The Use of STL and STL Extensions in CGAL

Release 3.3.1

25 August 2007

Preface

CGAL is a *Computational Geometry Algorithms Library* written in C++, developed by a consortium consisting of ETH Zürich (Switzerland), Freie Universität Berlin (Germany), INRIA Sophia-Antipolis (France), Martin-Luther-Universität Halle-Wittenberg (Germany), Max-Planck Institut für Informatik, Saarbrücken (Germany), RISC Linz (Austria) Tel-Aviv University (Israel), and Utrecht University (The Netherlands). You can find more information about the project on the CGAL home page at URL `http://www.cgal.org`.

Should you have any questions, comments, remarks or criticism concerning CGAL, please send a message to the email adresses specified on our web site.

Editorial Committee

Authors

Lutz Kettner (ETH Zürich)
Andreas Fabri (INRIA Sophia-Antipolis).

Acknowledgement

This work was supported by the ESPRIT IV Long Term Research Projects No. 21957 (CGAL) and No. 28155 (GALIA).

Contents

Chapter 1

Introduction

CGAL is the *Computational Geometry Algorithms Library* that is developed by the ESPRIT project CGAL. The library is written in C++ and makes heavily use of *templates*, which are a means to obtain generic code.

STL is the Standard Template Library. Its main components are *containers*, *algorithms*, *iterators* and *function objects*. This library is part of the ISO C++ standard [C++98]. STL is more than a library, it is a framework and a programming paradigm which was adopted by the CGAL project for its library of geometric algorithms.

This document describes in a simplified way the basic features of STL. After reading this document you should be able to use these features which are used throughout the CGAL library. This document is neither a reference manual nor a tutorial for STL. For the sake of simplicity we sometimes sacrifice exactness. If you compare what is written in this document with what is written in the reference manual you will see that in reality things are slightly more general and hence slightly more complicated.

If you want to develop your own iterators or containers, this is definitely the wrong document for you. We recommend to have a look at the header files themselves as the code is extremely instructive.

Chapter 2

Preliminaries

2.1 Pair (*pair<T1, T2>*)

Definition

A struct *pair* is a heterogeneous pair of values. Its data members are *first* and *second*.

#include <pair>

Creation

pair<T1, T2> p(T1 x, T2 y);

Introduces a pair.

Operations

template <class T1, class T2>
bool *pair<T1,T2> p == pair<T1,T2> p1*

Test for equality: Two pairs are equal, iff their data members are equal.

template <class T1, class T2>
bool *pair<T1,T2> p < pair<T1,T2> p1*

Lexicographical comparison of two pairs.

Example

```
Employee irene("Irene", "Eneri");
Employee leo("Leonard", "Eneri");
typedef int Social_security_number;
```

```
pair<Social_security_number, Employee> p1(7812, irene),  p2(5555, leo);

assert( p1.first == 7812 );
assert( p2.second == leo );
```

Chapter 3

Iterators

Iterators are a generalization of pointers that allow a programmer to work with different data structures (containers) in a uniform manner. An iterator is the glue that allows to write a single implementation of an algorithm that will work for data contained in an array, a list or some other container – even a container that did not yet exist when the algorithm was implemented.

An iterator is a concept, not a programming language construct. It can be seen as a set of requirements. A type is an iterator if it satisfies those requirements. So, for instance, a pointer to an element of an array is an iterator. We will check this later.

Depending on the operations defined for an iterator, there are five categories: *input, output, forward, bidirectional* and *random access iterators*. We first have to introduce some terminology.

Mutable versus constant: There is an additional attribute that forward, bidirectional and random access iterators might have, that is, they can be *mutable* or *constant* depending on whether the result of the `operator *` behaves as a reference or as a reference to a constant.

Past-the-end value: Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding container. These values are called *past-the-end* values. Values of the iterator for which the `operator *` is defined are called *dereferenceable*. The library never assumes that past-the-end values are dereferenceable.

Reachability An iterator j is called *reachable* from an iterator i if and only if there is a finite sequence of applications of *operator++* to i that makes $i == j$. If i and j refer to the same container, then either j is reachable from i, or i is reachable from j, or both ($i == j$).

Range: Most of the library's algorithmic templates that operate on data structures have interfaces that use *ranges*. A range is a pair of iterators that designate the beginning and end of the computation. A range $[i,i)$ is an *empty range*; in general, a range $[i,j)$ refers to the elements in the data structure starting with the one pointed to by i and up to but not including the one pointed to by j. Range $[i,j)$ is valid if and only if j is reachable from i. The result of the application of the algorithms in the library to invalid ranges is undefined.

As we mentioned in the introduction we are a little bit sloppy in the presentation of STL, in order to make it easier to understand. A class is said to be an iterator if it fulfills a set of requirements. In the following sections we do not present the requirements, but we state properties that are true, if the requirements are fulfilled. The difference is best seen by an example: we write that the return value of the test for equality returns a *bool*, but the requirement is only that the return value is convertible to *bool*.

3.1 Forward Iterator (*forward_iterator*)

Definition

A class *forward_iterator* that satisfies the requirements of a forward iterator for the value type *T*, supports the following operations.

Creation

forward_iterator it;

forward_iterator it(iterator it1);

Operations

iterator&	*it = iterator it1*	Assignment.
bool	*it == iterator it1*	Test for equality: Two iterators are equal if they refer to the same item.
bool	*it != iterator it1*	Test for inequality. The result is the same as *!(it == it1)*.
T&	**it*	Returns the value of the iterator. If *forward_iterator* is mutable **it = t* is valid. *Precondition*: *it* is dereferenceable.
iterator&	*++it*	Prefix increment operation. *Precondition*: *it* is dereferenceable.
iterator	*it++*	Postfix increment operation. The result is the same as that of *iterator tmp = it; ++it; return tmp;*. *Precondition*: *it* is dereferenceable.

3.2 Bidirectional Iterator (*bidirectional_iterator*)

Definition

A class *bidirectional_iterator* that satisfies the requirements of a bidirectional iterator for the value type *T*, supports the following operations in addition to the operations supported by a forward iterator.

Operations

iterator&	*−−it*	Prefix decrement operation. *Precondition*: *it* is dereferenceable.
iterator	*it−−*	Postfix decrement operation. The result is the same as that of *iterator tmp = it; --it; return tmp;*. *Precondition*: *it* is dereferenceable.

3.3 Random Access Iterator (*random_access_iterator*)

Definition

A class *random_access_iterator* that satisfies the requirements of a random access iterator for the value type *T*, supports the following operations in addition to the operations supported by a bidirectional iterator.

Operations

iterator&	$it += int\ n$	The result is the same as if the prefix increment operation was applied *n* times, but it is computed in constant time.
iterator	$it + int\ n$	Same as above, but returns a new iterator.
iterator	$int\ n + iterator\ it$	Same as above.
iterator&	$it -= int\ n$	The result is the same as if the prefix decrement operation was applied *n* times, but it is computed in constant time.
iterator	$it - int\ n$	Same as above, but returns a new iterator.
int	$it - iterator\ it1$	The result *n* is such that *it1 + n == it*. *Precondition*: *it* is reachable from *it1*.
T&	*it*[*int n*]	Returns **(it + n)*. *Precondition*: **(it + n)* is dereferenceable
bool	$it < iterator\ it1$	$<$ is a total ordering relation.
bool	$it > iterator\ it1$	$>$ is a total ordering relation opposite to <.
bool	$it <= iterator\ it1$	Result is the same as *! it > it1*.
bool	$it >= iterator\ it1$	Result is the same as *! it < it1*.

Example

We promised to show why a pointer (in this case *V**) is a random access iterator. In order to show that it supports the required operations, we give a program that uses them. Not all operations are used — so it is not a complete proof — but hopefully enough to convince the reader.

Notice that the program is nothing special. It was valid C++ (with minor changes even valid C), long before the iterator concept was invented! The comments point to the requirements.

```
const int N = 10;

struct V {
   float val;
};

float fill_and_add(V *b, V *e)
{
    V *c;                           /* creation, first way (ForwardIterator) */
    float sum;
    for (int i=0; i<e−b; i++)       /* subtraction (RandomAccessIterator) */
        b[i].val = i*i/2.0;         /* indexing (RandomAccessIterator) */
    for (c=b; c ≠ e; ++c)           /* assignment, inequality test, increment (ForwardIterator) */
        sum += (*c).val;            /* dereference (ForwardIterator) */
    return sum;
}

float foo()
{
    V a[N];
    V *e(a+N);                      /* creation, second way (ForwardIterator) and */
                                    /* integer adding (RandomAccessIterator) */
    return fill_and_add(a,e);
}
```

Example

We give a second example for the more advanced STL user. We stated that iterators allow us to work with different data structures in a unform manner. But the function *fill_and_add* in the previous example only takes pointers as argument. Here we rewrite it so that it takes any random access iterator as argument, provided it has the right value type (*V*).

```
template <class RandomAccessIterator>
float fill_and_add(RandomAccessIterator b, RandomAccessIterator e)
{
    RandomAccessIterator c;
    float sum;
    for (int i=0; i<e−b; i++)
        b[i].val = i*i/2.0;
    for (c=b; c ≠ e; ++c)
        sum += (*c).val;
    return sum;
}
```

3.4 Input Iterator (*input_iterator*)

Definition

A class *input_iterator* that satisfies the requirements of an input iterator for the value type *T*, supports the following operations.

Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be single pass algorithms.

Creation

input_iterator it(iterator it1);

Operations

iterator&	*it = iterator it1*	Assignment.
T	$*it$	Returns the value of the iterator. *Precondition*: *it* is dereferenceable.
iterator&	$++\,it$	Prefix increment operation. *Precondition*: *it* is dereferenceable.
iterator	$it++$	Postfix increment operation. The result is the same as that of *(void)++it;*. *Precondition*: *it* is dereferenceable.

Example

The following code fragment reads numbers of the type *double* from *cin* and computes their sum. The STL provides an *istream_iterator* that fulfills the input iterator requirements. As the iterator is a kind of file pointer it should be clear why only single pass algorithms can be applied on this iterator.

```
{
    istream_iterator<double> it(cin),
                             end();
    double sum = 0.0;
    while(it ≠ end){
        sum += *it;
        ++it;
    }
    cout ≪ sum ≪ endl;
}
```

3.5 Output Iterator (*output_iterator*)

Definition

A class *output_iterator* that satisfies the requirements of an output iterator for the value type T, supports the following operations.

Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be single pass algorithms.

Creation

output_iterator it;

output_iterator it(iterator it1);

Operations

iterator&	*it = iterator it1*	Assignment.
bool	*it == iterator it1*	Test for equality: Two iterators are equal if they refer to the same item.
bool	*it != iterator it1*	Test for inequality. The result is the same as *!(it == it1)*.
T&	**it*	Returns a reference to the value of the iterator. This operator can only be used in order to assign a value to this reference.
iterator&	*++it*	Prefix increment operation. *Precondition*: *it* is dereferenceable.
void	*it++*	Postfix increment operation. The result is the same as that of *iterator tmp = it; ++it; return tmp;*. *Precondition*: *it* is dereferenceable.

Example

The following code fragment reads numbers of the type *double* from *cin* and computes their sum. The STL provides an *ostream_iterator* that fulfills the output iterator requirements. As the iterator is a kind of file pointer it should be clear why only single pass algorithms can be applied on this iterator.

```
{
    ostream_iterator<double> it(cout);
    for(int r = 0; r < 10; r++){
        *it = 3.1415 * r * r;
        ++it;
    }
}
```

The above code fragment is equivalent to:

```
{
    for(int r = 0; r < 10; r++){
        cout << 3.1415 * r * r;
    }
}
```

The use of output iterators is better illustrated with a function that can write into arbitrary containers:

```
template < class OutputIterator >
void
generator(OutputIterator it)
{
    for(int r = 0; r < 10; r++){
        *it = 3.1415 * r * r;
        ++it;
    }
}
```

and here comes its usage.

```
{
    ostream_iterator<double> it(cout);
    generator(it);
    double R[10];
    generator(R);
}
```

Note that the memory where the function *generator* writes to must be allocated. If you want to insert the generated doubles at the end of a list you have to use a *back_insert_iterator*. To explain this is out of the scope of this introduction to the STL. Please refer to the STL reference manuals.

Chapter 4

Circulators

Circulators are quite similar to iterators that are described in Chapter 3. Circulators are a generalization of pointers that allow a programmer to work with different circular data structures like a ring list in a uniform manner. Please note that circulators are not part of the STL, but of CGAL. A summary of the requirements for circulators is presented here. Thereafter, a couple of adaptors are described that convert between iterators and circulators. For the complete description of the requirements, support to develop own circulators, and the adaptors please refer to the CGAL Reference Manual: Part 3: Support Library.

The specialization on circular data structures gives the reason for the slightly different requirements for circulators than for iterators. A circular data structure has no natural past-the-end value. In consequence, a container supporting circulators will not have an `end()`-member function, only a `begin()`-member function. The semantic of a range is different for a circulator c: The range $[c,c)$ denotes the sequence of all elements in the data structure. For iterators, this range would be empty. A separate test for an empty sequence has been added to the requirements. A comparison $c ==$ `NULL` for a circulator c tests whether the data structure is empty or not. As for C++, we recommend the use of 0 instead of `NULL`. An example function demonstrates a typical use of circulators. It counts the number of elements in the range $[c,d)$:

```
template <class Circulator, class Size>
void count( Circulator c, Circulator d, Size& n) {
    n = 0;
    if ( c != 0) {
        do {
            ++n;
        } while (++c != d);
    }
}
```

Given a circular data structure S, the expression `count(`S`.begin(),` S`.begin(), counter)` returns the number of elements of S in the `counter` parameter.

As for iterators, circulators come in different flavors. There are *forward, bidirectional* and *random access circulators*. They are either *mutable* or *constant*. The past-the-end value is not applicable for circulators.

Reachability: A circulator d is called *reachable* from a circulator c if and only if there is a finite sequence of applications of *operator++* to c that makes $c == d$. If c and d refer to the same non-empty data structure, then d is reachable from c, and c is reachable from d. In particular, any circulator c referring to a non-empty data structure will return to itself after a finite sequence of applications of *operator++* to c.

Range: Most of the library's algorithmic templates that operate on data structures have interfaces that use *ranges*. A range is a pair of circulators that designate the beginning and end of the computation. A range $[c,c)$ is a *full range*; in general, a range $[c,d)$ refers to the elements in the data structure starting with the one pointed to by c and up to but not including the one pointed to by d. Range $[c,d)$ is valid if and only if both refer to the same data structure. The result of the application of the algorithms in the library to invalid ranges is undefined.

Warning: Please note that the definition of a range is different to that of iterators. An interface of a data structure must declare whether it works with iterators, circulators, or both. STL algorithms always specify only iterators in their interfaces. A range $[c,d)$ of circulators used in an interface for iterators will work as expected as long as $c\ != d$. A range $[c,c)$ will be interpreted as the empty range like for iterators, which is different than the full range that it should denote for circulators.

Algorithms could be written to support both, iterators and circulators, in a single interface. Here, the range $[c,c)$ would be interpreted correctly. For more information how to program functions with this behavior, please refer to the CGAL Reference Manual.

As we said in the introduction, we are a little bit sloppy in the presentation, in order to make it easier to understand. A class is said to be a circulator if it fulfills a set of requirements. In the following sections we do not present the requirements, but we state properties that are true, if the requirements are fulfilled. The difference is best seen by an example: We write that the return value of the test for equality returns a *bool*. The requirement is only that the return value is convertible to *bool*.

4.1 Forward Circulator (*Circulator*)

Definition

A class *Circulator* that satisfies the requirements of a forward circulator for the value type *T*, supports the following operations.

Creation

Circulator c;

Circulator c(Circulator d);

Operations

bool	$c = Circulator\ d$	Assignment.
bool	$c == NULL$	Test for emptiness.
bool	$c\ != NULL$	Test for non-emptiness. The result is the same as *!(c == NULL)*.
bool	$c == Circulator\ d$	Test for equality: Two circulators are equal if they refer to the same item.

bool	$c\ != Circulator\ d$	Test for inequality. The result is the same as *!(c == d)*.
T&	$*c$	Returns the value of the circulator. If *Circulator* is mutable **c = t* is valid. *Precondition*: *c* is dereferenceable.
Circulator&	$++c$	Prefix increment operation. *Precondition*: *c* is dereferenceable. *Postcondition*: *c* is dereferenceable.
Circulator	$c++$	Postfix increment operation. The result is the same as that of *Circulator tmp = c; ++c; return tmp;* .

4.2 Bidirectional Circulator (*Circulator*)

Definition

A class *Circulator* that satisfies the requirements of a bidirectional circulator for the value type *T*, supports the following operations in addition to the operations supported by a forward circulator.

Operations

Circulator&	$--c$	Prefix decrement operation. *Precondition*: *c* is dereferenceable. *Postcondition*: *c* is dereferenceable.
Circulator	$c--$	Postfix decrement operation. The result is the same as that of *Circulator tmp = c; --c; return tmp;* .

4.3 Random Access Circulator (*Circulator*)

Definition

A class *Circulator* that satisfies the requirements of a random access Circulator for the value type *T*, supports the following operations in addition to the operations supported by a bidirectional Circulator.

Operations

Circulator&	$c += int\ n$	The result is the same as if the prefix increment operation was applied *n* times, but it is computed in constant time.
Circulator	$c + int\ n$	Same as above, but returns a new circulator.
Circulator	$int\ n + c$	Same as above.

Circulator&	$c -= int\ n$	The result is the same as if the prefix decrement operation was applied n times, but it is computed in constant time.
Circulator	$c - int\ n$	Same as above, but returns a new circulator.
T&	$c[\ int\ n]$	Returns **(c + n)*.
int	$c - Circulator\ d$	returns the difference between the two circulators within the interval $[1-s, s-1]$ for a sequence size s. The difference for a fixed circulator c (or d) with all other circulators d (or c) is a consistent ordering of the elements in the data structure. There has to be a minimal circulator d_{min} for which the difference $c-d_{\text{min}}$ to all other circulators c is non negative.
Circulator	*c.min_circulator()*	Returns the minimal circulator c_{min} in constant time. If c has a singular value, a singular value is returned.

There are no comparison operators required.

4.4 Adaptor: Container with Iterators from Circulator

Algorithms working on iterators cannot be applied to circulators in full generality, only to subranges (see the warning in Section 4). The following adaptors convert circulators to iterators (with the unavoidable space and time drawback) to reestablish this generality.

Definition

The adaptor *Container_from_circulator<C>* is a class that converts any circulator type *C* to a kind of container class, i.e. a class that provides an *iterator* and a *const_iterator* type and two member functions – *begin()* and *end()* – that return the appropriate iterators. In analogy to STL container classes these member functions return a *const_iterator* in the case that the container itself is constant and a mutable *iterator* otherwise.

#include <CGAL/circulator.h>

Types

Container_from_circulator<C>:: Circulator	the template argument *C*.
Container_from_circulator<C>:: iterator	
Container_from_circulator<C>:: const_iterator	

Creation

Container_from_circulator<C> container;	the resulting iterators will have a singular value.
Container_from_circulator<C> container(C c);	the resulting iterators will have a singular value if the circulator *c* is singular.

Operations

iterator	*container.begin()*	the start iterator.
const_iterator	*container.begin() const*	the start const iterator.
iterator	*container.end()*	the past-the-end iterator.
const_iterator	*container.end() const*	the past-the-end const iterator.

The *iterator* and *const_iterator* types are of the appropriate iterator category. In addition to the operations required for their category, they have a member function *current_circulator()* that returns a circulator pointing to the same position as the iterator does.

Example

The generic `reverse()` algorithm from the STL can be used with an adaptor if at least a bidirectional circulator `c` is given.

```
Circulator c;  // c is assumed to be a bidirectional circulator.
CGAL::Container_from_circulator<Circulator> container(c);
reverse( container.begin(), container.end());
```

Implementation

The forward and bidirectional iterator adaptors keep track of the number of rounds a circulator has done around the ring-like data structure. This is a kind of winding number. It is used to distinguish between the start position and the end position which will be denoted by the same circulator. This winding number is zero for the *begin()*-iterator and one for the *end()*-iterator. It is incremented whenever a circulator passes the *begin()* position. Two iterators are equal if their internally used circulators and winding numbers are equal. This is more general than necessary since the *end()*-iterator is not supposed to move any more.

The random access iterator has to be able to compute the size of the data structure. It is needed for the difference of a past-the-end iterator and the begin iterator. Therefore, the constructor for the random access iterator choose the minimal circulator for the internal anchor position. The minimal circulator is part of the random access circulator requirements, see Section 4.3.

4.5 Adaptor: Circulator from Iterator

To obtain circulators, one could use a container class like those in the Standard Template Library (STL) or a pair of *begin()*-, *end()*-iterators and one of the following adaptors. The adaptor for iterator pairs is described here, the adaptor for container classes is described in the next section.

Definition

The adaptor *Circulator_from_iterator<I>* converts two iterators of type *I*, a begin and a past-the-end value, to a circulator of equal category. The iterator must be at least of the forward iterator category. The circulator will be mutable or non-mutable according to the iterator. Iterators provide no *size_type*. This adapter assumes *std::size_t* instead.

#include <CGAL/circulator.h>

Types

typedef I   *iterator;*

In addition all types required for circulators are provided.

Creation

Circulator_from_iterator<I> c;	a circulator *c* with a singular value.
Circulator_from_iterator<I> c(I begin, I end);	a circulator *c* initialized to refer to the element **begin* in a range [*begin*,*end*). The circulator *c* contains a singular value if *begin==end*.

Operations

The adaptors conform to the requirements of the different circulator categories. An additional member function *current_iterator()* is provided that returns the current iterator that points to the same position as the circulator does.

Example

This program uses two adaptors, iterators to circulators and back to iterators. It applies an STL sort algorithm on a STL vector with three elements. The resulting vector will be `[2 5 9]` as it will be checked by the assertions. The program is part of the CGAL distribution.

```
#include <CGAL/basic.h>
#include <cassert>
#include <vector>
#include <algorithm>
#include <CGAL/circulator.h>

typedef  std::vector<int>::iterator                  I;
typedef  CGAL::Circulator_from_iterator<I>           Circulator;
typedef  CGAL::Container_from_circulator<Circulator> Container;
typedef  Container::iterator                         Iterator;

int main() {
    std::vector<int> v;
    v.push_back(5);
    v.push_back(2);
    v.push_back(9);
    Circulator c( v.begin(), v.end());
    Container  container( c);
    std::sort( container.begin(), container.end());
    Iterator i = container.begin();
    assert( *i == 2);
    i++;    assert( *i == 5);
    i++;    assert( *i == 9);
    i++;    assert(  i == container.end());
    return 0;
}
```

File: `examples/Circulator/circulator_prog1.cpp`

Another example usage for this adaptor are random access circulators over the built-in C arrays. Given an array of type `T*` with a begin pointer `b` and a past-the-end pointer `e` the adaptor *Circulator_from_iterator< T*> c(b,e)* is a random circulator *c* over this array.

4.6 Adaptor: Circulator from Container

To obtain circulators, one could use a container class like those in the Standard Template Library (STL) or a pair of *begin()*-, *end()*-iterators and one of the provided adaptors here. The adaptor for iterator pairs is described in the previous section, the adaptor for container classes is described here.

Definition

The adaptor *Circulator_from_container<C>* provides a circulator for an STL container *C* of equal category as the iterator provided by the container. The iterator must be at least of the forward iterator category. The corresponding non-mutable circulator is called *Const_circulator_from_container<C>*.

The container type *C* is supposed to conform to the STL requirements for container (i.e. to have a *begin()* and an *end()* iterator as well as the local types *reference*, *const_reference*, *value_type*, *size_type*, and *difference_type*).

#include <CGAL/circulator.h>

Types

All types required for circulators are provided.

Creation

Circulator_from_container<C> c;	a circulator *c* with a singular value.
Circulator_from_container<C> c(C container);*	a circulator *c* initialized to refer to the first element in *container*, i.e. *container.begin()*. The circulator *c* contains a singular value if the *container* is empty.
Circulator_from_container<C> c(C container, C::iterator i);*	a circulator *c* initialized to refer to the element **i* in *container*. *Precondition*: **i* is dereferenceable and refers to *container*.

Operations

The adaptors conform to the requirements of the different circulator categories. An additional member function *current_iterator()* is provided that returns the current iterator that points to the same position as the circulator does.

Example

This program uses two adaptors, container to circulators and back to iterators. It applies an STL sort algorithm on a STL vector with three elements. The resulting vector will be `[2 5 9]` as it will be checked by the assertions. The program is part of the CGAL distribution.

```
#include <CGAL/basic.h>
#include <cassert>
#include <vector>
#include <algorithm>
#include <CGAL/circulator.h>

typedef CGAL::Circulator_from_container< std::vector<int> >  Circulator;
typedef CGAL::Container_from_circulator<Circulator>          Container;
typedef Container::iterator                                  Iterator;

int main() {
    std::vector<int> v;
    v.push_back(5);
    v.push_back(2);
    v.push_back(9);
    Circulator c( &v);
    Container  container( c);
    std::sort( container.begin(), container.end());
    Iterator i = container.begin();
    assert( *i == 2);
    i++;    assert( *i == 5);
    i++;    assert( *i == 9);
    i++;    assert(  i == container.end());
    return 0;
}
```

File: examples/Circulator/circulator_prog2.cpp

Chapter 5

Function Objects

Function objects are objects with an *operator()(..)* defined. This results in faster code than passing function pointers, as the operator can even be inlined. The following function object classes are defined in STL.

5.1 Arithmetic operations

#include <functional>

STL defines the following function object classes *plus<T>*, *minus<T>*, *times<T>*, *divides<T>*, and *modulus<T>*, which have an operator() with two arguments. Furthermore, there is the function object class *negate<T>* with a single argument. The arguments as well as the return value are of type *T*.

Pars pro toto we give the more formal definition for tha class *plus<T>*.

Definition

An object of the class *plus* is a function object that allows to add two objects of type *T*.

Operations

T	*add(T t1, T t2)*	returns *t1* + *t2*. *Precondition*: '+' must be defined for type *T*.

Example

The following example shows how the function object *negate<T>* is applied on each element of an array.

```
{
    const int n = 10;
    int A[n];
    A[0] = 23;
    ...
    A[9] = 56;
    for_each(A, A+n, negate<int>());
}
```

5.2 Comparisons

#include <functional>

STL defines the following function object classes *equal_to<T>*, *not_equal_to<T>*, *greater<T>*, *less<T>*, *greater_equal<T>*, *less_equal<T>*. They all have an operator() with two arguments of type *T* and the return value is of type *bool*.

Definition

An object of the class *greater* is a function object that allows to compare two objects of type *T*.

Operations

T	*g(T t1, T t2)*	returns *t1 > t2*. *Precondition*: '>' must be defined for type *T*.

Example

A *set* is a container that stores objects in a linear order. Instead of having global *compare* functions for a type we pass a function object as template argument. Set *S* stores integers in a decreasing order. The first template argument is the type of the data in the set, the second template argument is the type of the comparison function object class.

```
{
    set< int, greater<int> > S;
}
```

The following code fragment shows how to sort an array using the STL function *sort*.

```
{
    const int n = 10;
    int A[n];
    A[0] = 23;
    ...
    A[9] = 56;
    sort(A, A+n, greater<int>());
}
```

Chapter 6

Sequence Containers

Sequence containers are objects that store other objects of a single type, organized in a strictly linear arrangement.

6.1 list (*list<T>*)

Definition

An object of the class *list* is a sequence that supports bidirectional iterators and allows constant time insert and erase operations anywhere within the sequence.

#include <list>

Types

list<T>:: iterator	A mutable bidirectional iterator.
list<T>:: const_iterator	A const bidirectional iterator.

Creation

list<T> L;	Introduces an empty list.
list<T> L(list<T> q);	Copy constructor.
list<T> L(int n, T t = T());	Introduces a list with n items, all initialized to t.

Operations

list<T> &	*L* = *list<T> L1*	Assignment.

bool	*L == list<T> L1*	Test for equality: Two lists are equal, iff they have the same size and if their corresponding elements are equal.
bool	*L != list<T> L1*	Test for inequality.
iterator	*L.begin()*	Returns a mutable iterator referring to the first element in list *L*.
const_iterator	*L.begin() const*	Returns a constant iterator referring to the first element in list *L*.
iterator	*L.end()*	Returns a mutable iterator which is the past-end-value of list *L*.
const_iterator	*L.end() const*	Returns a constant iterator which is the past-end-value of list *L*.
bool	*L.empty()*	Returns *true* if *L* is empty.
int	*L.size()*	Returns the number of items in list *L*.
T&	*L.front()*	Returns a reference to the first item in list *L*.
T	*L.front() const*	Returns a const reference to the first item in list *L*.
T&	*L.back()*	Returns a reference to the last item in list *L*.
T	*L.back() const*	Returns a const reference to the last item in list *L*.

Insertion

void	*L.push_front(T)*	Inserts an item in front of list *L*.
void	*L.push_back(T)*	Inserts an item at the back of list *L*.
iterator	*L.insert(iterator pos, T t)*	Inserts a copy of *t* in front of iterator *pos*. The return value points to the inserted item.
void	*L.insert(iterator pos, int n, T t = T())*	Inserts *n* copies of *t* in front of iterator *pos*.
void	*L.insert(iterator pos, const_iterator first, const_iterator last)*	Inserts a copy of the range $[first, last)$ in front of iterator *pos*.

Removal

void	*L.pop_front()*	Removes the first item from list *L*.
void	*L.pop_back()*	Removes the last item from list *L*.
void	*L.erase(iterator pos)*	Removes the item from list *L*, where *pos* refers to.
void	*L.erase(iterator first, iterator last)*	Removes the items in the range[*first*, *last*) from list *L*.

6.2 vector (*vector<T>*)

Definition

An object of the class *vector* is a sequence that supports random access iterators. In addition it supports (amortized) constant time insert and erase operations at the end. Insert and erase in the middle take linear time.

#include <vector>

Types

vector<T>:: iterator	A mutable random access iterator.
vector<T>:: const_iterator	A const random access iterator.

Creation

vector<T> V;	Introduces an empty vector.
vector<T> V(vector<T> q);	Copy constructor.
vector<T> V(int n, T t = T());	Introduces a vector with *n* items, all initialized to *t*.

Operations

vector<T> &	*V = vector<T> V1*	Assignment.
bool	*V == vector<T> V1*	Test for equality: Two vectors are equal, iff they have the same size and if their corresponding elements are equal.
bool	*V != vector<T> V1*	Test for inequality.
bool	*V < vector<T> V1*	Test for lexicographically smaller.
iterator	*V.begin()*	Returns a mutable iterator referring to the first element in vector *V*.
const_iterator	*V.begin() const*	Returns a constant iterator referring to the first element in vector *V*.
iterator	*V.end()*	Returns a mutable iterator which is the past-end-value of vector *V*.
const_iterator	*V.end() const*	Returns a constant iterator which is the past-end-value of vector *V*.

bool	*V.empty()*	Returns *true* if *V* is empty.
int	*V.size()*	Returns the number of items in vector *V*.
T&	*V*[*int pos*]	Random access operator.
T	*V*[*int pos*]	Random access operator.
T&	*V.front()*	Returns a reference to the first item in vector *V*.
T	*V.front() const*	Returns a const reference to the first item in vector *V*.
T&	*V.back()*	Returns a reference to the last item in vector *V*.
T	*V.back() const*	Returns a const reference to the last item in vector *V*.

Insert and Erase

void	*V.push_back(T)*	Inserts an item at the back of vector *V*.
iterator	*V.insert(iterator pos, T t)*	Inserts a copy of *t* in front of iterator *pos*. The return value points to the inserted item.
void	*V.insert(iterator pos, int n, T t = T())*	Inserts *n* copy of *t* in front of iterator *pos*.
void	*V.insert(iterator pos, const_iterator first, const_iterator last)*	Inserts a copy of the range $[first, last)$ in front of iterator *pos*.
void	*V.pop_back()*	Removes the last item from vector *V*.
void	*V.erase(iterator pos)*	Removes the item from vector *V*, where *pos* refers to.
void	*V.erase(iterator first, iterator last)*	Removes the items in the range$[first, last)$ from vector *V*.

6.3 deque (*deque<T>*)

Definition

An object of the class *deque* is a sequence that supports random access iterators. In addition it supports constant time insert and erase operations at both ends. Insert and erase in the middle take linear time.

#include <deque>

Types

deque<T>:: iterator	A mutable random access iterator.
deque<T>:: const_iterator	A const random access iterator.

Creation

deque<T> D;	Introduces an empty deque.
deque<T> D(deque<T> q);	Copy constructor.
deque<T> D(int n, T t = T());	Introduces a deque with *n* items, all initialized to *t*.

Operations

deque<T> &	*D = deque<T> D1*	Assignment.
bool	*D == deque<T> D1*	Test for equality: Two deques are equal, iff they have the same size and if their corresponding elements are equal.
bool	*D != deque<T> D1*	Test for inequality.
bool	*D < deque<T> D1*	Test for lexicographically smaller.
iterator	*D.begin()*	Returns a mutable iterator referring to the first element in deque *D*.
const_iterator	*D.begin() const*	Returns a constant iterator referring to the first element in deque *D*.
iterator	*D.end()*	Returns a mutable iterator which is the past-end-value of deque *D*.
const_iterator	*D.end() const*	Returns a constant iterator which is the past-end-value of deque *D*.

bool	*D.empty()*	Returns *true* if *D* is empty.
int	*D.size()*	Returns the number of items in deque *D*.
T&	*D*[*int pos*]	Random access operator.
T	*D*[*int pos*]	Random access operator.
T&	*D.front()*	Returns a reference to the first item in deque *D*.
T	*D.front() const*	Returns a const reference to the first item in deque *D*.
T&	*D.back()*	Returns a reference to the last item in deque *D*.
T	*D.back() const*	Returns a const reference to the last item in deque *D*.

Insert and Erase

void	*D.push_front(T)*	Inserts an item at the beginning of deque *D*.
void	*D.push_back(T)*	Inserts an item at the end of deque *D*.
iterator	*D.insert(iterator pos, T t = T())*	Inserts a copy of *t* in front of iterator *pos*. The return value points to the inserted item.
iterator	*D.insert(iterator pos, int n, T t = T())*	Inserts *n* copy of *t* in front of iterator *pos*. The return value points to the inserted item.
void	*D.insert(iterator pos, const_iterator first, const_iterator last)*	Inserts a copy of the range [*first*, *last*) in front of iterator *pos*.
void	*D.pop_front()*	Removes the first item from deque *D*.
void	*D.pop_back()*	Removes the last item from deque *D*.
void	*D.erase(iterator pos)*	Removes the item from deque *D*, where *pos* refers to.
void	*D.erase(iterator first, iterator last)*	Removes the items in the range[*first*, *last*) from deque *D*.

Chapter 7

Associative Containers

Associative containers are objects that store other objects of a single type. They allow for the fast retrieval of data based on keys.

7.1 set (*set<Key, Compare>*)

Definition

An object of the class *set<Key, Compare>* stores unique elements of type *Key*. It allows for the retrieval for the elements themselves. The elements in the set are ordered by the ordering relation *Compare*.

#include <set>

Types

set<Key, Compare>:: iterator	A const bidirectional iterator.

Creation

set<Key, Compare> S(Compare comp = Compare());

Introduces an empty set.

set<Key, Compare> S(set<Key, Compare> S1);	Copy constructor.

Operations

set<Key, Compare> & *S = set<Key, Compare> S1*

Assignment.

bool	*S == set<Key, Compare> S1*	
		Equality test: Two sets are equal, if the sequences *S* and *S1* are elementwise equal.
bool	*S < set<Key, Compare> S1*	
		Returns *true* if *S* is lexicographically less than *S1*, *false* otherwise.
set<Key, Compare>::iterator		
	S.begin()	Returns a constant iterator referring to the first element in set *S*.
set<Key, Compare>::iterator		
	S.end()	Returns a constant past-the-end iterator of set *S*.
bool	*S.empty()*	Returns *true* if *S* is empty.
int	*S.size()*	Returns the number of items in set *S*.

Insert and Erase

set<Key, Compare>::iterator		
	S.insert(set<Key, Compare>::iterator pos, Key k)	
		Inserts *k* in the set if *k* is not already present in *S*. The iterator *pos* is the starting point of the search. The return value points to the inserted item.
pair<set<Key, Compare>::iterator, bool>		
	S.insert(Key k)	Inserts *k* in the set if *k* is not already present in *S*. Returns a pair, where *first* is the iterator that points to the inserted item or to the item that is already present in *S*, and where *second* is *true* if the insertion took place.
void	*S.erase(set<Key, Compare>::iterator pos)*	
		Erases the element where pos points to.
int	*S.erase(Key k)*	Erases the element *k*, if present. Returns the number of erased elements.

Miscellaneous

set<Key, Compare>::iterator		
	S.find(Key k)	Returns an iterator that either points to the element *k*, or *end()* if *k* is not present in set *S*.
int	*S.count(Key k)*	Returns the number of occurrences of *k* in set *S*.
set<Key, Compare>::iterator		
	S.lower_bound(Key k)	
		Returns an iterator that points to the first element of *S* that is not less than *k*. If all elements are less than *k* then *end()* is returned. If *k* is present in the set the returned iterator points to *k*.
set<Key, Compare>::iterator		
	S.upper_bound(Key k)	
		Returns an iterator that points to the first element of the set that is greater than *k*. If no element is greater than *k* then *end()* is returned.

7.2 multiset (*multiset<Key, Compare>*)

Definition

An object of the class *multiset<Key, Compare>* can store multiple copies of the same element of type *Key*. The elements in the multiset are ordered by the ordering relation *Compare*.

The interface of the class *multiset<Key, Compare>* is almost the same as of the class *set<Key, Compare>*. We only list the functions that have a different syntax or semantics.

#include <set>

Types

multiset<Key, Compare>:: iterator	A const bidirectional iterator.

Operations

iterator	*M.insert(iterator pos, Key k)*	
		Inserts *k* in the set. The iterator *pos* is the starting point of the search. The return value points to the inserted item.

iterator	*M.insert(Key k)*	Inserts *k* in the set. Returns an iterator that points to the inserted item.
void	*M.erase(iterator pos)*	Erases the element where pos points to. This erases only one element
int	*M.erase(Key k)*	Erases all elements that are equal to *k*. Returns the number of erased elements.

7.3 map (*map<Key, T, Compare>*)

Definition

An object of the class *map<Key, T, Compare>* supports unique keys of type *Key*, and provides retrieval of values of type *T* based on the keys. The keys into the map are ordered by the ordering relation *Compare*.

Elements are stored in maps as *pairs* of *Key* and *T*.

#include <map>

Types

map<Key, T, Compare>:: iterator	A const bidirectional iterator.

Creation

map<Key, T, Compare> M(Compare comp = Compare());

Introduces an empty map.

map<Key, T, Compare> M(map<Key, T, Compare> M1);

Copy constructor.

Operations

map<Key, T, Compare> &

M = map<Key, T, Compare> M1

Assignment.

bool	*M == map<Key, T, Compare> M1*	
		Equality test: Two maps are equal, if the sequences *M* and *M1* are elementwise equal.
bool	*M < map<Key, T, Compare> M1*	
		Returns *true* if *M* is lexicographically less than *M1*, *false* otherwise.
iterator	*M.begin()*	Returns a constant iterator referring to the first element in map *M*.

iterator	*M.end()*	Returns a constant past-the-end iterator of map *M*.
bool	*M.empty()*	Returns *true* if *M* is empty.
int	*M.size()*	Returns the number of items in map *M*.
T&	*M*[*Key k*]	Returns a reference to the type *T* value associated with key *k*. If the map is constant then a const reference is returned. In contrast to vector or deque, the *pair(k,T())* is inserted into the map, if no element is associated with the key.

Insert and Erase

iterator	*M.insert(iterator pos, pair< Key ,T> val)*	
		Inserts *val* into the map if *val* is not already present in *M*. The iterator *pos* is the starting point of the search. The return value points to the inserted item.
pair<iterator, bool>	*M.insert(pair< Key ,T> val)*	
		Inserts *val* into the map if *val* is not already present in *M*. Returns a pair, where *first* is the iterator that points to the inserted item or to the item that is already present in *M*, and where *second* is *true* if the insertion took place.
void	*M.erase(iterator pos)*	
		Erases the element where pos points to.
int	*M.erase(Key k)*	Erases all elements that equal *k*. Returns the number of erased elements.

Miscellaneous

iterator	*M.find(Key k)*	Returns an iterator that either points to the element *k*, or *end()* if *k* is not present in map *M*.
int	*M.count(Key k)*	Returns the number of occurrences of *k* in map *M*.
iterator	*M.lower_bound(Key k)*	
		Returns an iterator that points to the first element of *M* that is not less than *k*. If all elements are less than *k* then *end()* is returned. If *k* is present in the map the returned iterator points to *k*.

iterator *M.upper_bound(Key k)*

Returns an iterator that points to the first element of the map that is greater than *k*. If no element is greater than *k* then *end()* is returned. If *k* is present in the map the returned iterator points to *k*.

7.4 **multimap** (*multimap<Key, T, Compare>*)

Definition

An object of the class *multimap<Key, T, Compare>* can store multiple equivalent keys of type *Key*, and provides retrieval of values of type *T* based on the keys. The keys in the multimap are ordered by the ordering relation *Compare*.

The interface of the class *multimap<Key, T, Compare>* is almost the same as of the class *map<Key, Compare>*. We only list the functions that have a different syntax or semantics.

#include <map>

Types

multimap<Key, T, Compare>:: iterator	A const bidirectional iterator.

Operations

iterator	*M.insert(iterator pos, pair<constKey,T> val)*	
		Inserts *val* in the set. The iterator *pos* is the starting point of the search. The return value points to the inserted item.
iterator	*M.insert(pair<constKey,T> val)*	
		Inserts *val* in the set. Returns an iterator that points to the inserted item.
void	*M.erase(iterator pos)*	
		Erases the element where pos points to. This erases only one element
int	*M.erase(Key k)*	Erases all elements that are equal to *k*. Returns the number of erased elements.

Bibliography

[C++98] International standard ISO/IEC 14882: Programming languages – C++. American National Standards Institute, 11 West 42nd Street, New York 10036, 1998.

Index

Pages on which definitions are given are presented in **boldface**.