Taris presents

Free words concerning C++ STL

Contents

Part I

Presentation of the C++ STL

Chapter 1

What is the C++ STL ?

1.1 Introduction to C++ STL

The C++ STL (which stands for *Standard templates library*) is a collection of classes and functions. These are very useful in many cases and allow developers to use them instead of creating their own classes to do the same work.

The STL is different depending on the platform you are working on. However the basic classes and functions are the same. This documentation is about GNU STL provided with the *gcc* compiler. Most of the classes described in this document exist on all good C++ compilers.

1.2 Why use the STL ?

There are many reasons to use it. First of all, compiler creators have had many reflexions on how to improve the STL implementation. Probably more than any developers who wants to use some classes or functions for just one project.

Moreover, your STL implementation may have some bugs (in fact, almost all libraries and programs have). But, generally, due to the number of developers using it, those bugs are known. You can probably browse the website of the creator of your STL and see what is not implemented and what is not working like it should be.

Finally, the STL is generally optimized for the system your working on. In many cases, the STL provided with your compiler will be faster than anything (portable) you can write on your own.

1.3 Why not use the STL ?

Perhaps you are a very strong developer. Perhaps the STL you have is very buggy and very slow. Or perhaps, you think you can write your own STL, using some assembly lines, which will be faster than the basic STL. In all those cases, you can join the GNU project (see *www.gnu.org*) and improve your STL or create your own. All the developers using it will gladly receive your contribution.

1.4 What is a template class ?

A template class is a class that can be instantiated with any type, unless this type does not satisfy the constraints required to be used by the template class. In other words, a template class is a class which will be given a type in argument at the creation of an object.

For example, the *vector* class can contain objects of a given type. This type is given when a new *vector* object is created. If you want your *vector* to contain *int*s, you can declare it like this :

```
vector<int> v;
```

But the same class *vector* can be used to contain *float* or to contain a *vector* of *int* :

```
vector<float> v_f;
vector< vector<int> > v_of_v_int;
```

Thus, a template class has only one implementation but can be used with multiple types. The problem is that creating a template class generally creates constraints on the type.

In the *vector* case the type must have a public copy constructor. Many classes have it, but not all. So, you cannot store all classes in a vector.

1.5 STL headers

Here is a description of all STL headers and their contents :

- **deque** : double access queue
- **functional** : template and utility function definitions
- **hash_map** : hash map container definition
- **hash_set** : hash set container definition
- **iomanip** : input and output stream manipulation functions
- **iosfwd** : input and output types forward declaration
- **iostream** : input and output stream classes
- **iterator** : various iterator types
- **list** : double linked list
- **map** : associative array structure
- **memory** : *auto_ptr* class definition
- **numeric** : numeric functions
- **queue** : a queue structure (first in, first out)
- **rope** : *rope* class definition
- **set** : a set structure (in the mathematical way)
- **slist** : *slist* class definition
- **sstream** : *stringbuf* class definition
- **stack** : a stack structure (first in, last out)
- **stdexcept** : standard exceptions
- **string** : string class definition

- **strstream** : string stream class definition
- **utility** : utility classes
- **valarray** : *valarray* class definition
- **vector** : equivalent of an array structure with an automatic reallocation

1.6 STL algorithms

The C++ STL provide an entire set of algorithms : from searching to sorting, the most used algorithms are included with an optimized implementation.

Those algorithms are mostly used on STL containers. Thus, you can create your own container and apply the STL algorithms on it if you follow a few rules in your implementation.

1.7 STL containers

A container is an object that contains objects. Containers are data structures useful in many cases. There is some algorithms which performances depend only of the container used in the implementation.

The C++ STL offers many containers. Each container grants fast operations of one type : fast access to the beginning and the end, fast random access, etc.

1.8 Iterators

An iterator is something pretty much like a pointer on an element of a container. Exactly, an iterator is an object which offers the same operations as a *C* pointer : ++, != and =.

For example, consider the use of an array of *int*. If you use the following code to display all the array elements :

```
int array[10];
int* p;

for(p = &array[0]; p != &array[10]; p++)
{
        cout << *p << endl;
}
```

With a STL container (say *vector*) and an iterator you can write :

```
vector<int> array;
vector<int>::iterator it;

for(it = array.begin(); it != array.end(); it++)
{
        cout << *it << endl;
}
```

Thus, iterators allow you to use all STL containers the same way.

1.9 Allocators

An allocator is an object used to allocate and deallocate memory. The allocators encapsulate these operations.

The STL offers standard allocators but it may be interesting to implement your own. For example, you can implement your own allocator for allocating small objects, because the standard ones are not very effective in allocating small objects.

1.10 Adaptors

An adaptor is a class that uses another underlying class to work. An adaptor usually takes a class name in a template parameter. Then, it can use any class that provides the functions called by the adaptor.

1.11 Functors

A functor is an object used like a function. The class implements the *operator()*. Then you can use the object as if it was a function.

For example :

```
#include <iostream>

class succ_functor
{
    public:
        int operator()(int an_int)
        {
                return an_int + 1;
        }
};

int main(void)
{
        int i = 12;
        succ_functor succ;

        /* Displays '13' */
        cout << succ(i) << endl;
}
```

There is a great terminology problem with the word *functor*. Firstly, *function object* is often used for objects that are used like functions and *functor* for both classes that are used like functions and their objects. So, often you will see the term *function object* instead of *functor*.

Secondly, many C++ users prefer the term *function object* to *functor*, because *functor* has a defined meaning in mathematics, and it is not exactly the same in C++. In spite of this problem, the word *functor* will be used in this documentation : there is nothing in this documentation concerning the mathematical functors, so you can be sure that the word *functor* means C++ functor.

Part II

Iterators

Chapter 1

What is an iterator ?

An iterator is an abstraction used to represent a pointer on a sequence. It means that you can access sequence elements by using iterators without knowing anything about element type or position.

You can use anything as an iterator. But, usually, to be called an iterator an object implements three basic operations :

- indirection (done by operator * and ->)
- incrementation (operator ++)
- comparison (operator == and ! =)

1.1 Why are iterators so important ?

Iterators are important because they provide an abstraction that makes all containers identical when viewed from the outside. For example, almost all STL algorithm use iterators to work using the same manner on all container flavour.

1.2 Iterators example

Usually, the first time you see a C++ code part using an iterator you cannot miss the huge word used to declare the iterator. Although it seems complicated at first sight, it is really very simple. You just have to remember that iterator types is often contained within a class declaration.

For example, if you want to create an iterator on a vector of *int* you can use :

```
vector<int>::iterator it;
```

It means : "I want to use the iterator type declared within the class *vector*".

Now you can use your iterator to go across the vector by using *operator++* and you can display an element by dereferencing the iterator :

```
vector<int> my_vector;
vector<int>::iterator it;

for(it = my_vector.begin(); it != my_vector.end(); it++)
{
        cout << *it << endl;
}

/* Another way to write it */
it = my_vector.begin();
while(it != my_vector.end())
{
        cout << *it << endl;
        it++;
}
```

This can be read : "From the beginning of *my_vector* and while the iterator does not reach the end, send the current element on *cout*."

1.3 What are the different types of iterator ?

There are different types of iterator. Each type provides a different set of access operations. All iterator types provide the operations ==, *!=* and =. In the following array *it* and *it2* are iterators and *x* is a variable pointed to by an iterator. *n* is a variable of type *int* :

Operations provided by the different types of iterator are :

Iterator type	Operations provided
InputIterator	++it, it++, x = *it
OutputIterator	++it, it++, *it = x
ForwardIterator	++it, it++, x = *it, *it = x
BidirectionalIterator	++it, it++, it–, –it, x = *it, *it = x
RandomAccessIterator	++it, it++, it–, –it, x = *it, *it = x, it + n, it - n, it += n, it -= n, it < it2, it > it2, it <= it2, it >= it2

Part III

Algorithms

Chapter 1

Introduction on STL algorithms

The STL library provides a complete set of algorithms that can be used to avoid painful reimplementations.

All algorithms can be used by including the header <*algorithm*>.

Chapter 2

Comparing algorithms

2.1 Lexicographical compare

Lexicographical algorithms use *operator<* to compare iterators of two sequences

```
/* Compares two sequences. When a difference
is found, returns 'true' if the
first sequence is less than the second
(using 'operator<' to compare iterators)
or 'false' in the other case .
If the sequences are the same it returns
'false'. */
template <class InputIter1, class InputIter2>
bool lexicographical_compare(InputIter1 first1, InputIter1 last1,
                             InputIter2 first2, InputIter2 last2);

/* Uses 'comp' instead of 'operator<' */
template <class InputIter1, class InputIter2, class Compare>
bool lexicographical_compare(InputIter1 first1, InputIter1 last1,
                             InputIter2 first2, InputIter2 last2, Compare comp);

/* This one uses 'memcmp' */
bool lexicographical_compare(const unsigned char* first1, const unsigned char* last1,
                             const unsigned char* first2, const unsigned char* last2);

/* Calls the previous function */
bool lexicographical_compare(const char* first1, const char* last1, const char* first2,
                             const char* last2);

/* This function is not part of the C++ standard.
It returns '-1' if the first sequence is less
than the second. '0' if they are the same.
Or '1' in the other case. */
template <class InputIter1, class InputIter2>
int lexicographical_compare_3way(InputIter1 first1, InputIter1 last1,
                                 InputIter2 first2, InputIter2 last2);
```

2.2 Equal and mismatch

Those algorithms can be used to compare two sequences :

```
/* Returns the first identical iterators
(using 'operator==' of the iterator) */
template <class InputIter1, class InputIter2>
pair<InputIter1, InputIter2> mismatch(InputIter1 first1, InputIter1 last1,
                                      InputIter2 first2);

/* Uses 'bin_pred' instead of 'operator==' */
template <class InputIter1, class InputIter2>
pair<InputIter1, InputIter2> mismatch(InputIter1 first1, InputIter1 last1,
                                      InputIter2 first2, BinaryPredicate bin_pred);

/* Returns 'true' if sequences are equal
or 'false' either (using 'operator==' of the iterator) */
template <class InputIter1, class InputIter2>
bool equal(InputIter1 first1, InputIter1 last1,
           InputIter2 first2);

/* Uses 'bin_pred' instead of 'operator==' */
template <class InputIter1, class InputIter2>
bool equal(InputIter1 first1, InputIter1 last1,
           InputIter2 first2, BinaryPredicate bin_pred);
```

Chapter 3

Copying algorithms

3.1 *copy*

The copy function copies a sequence to another location, this new sequence will start at *result* :

```
/* Returns an iterator on the element following
the last copied element */
template <class InputIter, class OutputIter>
OutputIter copy(InputIter first, InputIter last, OutputIter result);
```

3.2 *copy_backward*

This function is the same as *copy*, but the copy is made from the end of the source sequence to its beginning :

```
/* Returns an iterator on the element following
the last copied element */
template <class BidirectionalIter1, class BidirectionalIter2>
BidirectionalIter2 copy_backward(BidirectionalIter1 first, BidirectionalIter1 last,
                                 BidirectionalIter2 result);
```

3.3 *copy_n*

This function is not part of the C++ standard, it copies n elements from the source sequence to another sequence :

```
/* Returns a pair composed with the element
following the last element in the
source sequence and the element following
the last in the destination sequence */
template <class InputIter, class Size, class OutputIter>
pair<InputIter, OutputIter> copy_n(InputIter first, Size count, OutputIter result);
```

Chapter 4

Counting algorithms

4.1 *count*

The *count* algorithm counts the element in a range equals to a value using *operator==* :

```
/* Return the number of element
equals to 'value' using 'T::operator==' */
template <class InputIter, class T>
iterator_traits<InputIter>::difference_type count(InputIter first, InputIter last, const T& value);
```

4.2 *count_if*

The *count_if* algorithm counts the element in a range using a predicate instead of *operator==* :

```
/* Return the number of element
equals to 'value' using 'pred' */
template <class InputIter, class Predicate>
iterator_traits<InputIter>::difference_type count_if(InputIter first, InputIter last, Predicate pred);
```

Chapter 5

Filling algorithms

5.1 *fill*

This algorithm fills a range with a given value by using *operator=* :

```
template <class ForwardIter, class T> void fill(ForwardIter first, ForwardIter last, const T& value);
```

5.2 *fill_n*

This algorithms does the same thing that *fill* but fills *n* elements :

```
/* Returns an iterator on the last filled element */
template <class OutputIter, class Size, class T>
OutputIter fill_n(OutputIter first, Size n, const T& value);
```

Chapter 6

for_each

6.1 *for_each*

This algorithm applies a function to every element of a range :

```
/* 'f' will be applied to each element.
This function returns 'f' */
template <class InputIter, class Function>
Function for_each(InputIter first, InputIter last, Function f);
```

Chapter 7

generate

7.1 *generate*

This algorithm uses a generator to generate each element of a range :

```
/* Uses 'operator=' of the element contained in the
range to set elements.
'gen' is called by simply using 'gen()' */
template <class ForwardIter, class Generator>
void generate(ForwardIter first, ForwardIter last, Generator gen);
```

7.2 *generate_n*

This algorithm does the same thing that *generate* but with n elements after the first :

```
/* Returns an iterator on the last element generated */
template <class OutputIter, class Size, class Generator>
OutputIter generate_n(OutputIter first, Size n, Generator gen);
```

Chapter 8

Heap algorithms

All heap algorithms work on a range assumed to be a heap created by *make_heap*.

A heap is a particular way of ordering elements. The first element is the greatest and the last is the lowest.

Adding and deleting element from a heap is done in a logarithmic time. Sorting a heap by using the *sort_heap* function is done in $N * log(N)$ in the worst case (where N is the number of elements in the heap).

8.1 *push_heap*

push_heap adds an element to a heap. The element added is **(last - 1)*.

```
/* Uses 'operator<' to compare elements */
template <class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);

/* Uses 'comp' instead of 'operator<' to compare elements */
template <class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

8.2 *pop_heap*

This function removes the greatest element from the heap (the first element).

```
/* Uses 'operator<' to compare elements */
template <class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

/* Uses 'comp' instead of 'operator<' */
template <class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

8.3 *make_heap*

This function reorders the elements between *first* and *last* in order to create a heap.

```
/* Uses 'operator<' to compare elements */
template <class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);

/* Uses 'comp' instead of 'operator<' */
template <class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

8.4 *sort_heap*

This function uses a heap to create a sorted range. It does the same thing than doing a *pop_heap* on the heap until no more element is in the heap.

```
/* Uses 'operator<' to compare elements */
template <class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

/* Uses 'comp' instead of 'operator<' */
template <class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

8.5 *is_heap*

This function checks if a range is a heap or not. It returns true if the range is a heap or false otherwise.

```
template <class RandomAccessIter>
bool is_heap(RandomAccessIter first, RandomAccessIter last);

template <class RandomAccessIter, class StrictWeakOrdering>
bool is_heap(RandomAccessIter first, RandomAccessIter last, StrictWeakOrdering comp);
```

Chapter 9

Merging algorithms

9.1 Merge

The *merge* function merges two ranges and returns an iterator on the first element of the merged range. It assumes that the two ranges are sorted. The output range is sorted.

```
/* Uses 'operator<' to compare elements */
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter merge(InputIter1 first1, InputIter1 last1, InputIter2 first2, InputIter2 last2,
                 OutputIter result);

/* Uses 'comp' instead of 'operator<' */
template <class InputIter1, class InputIter2, class OutputIter, class Compare>
OutputIter merge(InputIter1 first1, InputIter1 last1, InputIter2 first2, InputIter2 last2,
                 OutputIter result, Compare comp);
```

9.2 Inplace merge

This functions takes two parts of a range, each one sorted, and returns a new sorted range by merging the two parts.

```
/* Uses 'operator<' to compare elements */
template <class BidirectionalIter>
void inplace_merge(BidirectionalIter first, BidirectionalIter middle, BidirectionalIter last);

/* Uses 'comp' instead of 'operator<' */
template <class BidirectionalIter, class Compare>
void inplace_merge(BidirectionalIter first, BidirectionalIter middle, BidirectionalIter last, Compare comp);
```

Chapter 10

Minimum and maximum

10.1 Min

This function returns the lowest of its two arguments.

```
template <class T> const T& min(const T& a, const T& b);

/* Uses 'comp' to compare 'a' and 'b' */
template <class T, class Compare> const T& min(const T& a, const T& b, Compare comp);
```

10.2 Max

This function returns the greatest of its two arguments.

```
template <class T> const T& max(const T& a, const T& b);

/* Uses 'comp' to compare 'a' and 'b' */
template <class T, class Compare> const T& max(const T& a, const T& b, Compare comp);
```

10.3 Min element

This function returns the lowest element of a range.

```
template <class ForwardIter> ForwardIter min_element(ForwardIter first, ForwardIter last);

/* Uses 'comp' to compare elements of the range */
template <class ForwardIter, class Compare>
ForwardIter min_element(ForwardIter first, ForwardIter last, Compare comp);
```

10.4 Max element

This function returns the greatest element of a range.

```
template <class ForwardIter> ForwardIter max_element(ForwardIter first, ForwardIter last);

/* Uses 'comp' to compare elements of the range */
template <class ForwardIter, class Compare>
ForwardIter max_element(ForwardIter first, ForwardIter last, Compare comp);
```

Chapter 11

Mismatch

11.1 Mismatch

This function crosses two ranges and stops when the two ranges differ. It returns a pair containing two iterators on the differing elements.

```
/* Uses 'operator==' to compare elements */
template <class InputIter1, class InputIter2>
pair<InputIter1, InputIter2> mismatch(InputIter1 first1, InputIter1 last1, InputIter2 first2);

/* Uses 'pred' instead of 'operator==' */
template <class InputIter1, class InputIter2, class BinaryPredicate>
pair<InputIter1, InputIter2> mismatch(InputIter1 first1, InputIter1 last1,
                                      InputIter2 first2, BinaryPredicate pred);
```

Chapter 12

nth_element

12.1 Nth element

This function sorts the elements between the first and the nth element in ascendant order. The elements between the nth element and the last are not sorted, however no element in between the nth and the last is smaller than an element between the first and the nth element.

```
/* Uses 'operator<' to compare elements */
template <class RandomAccessIter>
void nth_element(RandomAccessIter first, RandomAccessIter nth, RandomAccessIter last);

/* Uses 'comp' to compare elements */
template <class RandomAccessIter, class Compare>
void nth_element(RandomAccessIter first, RandomAccessIter nth, RandomAccessIter last,
                 Compare comp);
```

Chapter 13

Numeric algorithms

13.1 Accumulate

This function does a sum of all elements in a range and add a given value to it. The sum may be replaced by a given binary operation. The function returns the sum :

```
template <class InputIterator, class T> T accumulate(InputIterator first, InputIterator last, T init);

/* Makes a 'bin_op' instead of a sum */
template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation bin_op);
```

13.2 Adjacent difference

This function makes the difference of adjacent elements between *first* and *last*, then stores the result, starting at *result*.

Thus, **result* is equal to **first*, **(result + 1)* is equal to **(first + 1)* - **first* and so on. The function returns an iterator on the element following the last :

```
template <class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(InputIterator first, InputIterator last, OutputIterator result);

/* Uses 'bin_op' instead of a difference */
template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                   OutputIterator result, BinaryOperation bin_op);
```

13.3 Inner product

This function computes the inner product of two ranges and adds the result to the *init* value. Inner product is the sum of the product of all elements at the same position in each range. For two ranges of size 3, say *first1* and *first2* the inner product is :

$$[*first1] * [*first2] + [*(first1 + 1)] * [*(first2 + 1)] + [*(first1 + 2)] * [*(first2 + 2)]$$

The function returns the result of the inner product :

```
template <class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init);

/* Uses 'bin_op1' instead of a sum and 'bin_op2' instead of a product */
template <class InputIterator1, class InputIterator2, class T, class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init,
                BinaryOperation1 bin_op1, BinaryOperation2 bin_op2);
```

13.4 Partial sum

This function goes through a range and sums all elements between the current and the first. Here is an example :

```
#include <algo.h>
#include <iostream>

int main(int argc, char* argv[])
{
int v1[] = { 1, 1, 1, 1, 1, 1};
int v2[] = { 0, 0, 0, 0, 0, 0 };

        partial_sum(v1, v1 + 6, v2);

        /* Displays :
              1 2 3 4 5 6
        */
        for(int i = 0; i < 6; i++)
            {
                  cout << v2[i] << " ";
            }
        cout << endl;
}
```

The function returns an iterator on the element following the last :

```
template <class InputIterator, class OutputIterator>
OutputIterator partial_sum(InputIterator first, InputIterator last, OutputIterator result);

/* Does a 'bin_op' instead of a sum */
template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result, BinaryOperation bin_op);
```

13.5 Power

This function raises a value to a given power :

```
template <class T, class Integer> T power(T x, Integer n);

/* Uses 'oper' instead of a power */
template <class T, class Integer, class MonoidOperation>
T power(T x, Integer n, MonoidOperation oper);
```

Chapter 14

partition and *stable_partition*

14.1 *partition*

This function cuts a range in two parts : all elements in the first part satisfy a predicate and elements in the second part doesn't. This function returns the first element that doesn't satisfy the predicate. After a call to *partition*, relative order between elements may be changed.

```
template <class ForwardIter, class Predicate>
ForwardIter partition(ForwardIter first, ForwardIter last, Predicate pred);
```

14.2 *stable_partition*

This function does the same thing that *partition*, except that the relative order between elements is kept. Thus, *stable_partition* uses more computation time than *partition*.

```
template <class ForwardIter, class Predicate>
ForwardIter stable_partition(ForwardIter first, ForwardIter last, Predicate pred);
```

Chapter 15

Randomizing algorithms

15.1 Random shuffle

This function reorders the elements of a range by putting them at random places :

```
template <class RandomAccessIter>
void random_shuffle(RandomAccessIter first, RandomAccessIter last);

/* Uses 'rand' instead of the default random number generator */
template <class RandomAccessIter, class RandomNumberGenerator>
void random_shuffle(RandomAccessIter first, RandomAccessIter last, RandomNumberGenerator& rand);
```

Chapter 16

Removing algorithms

16.1 *remove*

This function removes from a range all elements equal to a given value :

```
/* Returns the last removed element, or
'first' if 'first' and 'last' are equal.
Uses 'operator!=' to compare elements */
ForwardIter remove(ForwardIter first, ForwardIter last, const T& value);
```

16.2 *remove_if*

This function removes all elements from a range that returns *true* using a given predicate.

```
template <class ForwardIter, class Predicate>
ForwardIter remove_if(ForwardIter first, ForwardIter last, Predicate pred);
```

16.3 *remove_copy*

This function copies all elements not equal to a given value to *result* :

```
/* Returns an iterator on element following
the last element of 'result' */
template <class InputIter, class OutputIter, class Tp>
OutputIter remove_copy(InputIter first, InputIter last, OutputIter result, const T& value);
```

16.4 *remove_copy_if*

This function copies all elements that doesn't verify the predicate *pred* into *result* :

```
/* Returns an iterator on element following
the last element of 'result' */
template <class InputIter, class OutputIter, class Predicate>
```

OutputIter remove_copy_if(InputIter first, InputIter last, OutputIter result, Predicate pred);

Chapter 17

Replacing algorithms

17.1 *replace*

This function replaces all value equal to *old_value* (using *operator==* of the iterator) by the value *new_value* in a range. The replacement is done by using *operator=* of the iterator.

```
template <class ForwardIter, class T>
void replace(ForwardIter first, ForwardIter last, const T& old_value, const T& new_value);
```

17.2 *replace_if*

This function does the same thing that the *replace* function but uses the *pred* predicate instead of *operator==* :

```
template <class ForwardIter, class _Predicate, class T>
void replace_if(ForwardIter first, ForwardIter last, Predicate pred, const T& new_value);
```

17.3 *replace_copy*

This function makes a copy of a range and replaces all *old_value* values by the value *new_value* into it. It uses *operator=* to make the copy and *operator==* to compare elements.

```
/* Returns an iterator on the element following the last
copied element */
template <class InputIter, class OutputIter, class Tp>
OutputIter replace_copy(InputIter first, InputIter last, OutputIter result,
                        const T& old_value, const T& new_value);
```

17.4 *replace_copy_if*

This function does the same thing that *replace_copy* but uses the predicate *pred* instead of *operator==* to compare elements :

```
template <class Iterator, class OutputIter, class Predicate, class T>
OutputIter replace_copy_if(Iterator first, Iterator last, OutputIter result,
                           Predicate pred, const T& new_value);
```

Chapter 18

Reversing and rotating algorithms

18.1 *reverse*

This function reverses a range :

```
template <class BidirectionalIter>
void reverse(BidirectionalIter first, BidirectionalIter last);
```

18.2 *reverse_copy*

This function makes a reversed copy of a range. It returns an iterator on the element following the last element copied :

```
/* Returns an iterator on the element
following the last copied element */
template <class BidirectionalIter, class OutputIter>
OutputIter reverse_copy(BidirectionalIter first, BidirectionalIter last, OutputIter result);
```

18.3 *rotate*

This function rotates a sequence. It means the sequence will start at the element in the middle of the source sequence and the last element will be followed by the first (it appends the element between the first and the middle element to the elements between the middle and the last element).

```
template <class ForwardIter>
ForwardIter rotate(ForwardIter first, ForwardIter middle,
                   ForwardIter last);
```

18.4 *rotate_copy*

This function make a rotated copy of a sequence. It means the resulting sequence will start at the element in the middle of the source sequence and the last element will be followed by the first (it appends the element between the first and the middle element to the elements between the middle and the last element).

```
/* Returns an iterator on the element
following the last copied element */
template <class ForwardIter, class OutputIter>
OutputIter rotate_copy(ForwardIter first, ForwardIter middle, ForwardIter last, OutputIter result);
```

Chapter 19

Searching algorithms

19.1 *adjacent_find*

adjacent_find searches two adjacent elements in a range that are equal (or satisfy a binary predicate) and returns an iterator on the first of those two elements. If such elements do not exist, it returns an iterator on the element following the last (that is, it returns *last*) :

```
/* Uses 'operator==' to compare elements */
template <class _ForwardIter>
ForwardIter adjacent_find(ForwardIter first, ForwardIter last);

/* Calls 'bin_pred' instead of 'operator==' */
template <class ForwardIter, class BinaryPredicate>
ForwardIter adjacent_find(ForwardIter first, ForwardIter last,
                          BinaryPredicate bin_pred);
```

19.2 *binary_search*

binary_search searches the element *val* in an ordered range and returns true if it is in the range or false otherwise :

```
/* Uses 'operator<' to compare elements */
template <class ForwardIter, class T>
bool binary_search(ForwardIter first, ForwardIter last,
                   const T& val);

/* Uses 'comp' instead of 'operator<' */
template <class ForwardIter, class T, class Compare>
bool binary_search(ForwardIter first, ForwardIter last,
                   const T& val, Compare comp);
```

19.3 *equal_range*

This function returns the lower bound and the upper bound of a range where a new value can be inserted without misordering the elements between *first* and *last* :

```
/* Uses 'operator<' to compare elements */
template <class ForwardIter, class T>
pair<ForwardIter, ForwardIter> equal_range(ForwardIter first, ForwardIter last, const T& val);

/* Uses 'comp' instead of 'operator<' */
template <class ForwardIter, class T, class Compare>
pair<ForwardIter, ForwardIter> equal_range(ForwardIter first, ForwardIter last,
                                           const T& val, Compare comp);
```

19.4 *find*

This function searches an element in a range and returns an iterator on it. If the element wasn't found it returns an iterator on the element following the last (that is, the *last* iterator) :

```
/* Uses 'operator!=' to compare elements */
template <class InputIter, class T>
InputIter find(InputIter first, InputIter last, const T& val);

/* Uses 'pred' instead of 'operator!=' */
template <class InputIter, class Predicate>
InputIter find_if(InputIter first, InputIter last, Predicate pred);
```

19.5 *find_end*

This function finds the last subsequence (elements between *first2* and *last2*) contained into a sequence (elements between *first1* and *last1*). It returns an iterator on the first element of the last occurence of the subsequence.

```
/* Uses 'operator==' to compare elements */
template <class ForwardIter1, class ForwardIter2>
ForwardIter1 find_end(ForwardIter1 first1, ForwardIter1 last1,
                      ForwardIter2 first2, ForwardIter2 last2);

/* Uses 'comp' instead of 'operator==' */
template <class ForwardIter1, class ForwardIter2, class BinaryPredicate>
ForwardIter1 find_end(ForwardIter1 first1, ForwardIter1 last1,
                      ForwardIter2 first2, ForwardIter2 last2,
                      BinaryPredicate comp);
```

19.6 *find_first_of*

This function searches any element of a sequence (elements between *first2* and *last2*) in another sequence (between *first* and *last*). It returns an iterator on the element found in the sequence (between *first1* and *last1*) or an iterator on the element following the last (that is, *last1*) if none of the elements was found in the range.

```
/* Uses 'operator==' to compare elements */
template <class InputIter, class ForwardIter>
InputIter find_first_of(InputIter first1, InputIter last1,
                        ForwardIter first2, ForwardIter last2);

/* Uses 'comp' instead of 'operator==' */
template <class InputIter, class ForwardIter, class BinaryPredicate>
InputIter find_first_of(InputIter first1, InputIter last1,
                        ForwardIter first2, ForwardIter last2,
                        BinaryPredicate comp);
```

19.7 *lower_bound*

This function returns the lower bound of a range. That is, the first position where a new value can be inserted without misordering the elements between *first* and *last* :

```
/* Uses 'operator<' to compare elements */
template <class ForwardIter, class T>
ForwardIter lower_bound(ForwardIter first, ForwardIter last, const T& val);

/* Uses 'comp' instead of 'operator<' */
template <class ForwardIter, class T, class Compare>
ForwardIter lower_bound(ForwardIter first, ForwardIter last,
                        const T& val, Compare comp);
```

19.8 *search*

This function searches a subsequence (elements from *first2* to *last2*) into sequence (elements from *first1* to *last1*). This function returns an iterator on the first element of the subsequence found in the sequence. If the subsequence wasn't found, *last1* is returned :

```
/* Uses 'operator==' to compare elements */
template <class ForwardIter1, class ForwardIter2>
ForwardIter1 search(ForwardIter1 first1, ForwardIter1 last1,
                    ForwardIter2 first2, ForwardIter2 last2);

/* Uses 'predicate' instead of 'operator==' */
template <class ForwardIter1, class ForwardIter2, class BinaryPred>
ForwardIter1 search(ForwardIter1 first1, ForwardIter1 last1,
                    ForwardIter2 first2, ForwardIter2 last2,
                    BinaryPred predicate);
```

19.9 *search_n*

This function searches *count* identical consecutive elements in a sequence, all equal to *val* :

```
/* Uses 'operator!=' to compare elements */
template <class ForwardIter, class Integer, class T>
ForwardIter search_n(ForwardIter first, ForwardIter last,
                     Integer count, const T& val);

/* Uses 'binary_pred' instead of 'operator!=' */
template <class ForwardIter, class Integer, class T, class BinaryPred>
ForwardIter search_n(ForwardIter first, ForwardIter last,
                     Integer count, const T& val,
                     BinaryPred binary_pred);
```

19.10 *upper_bound*

This function returns the upper bound of a range. That is, the last position where a new value can be inserted without misordering the elements between *first* and *last* :

```
/* Uses 'operator<' to compare elements */
template <class ForwardIter, class T>
ForwardIter upper_bound(ForwardIter first, ForwardIter last, const T& val);

/* Uses 'comp' instead of 'operator<' */
template <class ForwardIter, class T, class Compare>
ForwardIter upper_bound(ForwardIter first, ForwardIter last,
                   const T& val, Compare comp);
```

Chapter 20

Set algorithms

All set operations assume that the used ranges are sorted.

20.1 *includes*

This functions returns true if the elements between *first2* and *last2* are included between *first1* and *last1*, or false otherwise :

```
/* Uses 'operator<' to compare elements */
template <class InputIter1, class InputIter2>
bool includes(InputIter1 first1, InputIter1 last1,
              InputIter2 first2, InputIter2 last2);

/* Uses 'comp' instead of 'operator<' */
template <class InputIter1, class InputIter2, class Compare>
bool includes(InputIter1 first1, InputIter1 last1,
              InputIter2 first2, InputIter2 last2, Compare comp);
```

20.2 *set_difference*

This function creates the set difference of two sequences. It returns an iterator on the element following the last of the build sequence :

```
/* Uses 'operator<' to compare elements */
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_difference(InputIter1 first1, InputIter1 last1,
                          InputIter2 first2, InputIter2 last2,
                          OutputIter result);

/* Uses 'comp' instead of 'operator' */
template <class InputIter1, class InputIter2, class OutputIter, class Compare>
OutputIter set_difference(InputIter1 first1, InputIter1 last1,
                          InputIter2 first2, InputIter2 last2,
                          OutputIter result, Compare comp);
```

20.3 *set_intersection*

This function creates the set intersection of two sequences. It returns an iterator on the element following the last of the build sequence :

```
/* Uses 'operator<' to compare elements */
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_intersection(InputIter1 first1, InputIter1 last1,
                            InputIter2 first2, InputIter2 last2,
                            OutputIter result);

/* Uses 'comp' instead of 'operator' */
template <class InputIter1, class InputIter2, class OutputIter, class Compare>
OutputIter set_intersection(InputIter1 first1, InputIter1 last1,
                            InputIter2 first2, InputIter2 last2,
                            OutputIter result, Compare comp);
```

20.4 *set_symmetric_difference*

This function creates the set symmetric differenct of two sequences. It returns an iterator on the element following the last of the build sequence. The symmetric difference of two sets A and B is the union of $(A - B)$ and $(B - A)$.

```
/* Uses 'operator<' to compare elements */
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_symmetric_difference(InputIter1 first1, InputIter1 last1,
                                    InputIter2 first2, InputIter2 last2,
                                    OutputIter result);

/* Uses 'comp' instead of 'operator' */
template <class InputIter1, class InputIter2, class OutputIter, class Compare>
OutputIter set_symmetric_difference(InputIter1 first1, InputIter1 last1,
                                    InputIter2 first2, InputIter2 last2,
                                    OutputIter result, Compare comp);
```

20.5 *set_union*

This function creates the set union of two sequences. It returns an iterator on the element following the last of the build sequence :

```
/* Uses 'operator<' to compare elements */
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_union(InputIter1 first1, InputIter1 last1,
                     InputIter2 first2, InputIter2 last2,
                     OutputIter result);

/* Uses 'comp' instead of 'operator' */
template <class InputIter1, class InputIter2, class OutputIter, class Compare>
OutputIter set_union(InputIter1 first1, InputIter1 last1,
InputIter2 first2, InputIter2 last2,
OutputIter result, Compare comp);
```

Chapter 21

Sorting algorithms

21.1 *sort*

This function sorts a range in ascending order :

```
/* Uses 'operator<' to compare elements */
template <class RandomAccessIter>
void sort(RandomAccessIter first, RandomAccessIter last);

/* Uses 'comp' to compare elements */
template <class RandomAccessIter, class Compare>
void sort(RandomAccessIter first, RandomAccessIter last, Compare comp);
```

21.2 *partial_sort_copy*

This function partially sort a range and put the result in a new range (to know what means *partially sort* see *partial_sort* on page 54). The number of elements copied is the minimum of *result_last* - *result_first* and *last* - *first*. It returns an iterator on the element following the last element copied.

```
/* Uses 'operator<' to compare elements */
template <class InputIter, class RandomAccessIter>
RandomAccessIter partial_sort_copy(InputIter first, InputIter last,
                                   RandomAccessIter result_first,
                                   RandomAccessIter result_last);

/* Uses 'comp' to compare elements */
template <class InputIter, class RandomAccessIter, class Compare>
RandomAccessIter partial_sort_copy(InputIter first, InputIter last,
                                   RandomAccessIter result_first,
                                   RandomAccessIter result_last, Compare comp);
```

21.3 *partial_sort*

This function partially sorts a range. After calling this function, the elements between *first* and *middle* will be sorted and the elements between *middle* and *last* will be in an unspecified order. That is, this function sorts a range of size *middle* - *first* by taking its elements between *first* and *last* :

```
/* Uses 'operator<' to compare elements */
template <class RandomAccessIter>
void partial_sort(RandomAccessIter first, RandomAccessIter middle, RandomAccessIter last);

/* Uses 'comp' to compare elements */
template <class RandomAccessIter, class Compare>
void partial_sort(RandomAccessIter first,
                  RandomAccessIter middle,
                  RandomAccessIter last, Compare comp);
```

21.4 *stable_sort*

This function sorts a range (like the *sort* function) but keeps the order of equivalent elements :

```
/* Uses 'operator<' to compare elements */
template <class RandomAccessIter>
void stable_sort(RandomAccessIter first,
                 RandomAccessIter last);

/* Uses 'comp' to compare elements */
template <class RandomAccessIter, class Compare>
void stable_sort(RandomAccessIter first,
                 RandomAccessIter last, Compare comp);
```

Chapter 22

Swaping algorithms

22.1 *iter_swap*

This functions swaps two values by taking iterators on them :

```
/* Equivalent to 'swap(*a, *b)' */
template <class ForwardIter1, class ForwardIter2>
void iter_swap(ForwardIter1 a, ForwardIter2 b);
```

22.2 *swap*

This functions swaps the value of its two parameters :

```
/* Uses 'T::operator=' to swap elements */
template <class T> void swap(T& a, T& b);
```

22.3 *swap_ranges*

This function swaps the elements of two sequence, that is each element at a position in the first sequence is replaced by the element at the same position in the second sequence and vice-versa :

```
/* Returns on the element following the last of the
second sequence */
template <class ForwardIter1, class ForwardIter2>
ForwardIter2 swap_ranges(ForwardIter1 first1, ForwardIter1 last1,
                         ForwardIter2 first2);
```

Chapter 23

transform

23.1 Transform

This algorithm applies a function to each element of a sequence and puts the result starting at the *result* iterator :

```
template <class InputIter, class OutputIter, class UnaryOperation>
OutputIter transform(InputIter first, InputIter last,
                     OutputIter result, UnaryOperation oper);

/* Applies a binary operation with in first parameter
an element from the first sequence and in second parameter
an element from the second sequence */
template <class InputIter1, class InputIter2, class OutputIter, class BinaryOperation>
OutputIter transform(InputIter1 first1, InputIter1 last1,
                     InputIter2 first2, OutputIter result,
                     BinaryOperation bin_op);
```

Chapter 24

unique

24.1 *unique*

This function transforms a sequence such as each duplicate consecutive elements become a unique element :

```
/* Returns an iterator on the element
following the last (that is 'last')
(uses 'operator!=' to compare elements) */
template <class ForwardIter>
ForwardIter unique(ForwardIter first, ForwardIter last);

/* Uses 'pred' to compare element */
template <class ForwardIter, class BinaryPredicate>
ForwardIter unique(ForwardIter first, ForwardIter last, BinaryPredicate binary_pred);
```

24.2 *unique_copy*

This function copies a sequence such as each duplicate consecutive elements become a unique element.

```
/* Returns an iterator on the element
following the last
(uses 'operator!=' to compare elements) */
template <class InputIter, class OutputIter>
OutputIter unique_copy(InputIter first, InputIter last, OutputIter result);

/* Uses 'bin_pred' to compare elements */
template <class InputIter, class OutputIter, class BinaryPredicate>
OutputIter unique_copy(InputIter first, InputIter last, OutputIter result, BinaryPredicate bin_pred);
```

Part IV

Sequence Containers

Chapter 1

Introduction on STL containers

1.1 What is a container ?

A container is a structure designed to contain other objects. In the C++ case, containers contain objects of the same type. This type is fixed when the container is first created and cannot be changed after.

What is so interesting about STL containers ? The *thing* is that all STL containers have the same interface (only a few functions are specific to a type of container). Moreover, all containers are optimized for the type of operation they were created for.

All the containers are defined in the namespace *std*.

Note that STL containers take an object by reference when adding it to them. It means that the type used your container must have a public copy constructor.

1.2 What is an associative container ?

An associative container is a template class with two types : the type of the *keys* and the type of the *values*. Each value can be accessed using a unique key.

1.3 What is a container adaptor ?

A container adaptor is a template class which use a sequence container (*vector*, *list* or *deque*) to create a new type of container without implementing new containers.

The container adaptors only use member function to access their content. They do not have iterators like sequence containers.

Chapter 2

Bit vector : <bvector.h>

The bit vector is used like a *vector<bool>*, nevertheless, the bit vector is optimized in space : it requires only one bit per element.

If the header *<bvector.h>* does not exist on your system, it means that the class is implemented by specializing the vector class. All systems does not offer support for partially specialized template, so some compiler still use the *<bvector.h>* header.

2.1 Bvector types

A bvector container has the following types :

```
/* Type of the elements */
typedef bool value_type;
typedef bool* pointer;
typedef const bool* const_pointer;

/* Iterator types are implementation dependant */
typedef Bit_iterator iterator;
typedef const Bit_const_iterator const_iterator;
typedef Bit_reference reference;
typedef Bit_const_reference const_reference;
typedef reverse_iterator<const_iterator> const_reverse_iterator;
typedef reverse_iterator<iterator> reverse_iterator;

/* This is implementation dependant */
typedef size_t size_type;
typedef ptrdiff_t difference_type;
typedef typename Bvector_base::allocator_type allocator_type;
```

2.2 Bvector constructors

A bvector container has the following constructors :

```
explicit bit_vector(const allocator_type& a = allocator_type());

/* Inits the bit_vector with 'n' copies of the 'value' */
bit_vector(size_type n, bool value, const allocator_type& a = allocator_type());
```

```
/* Allocates 'n' area of memory */
explicit bit_vector(size_type n);

/* Copy constructor */
bit_vector(const bit_vector& x);

/* 'first' and 'last' are iterators on a sequence.
The values between 'first and 'last' are copied
in the new bit_vector */
template <class InputIterator>
bit_vector(InputIterator first, InputIterator last, const allocator_type& a = allocator_type());
```

2.3 Bvector destructor

A bvector container has the following destructor :

```
~bit_vector();
```

2.4 Bvector operators

A bvector container has the following operators :

```
/* Unchecked access */
reference operator[](size_type n);
/* Constant access */
const_reference operator[](size_type n) const;

/* Copy operator */
bit_vector& operator=(const bit_vector& v);

/* Compares two bit vectors using 'equal' function */
bool operator==(const bit_vector& x, const bit_vector& y);
/* Lexicographical compare between bit vectors */
bool operator<(const bit_vector& x, const bit_vector& y);
```

2.5 Bvector iterators

A bvector has the following iterators :

```
/* Returns an iterator on the first element */
iterator begin();
const_iterator begin() const;
/* Returns an iterator on the element following the last */
iterator end();
const_iterator end() const;
```

```
/* Returns an iterator on the last element */
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
/* Returns an iterator on the element preceding the first */
reverse_iterator rend();
const_reverse_iterator rend() const;
```

2.6 Bvector size and capacity

A bvector size and capacity can be checked and changed using those functions :

```
/* The number of element */
size_type size() const;
/* The largest possible size
of this bit_vector (based on the size
of boolean type) */
size_type max_size() const;

/* Number of allocated elements (initialized
elements plus non initialized) */
size_type capacity() const;

/* Returns true if this bit_vector contains
no element, in this case 'begin() == end()' */
bool empty() const;

/* Tries to allocate more memory for 'n' more
elements. If 'n' is less than or equal to
'capacity()' this call does nothing */
void reserve(size_type n);

/* If the 'new_size' is less than
the current size it erases element from
the end to set the size to 'new_size'.
If the 'new_size' is greater then
the current size it inserts elements
at the end (initialized with 'x' value). */
void resize(size_type new_size, bool x = bool());
```

2.7 Accessing elements

Elements of a bvector can be accessed using those functions :

```
/* Returns the first element */
reference front();
/* Returns the first element */
const_reference front();
```

```
/* Returns the last element */
reference back();
/* Returns the last element */
const_reference back();
```

2.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
/* Appends the element 'x' at the end */
void push_back(bool x);
/* Appends a new unitialized element at
the end */
void push_back();

/* Inserts 'x' after 'pos' and return
'x' position */
iterator insert(iterator pos, bool x);
/* Adds a new non initialized element */
iterator insert(iterator pos);
/* Inserts all the elements between 'first'
and 'last' immediatly after 'pos' */
void insert(iterator pos, const_iterator first, const_iterator_last);
/* Inserts 'n' copies of 'x' immediatly after 'pos' */
void insert(iterator pos, size_type n, bool x);

/* Erases the last element */
void pop_back();

/* Erases the element at 'pos'.
Returns the new element at 'pos' */
iterator erase(iterator pos);
/* Erases all elements between 'first'
and 'last'.
Returns 'first' after deletion */
iterator erase(iterator first, iterator last);

/* Equivalent to 'erase(begin(), end())' */
void clear();
```

2.9 Misc functions

The bvector class has those functions too :

```
/* Destroys the content of 'this' bit_vector
and replaces it by 'n' copies of
'val' */
void assign(size_type n, bool val);
/* Destroys the content of 'this' bit_vector
```

```
and replaces it by the elements
between 'first' and 'last' */
template <class InputIterator>
void assign(InputIterator first, InputIterator last);

/* Swaps the content of    'this' bit_vector
with the content of 'x' */
void swap(bit_vector& x);
/* Swaps the content of 'x' and 'y' */
void swap(bit_vector& x, bit_vector& y);

/* Returns this bit_vector allocator */
allocator_type get_allocator() const;
```

Chapter 3

Deque : <deque>

The deque container is pretty much like the *vector* container. It can be used like an array (any element given by his index can be accessed).

Like *vector*, insertion and deletion are done in a constant time at the end of a *deque*.

Unlike *vector*, the insertion and deletion of element at the beginning of a *deque* are done in a constant time.

Operations on others position are done in linear time.

In the following sections, we will consider a deque constructed with the template $<T,\ A>$. Where T is the value type of the deque and A is the allocator type. The allocator type is optionnal.

3.1 Deque types

A deque container has the following types :

```
/* Type of the elements */
typedef T value_type;
typedef value_type* pointer;
typedef const value_type* const_pointer;

/* Iterator types are implementation dependant */
typedef value_type* iterator;
typedef const value_type* const_iterator;
typedef value_type& reference;
typedef const value_type& const_reference;
typedef reverse_iterator<const_iterator> const_reverse_iterator;
typedef reverse_iterator<iterator> reverse_iterator;

/* This is implementation dependant */
typedef size_t size_type;
typedef ptrdiff_t difference_type;
typedef typename Vector_base::allocator_type allocator_type;
```

3.2 Deque constructors

A deque container has the following constructors :

```
explicit deque(const allocator_type& a = allocator_type());

/* Copy constructor */
deque(const deque& x);

/* Inits the deque with 'n' copies of the 'value' */
deque(size_type n, const T& value, const allocator_type& a = allocator_type());

/* Allocates 'n' area of memory */
explicit deque(size_type n);

/* 'first' and 'last' are iterators on a sequence.
The values between 'first and 'last' are copied
in the new vector */
template <class InputIterator>
deque(InputIterator first, InputIterator last, const allocator_type& a = allocator_type());
```

3.3 Deque destructor

A deque container has the following destructor :

```
~deque();
```

3.4 Deque operators

A deque container has the following operators :

```
/* Unchecked access */
reference operator[](size_type n);
/* Constant access */
const_reference operator[](size_type n) const;

/* Copy operator */
deque<T, A>& operator=(const deque<T, A>& d);

/* Compare two vectors using 'equal' function */
bool operator==(const deque<T, A>& x, const deque<T, A>& y);
bool operator!=(const deque<T, A>& x, const deque<T, A>& y);
/* Lexicographical compare between vectors */
bool operator<(const deque<T, A>& x, const deque<T, A>& y);
```

3.5 Deque iterators

A deque has the following iterators :

```
/* Returns an iterator on the first element */
iterator begin();
const_iterator begin() const;
/* Returns an iterator on the element following the last */
iterator end();
const_iterator end() const;

/* Returns an iterator on the last element */
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
/* Returns an iterator on the element preceding the first */
reverse_iterator rend();
const_reverse_iterator rend() const;
```

3.6 Deque size and capacity

A deque size and capacity can be checked and changed using those functions :

```
/* The number of element */
size_type size() const;
/* The largest possible size
of this deque (based on the size
of element's type) */
size_type max_size() const;

/* Returns true if this vector contains
no element, in this case 'begin() == end()' */
bool empty() const;

/* If the 'new_size' is less than
the current size it erases element from
the end to set the size to 'new_size'.
If the 'new_size' is greater then
the current size it inserts elements
at the end (initialized with 'x' value). */
void resize(size_type new_size, const T& x);

/* Calls 'resize(size_type, value_type())' */
void resize(size_type new_size);
```

3.7 Accessing elements

Elements of a deque can be accessed using those functions :

```
/* Returns the first element */
reference front();
/* Returns the first element */
const_reference front();
```

```
/* Returns the last element */
reference back();
/* Returns the last element */
const_reference back();
```

3.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
/* Appends the element 'x' at the end */
void push_back(const T& x);
/* Appends a new unitialized element at
the end */
void push_back();

/* Appends the element 'x' at the beginning */
void push_front(const value_type& x);
/* Appends a new unitialized element at
the beginning */
void push_front();

/* Inserts 'x' after 'pos' and return
'x' position */
iterator insert(iterator pos, const T& x);
/* Adds a new non initialized element */
iterator insert(iterator pos);
/* Inserts all the elements between 'first'
and 'last' immediatly after 'pos' */
void insert(iterator pos, const_iterator first, const_iterator_last);
/* Inserts 'n' copies of 'x' immediatly after 'pos' */
void insert(iterator pos, size_type n, const T& x);

/* Erases the last element */
void pop_back();
/* Erases the first element */
void pop_front();

/* Erases the element at 'pos'.
Returns the new element at 'pos' */
iterator erase(iterator pos);
/* Erases all elements between 'first'
and 'last'.
Returns 'first' after deletion */
iterator erase(iterator first, iterator last);

/* Equivalent to 'erase(begin(), end())' */
void clear();
```

3.9 Misc functions

The deque class has those functions too :

```
/* Destroys the content of 'this' deque
and replaces it by 'n' copies of
'val' */
void assign(size_type n, const T& val);
/* Destroys the content of 'this' deque
and replaces it by the elements
between 'first' and 'last' */
template <class InputIterator>
void assign(InputIterator first, InputIterator last);

/* Swaps the content of    'this' deque
with the content of 'x' */
void swap(deque<T, A>& x);
/* Swaps the content of 'x' and 'y' */
void swap(deque<T, A>& x, deque<T, A>& y);

/* Returns this list allocator */
allocator_type get_allocator() const;
```

Chapter 4

List : <list>

The list container is a double linked list. For each element of the list, the first link corresponds to the previous element and the second link corresponds to the next element.

The list is circular, thus, the element following the last is the first ; and the element preceding the first is the last.

The insertion and deletion in the list are done in constant time, no matter the position.

In the following sections, we will consider a list constructed with the template $<T, A>$. Where T is the value type of the list and A is the allocator type. The allocator type is optionnal.

4.1 List types

A list container has the following types :

```
/* Type of the elements */
typedef T value_type;
typedef value_type* pointer;
typedef const value_type* const_pointer;
typedef value_type& reference;
typedef const value_type& const_reference;

/* Iterator types are implementation dependant */
typedef List_iterator<T, T&, T*>* iterator;
typedef List_iterator<T, const T&, const T*> const_iterator;
typedef reverse_iterator<const_iterator> const_reverse_iterator;
typedef reverse_iterator<iterator> reverse_iterator;

/* Bidirectionnal iterators */
typedef reverse_bidirectional_iterator<const_iterator, value_type,
                                       const_reference, difference_type>
const_reverse_iterator;
typedef reverse_bidirectional_iterator<iterator, value_type,
                                       reference, difference_type>
reverse_iterator;

/* This is implementation dependant */
typedef size_t size_type;
typedef ptrdiff_t difference_type;
typedef typename List_base::allocator_type allocator_type;
```

4.2 List constructors

A list container has the following constructors :

```
explicit list(const allocator_type& a = allocator_type());

/* Inits the list with 'n' copies of the 'value' */
list(size_type n, const T& value, const allocator_type& a = allocator_type());

/* Allocates 'n' area of memory */
explicit list(size_type n);

/* Copy constructor */
list(const list<T, A>& x);

/* 'first' and 'last' are iterators on a sequence.
The values between 'first and 'last' are copied
in the new list */
template <class InputIterator>
list(InputIterator first, InputIterator last, const allocator_type& a = allocator_type());
```

4.3 List destructor

A list container has the following destructor :

```
~list();
```

4.4 List operators

A list container has the following operators :

```
/* Copy operator */
list<T, A>& operator=(const list<T, A>& l);

/* Compares two lists */
bool operator==(const list<T, A>& x, const list<T, A>& y);
/* Lexicographical compare between lists */
bool operator<(const list<T, A>& x, const list<T, A>& y);
```

4.5 List iterators

A list has the following iterators :

```
/* Returns an iterator on the first element */
iterator begin();
const_iterator begin() const;
```

```
/* Returns an iterator on the element following the last */
iterator end();
const_iterator end() const;

/* Returns an iterator on the last element */
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
/* Returns an iterator on the element preceding the first */
reverse_iterator rend();
const_reverse_iterator rend() const;
```

4.6 List size and capacity

A list size and capacity can be checked and changed using those functions :

```
/* The number of elements */
size_type size() const;
/* The largest possible size
of this list (based on the size
of element's type) */
size_type max_size() const;

/* Returns true if this list contains
no element, in this case 'begin() == end()' */
bool empty() const;

/* If the 'new_size' is less than
the current size it erases element from
the end to set the size to 'new_size'.
If the 'new_size' is greater then
the current size it inserts elements
at the end (initialized with 'x' value). */
void resize(size_type new_size, const T& x);

/* Calls 'resize(size_type, value_type())' */
void resize(size_type new_size);
```

4.7 Accessing elements

Elements of a list can be accessed using those functions :

```
/* Returns the first element */
reference front();
/* Returns the first element */
const_reference front();

/* Returns the last element */
reference back();
/* Returns the last element */
```

```
const_reference back();
```

4.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
/* Appends the element 'x' at the end */
void push_back(const T& x);
/* Appends a new unitialized element at
the end */
void push_back();

/* Appends the element 'x' at the beginning */
void push_front(const value_type& x);
/* Appends a new unitialized element at
the beginning */
void push_front();

/* Inserts 'x' after 'pos' and return
'x' position */
iterator insert(iterator pos, const T& x);
/* Inserts all the elements between 'first'
and 'last' immediatly after 'pos' */
void insert(iterator pos, const_iterator first, const_iterator_last);
/* Inserts 'n' copies of 'x' immediatly after 'pos' */
void insert(iterator pos, size_type n, const T& x);

/* Erases the last element */
void pop_back();
/* Erases the first element */
void pop_front();

/* Erases the element at 'pos'.
Returns the new element at 'pos' */
iterator erase(iterator pos);
/* Erases all elements between 'first'
and 'last'.
Returns 'first' after deletion */
iterator erase(iterator first, iterator last);

/* Equivalent to 'erase(begin(), end())' */
void clear();
```

4.9 Misc functions

The list class has those functions too :

```
/* Destroys the content of 'this' list
and replaces it by 'n' copies of
```

```
'val' */
void assign(size_type n, const T& val);
/* Destroys the content of 'this' list
and replaces it by the elements
between 'first' and 'last' */
template <class InputIterator>
void assign(InputIterator first, InputIterator last);

/* Swaps the content of    'this' list
with the content of 'x' */
void swap(list<T, A>& x);
/* Swaps the content of 'x' and 'y' */
void swap(list<T, A>& x, list<T, A>& y);

/* Returns this list allocator */
allocator_type get_allocator() const;

/* Transfers all elements from 'x'
to 'this' list, placing them at position
'pos' */
void splice(iterator pos, list<T, A>& x);
/* Transfers all elements in a list starting
at position 'i' to 'this' list, starting
at position 'pos'.
'x' is unused in many implementations */
void splice(iterator pos, list<T, A>& x, iterator i);
/* Transfers all elements in a list starting
at position 'first' stopping at 'last',
to 'this' list, starting at position 'pos'.
'x' is unused in many implementations */
void splice(iterator pos, list<T, A>& x, iterator first, iterator last);

/* Removes all elements from 'this' list
of value 'value'. Uses 'operator=='
of type 'T' */
void remove(const T& value);
/* Removes all elements from 'this' list
who gives a positive result when
passed to 'p' */
template<class Pred> void remove_if(Pred p);

/* Deletes all element with the same value
except one (generally the first).
Same elements are recognized using
'operator==' of the type 'T') */
void unique();
/* Uses 'b' instead of 'operator==' */
template<class BinPred> void unique(BinPred b);

/* Merges 'x' with 'this'.
Both lists must be sorted.
This function uses 'operator<'
of type 'T' */
void merge(list<T, A>& x);
```

```
/* Uses 'cmp' instead of
'operator<' */
void merge(list<T, A>& x, cmp);

/* Reverses 'this' list */
void reverse();

/* Sorts 'this' list using 'operator<'
of type 'T' */
void sort();
/* Uses 'cmp' instead of
'operator<' */
void sort(cmp);
```

Chapter 5

SList : <slist>

The slist container corresponds to a single linked list. Each element is linked to the next. Compared to *list*, which has a bidirectionnal iterator, *slist* has a forward iterator.

Thus, if you do not need the fonctionnalities of a double linked list, you should use *slist* instead of *list*.

5.1 How to know if I should use *list* or *slist* ?

The main difference between *list slist* is the time of function like *insert* and *erase*. If you need to work on element that are far from the beginning, all the slist must be crossed before finding the position to put the element at.

Thus, if you need to work on position far from the beginning of the list you should use a *list* container.

5.2 Slist types

A slist container has the following types :

```
/* Type of the elements */
typedef T value_type;
typedef value_type* pointer;
typedef const value_type* const_pointer;
typedef value_type& reference;
typedef const value_type& const_reference;

/* Iterator types are implementation dependant */
typedef Slist_iterator<T, T&, T*>* iterator;
typedef Slist_iterator<T, const T&, const T*> const_iterator;

/* This is implementation dependant */
typedef size_t size_type;
typedef ptrdiff_t difference_type;
typedef typename Slist_base::allocator_type allocator_type;
```

5.3 Slist constructors

A slist container has the following constructors :

```
explicit slist(const allocator_type& a = allocator_type());

/* Inits the slist with 'n' copies of the 'value' */
slist(size_type n, const T& value, const allocator_type& a = allocator_type());

/* Allocates 'n' area of memory */
explicit slist(size_type n);

/* Copy constructor */
slist(const slist<T, A>& x);

/* 'first' and 'last' are iterators on a sequence.
The values between 'first and 'last' are copied
in the new slist */
template <class InputIterator>
slist(InputIterator first, InputIterator last, const allocator_type& a = allocator_type());
```

5.4 Slist destructor

A slist container has the following destructor :

```
~slist();
```

5.5 Slist operators

A slist container has the following operators :

```
/* Copy operator */
slist<T, A>& operator=(const slist<T, A>& l);

/* Compares two slists */
bool operator==(const slist<T, A>& x, const slist<T, A>& y);
/* Lexicographical compare between slists */
bool operator<(const slist<T, A>& x, const slist<T, A>& y);
```

5.6 Slist iterators

A slist has the following iterators :

```
/* Returns an iterator on the first element */
iterator begin();
const_iterator begin() const;
/* Returns an iterator on the element following the last */
iterator end();
const_iterator end() const;
```

5.7 Slist size and capacity

A slist size and capacity can be checked and changed using those functions :

```
/* The number of elements */
size_type size() const;
/* The largest possible size
of this slist (based on the size
of element's type) */
size_type max_size() const;

/* Returns true if this slist contains
no element, in this case 'begin() == end()' */
bool empty() const;

/* If the 'new_size' is less than
the current size it erases element from
the end to set the size to 'new_size'.
If the 'new_size' is greater then
the current size it inserts elements
at the end (initialized with 'x' value). */
void resize(size_type new_size, const T& x);

/* Calls 'resize(size_type, value_type())' */
void resize(size_type new_size);
```

5.8 Accessing elements

Elements of a slist can be accessed using those functions :

```
/* Returns the first element */
reference front();
/* Returns the first element */
const_reference front();

/* Returns an iterator on the element
preceding 'pos'*/
iterator previous(const_iterator pos);
const_iterator previous(const_iterator pos);
```

5.9 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
/* Appends the element 'x' at the beginning */
void push_front(const value_type& x);
/* Appends a new unitialized element at
the beginning */
void push_front();
/* Erases the first element */
void pop_front();

/* Returns an iterator on the element
preceding 'pos' */
iterator previous(const_iterator pos);
const_iterator previous(const_iterator pos);

/* Inserts 'x' after 'pos' and returns
'x' position */
iterator insert_after(iterator pos, const value_type& x);
/* Inserts an unitialized element immediatly after 'pos'
and returns its position */
iterator insert_after(iterator pos);
/* Inserts 'n' copies of 'x' immediatly after 'pos' */
void insert_after(iterator pos, size_type n, const value_type& x);
/* Inserts all the elements between 'first'
and 'last' immediatly after 'pos' */
void insert_after(iterator pos, InIter first, InIter last);

/* Inserts 'x' after 'pos' and returns
'x' position
Calls 'insert_after' */
iterator insert(iterator pos, const T& x);
/* Inserts 'n' copies of 'x' immediatly after 'pos'
Calls 'insert_after' */
void insert(iterator pos, size_type n, const T& x);
/* Inserts an unitialized element immediatly after 'pos'
and returns its position
Calls 'insert_after' */
iterator insert(iterator pos);
/* Inserts all the elements between 'first'
and 'last' immediatly after 'pos'
Calls 'insert_after' */
void insert(iterator pos, InIter first, InIter last);


/* Erases the element after 'pos'.
Returns the an iterator on the
new element at 'pos' */
iterator erase_after(iterator pos);
/* Erases all elements between 'first'
and 'last'.
Returns 'first' after deletion */
iterator erase_after(iterator first, iterator last);

/* Erases the element at 'pos'.
```

```
Returns the new element at 'pos'
Calls 'erase_after' */
iterator erase(iterator pos);
/* Erases all elements between 'first'
and 'last'.
Returns 'first' after deletion
Calls 'erase_after' */
iterator erase(iterator first, iterator last);

/* Equivalent to 'erase(begin(), end())' */
void clear();
```

5.10 Misc functions

The slist class has those functions too :

```
/* Destroys the content of 'this' slist
and replaces it by 'n' copies of
'val' */
void assign(size_type n, const T& val);
/* Destroys the content of 'this' slist
and replaces it by the elements
between 'first' and 'last' */
template <class InputIterator>
void assign(InputIterator first, InputIterator last);

/* Swaps the content of    'this' slist
with the content of 'x' */
void swap(slist<T, A>& x);
/* Swaps the content of 'x' and 'y' */
void swap(slist<T, A>& x, slist<T, A>& y);

/* Returns this slist allocator */
allocator_type get_allocator() const;


/* Transfers all elements in a slist starting
at position 'first + 1' stopping at 'last - 1',
to 'this' slist, immediatly after 'pos' */
void splice_after(iterator pos, iterator before_first, iterator before_last);
/* Moves the element that follows 'prev' to 'this' slist,
inserting it immediately
after 'pos'.    This is done in constant time */
void splice_after(iterator pos, iterator prev);

/* Transfers all elements from 'x'
to 'this' slist, placing them at position
'pos' */
void splice(iterator pos, slist<T, A>& x);
/* Transfers all elements in a slist starting
at position 'i' to 'this' slist, starting
at position 'pos'.
```

```
'x' is unused in many implementations */
void splice(iterator pos, slist<T, A>& x, iterator i);
/* Transfers all elements in a slist starting
at position 'first' stopping at 'last',
to 'this' slist, starting at position 'pos'.
'x' is unused in many implementations */
void splice(iterator pos, slist<T, A>& x, iterator first, iterator last);

/* Removes all elements from 'this' slist
of value 'value'. Uses 'operator=='
of type 'T' */
void remove(const T& value);
/* Removes all elements from 'this' slist
who gives a positive result when
passed to 'p' */
template<class Pred> void remove_if(Pred p);

/* Deletes all element with the same value
except one (generally the first).
Same elements are recognized using
'operator==' of the type 'T') */
void unique();
/* Uses 'b' instead of 'operator==' */
template<class BinPred> void unique(BinPred b);

/* Merges 'x' with 'this'.
Both slists must be sorted.
This function uses 'operator<'
of type 'T' */
void merge(list<T, A>& x);
/* Uses 'cmp' instead of
'operator<' */
void merge(list<T, A>& x, cmp);

/* Reverses 'this' slist */
void reverse();

/* Sorts 'this' list using 'operator<'
of type 'T' */
void sort();
/* Uses 'cmp' instead of
'operator<' */
void sort(cmp);
```

Chapter 6

Vector : <vector>

The vector container is the more generic container. It can be assimilated to an array (it may be used like an array). The main difference is the possibility of the vector to be extended while adding more elements.

The two important thing with a vector is its capacity and its size. The size is the number of elements in the vector. The vector is the number of elements that can be added to a vector before resizing it (to allocate more memory to add the new elements).

The time of insertion and removal of elements at the end of a vector is constant. At the beginning and at any other position the time of insertions and removals are linear.

In the following sections, we will consider a vector constructed with the template <*T*, *A*>. Where *T* is the value type of the vector and *A* is the allocator type. The allocator type is optionnal.

6.1 Vector types

A vector container has the following types :

```
/* Type of the elements */
typedef T value_type;
typedef value_type* pointer;
typedef const value_type* const_pointer;

/* Iterator types are implementation dependant */
typedef value_type* iterator;
typedef const value_type* const_iterator;
typedef value_type& reference;
typedef const value_type& const_reference;
typedef reverse_iterator<const_iterator> const_reverse_iterator;
typedef reverse_iterator<iterator> reverse_iterator;

/* This is implementation dependant */
typedef size_t size_type;
typedef ptrdiff_t difference_type;
typedef typename Vector_base::allocator_type allocator_type;
```

6.2 Vector constructors

A vector container has the following constructors :

```
explicit vector(const allocator_type& a = allocator_type());

/* Inits the vector with 'n' copies of the 'value' */
vector(size_type n, const T& value, const allocator_type& a = allocator_type());

/* Allocates 'n' area of memory */
explicit vector(size_type n);

/* Copy constructor */
vector(const vector<T, A>& x);

/* 'first' and 'last' are iterators on a sequence.
The values between 'first and 'last' are copied
in the new vector */
template <class InputIterator>
vector(InputIterator first, InputIterator last, const allocator_type& a = allocator_type());
```

6.3 Vector destructor

A vector container has the following destructor :

```
~vector();
```

6.4 Vector operators

A vector container has the following operators :

```
/* Unchecked access */
reference operator[](size_type n);
/* Constant access */
const_reference operator[](size_type n) const;

/* Copy operator */
vector<T, A>& operator=(const vector<T, A>& v);

/* Compares two vectors using 'equal' function */
bool operator==(const vector<T, A>& x, const vector<T, A>& y);
/* Lexicographical compare between vectors */
bool operator<(const vector<T, A>& x, const vector<T, A>& y);
```

6.5 Vector iterators

A vector has the following iterators :

```
/* Returns an iterator on the first element */
iterator begin();
const_iterator begin() const;
/* Returns an iterator on the element following the last */
iterator end();
const_iterator end() const;

/* Returns an iterator on the last element */
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
/* Returns an iterator on the element preceding the first */
reverse_iterator rend();
const_reverse_iterator rend() const;
```

6.6 Vector size and capacity

A vector size and capacity can be checked and changed using those functions :

```
/* The number of element */
size_type size() const;
/* The largest possible size
of this vector (based on the size
of element's type) */
size_type max_size() const;

/* Number of allocated elements (initialized
elements plus non initialized) */
size_type capacity() const;

/* Returns true if this vector contains
no element, in this case 'begin() == end()' */
bool empty() const;

/* Tries to allocate more memory for 'n' more
elements. If 'n' is less than or equal to
'capacity()' this call does nothing */
void reserve(size_type n);

/* If the 'new_size' is less than
the current size it erases element from
the end to set the size to 'new_size'.
If the 'new_size' is greater then
the current size it inserts elements
at the end (initialized with 'x' value). */
void resize(size_type new_size, const T& x);

/* Calls 'resize(size_type, value_type())' */
void resize(size_type new_size);
```

6.7 Accessing elements

Elements of a vector can be accessed using those functions :

```
/* Returns the first element */
reference front();
/* Returns the first element */
const_reference front();

/* Returns the last element */
reference back();
/* Returns the last element */
const_reference back();
```

6.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
/* Appends the element 'x' at the end */
void push_back(const T& x);
/* Appends a new unitialized element at
the end */
void push_back();

/* Inserts 'x' after 'pos' and return
'x' position */
iterator insert(iterator pos, const T& x);
/* Adds a new non initialized element */
iterator insert(iterator pos);
/* Inserts all the elements between 'first'
and 'last' immediatly after 'pos' */
void insert(iterator pos, const_iterator first, const_iterator_last);
/* Inserts 'n' copies of 'x' immediatly after 'pos' */
void insert(iterator pos, size_type n, const T& x);

/* Erases the last element */
void pop_back();

/* Erases the element at 'pos'.
Returns the new element at 'pos' */
iterator erase(iterator pos);
/* Erases all elements between 'first'
and 'last'.
Returns 'first' after deletion */
iterator erase(iterator first, iterator last);

/* Equivalent to 'erase(begin(), end())' */
void clear();
```

6.9 Misc functions

The vector class has those functions too :

```
/* Destroys the content of 'this' vector
and replaces it by 'n' copies of
'val' */
void assign(size_type n, const T& val);
/* Destroys the content of 'this' vector
and replaces it by the elements
between 'first' and 'last' */
template <class InputIterator>
void assign(InputIterator first, InputIterator last);

/* Swaps the content of    'this' vector
with the content of 'x' */
void swap(vector<T, A>& x);
/* Swaps the content of 'x' and 'y' */
void swap(vector<T, A>& x, vector<T, A>& y);

/* Returns this vector allocator */
allocator_type get_allocator() const;
```

Chapter 7

Bitset : <bitset>

The bitset container is pretty much like a *vector<bool>*. It is a container optimized to contain bits and to offer constant access to each element.

A bitset does not follow the STL container way. It is a special container, which is between a real container and an unsigned int.

In the following sections we will consider a bitset constructed with the template *<N, WordT>* where :

- *N* is the size of the bitset (must be superior to 0)
- *WordT* is a type used to know the size of a word, it must be an unsigned type (this parameter is optionnal, default is *unsigned long*)

7.1 Bitset types

A bitset has the following types :

```
class reference
{
   public:
      reference& operator=(const reference&);
      reference& operator=(bool b);
      bool operator~() const;
      operator bool() const;
      reference& flip();
};

/* 0 is the least significant word */
WordT M_w[N];
```

7.2 Bitset constructors

A bitset has the following constructors :

```
bitset();
bitset(unsigned long val);
```

```
/* Creates a bitset using characters in a string
(begins at 'pos' and using 'n' as the size
of the element contained in the string) */
template<class CharT, class Traits, class Alloc>
explicit bitset(const basic_string<CharT, Traits, Alloc>& s, size_t pos = 0,
                size_t n = size_t(basic_string<CharT, Traits, Alloc>::npos));
```

7.3 Bitset destructor

A bitset uses the default destructor.

7.4 Bitset operators

A bitset has the following operators :

```
/* Bitwise operators */
bitset<N, WordT>& operator&=(const bitset<N, WordT>& bs);
bitset<N, WordT>& operator|=(const bitset<N, WordT>& bs);
bitset<N, WordT>& operator^=(const bitset<N, WordT>& bs);
bitset<N, WordT>& operator<<=(const bitset<N, WordT>& bs);
bitset<N, WordT>& operator>>=(const bitset<N, WordT>& bs);

template <size_t N, class WordT>
bitset<N, WordT> operator&(const bitset<N, WordT>& x, const bitset<N, WordT>& y);

template <size_t N, class WordT>
bitset<N, WordT> operator|(const bitset<N, WordT>& x, const bitset<N, WordT>& y);

template <size_t N, class WordT>
bitset<N, WordT> operator^(const bitset<N, WordT>& x, const bitset<N, WordT>& y);

bitset<N, WordT> operator<<(size_t pos) const;
bitset<N, WordT> operator>>(size_t pos) const;

bitset<N, WordT> operator~() const;


reference operator[](size_t pos);
bool operator[](size_t pos) const;

bool operator==(const bitset<N, WordT>& bs) const;

template <size_t N, class WordT>
istream& operator>>(istream& is, bitset<N, WordT>& x);
template <size_t N, class WordT>
ostream& operator<<(ostream& os, const bitset<N, WordT>& x);
```

7.5 Bitset iterators

You cannot iterate on a bitset.

7.6 Bitset size and capacity

A bitset size and capacity can be checked and changed using those functions :

```
/* Returns the number of bits that are set */
size_t count() const;

/* Returns the template parameter 'N' */
size_t size() const;
```

7.7 Testing and changing bits

Bits value can be tested and changed using those functions :

```
/* Returns true if bit at 'pos' is set */
bool test(size_t pos) const;
/* Returns true if any bit is set */
bool any() const;
/* Returns true if none of the bit is set */
bool none() const;

/* Sets all bits */
bitset<N, WordT>& set();
/* Set bit at 'pos' */
bitset<N, WordT>& set(size_t pos);
/* Set the bit at 'pos' making it becomes 'value' */
bitset<N, WordT>& set(size_t pos, int val);

/* Unsets all bits */
bitset<N, WordT>& reset();
/* Unset bit at 'pos' */
bitset<N, WordT>& reset(size_t pos);

/* Flips all bits */
bitset<N, WordT>& flip();
/* Flips bit at 'pos' */
bitset<N, WordT>& flip(size_t pos);
```

7.8 Converting a bitset

A bitset can be converted in a string or in an unsigned long using those function :

unsigned long to_ulong() **const**;

template <class CharT, class Traits, class Alloc> basic_string<CharT, Traits, Alloc> to_string() **const**;

Part V

Associative Containers

Chapter 1

What are associative containers ?

Associative containers are containers like sequence container, but each value contained in it is associated with something called a key.

1.1 Introduction

1.2 When use an associative container ?

Chapter 2

Hash : <hashtable.h>

2.1 What is a hash table ?

A hash table (shortly known as hash), is an associative container. It is a container that associates a **value** and a **key**. Each key can have more than one value.

The keys and their values are put into place known as **buckets**. Each bucket corresponds to a unique key.

The difference between a hash table and others associative containers is that you MUST specify the functions used by the hash table on its creation. The advantage of this, is that you can choose yourself the type of the argument and of the returned object of those functions.

The only constraint is that your *operator()* must be *const* because the hash table will pass constant references to it when doing something.

The table is filled by using a **hash function** (called *hf* in constructors). This function returns a position in the table by doing something on the key.

The function **equals** (called *eql* in constructors) is used to test if two keys are equal.

The function *extract key* (called *ext* in constructors) is used to obtain a key by giving it a value.

The prototypes of those functors are choosen on the hash table creation (you put it in the template).

In the following sections, all hash tables will be constructed with the template : *<val, key, hf, ext, eql, a>*. Where :

- *val* is the value type
- *key* is the key type
- *hf* is the hash function
- *ext* is the **extract key** function
- *eql* is the **equals** function
- *a* is the allocator type (optionnal)

2.2 Hash types

A hash container has the following types :

```
typedef Key key_type;
typedef T value_type;
typedef EqualKey key_equal;
```

```
/* Hash function */
typedef HashFcn hasher;

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

2.3 Hash constructors

A hash container has the following constructors :

```
hashtable(size_type n, const HashFcn& hf, const EqualKey& eql, const ExtractKey& ext,
          const allocator_type& a = allocator_type());

hashtable(size_type n, const HashFcn& hf, const EqualKey& eql, const allocator_type& a = allo-
cator_type());

/* Copy constructor */
hashtable(const hashtable& ht);
```

In those constructors *hf* is a hash function, *ext* allow to extract a key by giving it a complete bucket.

2.4 Hash destructor

A hash container has the following destructor :

```
~hashtable();
```

2.5 Hash operators

A hash container has the following operators :

```
/* For each common key check if the values
are equal. Returns false if the number
of key is different, if there is
key which is not in common or if
two values are not equal but have
the same key */
bool operator==(const hashtable<val, key, hf, ext, eql, a>& ht1,
              const hashtable<val, key, hf, ext, eql, a>& ht2);
```

```
/* Copy operator */
hashtable& operator=(const hashtable& ht);
```

2.6 Hash iterators

Hash iterators can be used to move accross the value contained in a hash table. To get the key corresponding to a value, you may use the **extract** function that you used to create your hash table.

```
typedef hashtable_iterator<val, key, hf, ext, eql, a> iterator;
typedef hashtable_const_iterator<val, key, hf, ext, eql, a> const_iterator;

/* Returns an iterator on the first element */
iterator begin();
const_iterator begin();

/* Returns an iterator on the element following the last */
iterator end();
const_iterator end();
```

2.7 Hash size and capacity

A hash size and capacity can be checked and changed using those functions :

```
/* The number of element */
size_type size() const;
/* The largest possible size
of this hash table (based on the size
of element's type) */
size_type max_size() const;

/* Returns true if this hash table contains
no element, in this case 'begin() == end()' */
bool empty() const;

void resize(size_type num_element_hint);
```

2.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
/* If the value corresponding to 'obj' already
exists, returns it.
In other case inserts it then
returns an iterator on it.
The boolean is the pair is false if
```

```
the value existed or true in other case */
pair<iterator, bool> insert_unique(const value_type& obj);
/* Do not resize the hash table (do not
check the hash table size) */
pair<iterator, bool> insert_unique_no_resize(const value_type& obj);
/* Uses 'insert_unique' to insert all element between 'f' and 'l' */
template<class InputIterator> void insert_unique(InputIterator f, InputIterator l);
/* The last argument specifies the iterator category
(not used) */
template<class InputIterator> void insert_unique(InputIterator f, InputIterator l,    input_iterator_tag);
template<class ForwardIterator> void insert_unique(ForwardIterator f, ForwardIterator l,    forward_iterator_tag);

/* If the value corresponding to 'obj' already
exists, add a new value corresponding
to this key using a flavor of linkes list.
In other case inserts it then
returns an iterator on it. */
iterator insert_equal(const value_type& obj);
/* Do not resize the hash table (do not
check the hash table size) */
iterator insert_equal_no_resize(const value_type& obj);
/* Uses 'insert_equal' to insert all element between 'f' and 'l' */
template<class InputIterator> void insert_equal(InputIterator f, InputIterator l);
/* The last argument specifies the iterator category
(not used) */
template<class InputIterator> void insert_equal(InputIterator f, InputIterator l, input_iterator_tag);
template<class ForwardIterator> void insert_equal(ForwardIterator f, ForwardIterator l, forward_iterator_tag);

/* Seeks for a value in 'this'. If founded
returns it. In other case inserts it
then returns it */
reference find_or_insert(const value_type& obj);

/* Erases the bucket corresponding to 'key' */
size_type erase(const_type& key);
/* Erases the value pointed by 'it' */
void erase(const iterator& it);
void erase(const const_iterator& it);
/* Erases all elements between 'first'
and 'last' (in the same bucket) */
void erase(iterator first, iterator last);
void erase(const_iterator first, const_iterator last);

/* Empties the hash table */
void clear();
```

2.9 Misc functions

The hash class has those functions too :

```
/* Swaps the content of    'this' hash table
with the content of 'x' */
```

```
void swap(hashtable<val, key, hf, ext, eql, a>& x);
/* Swaps the content of 'x' and 'y' */
void swap(hashtable<val, key, hf, ext, eql, a>& x, hashtable<val, key, hf, ext, eql, a>& y);

/* Returns this hash table allocator */
allocator_type get_allocator() const;

/* Returns the number of buckets allocated
for this hash table */
size_type bucket_count() const;
/* Returns the maximum number of buckets
that can be allocated */
size_type max_bucket_count() const;
/* Returns the total number of elements
in a bucket */
size_type elems_in_bucket(size_type bucket) const;

/* Returns an iterator on the value
corresponding to'key' */
iterator find(const key_type& key);
const_iterator find(const key_type& key) const;

/* Returns the number of elements in the bucket
corresponding to 'key' */
size_type count(const key_type& key) const;

/* Returns a range of value where all elements
correspond to 'key' */
pair<iterator, iterator> equal_range(const key_type& key);
pair<iterator, iterator> equal_range(const key_type& key) const;

void resize(size_type num_element_hint);
```

2.10 Hash table exemple

Here is an example of how to use a hash table.

```
#include <hashtable.h>
#include <iostream>

class hash_func
{
    public:
        char operator()(const int& value) const
        {
                return static_cast<char>(value);
        }
};

class extractor
```

```
{
  public:
    int operator()(const char& value) const
    {
        return static_cast<int>(value);
    }
};

class equal_key
{
  public:

    bool operator()(const int& key1, const int& key2) const
    {
        return key1 == key2;
    }
};

/* Main program */
int main(int argc, char* argv[])
{
    hash_func hf;
    extractor ex;
    equal_key eq;

    hashtable<char, int, hash_func,
      extractor, equal_key> ht(sizeof(char), hf, eq, ex);

    hashtable<char, int, hash_func,
      extractor, equal_key>::iterator it;

    ht.insert_unique('a');
    ht.insert_unique('b');
    ht.insert_unique('x');

   /* Displays :
        120 : x
        97 : a
        98 : b
    */
    for(it = ht.begin();
        it != ht.end();
        it++)
      {
          cout << ex(*it) << " : ";
          cout << *it << endl;
      }

    return 0;
}

/* — >>> EOF <<< — */
```

Chapter 3

Hash map : <hash_map>

3.1 What is a hash map ?

A hash map is an associative container which contains element of type *pair<const Key, value>*.

Each value is identified by a unique key.

The hash map container use a *hash table*, thus the two container are very identical.

In the following sections we will consider a hash map constructed with the template : *<K, T, hf, eq, a>* where :

- *K* is the key type
- *T* is the value type
- *hf* is the hash function. This argument is optionnal (default is *hash<K>*)
- *eq* is the equal function to compare values. This parameter is optionnal (default is *equal_to<T>*)
- *a* is the hash map allocator. This argument is optionnal (default is the stl default allocator for type *T*)

3.2 Hash map types

A hash map container has the following types :

```
/* This type is private */
typedef hashtable<pair<const K,T>, K, hf, select1st< pair<const K, T> >, eq, a> ht;
/* This member is private */
ht _ht;

/* Type of the keys */
typedef ht::Key key_type;
/* Type of the values */
typedef T data_type;
typedef T mapped_type;
typedef ht::value_type value_type;
typedef ht::hasher hasher;
typedef ht::key_equal key_equal;
typedef ht::size_type size_type;
typedef ht::difference_type difference_type;
typedef ht::pointer pointer;
typedef ht::const_pointer const_pointer;
typedef ht::reference reference;
typedef ht::const_reference const_reference;

typedef ht::iterator iterator;
typedef ht::const_itertator const_iterator;

typedef ht::allocator_type allocator_type;

/* These functions return types
of 'ht' member (because it is
a private member) */
/* Returns the hash function */
hasher hash_funct() const;
/* Returns the equal function */
key_equal key_eq() const;
```

3.3 Hash map constructors

A hash map container has the following constructors :

```
/* Default constructor */
hash_map() : _ht(100, hasher(), key_equal(), allocator_type());

explicit hash_map(size_type n) : _ht(n, hasher(), key_equal(), allocator_type());

hash_map(size_type n, const hasher& hf) : _ht(n, hf, key_equal(), allocator_type());

hash_map(size_type n, const hasher& hf, const key_equal& eql,
          const allocator_type& a = allocator_type()):
_ht(n, hf, eql, a);


/* Inits the hash map with the elements bewteen 'first' and 'last' */
template <class InputIterator> hash_map(InputIterator first, InputIterator last);
template <class InputIterator> hash_map(InputIterator first, InputIterator last, size_type n);
template <class InputIterator> hash_map(InputIterator first, InputIterator last, size_type n,
                                         const hasher& hf);
template <class InputIterator> hash_map(InputIterator first, InputIterator last, size_type n,
                                         const hasher& hf, const key_equal& eql,
                                         const allocator_type& a = allocator_type());
```

3.4 Hash map destructor

A hash map container uses the default destructor (it calls the destructor of *hash*).

3.5 Hash map operators

A hash map container has the following operators :

```
/* Uses 'hash::operator==' to compare 'h1._ht'
and 'h2._ht' */
bool operator==(const hash_map<K, T, hf, eq, a>& h1, const hash_map<K, T, hf, eq, a>& h2);

/* Returns a value by giving its key */
T& operator[](const key_type& key);
```

3.6 Hash Map iterators

Hash map iterators are of type *pair* the first element is the key and the second element is the value. Thus, if you want to get the key and the value contained in an iterator *it* use *it-¿first* and *it-¿second*. Iterators can be obtained using the following functions :

```
iterator begin();
const_iterator begin() const;

iterator end();
const_iterator end() const;
```

3.7 Hash map size and capacity

A hash map size and capacity can be checked and changed using those functions :

```
/* The number of element */
size_type size() const;
/* The largest possible size
of this hash map (based on the size
of element's type) */
size_type max_size() const;

/* Returns true if this hash table contains
no element, in this case 'begin() == end()' */
bool empty() const;

void resize(size_type hint);
```

3.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
/* Inserts a new value in this hash map
and returns a pair containing an iterator
```

```
on the value in first, and a bool in
second. The second value is true if
insertion was successful or false
in other case */
pair<iterator, bool> insert(const value_type& x);
/* Inserts a new value with no size check */
pair<iterator, bool> insert_no_resize(const value_type& obj);

/* Inserts the elements between first and last */
template <class InputIterator> void insert(InputIterator first, InputIterator last);

/* Returns the number of elements deleted (0 or 1) */
size_type erase(const key_type& key);

void erase(iterator it);
void erase(iterator first, iterator last);

void clear();
```

3.9 Misc functions

The hash map class has those functions too :

```
/* Swaps the content of    'this' hash map
with the content of 'x' */
void swap(hash_map<K, T, hf, eq, a>& x);
/* Swaps the content of 'x' and 'y' */
void swap(hash_map<K, T, hf, eq, a>& h1, hash_map<K, T, hf, eq, a>& h2);

/* Returns this hash map allocator */
allocator_type get_allocator() const;

/* Returns the number of buckets allocated
for this hash map */
size_type bucket_count() const;
/* Returns the maximum number of buckets
that can be allocated */
size_type max_bucket_count() const;
/* Returns the total number of elements
in a bucket */
size_type elems_in_bucket(size_type bucket) const;

/* Returns an iterator on the value
corresponding to 'key' */
iterator find(const key_type& key);
const_iterator find(const key_type& key) const;

/* Returns the number of elements in the bucket
corresponding to 'key' */
size_type count(const key_type& key) const;
```

```
/* Returns a range of value where all elements
correspond to 'key' */
pair<iterator, iterator> equal_range(const key_type& key);
pair<const_iterator, const_iterator> equal_range(const key_type& key) const;
```

3.10 Hash map exemple

Here is an example of how to use a hash map.

```
#include <hash_map>
#include <string>

int main(int argc, char* argv[])
{
        hash_map<int, string> h;

        h[1] = "toto";
        h[2] = "tata";
        h[3] = "titi";
        h.insert(pair<int, string>(4, "toto_is_back"));

       /* Displays:
              1 = toto
              2 = tata
              3 = titi
              4 = toto_is_back
        */
        for(hash_map<int, string>::iterator it = h.begin();
               it != h.end();
               it++)
            {
                   cout << it->first << " = " << it->second << endl;
            }

        return 0;
}

/* — >>> EOF <<< — */
```

Chapter 4

Hash multimap : <hash_map>

4.1 What is a hash multimap ?

The hash multimap is a hash map that can contain multiple for a same key.

Most of hash multimap function call the corresponding hash map function (on his private member).

In the following section, we will consider a hash multimap created with the template :
<pair<const K, T>, K, hf, select1st<pair<const K, T>>, eq, A> where :

- *K* is the key type
- *T* is the value type
- *hf* is the hash function
- *eq* is the equal function to compare values. This parameter is optionnal (default is *equal_to<T>*)
- *a* is the hash multimap allocator. This argument is optionnal (default is the stl default allocator for type *T*)

4.2 Hash multimap types

A hash multimap container has the following types :

```
/* This type is private */
typedef hashtable<pair<const K, T>, K, hf, select1st<pair<const K, T> >, eq, A> ht;
/* This member is private. It is used
to define hash multimap types */
ht M_ht;

typedef ht::key_type key_type;
typedef ht::value_type value_type;
typedef T data_type;
typedef T mapped_type;
typedef ht::hasher hasher;
typedef ht::key_equal key_equal;
typedef ht::size_type size_type;
typedef ht::difference_type difference_type;
typedef ht::pointer pointer;
typedef ht::const_pointer const_pointer;
typedef ht::reference reference;
typedef ht::const_reference const_reference;
typedef ht::iterator iterator;
typedef ht::const_iterator const_iterator;
typedef ht::allocator_type allocator_type;
```

4.3 Hash multimap constructors

A hash multimap container has the following constructors :

```
/* Default constructor */
hash_multimap();
explicit hash_multimap(size_type n);
hash_multimap(size_type n, const hasher& hf);
hash_multimap(size_type n, const hasher& hf, const key_equal& eql,
              const allocator_type& a = allocator_type());


/* Inits the hash multimap with the elements bewteen 'first' and 'last' */
template <class InputIterator> hash_multimap(InputIterator first, InputIterator last);
template <class InputIterator> hash_multimap(InputIterator first, InputIterator last, size_type n);
template <class InputIterator> hash_multimap(InputIterator first, InputIterator last, size_type n,
                                             const hasher& hf);
template <class InputIterator> hash_multimap(InputIterator first, InputIterator last, size_type n,
                                             const hasher& hf, const key_equal& eql,
                                             const allocator_type& a = allocator_type());
```

4.4 Hash multimap destructor

A hash multimap container uses the default destructor (it calls the destructor of *hash_map*).

4.5 Hash multimap operators

A hash multimap container has the following operators :

```
bool operator==(const hash_multimap& h1, const hash_multimap& h2);
```

4.6 Hash Multimap iterators

Here is the hash multimap access functions to iterators :

```
iterator begin();
const_iterator begin() const;

iterator end();
```

```
const_iterator end() const;
```

4.7 Hash multimap size and capacity

A hash multimap size and capacity can be checked and changed using those functions :

```
/* The number of element */
size_type size() const;
/* The largest possible size
of this hash multimap (based on the size
of element's type) */
size_type max_size() const;

/* Returns true if this hash multimap contains
no element, in this case 'begin() == end()' */
bool empty() const;

void resize(size_type hint);
```

4.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
/* Inserts the elements between first and last */
template <class InputIterator> void insert(InputIterator first, InputIterator last);

/* Inserts a new '(key, value)' couple
use 'pair' to insert it. */
iterator insert(const value_type& obj);
iterator insert_no_resize(const value_type& obj);

/* Returns the number of elements deleted */
size_type erase(const key_type& key);

void erase(iterator it);
void erase(iterator first, iterator last);

void clear();
```

4.9 Misc functions

The hash multimap class has those functions too :

```
/* Swaps the content of    'this' hash map
with the content of 'x' */
void swap(hash_multimap& x);
/* Swaps the content of 'x' and 'y' */
```

```
void swap(hash_multimap<K, T, hf, eq, a>& h1, hash_map<K, T, hf, eq, a>& h2);

/* Returns this hash multimap allocator */
allocator_type get_allocator() const;

/* Returns the number of buckets allocated
for this hash multimap */
size_type bucket_count() const;
/* Returns the maximum number of buckets
that can be allocated */
size_type max_bucket_count() const;
/* Returns the total number of elements
in a bucket */
size_type elems_in_bucket(size_type bucket) const;

/* Returns an iterator on the value
corresponding to 'key' */
iterator find(const key_type& key);
const_iterator find(const key_type& key) const;

/* Returns the number of elements in the bucket
corresponding to 'key' */
size_type count(const key_type& key) const;

/* Returns a range of value where all elements
correspond to 'key' */
pair<iterator, iterator> equal_range(const key_type& key);
pair<iterator, iterator> equal_range(const key_type& key) const;
```

Chapter 5

Hash set : <hash_set>

5.1 What is a hash set ?

A hash set is a set that uses a hash table to provide faster searching functionnality.

In the following sections we will consider a hash set constructed with the template : *<T, hf, eq, A>* where :

- *T* is the value type (since set has no key value type and key type are the same)
- *hf* is the hash function. This parameter is optionnal (default is *hash<T>*)
- *eq* is the equal function to compare values. This parameter is optionnal (default is *equal_to<T>*)
- *A* is the hash set allocator. This parameter is optionnal (default is the stl default allocator for type *T*)

5.2 Hash set types

A hash set container has the following types :

```
/* This type is private */
typedef hashtable<T, T, hf, identity<T>, eq, A> ht;
/* This member is private */
ht _ht;

/* Type of the keys */
typedef ht::Key key_type;
/* Type of the values */
typedef ht::value_type value_type;
typedef ht::hasher hasher;
typedef ht::key_equal key_equal;
typedef ht::size_type size_type;
typedef ht::difference_type difference_type;

typedef ht::const_pointer pointer;
typedef ht::const_pointer const_pointer;

typedef ht::const_reference reference;
typedef ht::const_reference const_reference;
typedef ht::const_iterator iterator;
typedef ht::const_itertator const_iterator;

typedef ht::allocator_type allocator_type;


/* These functions return types
of 'ht' member (because it is
a private member) */
/* Returns the hash function */
hasher hash_funct() const;
/* Returns the equal function */
key_equal key_eq() const;
```

5.3 Hash set constructors

A hash set container has the following constructors :

```
/* Default constructor */
hash_set() : _ht(100, hasher(), key_equal(), allocator_type());

explicit hash_set(size_type n) : _ht(n, hasher(), key_equal(), allocator_type());

hash_set(size_type n, const hasher& hf) : _ht(n, hf, key_equal(), allocator_type());

hash_set(size_type n, const hasher& hf, const key_equal& eql,
         const allocator_type& a = allocator_type()):
_ht(n, hf, eql, a);


/* Inits the hash set with the elements bewteen 'first' and 'last' */
template <class InputIterator> hash_set(InputIterator first, InputIterator last);
template <class InputIterator> hash_set(InputIterator first, InputIterator last, size_type n);
template <class InputIterator> hash_set(InputIterator first, InputIterator last, size_type n,
                                        const hasher& hf);
template <class InputIterator> hash_set(InputIterator first, InputIterator last, size_type n,
                                        const hasher& hf, const key_equal& eql,
                                        const allocator_type& a = allocator_type());
```

5.4 Hash set destructor

A hash set container uses the default destructor (it calls the destructor of *hash*).

5.5 Hash set operators

A hash set container has the following operators :

```
/* Uses 'hash::operator==' to compare 'h1._ht'
and 'h2._ht' */
bool operator==(const hash_set<T, hf, eq, a>& h1, const hash_set<T, hf, eq, a>& h2);
```

5.6 Hash set iterators

Hash set iterators can be obtained using the following functions :

```
iterator begin() const;
iterator end() const;
```

5.7 Hash set size and capacity

A hash set size and capacity can be checked and changed using those functions :

```
/* The number of element */
size_type size() const;
/* The largest possible size
of this hash set (based on the size
of element's type) */
size_type max_size() const;

/* Returns true if this hash set contains
no element, in this case 'begin() == end()' */
bool empty() const;

void resize(size_type hint);
```

5.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
/* Inserts a new value in this hash set
and returns a pair containing an iterator
on the value in first, and a bool in
second. The second value is true if
insertion was successful or false
in other case */
pair<iterator, bool> insert(const value_type& x);
/* Inserts a new element with no size check */
pair<iterator_bool> insert_no_resize(const value_type& obj);

/* Inserts the elements between first and last */
template <class InputIterator> void insert(InputIterator first, InputIterator last);

/* Returns the number of elements deleted (0 or 1) */
```

```
size_type erase(const key_type& key);

void erase(iterator it);
void erase(iterator first, iterator last);

void clear();
```

5.9 Misc functions

The hash set class has those functions too :

```
/* Swaps the content of    'this' hash set
with the content of 'x' */
void swap(hash_set<T, hf, eq, a>& x);
/* Swaps the content of 'x' and 'y' */
void swap(hash_set<T, hf, eq, a>& h1, hash_set<T, hf, eq, a>& h2);

/* Returns this hash set allocator */
allocator_type get_allocator() const;

/* Returns the number of buckets allocated
for this hash set */
size_type bucket_count() const;
/* Returns the maximum number of buckets
that can be allocated */
size_type max_bucket_count() const;
/* Returns the total number of elements
in a bucket */
size_type elems_in_bucket(size_type bucket) const;

/* Returns an iterator on the value
corresponding to 'key' */
iterator find(const key_type& key) const;

/* Returns the number of elements in the bucket
corresponding to 'key' */
size_type count(const key_type& key) const;

/* Returns a range of value where all elements
correspond to 'key' */
pair<iterator, iterator> equal_range(const key_type& key) const;
```

Chapter 6

Hash multiset : <hash_set>

6.1 What is a hash multiset ?

The hash multiset is a hash set that can contain multiple identical values.

In the following section, we will consider a hash multiset created with the template <*T, hf, eq, A*> where :

- *T* is the value type (since set has no key value type and key type are the same)
- *hf* is the hash function. This parameter is optionnal (default is *hash<T>*)
- *eq* is the equal function to compare values. This parameter is optionnal (default is *equal_to<T>*)
- *A* is the hash multiset allocator. This parameter is optionnal (default is the stl default allocator for type *T*)

6.2 Hash multiset types

A hash multiset container has the following types :

```
/* This type is private */
typedef hashtable<T, T, hf, identity<T>, eq, A> ht;
/* This member is private */
ht _ht;

/* Type of the keys */
typedef ht::Key key_type;
/* Type of the values */
typedef ht::value_type value_type;
typedef ht::hasher hasher;
typedef ht::key_equal key_equal;
typedef ht::size_type size_type;
typedef ht::difference_type difference_type;

typedef ht::const_pointer pointer;
typedef ht::const_pointer const_pointer;

typedef ht::const_reference reference;
typedef ht::const_reference const_reference;
typedef ht::const_iterator iterator;
typedef ht::const_itertator const_iterator;

typedef ht::allocator_type allocator_type;


/* These functions return types
of 'ht' member (because it is
a private member) */
/* Returns the hash function */
hasher hash_funct() const;
/* Returns the equal function */
key_equal key_eq() const;
```

6.3 Hash multiset constructors

A hash multiset container has the following constructors :

```
/* Default constructor */
hash_multiset() : _ht(100, hasher(), key_equal(), allocator_type());

explicit hash_multiset(size_type n) : _ht(n, hasher(), key_equal(), allocator_type());

hash_multiset(size_type n, const hasher& hf) : _ht(n, hf, key_equal(), allocator_type());

hash_multiset(size_type n, const hasher& hf, const key_equal& eql,
              const allocator_type& a = allocator_type()):
_ht(n, hf, eql, a);


/* Inits the hash multiset with the elements bewteen 'first' and 'last' */
template <class InputIterator> hash_multiset(InputIterator first, InputIterator last);
template <class InputIterator> hash_multiset(InputIterator first, InputIterator last, size_type n);
template <class InputIterator> hash_multiset(InputIterator first, InputIterator last, size_type n,
                                             const hasher& hf);
template <class InputIterator> hash_multiset(InputIterator first, InputIterator last, size_type n,
                                             const hasher& hf, const key_equal& eql,
                                             const allocator_type& a = allocator_type());
```

6.4 Hash multiset destructor

A hash multiset container uses the default destructor (it calls the destructor of *hash*).

6.5 Hash multiset operators

A hash multiset container has the following operators :

```
/* Uses 'hash::operator==' to compare 'h1._ht'
and 'h2._ht' */
bool operator==(const hash_multiset<T, hf, eq, a>& h1, const hash_multiset<T, hf, eq, a>& h2);
```

6.6 Hash multiset iterators

Hash multiset iterators can be obtained using the following functions :

```
iterator begin() const;
iterator end() const;
```

6.7 Hash multiset size and capacity

A hash multiset size and capacity can be checked and changed using those functions :

```
/* The number of element */
size_type size() const;
/* The largest possible size
of this hash multiset (based on the size
of element's type) */
size_type max_size() const;

/* Returns true if this hash multiset contains
no element, in this case 'begin() == end()' */
bool empty() const;

void resize(size_type hint);
```

6.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
/* Inserts a new value in this hash multiset
and returns an iterator on it */
iterator insert(const value_type& x);
/* Inserts a new element with no size check */
iterator insert_no_resize(const value_type& obj);

/* Inserts the elements between first and last */
template <class InputIterator> void insert(InputIterator first, InputIterator last);

/* Returns the number of elements deleted (0 or 1) */
size_type erase(const key_type& key);

void erase(iterator it);
void erase(iterator first, iterator last);
void clear();
```

6.9 Misc functions

The hash multiset class has those functions too :

```
/* Swaps the content of    'this' hash multiset
with the content of 'x' */
void swap(hash_multiset<T, hf, eq, a>& x);
/* Swaps the content of 'x' and 'y' */
void swap(hash_multiset<T, hf, eq, a>& h1, hash_multiset<T, hf, eq, a>& h2);

/* Returns this hash multiset allocator */
allocator_type get_allocator() const;

/* Returns the number of buckets allocated
for this hash multiset */
size_type bucket_count() const;
/* Returns the maximum number of buckets
that can be allocated */
size_type max_bucket_count() const;
/* Returns the total number of elements
in a bucket */
size_type elems_in_bucket(size_type bucket) const;

/* Returns an iterator on the value
corresponding to 'key' */
iterator find(const key_type& key) const;

/* Returns the number of elements in the bucket
corresponding to 'key' */
size_type count(const key_type& key) const;

/* Returns a range of value where all elements
correspond to 'key' */
pair<iterator, iterator> equal_range(const key_type& key) const;
```

Chapter 7

Map : <map>

7.1 What is a map ?

A map is a container that associates a key and a value. All key are unique : two elements cannot have different keys.

The element type is *pair<const K, T>* where *K* is the key type and *T* the data type.

In the following sections we will consider a map constructed with the template <*K, T, Compare, A*> where :

- *K* is the key type
- *T* is the data type
- *Compare* is the comparator function. This parameter is optionnal (default is *less<K>*
- *A* is the allocator. This parameter is optionnal (default is the stl default allocator for type *T*)

7.2 Map types

A map container has the following types :

```
typedef K key_type;
typedef T data_type;
typedef T mapped_type;
typedef pair<const K, T> value_type;
typedef Compare key_compare;

/* Public internal class, used to compare
two values by overloading 'operator()'
*/
class value_compare :
public binary_function<value_type, value_type, bool>;

bool value_compare::operator()(const value_type& x, const value_type& y);

/* This type is private. It is used to store values */
typedef Rb_tree<key_type, value_type, select1st<value_type>, key_compare, A> Rep_type;
/* This member is private */
Rep_type M_t;
```

```
typedef   Rep_type::pointer pointer;
typedef   Rep_type::const_pointer const_pointer;
typedef   Rep_type::reference reference;
typedef   Rep_type::const_reference const_reference;
typedef   Rep_type::iterator iterator;
typedef   Rep_type::const_iterator const_iterator;
typedef   Rep_type::reverse_iterator reverse_iterator;
typedef   Rep_type::const_reverse_iterator const_reverse_iterator;
typedef   Rep_type::size_type size_type;
typedef   Rep_type::difference_type difference_type;
typedef   Rep_type::allocator_type allocator_type;
```

7.3 Map constructors

A map container has the following constructors :

```
map();
explicit map(const Compare& comp, const allocator_type& a = allocator_type());

template<class InputIterator> map(InputIterator first, InputIterator last);
template<class InputIterator> map(InputIterator first, InputIterator last,
                                  const Compare& comp, const allocator_type& a = allocator_type());

/* Copy constructor */
map(const map<K, T, Compare, A>& x);
```

7.4 Map destructor

A map container uses the default destructor (it calls the destructor of *rb_tree*).

7.5 Map operators

A map container has the following operators :

```
map<K, T, Compare, A>& operator=(const map<K, T, Compare, A>& x);
T& operator[](const key_type& k);
bool operator==(const map<K, T, Compare, A>& m1, const map<K, T, Compare, A>& m2);
bool operator<(const map<K, T, Compare, A>& m1, const map<K, T, Compare, A>& m2);
```

7.6 Map iterators

Map iterators can be obtained using the following functions :

```
/* Returns an iterator on the first element */
iterator begin();
```

```
const_iterator begin() const;
/* Returns an iterator on the element following the last */
iterator end();
const_iterator end() const;

/* Returns an iterator on the last element */
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
/* Returns an iterator on the element preceding the first */
reverse_iterator rend();
const_reverse_iterator rend() const;
```

7.7 Map size and capacity

A map size and capacity can be checked and changed using those functions :

```
/* The number of element */
size_type size() const;
/* The largest possible size
of this map (based on the size
of element's type) */
size_type max_size() const;

/* Returns true if this map contains
no element, in this case 'begin() == end()' */
bool empty() const;
```

7.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
pair<iterator,bool> insert(const value_type& x)
iterator insert(iterator position, const value_type& x);

/* Inserts all element between 'first' and 'last' */
template <class InputIterator> void insert(InputIterator first,    InputIterator last);

void erase(iterator position);
/* Returns the number of deleted
element (0 or 1) */
size_type erase(const key_type& x);
void erase(iterator first, iterator last);

void clear();
```

7.9 Misc functions

The map class has those functions too :

```
/* Returns an iterator on 'x' */
iterator find(const key_type& x);
const_iterator find(const key_type& x);

/* Counts the number of 'x' in this
map (0 or 1)    */
size_type count(const key_type& x) const;

/* Returns an iterator on the first
element whose key is not less than 'x' */
iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
/* Returns an iterator on the first element
whose key is greater the 'x' */
iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;

/* Returns a pair composed by
the 'lower_bound()' and the 'upper_bound()' */
pair<iterator, iterator> equal_range(const key_type& x);
pair<const_iterator, const_iterator> equal_range(const key_type& x) const;

/* Swaps the content of 'x' and 'y' */
template <class K, class T, class compare, class A>
inline void swap(map<K, T, compare, A>& x, map<K, T, compare, A>& y);
/* Swaps the content of    'this' map
with the content of 'x' */
void swap(map<K, T, compare, A>& x);

/* Returns this map allocator */
allocator_type get_allocator() const;

/* Returns this map key or value comparator */
key_compare key_comp() const;
value_compare value_comp() const;
```

Chapter 8

Multimap : <multimap.h>

8.1 What is a multimap ?

A multimap is very similar to a map. The difference is that you can store multiple values for a same key.

In the following sections we will consider a multimap constructed with the template : *<K, T, compare, A>* where

- *K* is the key type
- *T* is the value type
- *compare* is a function used to compare keys. This parameter is optionnal (default is *less<K>*).
- *A* is the multimap allocator. This argument is optionnal (default is the stl default allocator for type *T*)

Like the *set* container, multimap uses a **red-black tree**.
It uses the bucket mechanism like the *set* container.

8.2 Multimap types

A multimap container has the following types :

```
typedef K key_type;
typedef T data_type;
typedef T mapped_type;
typedef pair<const K, T> value_type;
typedef compare key_compare;


/* Hash multimap contains this
utility class */
class value_compare : public binary_function<value_type, value_type, bool>;
bool value_compare::operator()(const value_type& x, const value_type& y) const;

/* This type is private */
typedef rb_tree<key_type, value_type, select1st<value_type>, key_compare, A> rep_type;
/* This member is private */
rep_type m_t;
```

```
/* Public types are defined using 'rep_type' */
typedef rep_type::pointer pointer;
typedef rep_type::const_pointer const_pointer;
typedef rep_type::reference reference;
typedef rep_type::const_reference const_reference;
typedef rep_type::iterator iterator;
typedef rep_type::const_iterator const_iterator;
typedef rep_type::reverser_iterator reverse_iterator;
typedef rep_type::const_reverse_iterator const_reverse_iterator;
typedef rep_type::size_type size_type;
typedef rep_type::difference_type difference_type;
typedef rep_type::allocator_type allocator_type;
```

8.3 Multimap constructors

A multimap container has the following constructors :

```
multimap();
explicit multimap(const compare& comp, const allocator_type& a = allocator_type());

template <class InputIterator>
multimap(InputIterator first, InputIterator last);
template <class InputIterator>
multimap(InputIterator first, InputIterator last, const compare& comp,
          const allocator_type& a = allocator_type());

/* Copy constructor */
multimap(const multimap<K, T, compare, A>& x);
```

8.4 Multimap destructor

A multimap container uses the default destructor (it calls the destructor of *rb_tree*).

8.5 Multimap operators

A multimap container has the following operators :

```
multimap<K, T, compare, A>& operator=(const multimap<K, T, compare, A>& x);

bool operator==(const multimap<K, T, compare, A>& m1, multimap<K, T, compare, A>& m2);

/* Uses 'rb_tree::operator<' */
bool operator<(const multimap<K, T, compare, A>& m1, multimap<K, T, compare, A>& m2);
```

8.6 Multimap iterators

Multimap iterators use the type 'rb_tree::iterator' :

```
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
```

8.7 Multimap size and capacity

A multimap size and capacity can be checked and changed using those functions :

```
/* The number of element */
size_type size() const;
/* The largest possible size
of this hash multimap (based on the size
of element's type) */
size_type max_size() const;

/* Returns true if this hash multimap contains
no element, in this case 'begin() == end()' */
bool empty() const;
```

8.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
/* Inserts the elements between first and last */
template <class InputIterator> void insert(InputIterator first, InputIterator last);

/* Inserts a new '(key, value)' couple
use 'pair' to insert it. */
iterator insert(const value_type& x);
iterator insert(iterator position, const value_type& x);

/* Returns the number of elements deleted */
size_type erase(const key_type& key);

void erase(iterator pos);
void erase(iterator first, iterator last);

void clear();
```

8.9 Misc functions

The multimap class has those functions too :

```
/* Returns an iterator on 'x' */
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;

/* Counts the number of 'x' in this
multimap (0 or 1)    */
size_type count(const key_type& x) const;

/* Returns an iterator on the first
element whose key is not less than 'x' */
iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
/* Returns an iterator on the first element
whose key is greater the 'x' */
iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;

/* Returns a pair composed by
the 'lower_bound()' and the 'upper_bound()' */
pair<iterator, iterator> equal_range(const key_type& x);
pair<const_iterator, const_iterator> equal_range(const key_type& x) const;

/* Swaps the content of 'x' and 'y' */
template <class K, class T, class compare, class A>
inline void swap(multimap<K, T, compare, A>& x, multimap<K, T, compare, A>& y);
/* Swaps the content of    'this' multimap
with the content of 'x' */
void swap(multimap<K, T, compare, A>& x);

/* Returns this multimap allocator */
allocator_type get_allocator() const;

/* Returns this multimap key or value comparator */
key_compare key_comp() const;
value_compare value_comp() const;
```

8.10 Multimap exemple

Here is an example of how to use a multimap.

```
#include <multimap.h>
#include <string>

int main(int argc, char* argv[])
{
        multimap<int, char> h;
        h.insert(pair<int, char>(1, 't'));
        h.insert(pair<int, char>(1, 'o'));
        h.insert(pair<int, char>(1, 'i'));
        h.insert(pair<int, char>(2, 'b'));
        h.insert(pair<int, char>(5, 'c'));


       /* Displays:
               1 = t
               1 = o
               1 = i
               2 = b
               5 = c
        */
        for(multimap<int, char>::iterator it = h.begin();
                it != h.end();
                it++)
            {
                    cout << it->first << " = " << it->second << endl;
            }

        return 0;
}

/* — >>> EOF <<< — */
```

Chapter 9

Multiset : <set>

9.1 What is a multiset ?

A multiset is very similar to a set. The difference is that you can store multiple identical values.

In the following sections we will consider a multiset constructed with the template : *<K, Compare, A>* where

- *K* is the key type and the value type
- *Compare* is a function used to compare keys. This parameter is optionnal (default is *less<K>*).
- *A* is the multiset allocator. This argument is optionnal (default is the stl default allocator for type *T*)

Like the *set* container, multiset uses a **red-black tree**.

9.2 Multiset types

A multiset container has the following types :

```
typedef K key_type;
typedef K value_type;
typedef Compare key_compare;
typedef Compare value_compare;

/* This type is private */
typedef Rb_tree<key_type, value_type, Identity<value_type>, key_compare, A> Rep_type;
/* This member is private */
Rep_type M_t;

/* Public types defined using 'M_t' types */
typedef Rep_type::const_pointer pointer;
typedef Rep_type::const_pointer const_pointer;
typedef Rep_type::const_reference reference;
typedef Rep_type::const_reference const_reference;
typedef Rep_type::const_iterator iterator;
typedef Rep_type::const_iterator const_iterator;
typedef Rep_type::const_reverse_iterator reverse_iterator;
typedef Rep_type::const_reverse_iterator const_reverse_iterator;
typedef Rep_type::size_type size_type;
typedef Rep_type::difference_type difference_type;
typedef Rep_type::allocator_type allocator_type;
```

9.3 Multiset constructors

A multiset container has the following constructors :

```
multiset();
explicit multiset(const Compare& comp, const allocator_type& a = allocator_type());
template <class InputIterator> multiset(InputIterator first, InputIterator last);
template <class InputIterator> multiset(InputIterator first, InputIterator last, const Compare& comp,
                                        const allocator_type& a = allocator_type());

/* Copy constructor */
multiset(const multiset<K, Compare, A>& x);
```

9.4 Multiset destructor

A multiset container uses the default destructor (it calls the destructor of *rb_tree*).

9.5 Multiset operators

A multiset container has the following operators :

```
multiset<K, Compare, A>& operator=(const multiset<K, Compare, A>& x);

bool operator==(const multiset<K, Compare, A>& m1, const multiset<K, Compare, A>& m2);
bool operator<(const multiset<K, Compare, A>& m1, const multiset<K, Compare, A>& m2);
```

9.6 Multiset iterators

Multiset iterators use the type 'rb_tree::iterator' :

```
iterator begin() const;
iterator end() const;

reverse_iterator rbegin() const;
reverse_iterator rend() const;
```

9.7 Multiset size and capacity

A multiset size and capacity can be checked and changed using those functions :

```
/* The number of element */
size_type size() const;
/* The largest possible size
of this multiset (based on the size
of element's type) */
size_type max_size() const;

/* Returns true if this multiset contains
no element, in this case 'begin() == end()' */
bool empty() const;
```

9.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
/* Inserts the elements between first and last */
template <class InputIterator> void insert(InputIterator first, InputIterator last);

iterator insert(const value_type& x);
iterator insert(iterator position, const value_type& x);

/* Returns the number of elements deleted */
size_type erase(const key_type& key);

void erase(iterator pos);
void erase(iterator first, iterator last);

void clear();
```

9.9 Misc functions

The multiset class has those functions too :

```
/* Returns an iterator on 'x' */
iterator find(const key_type& x);

/* Counts the number of 'x' in this
multiset (0 or 1)    */
size_type count(const key_type& x) const;

/* Returns an iterator on the first
element whose key is not less than 'x' */
iterator lower_bound(const key_type& x) const;
/* Returns an iterator on the first element
whose key is greater the 'x' */
iterator upper_bound(const key_type& x) const;

/* Returns a pair composed by
the 'lower_bound()' and the 'upper_bound()' */
```

```
pair<iterator, iterator> equal_range(const key_type& x) const;

/* Swaps the content of 'x' and 'y' */
template <class K, class Compare, class A>
inline void swap(multiset<K, Compare, A>& x, multiset<K, Compare, A>& y);
/* Swaps the content of    'this' multiset
with the content of 'x' */
void swap(multiset<T, compare, A>& x);

/* Returns this multiset allocator */
allocator_type get_allocator() const;

/* Returns this multiset key or value comparator */
key_compare key_comp() const;
value_compare value_comp() const;
```

Chapter 10

Set : <set>

10.1 The set container

The set container corresponds to the mathematical definition of a set of elements : each element is unique in a set.

When adding or deleting an element, iterators on the set are not invalidated (unless the iterators were pointing on the deleted element).

In the following sections we will consider all sets constructed with the template : <*Key, Compare, Alloc*> where *Key* is the key type, *Compare* is the function used to compare two keys and *Alloc* is the set allocator.

The allocator is optionnal.

The set is implemented using a **red-black tree**, thus many set types depending on how this tree is created.

10.2 Set types

A set container has the following types :

```
typedef Key key_type;
typedef Key value_type;
typedef Compare key_compare;
typedef Compare value_compare;

/* This type is private */
typedef rb_tree<key_type, value_type, Identity<value_type>, key_compare, A> Rep_type;
/* This member is private, it
is the true set content */
Rep_type _M_t;

/* Set types are defined using Rb_tree types (contained in 'tree.h') */
typedef typename Rep_type::const_pointer pointer;
typedef typename Rep_type::const_pointer const_pointer;
typedef typename Rep_type::const_reference reference;
typedef typename Rep_type::const_reference const_reference;
typedef typename Rep_type::const_iterator iterator;
typedef typename Rep_type::const_iterator const_iterator;
typedef typename Rep_type::const_reverse_iterator iterator;
typedef typename Rep_type::const_reverse_iterator const_iterator;
typedef typename Rep_type::size_type size_type;
typedef typename Rep_type::difference_type difference_type;
typedef typename Rep_type::allocator_type allocator_type;
```

10.3 Set constructors

A set container has the following constructors :

```
/* Calls the default constructor of
'Compare' and 'allocator_type' */
set();

explicit set(const Compare& comp, const allocator_type& a = allocator_type());

/* Fills the set with elements between 'first'
and 'last' */
template<class InputIterator>
set(InputIterator first, InputIterator last);
template<class InputIterator>
set(InputIterator first, InputIterator last,
const Compare& comp, const allocator_type& a = allocator_type());

/* Copy constructor */
set(const set<Key, Compare, Alloc>& x);
```

10.4 Set destructor

A set container uses the default destructor (it calls the destructor of *rb_tree*).

10.5 Set operators

A set container has the following operators :

```
/* Copy operator */
set<Key, Compare, Alloc>& operator=(const set<Key, Compare, Alloc>& x);

/* Calls 'rb_tree' operators */
friend bool operator==(const set& s1, const set& s2);
friend bool operator<(const set& s1, const set& s2);
```

10.6 Set iterators

A set container has the following iterators :

```
/* Returns an iterator on the first element */
const_iterator begin() const;
```

```
/* Returns an iterator on the element following the last */
const_iterator end() const;

/* Returns an iterator on the last element */
const_reverse_iterator rbegin() const;
/* Returns an iterator on the element preceding the first */
const_reverse_iterator rend() const;
```

10.7 Set size and capacity

A set size and capacity can be checked and changed using those functions :

```
/* The number of element */
size_type size() const;
/* The largest possible size
of this set (based on the size
of element's type) */
size_type max_size() const;


/* Returns true if this set contains
no element, in this case 'begin() == end()' */
bool empty() const;
```

10.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
/* Returns a pair composed with an iterator
on the new element and a bool.
The bool is 'true' if the element was
already in the set or false in other case */
pair<iterator, bool> insert(const value_type& x);

/* Returns an iterator on the new element */
iterator insert(iterator pos, const value_type& x);
/* Inserts the element between 'first' and 'last' */
template <class InputIterator> void insert(InputIterator first, InputIterator last);

/* Erases the element at 'pos' */
void erase(iterator pos);
/* Returns the number of element deleted
(1 if there was an element 'x'
or 0 in other case) */
size_type erase(const key_type& x);
/* Erases the element between 'first'
and 'last' */
void erase(iterator first, iterator last);
```

```
/* Empties the set */
void clear();
```

10.9 Misc functions

The set class has those functions too :

```
/* Returns an iterator on 'x' */
iterator find(const key_type& x) const;

/* Counts the number of 'x' in this
set (0 or 1) */
size_type count(const key_type& x) const;

/* Returns an iterator on the first
element whose key is not less than 'x' */
iterator lower_bound(const key_type& x) const;
/* Returns an iterator on the first element
whose key is greater the 'x' */
iterator upper_bound(const key_type& x) const;

/* Returns a pair composed by
the 'lower_bound()' and the 'upper_bound()' */
pair<iterator, iterator> equal_range(const key_type& x) const;

/* Swaps the content of    'this' set
with the content of 'x' */
void swap(set<Key, Compare, Alloc>& x);

/* Returns this set allocator */
allocator_type get_allocator() const;

/* Returns this set key or value comparator */
key_compare key_comp() const;
value_compare value_comp() const;
```

10.10 Set table exemple

Here is an example of how to use a set table.

```
#include <set>

int main(int argc, char* argv[])
{
        set<int> a_set;

        a_set.insert(1);
        a_set.insert(2);
        a_set.insert(1);
```

```
    /* Displays :
            1
            2 */
    for(set<int>::iterator it = a_set.begin();
            it != a_set.end(); it++)
    {
            cout << *it << endl;
    }

    cout << a_set.erase(1) << endl;

    /* Displays :
            2 */
    for(set<int>::iterator it = a_set.begin();
            it != a_set.end(); it++)
    {
            cout << *it << endl;
    }

    return 0;
}
/* — >>> EOF <<< — */
```

Part VI

Container adaptors

Chapter 1

Priority queue : <queue>

The priority queue container is a queue that adapts a *vector* container. It is a *first in, first out* container but the elements are sorted using a compare predicate (given in third parameter of the template).

In the following sections we will consider a priority queue constructed with the template <*T, Sequence, Compare*> where :

- *T* is the element type
- *Sequence* is the sequence adapted by the priority queue (this parameter is optionnal, by default *vector*)
- *Compare* is used to make the comparisons between elements (this parameter is optionnal, by default *less<Sequence::value_type>*

Like the queue container, priority queue has no equality operator.

1.1 Priority queue types

A priority queue container has the following types :

```
typedef Sequence::value_type value_type;
typedef Sequence::size_type size_type;
typedef Sequence container_type;

typedef Sequence::reference reference;
typedef Sequence::const_reference const_reference;
```

1.2 Priority queue constructors

A priority queue container has the following constructors :

```
priority_queue();
explicit priority_queue(const Compare& x);
priority_queue(const Compare& x, const Sequence& s);
```

```
/* Inits the priority queue with elements
between 'first' and 'last' */
template <class InputIterator>
priority_queue(InputIterator first, InputIterator last);
template <class InputIterator>
priority_queue(InputIterator first, InputIterator last, const Compare& x);
template <class _InputIterator>
priority_queue(InputIterator first, InputIterator last, const Compare& x, const Sequence& s);
```

1.3 Priority queue destructor

A priority queue container uses the default destructor (which calls the destructor of the *Sequence* member).

1.4 Priority queue operators

A priority queue has no comparison operator.

1.5 Priority queue iterators

You cannot iterate on a priority queue container.

1.6 Priority queue size and capacity

A priority queue size and capacity can be checked and changed using those functions :

```
bool empty() const;
size_type size() const;
```

1.7 Accessing elements

Elements of a priority queue can be accessed using this function :

```
const_reference top() const;
```

1.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
void push(const value_type& x);
void pop();
```

Chapter 2

Queue : <queue>

The queue container is a container adaptor. By default, it uses the *deque* container, but you can build it with any container that provides the same interface.

A queue is a *first in first out* container. Last element added is the first to be removed.

In the following sections we will consider a queue constructed with the template <*T, Sequence*> where :

- *T* is the element type
- *Sequence* is the sequence adapted by the queue (this parameter is optionnal, by default *deque*)

The queue operator has no equality operator.

2.1 Queue types

A queue container has the following types :

```
typedef Sequence::value_type value_type;
typedef Sequence::size_type size_type;
typedef Sequence container_type;

typedef Sequence::reference reference;
typedef Sequence::const_reference const_reference;
```

2.2 Queue constructors

A queue container has the following constructors :

```
queue();

/* Inits the adapted container
with the content of 'c' */
explicit queue(const Sequence& c);
```

2.3 Queue destructor

A queue container uses the default destructor (which calls the destructor of the *Sequence* member).

2.4 Queue operators

A queue container has the following operators :

```
/* Those functions use the 'operator==' of the 'Sequence' type */
template <class T, class Sequence>
bool operator==(const queue<T, Sequence>& x, const queue<T, Sequence>& y);
template <class T, class Sequence>
bool operator!=(const queue<T, Sequence>& x, const queue<T, Sequence>& y);

/* Those functions use the 'operator<' of the 'Sequence' type */
template <class T, class Sequence>
bool operator<(const queue<T, Sequence>& x, const queue<T, Sequence>& y);
template <class T, class Sequence>
bool operator>(const queue<T, Sequence>& x, const queue<T, Sequence>& y);
template <class T, class Sequence>
bool operator<=(const queue<T, Sequence>& x, const queue<T, Sequence>& y);
template <class T, class Sequence>
bool operator>=(const queue<T, Sequence>& x, const queue<T, Sequence>& y);
```

2.5 Queue iterators

You cannot iterate on a queue container.

2.6 Queue size and capacity

A queue size and capacity can be checked and changed using those functions :

```
bool empty() const;
size_type size() const;
```

2.7 Accessing elements

Elements of a queue can be accessed using those functions :

```
reference front();
const_reference front() const;

reference back();
```

```
const_reference back() const;
```

2.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
void push(const value_type& x);
void pop();
```

Chapter 3

Stack

The stack container is a container adaptor. By default, it uses the *deque* container, but you can build it with any container that provides the same interface.

A stack is a first in, last out container. The last element addded is the first to be removed.

In the following sections we will consider a stack constructed with the template <*T, Sequence*> where :

- *T* is the element type
- *Sequence* is the sequence adapted by the stack (this parameter is optionnal, by default *deque*)

Like the queue container, stack has no equality operator.

3.1 Stack types

A stack container has the following types :

```
typedef Sequence::value_type value_type;
typedef Sequence::size_type size_type;
typedef Sequence container_type;

typedef Sequence::reference reference;
typedef Sequence::const_reference const_reference;
```

3.2 Stack constructors

A stack container has the following constructors :

```
stack();
explicit stack(const Sequence& s);
```

3.3 Stack destructor

A stack container uses the default destructor (which calls the destructor of the *Sequence* member).

3.4 Stack operators

A stack container has the following operators :

```
template <class T, class Sequence>
bool operator==(const stack<T, Sequence>& x, const stack<T, Sequence>& y);
template <class T, class Sequence>
bool operator<(const stack<T, Sequence>& x, const stack<T, Sequence>& y);

/* Calls 'operator==' */
template <class T, class Sequence>
bool operator!=(const stack<T, Sequence>& x, const stack<T, Sequence>& y);

/* Call 'operator<' */
template <class T, class Sequence>
bool operator>(const stack<T, Sequence>& x, const stack<T, Sequence>& y);
template <class T, class Sequence>
bool operator>=(const stack<T, Sequence>& x, const stack<T, Sequence>& y);
template <class T, class Sequence>
bool operator<=(const stack<T, Sequence>& x, const stack<T, Sequence>& y);
```

3.5 Stack iterators

You cannot iterate on a stack container.

3.6 Stack size and capacity

A stack size and capacity can be checked and changed using those functions :

```
bool empty() const;
size_type size();
```

3.7 Accessing elements

Elements of a stack can be accessed using those functions :

```
reference top();
const_reference top() const;
```

3.8 Inserting and deleting elements

Insertion and deletion are provided by those functions :

```
void push(const value_type& x);
void pop();
```

Part VII

Functors

Chapter 1

Introduction on functors

A functor (also called a function object), is an object that can be called as if it was a function. Generally speaking, a functor is an object that defines *operator()*. But it can also be a normal function or even a function pointer (called respectively *Generator* and *Unary function* or *Binary function*).

The STL uses only three types of functors, depending on the number of argument used to call them : no argument, one or two.

An *Adaptable function* is an object that can be used as a function but is, in fact, an object, with *operator()* overloaded and nested argument and return types.

Functors that return a boolean value are called *Predicate*. They are divided into two types of predicate : *Predicate* and *Binary predicate*. The first is an unary functor and the second is a binary functor.

On the following sections we will describe the functors provided by the C++ STL. All STL functors can be used including the *functionnal* header.

Chapter 2

Unary function

unary_function is the type inherited by all unary functors :

```
template <class Arg, class Result>
struct unary_function
{
        typedef Arg argument_type;
        typedef Result result_type;
};
```

Chapter 3

logical_not

logical_not is a unary function. It takes an object of type T and returns the value of a logical not on that object (using *T::operator!*).

```
template <class T>
struct logical_not : public unary_function<T, bool>
{
        bool operator()(const T& x) const;
};
```

Chapter 4

Binary function

binary_function is the type inherited by all binary functors :

```
template <class Arg1, class Arg2, class Result>
struct binary_function
{
        typedef Arg1 first_argument_type;
        typedef Arg2 second_argument_type;
        typedef Result result_type;
};
```

Chapter 5

plus

plus is a binary function. It takes two objects of type T, say x and y, and returns $x + y$:

```
template <class T>
struct plus : public binary_function<T, T, T>
{
        T operator()(const T& x, const T& y) const;
};
```

Chapter 6

minus

minus is a binary function. It takes two objects of type T, say x and y, and returns x - y :

```
template <class T>
struct minus : public binary_function<T, T, T>
{
        T operator()(const T& x, const T& y) const;
};
```

Chapter 7

multiplies

multiplies is a binary function. It takes two objects of type T, say x and y, and returns x * y :

```
template <class T>
struct multiplies : public binary_function<T, T, T>
{
        T operator()(const T& x, const T& y) const;
};
```

Chapter 8

divides

divides is a binary function. It takes two objects of type T, say x and y, and returns x / y :

```
template <class T>
struct divides : public binary_function<T, T, T>
{
        T operator()(const T& x, const T& y) const;
};
```

Chapter 9

modulus

modulus is a binary function. It takes two objects of type T, say x and y, and returns $x \% y$:

```
template <class T>
struct modulus : public binary_function<T, T, T>
{
        T operator()(const T& x, const T& y) const;
};
```

Chapter 10

negate

negate is an unary function. It takes one object of type T, say x, and returns - x :

```
template <class T>
struct negate : public unary_function<T, T>
{
        T operator()(const T& x) const;
};
```

Chapter 11

equal_to and *not_equal_to*

equal_to is a binary function. It takes two objects of type T and returns *true* if they are equal (using *T::operator==*) or *false* otherwise :

```
template <class T>
struct equal_to : public binary_function<T, T,bool>
{
        bool operator()(const T& x, const T& y) const;
};
```

not_equal_to is like *equal_to* but returns *true* if the two objects are different (using *T::operator!=*) or *false* otherwise :

```
template <class T>
struct not_equal_to : public binary_function<T, T, bool>
{
        bool operator()(const T& x, const T& y) const;
};
```

Chapter 12

less and *greater*

less is a binary function. It takes two objects of type T, say x and y, and returns *true* if $x < y$ (using *T::operator<*) or *false* otherwise. *less_equal* returns *true* if $x <= y$ (using *T::operator<=*).

greater does the same thing than *less* but returns *true* if $x > y$ (using *T::operator>*) or false otherwise. *greater_equal* returns *true* if $x >= y$ (using *T::operator>=*).

```
template <class T>
struct less : public binary_function<T, T, bool>
{
        bool operator()(const T& x, const T& y) const;
};

template <class T>
struct greater : public binary_function<T, T, bool>
{
        bool operator()(const T& x, const T& y) const;
};

template <class T>
struct greater_equal : public binary_function<T, T, bool>
{
        bool operator()(const T& x, const T& y) const;
};

template <class T>
struct less_equal : public binary_function<T, T, bool>
{
        bool operator()(const T& x, const T& y) const;
};
```

Chapter 13

logical_and and *logical_or*

logical_and is a binary function. It takes two objects of type T and returns the value of a logical and between the two objects (using *T::operator&&*). *logical_or* uses *T::operator*|| :

```
template <class T>
struct logical_and : public binary_function<T, T, bool>
{
        bool operator()(const T& x, const T& y) const;
};

template <class T>
struct logical_or : public binary_function<T, T, bool>
{
        bool operator()(const T& x, const T& y) const;
};
```

Chapter 14

binder1st and *binder2nd*

binder1st and *binder2nd* transform an adaptable binary function into an adaptable unary function by setting one of the function parameter : the first for *binder1st* and the second for *binder2nd*.

Often the constructor of those functors are not called : instead the helper functions *bind1st* and *bind2nd* are used :

```
template <class Operation>
class binder1st : public unary_function<typename Operation::second_argument_type,
                                       typename Operation::result_type>
{
    protected:
        Operation op;
        typename Operation::first_argument_type value;

    public:
        binder1st(const Operation& x, const typename Operation::first_argument_type& y);
        operator()(const typename Operation::second_argument_type& x) const;

        typename Operation::result_type
};

template <class Operation, class T>
binder1st<Operation> bind1st(const Operation& oper, const T& x);

/* _________ bind2nd _________ */
template <class Operation>
class binder2nd : public unary_function<typename Operation::first_argument_type,
                                       typename Operation::result_type>
{
    protected:
        Operation op;
        typename Operation::second_argument_type value;

    public:
        binder2nd(const Operation& x, const typename Operation::second_argument_type& y);
        operator()(const typename Operation::first_argument_type& x) const;

        typename Operation::result_type
};
```

```
template <class Operation, class T>
binder2nd<Operation> bind2nd(const Operation& oper, const T& x);
```

14.1 Example

Chapter 15

ptr_fun, *pointer_to_unary_function* and *pointer_to_binary_function*

pointer_to_unary_function and *pointer_to_binary_function* are types to create adaptable functions. Often they are not used directly, but by calling *ptr_fun*.

ptr_fun is used to create a function pointer adaptor using a function pointer. It returns either a *pointer_to_unary_function* or a *pointer_to_binary_function* depending of its argument type :

```
template <class Arg, class Result>
class pointer_to_unary_function : public unary_function<Arg, Result>
{
  protected:
    Result (*ptr)(Arg);

  public:
    pointer_to_unary_function();
    explicit pointer_to_unary_function(Result (*x)(Arg));

    Result operator()(Arg x) const;
};

template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function :
public binary_function<Arg1, Arg2, Result>
{
  protected:
      Result (*ptr)(Arg1, Arg2);
  public:
    pointer_to_binary_function();
    explicit pointer_to_binary_function(Result (*x)(Arg1, Arg2));

    Result operator()(Arg1 x, Arg2 y) const;
};

/* ---------- ptr_fun ---------- */
template <class Arg, class Result>
```

```
pointer_to_unary_function<Arg, Result> ptr_fun(Result (*x)(Arg));

template <class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1, Arg2, Result> ptr_fun(Result (*x)(Arg1, Arg2));
```

15.1 Examples

Chapter 16

unary_negate and *binary_negate*

unary_negate is a functor construted by passing it a *Unary predicate*, say *pred*. When applying the constructed *unary_negate* it is equivalent to call *!pred*.

binary_negate does the same thing than *unary_negate* but with a *Binary predicate* :

unary_negate and *binary_negate* are easily constructed by calling the *not1* and *not2* functions.

```
template <class Predicate>
class unary_negate
: public unary_function<typename Predicate::argument_type, bool>
{
    protected:
        Predicate pred;
    public:
        explicit unary_negate(const Predicate& pred);

        /* Returns '!pred(x)' */
        bool operator()(const typename Predicate::argument_type& x) const;
};

template <class Predicate>
unary_negate<Predicate> not1(const Predicate& pred);

/* __________ binary_negate __________ */
template <class Predicate>
class binary_negate
: public binary_function<typename Predicate::first_argument_type,
                         typename Predicate::second_argument_type,
                         bool>
{
    protected:
        Predicate pred;
    public:
        explicit binary_negate(const Predicate& pred);

        /* Returns '!pred(x, y)' */
        bool operator()(const typename Predicate::first_argument_type& x,
                        const typename Predicate::second_argument_type& y) const;
};

template <class Predicate>
binary_negate<Predicate> not2(const Predicate& pred);
```

16.1 Example

Chapter 17

Member function adaptors

All member function adaptors allow you to create a functor which will call a member function when using its *operator()*. Thus, you can use a member function as if it was a normal function.

There are 16 functors in this family depending on the form of the member function :

- member function taking no arguments or one argument
- call through pointer of through reference
- member function returning a void type or member function returning non-void type[1]
- const member function or non-const member function

Often the constructor of those functors are not called : instead the helper functions *mem_fun* and *mem_fun_ref* provides instances of member function adaptors.

```
template <class Ret, class T>
class mem_fun_t : public unary_function<T*, Ret>
{
   public:
      explicit mem_fun_t(Ret (T::*pf)());
      Ret operator()(T* p) const;
};

template <class Ret, class T>
class const_mem_fun_t : public unary_function<const T*, Ret>
{
   public:
      explicit const_mem_fun_t(Ret (T::*pf)() const);
      Ret operator()(const T* p) const;
};


template <class Ret, class T>
class mem_fun_ref_t : public unary_function<T, Ret>
{
   public:
```

[1]According to the draft those cases should be handled the same way. But you can use member function returning void type only if your compiler supports partially specialized templates.

```
        explicit mem_fun_ref_t(Ret (T::*pf)());
        Ret operator()(T& r) const;
};

template <class Ret, class T>
class const_mem_fun_ref_t : public unary_function<T, Ret>
{
    public:
        explicit const_mem_fun_ref_t(Ret (T::*pf)() const);
        Ret operator()(const T& r) const;
};

template <class Ret, class T, class Arg>
class mem_fun1_t : public binary_function<T*, Arg, Ret>
{
    public:
        explicit mem_fun1_t(Ret (T::*pf)(Arg));
        Ret operator()(T* p, Arg x) const;
};

template <class Ret, class T, class Arg>
class const_mem_fun1_t : public binary_function<const T*, Arg, Ret>
{
    public:
        explicit const_mem_fun1_t(Ret (T::*pf)(Arg) const);
        Ret operator()(const T* p, Arg x) const;
};

template <class Ret, class T, class Arg>
class mem_fun1_ref_t : public binary_function<T, Arg, Ret>
{
    public:
        explicit mem_fun1_ref_t(Ret (T::*pf)(Arg));
        Ret operator()(T& r, Arg x) const;
};

template <class Ret, class T, class Arg>
class const_mem_fun1_ref_t : public binary_function<T, Arg, Ret>
{
    public:
        explicit const_mem_fun1_ref_t(Ret (T::*pf)(Arg) const);
        Ret operator()(const T& r, Arg x) const;
};

template <class T>
class mem_fun_t<void, T> : public unary_function<T*, void>
{
    public:
        explicit mem_fun_t(void (T::*pf)());
        void operator()(T* p) const;
};

template <class T>
class const_mem_fun_t<void, T> : public unary_function<const T*, void>
```

```
{
   public:
      explicit const_mem_fun_t(void (T::*pf)() const);
      void operator()(const T* p) const;
};

template <class T>
class mem_fun_ref_t<void, T> : public unary_function<T, void>
{
   public:
      explicit mem_fun_ref_t(void (T::*pf)());
      void operator()(T& r) const;
};

template <class T>
class const_mem_fun_ref_t<void, T> : public unary_function<T, void>
{
   public:
      explicit const_mem_fun_ref_t(void (T::*pf)() const);
      void operator()(const T& r) const;
};

template <class T, class Arg>
class mem_fun1_t<void, T, Arg> : public binary_function<T*, Arg,void>
{
   public:
      explicit mem_fun1_t(void (T::*pf)(Arg));
      void operator()(T* p, Arg x) const;
};

template <class T, class Arg>
class const_mem_fun1_t<void, T, Arg> : public binary_function<const T*, Arg, void>
{
   public:
      explicit const_mem_fun1_t(void (T::*pf)(Arg) const);
      void operator()(const T* p, Arg x) const;
};

template <class T, class Arg>
class mem_fun1_ref_t<void, T, Arg> : public binary_function<T, Arg, void>
{
   public:
      explicit mem_fun1_ref_t(void (T::*pf)(Arg));
      void operator()(T& r, Arg x) const;
};

template <class T, class Arg>
class const_mem_fun1_ref_t<void, T, Arg> : public binary_function<T, Arg, void>
{
   public:
      explicit const_mem_fun1_ref_t(void (T::*pf)(Arg) const);
      void operator()(const T& r, Arg x) const;
};
```

```
/* Helper functions */
template <class Ret, class T>
mem_fun_t<Ret, T> mem_fun(Ret (T::*f)());

template <class Ret, class T>
const_mem_fun_t<Ret, T> mem_fun(Ret (T::*f)() const);

template <class Ret, class T>
mem_fun_ref_t<Ret, T> mem_fun_ref(Ret (T::*f)());

template <class Ret, class T>
const_mem_fun_ref_t<Ret, T> mem_fun_ref(Ret (T::*f)() const);

template <class Ret, class T, class Arg>
mem_fun1_t<Ret, T, Arg> mem_fun(Ret (T::*f)(Arg));

template <class Ret, class T, class Arg>
const_mem_fun1_t<Ret, T, Arg> mem_fun(Ret (T::*f)(Arg) const);

template <class Ret, class T, class Arg>
mem_fun1_ref_t<Ret, T, Arg> mem_fun_ref(Ret (T::*f)(Arg));

template <class Ret, class T, class Arg>
const_mem_fun1_ref_t<Ret, T, Arg> mem_fun_ref(Ret (T::*f)(Arg) const);
```

17.1 Example

Part VIII

Misc

Chapter 1

auto_ptr

Part IX

Frequently Asked Question

Chapter 1

FAQ about iterators

---

1 - But damned, why must I write this dusty template type. I know on which vector I will use my iterator, I know which data type this vector is containing. Why must I specify it again in the iterator declaration ?

A good answer to this question is : “Go on page 82 and look at the *iterator* type. As you can see the value type of the vector is used. So, how can your compiler know what to put in this type declaration if you do not tell him ?”

---

2 - But I use an object oriented language where I do not have to do this type of thing. This language, which name comes from a coffee (or something coming from Smalltalk family), allows me to play with the vectors without knowing the type(s) contained in it. Why can’t I do the same thing with the C++ iterators ?

It is because all C++ class does not inherit from a same great elder parent class (usually called *Object* or something like that). However, the C++ provides the templates to allow you to use multiple types with a same implementation. There are many reasons, but we are not here to discuss the advantage of one approach over the other.

The thing is that , you are supposed to know the type contained in your container and declare you iterator keeping this type in mind.

---

3 - But why can’t I use the same iterator type for all container. For example, if I have a *vector<int>* and a *list<int>*, I should only write : *iterator<int>* and reuse my iterator while doing something on the vector or on the list. Why can’t I do that ?

---

4 - I don’t feel right with all these templates around me. I’d rather use the size of my container to go through it. Why do I have to use templates when I can make it more simple ?

Yes, you can use something like :

```
vector<int> v;
```

```
for(int i = 0; i < v.size(); i++)
{
        // Do something with 'v[i]'
}
```

In some cases, iterators are faster. It is because when you use something like *operator[]*, you must go through at least one element (unless you are using the first). When using an iterator on element, say 19, you just have to increment the iterator from element 18 to element 19. Your iterator was already near the good place, so why restart the search from the beginning of your container ?

Moreover, if your iterator is only a pointer, the operators are most likely inlined, so the iterator is as faster as if it was a pointer.

Chapter 2

FAQ about containers

1 - Why are *queue*, *priority queue* and *stack* container adaptors and not like all other containers ?

It is to follow a basic principle of oriented object programming : reusability.

2 - Reusability ? But wasn't it possible to implement all container using a unique base class ? Or to implement some container, like *list*, using only one, like *vector* for example ?

The main purpose of the STL is to provide efficient generic classes to be used instead of rewriting yours. It is possible to implement all containers using a base class. It is also possible to implement some containers using others. But it makes the implementation less efficient than those provided by compiler writers.

For example, it is hard to use the same base class for all containers without having virtual functions. And virtual functions don't make the code very efficient.

Thus, container adaptors are those container that could have been written without losing efficiency and with reusing the existing classes.

Chapter 3

FAQ about algorithms

Chapter 4

FAQ about relations between C and C++

1 - What is the type of a pointer on a member function ?

2 - Can I use a pointer on a member function where a pointer on traditionnal function is required (for example, to use it in a callback or something like that) ?

Part X

GNU Free Documentation License

Chapter 1

GNU Free Documentation License

Version 1.2, November 2002

PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1.1 APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary

Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

1.2 VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 1.3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

1.3 COPYING IN QUANTITY

1.4 MODIFICATIONS

1.5 COMBINING DOCUMENTS

Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

1.6 COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

1.7 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 1.3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

1.8 TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 1.4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 1.4) to Preserve its Title (section 1.1) will typically require changing the actual title.