator<R, E,
      false> const& a)
    : position(a.position) {
  }

  // assignment

  template <typename R, typename E, bool is_const>
  entry_iterator<R, E, is_const>&
  entry_iterator<R, E, is_const>::operator=(entry_iterator<R, E,
      is_const> const& a) {
```

```
  (*this).position = a.position;
  return *this;
}

// destructor

template <typename R, typename E, bool is_const>
entry_iterator<R, E, is_const>::~entry_iterator() {
}

// operator*

template <typename R, typename E, bool is_const>
typename entry_iterator<R, E, is_const>::reference
entry_iterator<R, E, is_const>::operator*() const {
  return reference(e.content((*this).position));
}

// operator->

template <typename R, typename E, bool is_const>
typename entry_iterator<R, E, is_const>::pointer
entry_iterator<R, E, is_const>::operator->() const {
  return pointer(&e.content((*this).position));
}

// operator++; pre-increment

template <typename R, typename E, bool is_const>
entry_iterator<R, E, is_const>&
entry_iterator<R, E, is_const>::operator++() {
  e.advance((*this).position, 1);
  return *this;
}

// operator++; post-increment

template <typename R, typename E, bool is_const>
entry_iterator<R, E, is_const>
entry_iterator<R, E, is_const>::operator++(int) {
  entry_iterator<R, E, is_const> temporary = *this;
  e.advance((*this).position, 1);
  return temporary;
}

// operator--; pre-decrement

template <typename R, typename E, bool is_const>
entry_iterator<R, E, is_const>&
entry_iterator<R, E, is_const>::operator--() {
  e.advance((*this).position, -1);
  return *this;
}

// operator--; post-decrement

template <typename R, typename E, bool is_const>
entry_iterator<R, E, is_const>
entry_iterator<R, E, is_const>::operator--(int) {
  entry_iterator<R, E, is_const> temporary = *this;
  e.advance((*this).position, -1);
  return temporary;
}

// operator+=
```

```
template <typename R, typename E, bool is_const>
entry_iterator<R, E, is_const>&
entry_iterator<R, E, is_const>::operator+=(difference_type n) {
  e.advance((*this).position, n);
  return *this;
}

// operator-=

template <typename R, typename E, bool is_const>
entry_iterator<R, E, is_const>&
entry_iterator<R, E, is_const>::operator-=(difference_type n) {
  e.advance((*this).position, -n);
  return *this;
}

// operator+

template <typename R, typename E, bool is_const>
entry_iterator<R, E, is_const>
entry_iterator<R, E, is_const>::operator+(difference_type n) const {
  entry_iterator<R, E, is_const> temporary = *this;
  e.advance(temporary.position, n);
  return temporary;
}

// operator-

template <typename R, typename E, bool is_const>
entry_iterator<R, E, is_const>
entry_iterator<R, E, is_const>::operator-(difference_type n) const {
  entry_iterator<R, E, is_const> temporary = *this;
  e.advance(temporary.position, -n);
  return temporary;
}

// iterator distance

template <typename R, typename E, bool is_const>
typename entry_iterator<R, E, is_const>::difference_type
entry_iterator<R, E, is_const>::operator-(entry_iterator<R, E,
    is_const> const& a) const {
  return e.distance((*this).position, a.position);
}

// operator==

template <typename R, typename E, bool is_const>
template <bool both>
bool
entry_iterator<R, E, is_const>::operator==(entry_iterator<R, E, both
    > const& a) const {
  return e.equal((*this).position, a.position);
}

// operator!=

template <typename R, typename E, bool is_const>
template <bool both>
bool
entry_iterator<R, E, is_const>::operator!=(entry_iterator<R, E, both
    > const& a) const {
  return !(*this == a);
}
```

```
  // operator<

  template <typename R, typename E, bool is_const>
  template <bool both>
  bool
  entry_iterator<R, E, is_const>::operator<(entry_iterator<R, E, both>
       const& a) const {
    return e.distance((*this).position, a.position) < 0;
  }

  // operator>

  template <typename R, typename E, bool is_const>
  template <bool both>
  bool
  entry_iterator<R, E, is_const>::operator>(entry_iterator<R, E, both>
       const& a) const {
    return a < *this;
  }

  // operator<=

  template <typename R, typename E, bool is_const>
  template <bool both>
  bool
  entry_iterator<R, E, is_const>::operator<=(entry_iterator<R, E, both
      > const& a) const {
    return !(a < *this);
  }

  // operator>=

  template <typename R, typename E, bool is_const>
  template <bool both>
  bool
  entry_iterator<R, E, is_const>::operator>=(entry_iterator<R, E, both
      > const& a) const {
    return !(*this < a);
  }

  // operator+(int, iterator)

  template <typename R, typename E, bool is_const>
  entry_iterator<R, E, is_const> operator+(typename R::difference_type
       n, entry_iterator<R, E, is_const> const& a) {
    return a + n;
  }

  // parametrized constructor

  template <typename R, typename E, bool is_const>
  entry_iterator<R, E, is_const>::entry_iterator(position_type const&
      _position)
    : position(_position) {
  }

  // convertion operator

  template <typename R, typename E, bool is_const>
  entry_iterator<R, E, is_const>::operator position_type() const {
    return (*this).position;
  }

}
```

Appendix B.4 entry_iterator.h++

```
/*
  A rank-based iterator is just a (pointer, index) pair where the
  pointer points to the data structure containing the cell referred to
  and the index is the rank of that cell in the sequence of cells
  storing the elements.

  The idea of combining iterators and const iterators into the same
  class is taken from [Matt Austern. Defining iterators and const
  iterators. C/C++ User's Journal 19,1 (2001), 74-79].

  Author: Jyrki Katajainen, Bo Simonsen, April 2008
*/

#ifndef __CPHSTL_ENTRY_ITERATOR__
#define __CPHSTL_ENTRY_ITERATOR__

#include <iterator> // defines std::random_access_iterator_tag
#include <cstddef> // defines std::size_t and std::ptrdiff_t
#include <utility> // defines std::pair

#include "type.h++" // defines cphstl::if_then_else

namespace cphstl {

  template <typename V, typename A, typename R>
  class vector;

  template <typename R, typename E, bool is_const = false>
  class entry_iterator {

  public:

    // types

    typedef std::random_access_iterator_tag iterator_category;
    typedef typename R::value_type value_type;
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;
    typedef typename if_then_else<is_const, value_type const*,
        value_type*>::type pointer;
    typedef typename if_then_else<is_const, value_type const&,
        value_type&>::type reference;

    typedef E entry;

  protected:

    // types

    typedef typename entry::template iterator_entry<is_const>
        iterator_entry;
    typedef typename iterator_entry::position_type position_type;

  public:

    // friends

    friend class entry_iterator<R, E, !is_const>;

    template <typename V, typename A, typename S>
    friend class cphstl::vector;

    // structors
```

```
    entry_iterator();
    entry_iterator(entry_iterator<R, E, false> const&);
    entry_iterator& operator=(entry_iterator const&);
    ~entry_iterator();

    // operators

    reference operator*() const;
    pointer operator->() const;
    entry_iterator& operator++();
    entry_iterator operator++(int);
    entry_iterator& operator--();
    entry_iterator operator--(int);
    entry_iterator& operator+=(difference_type);
    entry_iterator& operator-=(difference_type);
    entry_iterator operator+(difference_type) const;
    entry_iterator operator-(difference_type) const;
    difference_type operator-(entry_iterator const&) const;

    template <bool both>
    bool operator==(entry_iterator<R, E, both> const&) const;

    template <bool both>
    bool operator!=(entry_iterator<R, E, both> const&) const;

    template <bool both>
    bool operator<(entry_iterator<R, E, both> const&) const;

    template <bool both>
    bool operator>(entry_iterator<R, E, both> const&) const;

    template <bool both>
    bool operator<=(entry_iterator<R, E, both> const&) const;

    template <bool both>
    bool operator>=(entry_iterator<R, E, both> const&) const;

  private:
    // converters to be used by the container friends
    entry_iterator(position_type const&);
    operator position_type() const;

    position_type position;

    iterator_entry e;
  };

  template<typename R, typename E, bool both>
  entry_iterator<R, E, both>
  operator+(typename R::difference_type, entry_iterator<R, E, both>
      const&);

}

#include "entry_iterator.c++" // implements cphstl::entry_iterator

#endif
```

Appendix C. Source code - Example implementations

Appendix C.1 avl.h++

```
/*
```

```
  Header file for the AVL implementation

  Authors: Bo Simonsen, May 2008

*/

/*
 * Copyright (c) 2003
 * Stephan Lynge, Department of Computer Science University of
     Copenhagen
 *
 * Copyright (c) 2002, 2003
 * Herve Bronnimann, Polytechnic University
 *
 * Copyright (c) 1996,1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee
     ,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
     in
 * supporting documentation.  Neither the author nor Polytechnic
 * University, nor Silicon Graphics, nor Hewlett Packard, makes any
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied
     warranty.
 *
 */

#ifndef CPHSTL_AVL_HPP
#define CPHSTL_AVL_HPP

/*
AVL search tree class, designed for use in implementing STL
associative containers (set, multiset, map, and multimap).
*/

#include <functional>
#include <algorithm>
#include <iterator>
#include <memory>

#include "node_iterator.h++"
#include "is_sorted.c++" // Defines cphstl::is_sorted generic
    algorithm
#include "avl_comp.h++"

#include <iostream>

namespace cphstl {

  namespace tree {

    /* Forward declarations */
    template <typename V>
    class avl_tree_node;
    template <typename K, typename V, typename F, typename C, typename
         A, typename N, bool is_multiset>
    class f_insert;
```

```
  template <typename K,typename V,typename F, typename C, typename A
      ,typename N, bool is_multiset>
  class f_lowerbound;
  template <typename K,typename V,typename F, typename C, typename A
      ,typename N, bool is_multiset>
  class f_upperbound;

template <
  typename K,
  typename V = K,
  typename F = cphstl::unnamed::identity<V>,
  typename C = std::less<K>,
  typename A = std::allocator<V>,
  typename N = cphstl::tree::avl_tree_node<V>,
  bool is_multiset = false
>
  class avl_tree
  {
    friend class f_upperbound<K,V,F,C,A,N,is_multiset>;
    friend class f_lowerbound<K,V,F,C,A,N,is_multiset>;

                           // public types
  public:
    typedef K key_type;
    typedef V value_type;
    typedef C key_compare;
    typedef N node_type;
    typedef F functor_type;
    typedef typename cphstl::if_then_else<cphstl::types<K, V>::
        are_same, C, cphstl::tree::value_compare<K, V, C> >::type
        value_compare;

    typedef typename A::template rebind<node_type>::other
        allocator_type;

    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef value_type* pointer;
    typedef value_type const* const_pointer;
    typedef value_type& reference;
    typedef value_type const& const_reference;

    typedef cphstl::node_iterator<node_type, true> const_iterator;
    typedef cphstl::node_iterator<node_type, false> iterator;

  private:
    key_compare comparator;
    allocator_type allocator;
    node_type*     header;
    size_type node_count;

  public:
    explicit avl_tree(const key_compare& k, const allocator_type& a)
        ;
    avl_tree(const avl_tree& x);
    ~avl_tree();


    avl_tree& operator=(const avl_tree& x);

                                 // size accessors

    size_type size() const;
    size_type max_size() const;
```

```
  allocator_type get_allocator() const;
  key_compare key_comp() const;
  value_compare value_comp() const;

  std::pair<node_type*, bool> insert_unique(const value_type& v);
  node_type* insert_equal(const value_type& v);

  node_type* insert_unique(node_type* position, const value_type&
      v);
  node_type* insert_equal(node_type* position, const value_type& v
      );

  template <typename I>
  void insert_equal(I x, I y);
  template <typename I>
  void insert_unique(I x, I y);

  template <typename I>
  void insert_sorted(I x, I y);

  node_type* lower_bound(const key_type& k) const;
  node_type* upper_bound(const key_type& k) const;

  node_type* begin() const;
  node_type* end() const;
  size_type erase(node_type* position);
  size_type erase(key_type const& position);

  void swap(avl_tree& t);

  operator std::string() const;

  typename f_insert<K, V, F, C, A, N, is_multiset>::return_type1
  insert(value_type const& v);

  typename f_insert<K, V, F, C, A, N, is_multiset>::return_type2
  insert(node_type* position, value_type const& v);

  template <typename I>
  void insert(I x, I y);

protected:
  node_type* make_node();
  void free_node(node_type* p);

  node_type* create_node(const value_type& x);

  void destroy_node(node_type* p);
  void destroy_tree(node_type* x);

  void construct(value_type* p, value_type const& v) const;
  void destroy(value_type* p)   const;
  void init();

  short get_side( node_type* x, node_type*& root);
  void rebalance(node_type* x, node_type*& root, int cmd, short
      side);
  void rotate_left( node_type* x, node_type*& root);
  void rotate_right( node_type* x, node_type*& root);

  short relink_for_erase(node_type* z, node_type*& root, node_type
      *& leftmost, node_type*& rightmost);

  node_type* copy_nodes(node_type* begin, node_type* end, int
      node_count);
```

```
      void copy(const avl_tree& x);
      node_type* build_tree(node_type* nodes[], int node_count);

      node_type*& root()      const;
      node_type*& leftmost()  const;
      node_type*& rightmost() const;

      node_type* tree_maximum(node_type* x) const;
      node_type* tree_minimum(node_type* x) const;

      node_type* _insert(node_type* x, node_type* y, const value_type&
           v, node_type* pred, node_type* succ);

      /* template parameter F is used, which would be appropiate, so Z
           is randomly selected */
      template <typename Z>
      node_type* find_node(const key_type& k, Z z) const;

    };


  } // namespace tree

} // namespace cphstl

#include "avl_func.h++"
#include "avl_node.h++"
#include "avl_set.c++"
#include "avl_multiset.c++"
#include "avl.c++"

#endif // CPHSTL_AVL_HPP
```

Appendix C.2 avl_node.h++

```
/*

  AVL node class

  Authors: Bo Simonsen, June 2008

*/

namespace cphstl {
  namespace tree {
    /* AVL tree node classes
     */
    template <typename V>
    class avl_tree_node {
    public:
      typedef V value_type;
      typedef avl_tree_node<value_type> node_type;

    private:
      node_type* parent;
      node_type* left;
      node_type* right;

      /* Pointers for an internal linked list which makes it possible
          to do iterator operations
         in O(1) time. */

      node_type* prev;
      node_type* next;
```

```
    short balance;
    value_type data;

  public:
    node_type*& get_parent() {
      return (*this).parent;
    }
    node_type*& get_left() {
      return (*this).left;
    }
    node_type*& get_right() {
      return (*this).right;
    }
    node_type*& get_prev() {
      return (*this).prev;
    }
    node_type*& get_next() {
      return (*this).next;
    }

    node_type* successor() const {
      return (*this).next;
    }
    node_type* predecessor() const {
      return (*this).prev;
    }


    value_type& get_data() {
      return (*this).data;
    }
    short& get_balance() {
      return (*this).balance;
    }

    value_type& content() {
      return data;
    }
    const value_type& content() const {
      return data;
    }
    operator std::string() const {
      std::stringstream ss, _ss;
      std::string balance_str;
      std::string data_str;

      ss << balance;
      ss >> balance_str;

      _ss << data;
      _ss >> data_str;

      std::string ret = "";

      if(left != 0) {
        ret += "Left\n" + (std::string) *left;
      }

      ret += "Node:␣" + data_str + "␣balance:␣" + balance_str + "\n"
          ;

      if(right != 0) {
        ret += "Right\n" + (std::string) *right;
      }
```

```
        return ret;
      }
    };

  }

}
```

Appendix C.3 dynamic_array.h++

```
/*
  A dynamic array is a container that doubles its capacity when it
  becomes full, and halves its capacity when it becomes only one-forth
  full.

  Authors: Tina A. G. Andersen, Jyrki Katajainen, Ulrik Schou
      Joergensen and Claus Ullerlund, Bo Simonsen,
  April 2008


  Important : This class cannot be used standalone. This class is a
      _realisation_ to be used with the _bridge-class_ vector(../../
      Vector).

  The design pattern is called a "bridge pattern":

  client <--> bridge <--> realisation

  The advantage of this, is you can change the realisation without
      confusing the client.

*/

#ifndef __CPHSTL_DYNAMIC_ARRAY__
#define __CPHSTL_DYNAMIC_ARRAY__

#include <memory> // defines std::allocator
#include <stdexcept> // defines std::out_of_range
#include <cstddef> // defines std::size_t and std::ptrdiff_t
#include <utility> // defines std::pair

#include "entry_iterator.h++" // defines cphstl::entry_iterator

namespace cphstl {

  template <typename V, typename A>
  class dynamic_array_entry;

  template <
    typename V,
    typename A = std::allocator<V>,
    typename E = dynamic_array_entry<V, A>
  >
  class dynamic_array {

  public:

    // types

    typedef V value_type;
    typedef A allocator_type;
    typedef V* pointer;
    typedef V const* const_pointer;
```

```
    typedef E entry;
    typedef std::ptrdiff_t difference_type;
    typedef std::size_t size_type;
    typedef cphstl::entry_iterator<dynamic_array<V, A, E>, E, false>
        iterator;
    typedef cphstl::entry_iterator<dynamic_array<V, A, E>, E, true>
        const_iterator;
    typedef typename entry::template iterator_entry<false>::
        position_type concrete_iterator;
    typedef typename entry::template iterator_entry<true>::
        position_type concrete_const_iterator;

    // structors

    explicit dynamic_array(A const& = A());
    explicit dynamic_array(size_type, V const& = V(), A const& = A());
    dynamic_array(dynamic_array<V, A, E> const&);
    ~dynamic_array();

    // iterators

    concrete_iterator begin();
    concrete_const_iterator begin() const;

    // accessors

    A get_allocator() const;
    size_type size() const;
    size_type max_size() const;
    size_type capacity() const;
    V const& operator[](size_type) const;

    dynamic_array<V,A>& operator=(const dynamic_array<V,A>&);

    // modifiers

    V& operator[](size_type);
    void reserve(size_type);
    concrete_iterator insert(concrete_iterator, V const&);
    void insert(concrete_iterator, size_type, V const&);

    template <typename I>
    void insert(concrete_iterator, I, I);

    concrete_iterator erase(concrete_iterator);
    concrete_iterator erase(concrete_iterator, concrete_iterator);
    void swap(dynamic_array<V, A, E>&);

  protected:

    typedef typename A::template rebind<V>::other value_allocator_type
        ;

    size_type current_capacity;
    size_type current_size;
    pointer array_p;
    value_allocator_type allocator;

  };

}

#include "dynamic_array_entry.h++"
#include "dynamic_array.c++" //implements cphstl::dynamic_array
```

```
#endif
```

Appendix C.4 dynamic_array_entry.h++

```
/*
  A definition of the "Entry" class for dynamic array which should be
      customizing the
  container (it's given as template parameter).

  The current use is only iterator related.

  Author: Bo Simonsen, May 2008

*/



namespace cphstl
{
  template <typename V, typename A>
  class dynamic_array_entry {
  public:
    template <bool is_const>
    class iterator_entry {
    public:
      typedef V value_type;
      typedef dynamic_array<V, A> container_type;
      typedef std::size_t size_type;
      typedef std::ptrdiff_t difference_type;

      typedef typename if_then_else<is_const, std::pair<const
          container_type*, size_type>, std::pair<container_type*,
          size_type> >::type position_type;
      typedef typename if_then_else<!is_const, std::pair<const
          container_type*, size_type>, std::pair<container_type*,
          size_type> >::type negated_position_type;
      typedef typename if_then_else<is_const, const value_type&,
          value_type&>::type v_return;

      v_return content(const position_type& pos) const {
        return (*pos.first)[pos.second];
      }
      void advance(position_type& pos, const difference_type n) const
          {
        pos.second += n;
      }

      /* First argument is _allways_ "position_type" but second
          argument can differ */

      difference_type distance(const position_type& pos1, const
          position_type& pos2) const {
        assert(pos1.first == pos2.first);
        return pos1.second - pos2.second;
      }
      difference_type distance(const position_type& pos1, const
          negated_position_type& pos2) const {
        assert(pos1.first == pos2.first);
        return pos1.second - pos2.second;
      }
      bool equal(const position_type& pos1, const position_type& pos2)
           const {
        return (pos1.first == pos2.first) && (pos1.second == pos2.
            second);
      }
```

```
      bool equal(const position_type& pos1, const
          negated_position_type& pos2) const {
        return (pos1.first == pos2.first) && (pos1.second == pos2.
            second);
      }
    };
  };
}
```