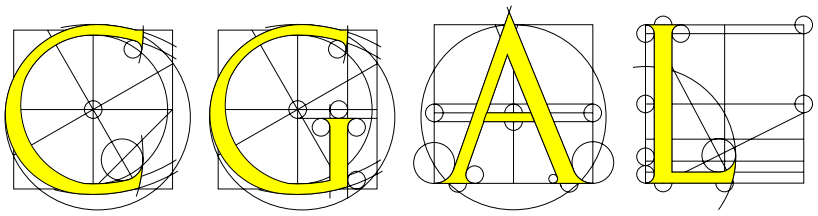

CGAL Reference Manual

Part 3: Support Library

Release 1.0, April 1998

Preface

CGAL is a *Computational Geometry Algorithms Library* written in C++, which is developed by the ESPRIT project CGAL. This project is carried out by a consortium consisting of Utrecht University (The Netherlands), ETH Zürich (Switzerland), Freie Universiät Berlin (Germany), INRIA Sophia-Antipolis (France), Max-Planck Institut für Informatik, Saarbrücken (Germany), RISC Linz (Austria) and Tel-Aviv University (Israel). You find more information on the project on the CGAL home page at URL `http://www.cs.uu.nl/CGAL/`.

Should you have any questions, comments, remarks or criticism concerning CGAL, please send a message to `cgal@cs.uu.nl`.

Editors

Hervé Brönnimann, Andreas Fabri (INRIA Sophia-Antipolis).
Stefan Schirra (Max-Planck Institut für Informatik).
Remco Veltkamp (Utrecht University).

Authors

Chapter 1: Lutz Kettner (ETH Zürich), Sylvain Pion (INRIA Sophia-Antipolis), Stefan Schirra (Max-Planck Institut für Informatik).

Chapter 2: Michael Hoffmann, Lutz Kettner (ETH Zürich).

Chapter 3: Lutz Kettner (ETH Zürich).

Chapter 4: Lutz Kettner (ETH Zürich).

Chapter 5: Michael Hoffmann, Lutz Kettner (ETH Zürich), Sven Schönherr (Freie Universiät Berlin).

Chapter 6: Lutz Kettner (ETH Zürich).

Chapter 7: Lutz Kettner (ETH Zürich).

Chapter 8: Andreas Fabri (INRIA Sophia-Antipolis).

Acknowledgement

This work was supported by the ESPRIT IV Long Term Research Project No. 21957 (CGAL).

Contents

Chapter 1

Number Type Support

CGAL representation classes are parameterized by number types. Depending on the problem and the input data that have to be handled, one has to make a trade-off between efficiency and accuracy in order to select an appropriate number type and representation class.

In homogeneous representation, two number types are involved, although only one of them appears as a template parameter in the homogeneous representation class. This type, for sake of simplicity and readability called ring type is used for the representation of homogeneous coordinates and all internal computations. If it is assured that the second operand divides the first one, these internal computations are basically division-free. The ring type is a placeholder for an integer type (or an integral domain type) rather than for elements of arbitrary rings. The name should remind you that the division operation is not really needed (with the exception mentioned above) for this number type. Of course, also more general number types can be used as a ring type in a homogeneous representation class. In some computations, e.g. accessing Cartesian coordinates, divisions cannot be avoided. In these computations a second number type, the field type, is used. CGAL automatically generates this number type as a *CGAL_Quotient*, cf. Subsection 1.4. For the Cartesian representation there is only one number type that is used for all calculations.

The representation classes provide access to the number types involved in the representation, although it is not expected that such access is needed at this level, since low-level geometric operations are wrapped in geometric primitives provided by CGAL. This access can be useful if appropriate primitives are missing. In a homogeneous representation class R ring type and field type can be accessed as $R::RT$ and $R::FT$, respectively. The number type used in Cartesian representation is considered as ring type and as field type depending on the context. If can be accessed as $R::RT$ and $R::FT$, according to the use of number types used in the homogeneous counterpart.

1.1 Required Functionality of Number Types

Number types must fulfill certain requirements, such that they can be successfully used in CGAL code. This section describes those requirements. We focus on the syntactical requirements. Of course, number types also have evident semantic constraints. They should be meaningful in the sense that they approximate the integers or the rationals or some other subfield of the real numbers.

The requirements are described as a class interface of a class NT, with constructors, methods and the like. This is only a matter of presentation. In fact *double* and *float* also fulfill the requirements.

Creation

NT ntvar;	Declaration of a variable.
NT ntvar(ntval);	Declaration and initialization.
NT ntvar(i);	Declaration and initialization with a small integer constant i, $0 \leq i \leq 127$. The neutral elements for addition (zero) and multiplication (one) are needed quite often, but sometimes other small constants are useful too. The value 127 was chosen such that even signed 8 bit number types can fulfill this condition.

Operations

NT &	*ntvar = ntval*	Assignment.
bool	*CGAL_is_valid(ntval)*	Not all values of a number type need be valid. The routine *CGAL_is_valid* checks this. For example, an expression like *nt(0)/nt(0)* can result in an invalid number. Routines often have as a precondition that all numerical values are valid.
bool	*CGAL_is_finite(ntval)*	When two large doubles are multiplied, the result may not fit in a *NT*. Some number types (the standard float and double type when they conform to standards) have a way to represent a too big value as infinity. *CGAL_is_finite* implies *CGAL_is_valid*.
bool	*ntvar == n*	
bool	*ntvar != n*	
bool	*ntvar < n*	
bool	*ntvar > n*	
bool	*ntvar <= n*	
bool	*ntvar >= n*	
NT	*ntval1 + ntval2*	
NT	*ntval1 − ntval2*	
NT	*ntval1 * ntval2*	
NT	*−ntval*	

bool	*ntvar += n*	
bool	*ntvar -= n*	
bool	*ntvar *= n*	
bool	*ntvar /= n*	
double	*CGAL_to_double(ntval)*	gives the double value for a number type. This is usually an approximation for the real (stored) value. It can be used to send numbers to a renderer or to store them in a file.
NT	*ntval1 /ntval2*	Division by zero need not be defined. It may result in a runtime error, an invalid value, a valid value or anything else. This basically means that the library tests for zero whenever it does a division.

1.2 Utility Routines

The previous section listed all the required functionality. For the user of a number type it is handy to have a larger set of operations available.

1.2.1 Utility Functions

#include <CGAL/number_utils.h>

NT	*CGAL_min(ntval1, ntval2)*	returns the smaller of the two values.
NT	*CGAL_max(ntval1, ntval2)*	returns the larger of the two values.
NT	*CGAL_abs(ntval)*	returns the absolute value.
int	*CGAL_sign(ntval)*	returns the sign: -1, 0, or +1.
bool	*CGAL_is_negative(ntval)*	
bool	*CGAL_is_positive(ntval)*	
bool	*CGAL_is_zero(ntval)*	
bool	*CGAL_is_one(ntval)*	
CGAL_Comparison_result	*CGAL_compare(n1, n2)*	

Those routines are implemented using the required operations from the previous section. They are defined by means of templates, so you do not have to supply all those operations when you write a new number type.

For the number types *bool*, *int*, and *double* there is the random numbers generator *CGAL_Random*.

1.2.2 Utility Function Classes

In addition to the utility routines listed above, there are function object class templates corresponding to these functions. Note that the function object class corresponding to *CGAL_sign* is named *CGAL_Sgn* in order to avoid a conflict with the type *CGAL_Sign*.

CGALprovides the following function object classes:

#include <CGAL/number_utils_classes.h>

class CGAL_Min<NT>;

class CGAL_Max<NT>;

class CGAL_Abs<NT>;

class CGAL_Sgn<NT>;

class CGAL_Is_negative<NT>;

class CGAL_Is_positive<NT>;

class CGAL_Is_zero<NT>;

class CGAL_Is_one<NT>;

class CGAL_Compare<NT>;

1.3 Built-in Number Types

The built-in number types *float* and *double* have the required arithmetic and comparison operators. They lack some required routines though which are automatically included by CGAL. [1]

All built-in number types of C++ can represent a discrete (bounded) subset of the rational numbers only. We assume that the floating-point arithmetic of your machine follows IEEE floating-point-standard.

[1] The functions can be found in the header files <*CGAL/double.h*> and <*CGAL/float.h*>.

Since the floating-point culture has much more infrastructural support (hardware, language definition and compiler) than exact computation, it is very efficient. Like with all number types with finite precision representation which are used as approximations to the infinite ranges of integers or real numbers, the built-in number types are inherently potentially inexact. Be aware of this if you decide to use the efficient built-in number types: you have to cope with numerical problems. For example, you can compute the intersection point of two lines and then check whether this point lies on the two lines. With floating point arithmetic, roundoff errors may cause the answer of the check to be *false*. With the built-in integer types overflow might occur.

1.4 Number Types Provided by CGAL

1.4.1 Quotient (*CGAL_Quotient<NT>*)

Definition

An object of the class *CGAL_Quotient<NT>* is an element of the ring of quotients of the integral domain type *NT*. If *NT* behaves like an integer, *CGAL_Quotient<NT>* behaves like the rational numbers. LEDA's class *rational* (see Section 1.5) has been the basis for *CGAL_Quotient<NT>*. A *CGAL_Quotient<NT>* q is represented as a pair of *NT*s, representing numerator and denominator.

#include <CGAL/Quotient.h>

Creation

CGAL_Quotient<NT>	*q;*	introduces an uninitialized variable q.
CGAL_Quotient<NT>	*q(int i);*	introduces the quotient $i/1$.
CGAL_Quotient<NT>	*q(NT n);*	introduces the quotient $n/1$.
CGAL_Quotient<NT>	*q(NT n, NT d);*	introduces the quotient n/d.

Operations

The arithmetic operations $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$, $-$(unary), and the comparison operations $<$ and $==$ are all available. If *CGAL_PROVIDE_STL_PROVIDED_REL_OPS* is defined, the comparison operations $<=$, $>$, $>=$, $!=$ are provided, too.

There are two access functions, namely to the numerator and the denominator of a quotient. Note that these values are not uniquely defined. It is guaranteed that *q.numerator()* and *q.denominator()* return values *nt_num* and *nt_den* such that $q = nt\_num/nt\_den$, only if *q.numerator()* and *q.denominator()* are called consecutively wrt q, i.e. q is not involved in any other operation between these calls.

NT	*q.numerator()*	returns a numerator of q.
NT	*q.denominator()*	returns a denominator of q.

The stream operations are available as well. They assume that corresponding stream operators for type *NT* exist.

ostream&	*ostream& out << q*	writes *q* to ostream *out* in format "`n/d`", where `n`==*q.numerator()* and `d`==*q.denominator()*.
istream&	*istream& in >> & q*	reads *q* from istream *in*. Expected format is "`n/d`", where `n` and `d` are of type *NT*. A single `n` which is not followed by a `/` is also accepted and interpreted as `n/1`.

The following functions are added to fulfill the CGAL requirements on number types.

double	*CGAL_to_double(q)*	returns some double approximation to *q*.
bool	*CGAL_is_valid(q)*	returns true, if numerator and denominator are valid.
bool	*CGAL_is_finite(q)*	returns true, if numerator and denominator are finite.

Implementation

The quotient class template is based on the class *leda_rational* in LEDA.

1.4.2 Interval Arithmetic (*CGAL_Interval_nt*)

This chapter describes briefly what interval arithmetic is, its implementation in CGAL, and its possible use by geometric programs. The main reason for having interval arithmetic in CGAL is its integration into a geometric predicate pre-compiler, but we also provide a number type so that the users can use it separately if they find any use for it. The purpose of interval arithmetic is to provide an efficient way to control the errors made by floating point computations.

Warning This number type is definitely for the advanced user, who understands what happens. This type is nothing more than doubles, but you can, when you ask for it, have a control on the computation error. That means that if you only use the classical operations on number types, it won't be useful to you. Specifically, if algorithms are not designed to work with interval arithmetic, it makes no sense to plug this number type into a representation class. For example, if you plug this number type into a representation class of say, *CGAL_Point_2*, and apply the *CGAL_convex_hull_points_2* algorithm, the interval arithmetic will not be used.

Definition

Interval arithmetic is a large concept and we will only consider here a simple arithmetic based on intervals whose bounds are *double*s. So each variable is an interval representing any value inside the interval. All arithmetic operations made on intervals give a result that contain all possible values. For example, if the final result of a sequence of arithmetic operations is an interval that does not contain zero, then you can surely decide the sign of the result.

#include <*CGAL/Interval_arithmetic.h*>

Creation

CGAL_Interval_nt (double d = 0);	introduces an interval whose bounds are d.
CGAL_Interval_nt (double d, double e);	introduces the interval [d;e].

Several functions should provide a cast from any numerical type used by CGAL, to an interval CGAL_Interval_nt.

Operations

All functions required by a class to be considered a CGAL number type are present, sometimes with a particular semantic, notably :

CGAL_Interval_nt	I / J	is defined with the following meaning for the degenerate case where the denominator is an interval containing zero: it returns the interval $]-\infty;+\infty[$ (the 'division by zero' exception can be raised, but the default is to ignore it, see next paragraph for details).
CGAL_Interval_nt	*sqrt(I)*	is defined and produces an error if the lower bound of the interval is a negative number.
bool	$I < J$	returns *true* when any value in I is inferior to any value in J.
bool	$I == J$	returns *true* when I and J overlap (or intersect).
bool	$I > J$	is defined logically from the previous ones
bool	$I <= J$	idem
bool	$I >= J$	idem
bool	$I != J$	idem
double	*CGAL_to_double(I)*	returns the middle of the interval, as a double approximation of the interval.
double	*I.lower_bound()*	returns the lower bound of the interval.
double	*I.upper_bound()*	returns the upper bound of the interval.

Implementation

CGAL_Interval_nt derives from *CGAL_Interval_nt_advanced*, see Section 1.4.3.

Example

Here a simple example is given showing how to make safe and fast predicates. This kind of problem is going to be resolved automatically by a predicate compiler, but before it is available, you can do it by hand. Let's consider the *which_side* predicate:

```
int which_side(Point<double> P, Point<double> Q, Point<double> R)
{
  double a = P.x-R.x;
  double b = Q.x-R.x;
  double c = P.y-R.y;
  double d = Q.y-R.y;
  double det = a*d-b*c;

  return (det>0) ? 1 : ( (det<0) ? -1 : 0 );
}
```

This function becomes the following, if you want it to be exact :

```
#include "CGAL/Interval_arithmetic.h"
typedef CGAL_Interval_nt Dis;

int which_side(Point<double> P, Point<double> Q, Point<double> R)
{
  Dis a = (Dis) P.x - (Dis) R.x;
  Dis b = (Dis) Q.x - (Dis) R.x;
  Dis c = (Dis) P.y - (Dis) R.y;
  Dis d = (Dis) Q.y - (Dis) R.y;
  Dis det = a*d-b*c;

  if (det == 0)
  {
    // Note: this doesn't work, but the idea is here.
    return which_side((Point<LEDA_Real>) P, (Point<LEDA_Real>) Q, (Point<LEDA_Real>) R);
  }
  else return (det>0) ? 1 : -1;
}
```

1.4.3 Advanced Interval Arithmetic (*CGAL_Interval_nt_advanced*)

CGAL_Interval_nt derives from *CGAL_Interval_nt_advanced*. It allows you to make faster computations, but you need to set the correct rounding mode of the FPU (towards infinity) before doing any computation with this number type. See for details above, and in the test programs.

#include <*CGAL/Interval_arithmetic.h*>

Implementation

The basic idea is to use the directed rounding modes specified by the *IEEE 754* standard, which are implemented by almost all processors nowadays. It states that you have the possibility, concerning the basic floating point operations ($+, -, *, /, \sqrt{}$) to specify the rounding mode of each operation instead of using the default, which is usually to round to the nearest. This feature allows us to compute easily on intervals. For example, to add the two intervals [a.i;a.s] and [b.i;b.s], compute $c.i = a.i + b.i$ rounded towards minus infinite, and $c.s = a.s + b.s$ rounded towards plus infinite, and the result is the interval [c.i;c.s]. This method can be extended easily to the other operations.

The problem is that we have to change the rounding mode very often, and the functions of the C library doing this operation are slow and different from one architecture to another. That's why assembly versions are used as often as possible. Another trick is to compute the opposite of the lower bound, instead of the normal lower bound, which allows us to change the rounding mode once less. Therefore, all basic operations, which are in the class *CGAL_Interval_nt_advanced* assume that the rounding mode is set to positive infinite, and everything work with this correctly set. The class *CGAL_Interval_nt* takes care of this, but is a bit slower.

Note also that NaNs are not handled, so be careful with that (especially if you 'divide by zero').

FPU rounding modes We provide the following functions (in C or assembly) to change the rounding mode.

- *CGAL_FPU_set_rounding_to_zero()*
- *CGAL_FPU_set_rounding_to_nearest()*
- *CGAL_FPU_set_rounding_to_infinity()*
- *CGAL_FPU_set_rounding_to_minus_infinity()*

Platform support This part of CGAL must be ported to each non-supported platform, because of the use of directed rounding modes. Here is the current state of supported platforms:

- Sparc
  - g++ uses the assembly versions, independent of the OS.
  - CC uses the C library functions to change the rounding modes. Works under Solaris (SunOS doesn't use the same files, sorry).
- Intel
  - g++ uses the assembly versions, independent of the OS (tested for Linux).
  - For Windows (NT, 95), Solaris, and others the situation is unknown.
- Alpha
  - g++ 2.7.2 doesn't support (g++ 2.8 and egcs do) the necessary command line option (to set the dynamic rounding mode). There are several possibilities to make it work, but all of them are painful.
  - CC: I don't know, but cc works when you specify the command line option "-fprm d".
- Mips
  - g++ and CC use the C library functions.

1.5 Number Types Provided by LEDA

LEDA provides number types that can be used for exact computation with both Cartesian and homogeneous representation. If you are using homogeneous representation with the built-in integer types *short*, *int*, and *long* as ring type, exactness of computations can be guaranteed only if your input data come from a sufficiently small integral range and the depth of the computations is sufficiently small. LEDA provides the number type *leda_integer* for integers of arbitrary length. (Of course the length is somehow bounded by the resources of your computer.) It can be used as ring type in homogeneous representation and leads to exact computation as long as all intermediate results are rational. For the same kind of problems Cartesian representation with number type *leda_rational* leads to exact computation as well. The number type *leda_bigfloat* in LEDA is a variable precision floating-point type. Rounding mode and precision (i.e. mantissa length) of *leda_bigfloat* can be set.

The most sophisticated number type in LEDA is the number type called *leda_real*. Like in Pascal, where the name *leda_real* is used for floating-point numbers, the name *leda_real* does not describe the number type precisely, but intentionally. *leda_reals* are a subset of real algebraic numbers. Any integer is *leda_real* and *leda_real*s are closed under the operations $+, -, *, /$ and k-th root computation. *leda_real*s guarantee that all comparisons between expressions involving *leda_real*s produce the exact result.

In the files <*CGAL/leda_integer.h*>, <*CGAL/leda_rational.h*>, <*CGAL/leda_bigfloat.h*>, and <*CGAL/leda_real.h*>, the LEDA types *leda_integer*, *leda_rational*, *leda_bigfloat*, and *leda_real* are made conform to the requirements presented in 1.1. Also, in these files the LEDA number types are included. For more details on the number types of LEDA we refer to the LEDA manual [MNU97].

1.6 Number Types Provided by GNU

CGAL provides wrapper classes for number types defined in the GNU Multiple Precision Arithmetic Library [Gra96]. The file `CGAL/Gmpz.h` provides the class *CGAL_Gmpz*, a wrapper class for the integer type *mpz_t*, that is compliant to the CGAL number type requirements. To use this, the GNU Multiple Precision Arithmetic Library must be installed.

1.6.1 GNU Multiple Precision Integer (*CGAL_Gmpz*)

Definition

An object of the class *CGAL_Gmpz* is an arbitrary precision integer based on the GNU Multiple Precision Arithmetic Library.

#include <*CGAL/Gmpz.h*>

Creation

CGAL_Gmpz q;	creates an uninitialized multiple precision integer q.
CGAL_Gmpz q(int i);	creates an multiple precision integer initialized with i.

CGAL_Gmpz q(double d);	creates an multiple precision integer initialized with the integral part of d.

Operations

The arithmetic operations $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$, $-$(unary), the comparison operations $<$, $<=$, $>$, $>=$, $==$, $!=$ and the stream operations are all available.

The following functions are added to fulfill the CGAL requirements on number types.

double	*CGAL_to_double(g)*	returns some double approximation to g.
bool	*CGAL_is_valid(g)*	returns true
bool	*CGAL_is_finite(g)*	returns true
CGAL_io_Operator	*CGAL_io_tag(g)*	

Implementation

*CGAL_Gmpz*s are reference counted.

1.7 User-supplied Number Types

You can also use your own number type with the CGAL representation classes, e.g. the BIGNUM package [SVH89]. Depending on the arithmetic operations carried out by the algorithms that you are going to use the number types must fulfill the requirements from 1.1.

Chapter 2

STL Extensions for CGAL

CGAL is designed in the spirit of the generic programming paradigm to work together with the Standard Template Library (STL) [C++96, MS96]. This chapter documents non-geometric STL-like components that are not provided in the STL standard but in CGAL: a doubly-connected list managing items in place (where inserted items are not copied), generic functions, function objects for projection and creation, classes for composing function objects and adaptor classes around iterators and circulators. See also circulators in Chapter 3.

2.1 Doubly-Connected List Managing Items in Place

The class *CGAL_In_place_list*<*T*,&*T::next*,&*T::prev*> manages a sequence of items in place in a doubly-connected list. Its goals are the flexible handling of memory management and performance optimization. The item type is supposed to provide the two necessary pointers &*T::next_link* and &*T::prev_link*. One possibility to obtain these pointers is to inherit them from the base class *CGAL_In_place_list_base*<*T*>.

The class *CGAL_In_place_list*<*T*> is a container and quite similar to STL containers, with the advantage that it is able to handle the stored elements by reference instead of copying them. It is possible to delete an element only knowing its address and no iterator to it. This simplifies mutually pointered data structures like a halfedge data structure for planar maps or polyhedral surfaces. The usual iterators are also available. Another container with this property of working with pointers to objects is the STL vector (at least in the current STL implementations).

2.1.1 Base Classes for List Nodes

Definition

The node base classes provides pointers to build linked lists. The class *CGAL_In_place_sl_list_base*<*T*> provides a pointer *next_link* for a single linked list. The class *CGAL_In_place_list_base*<*T*> provides

an additional pointer *prev_link* for doubly linked lists. These names conform to the default parameters used in the template argument lists of the container classes. The pointers are public members.

#include <*CGAL/In_place_list.h*>

*T**	*next_link;*	forward pointer
*T**	*prev_link;*	backwards pointer

2.1.2 Container Class (*CGAL_In_place_list*<*T,bool*>)

Definition

An object of the class *CGAL_In_place_list*<*T,bool*> represents a sequence of items of type T that supports bidirectional iterators and allows constant time insert and erase operations anywhere within the sequence. The functionality is similar to the *list*<*T*> in the STL.

The *CGAL_In_place_list*<*T,bool*> manages the items in place, i.e. inserted items are not copied. Two pointers of type T^* are expected to be reserved in T for the list management. The base class *CGAL_In_place_list_base*<*T*> can be used to obtain such pointers.

The *CGAL_In_place_list*<*T,bool*> does not copy element items during insertion (unless otherwise stated for a function). On removal of an item or destruction of the list the items are not deleted by default. The second template parameter *bool* is set to *false* in this case. If the *CGAL_In_place_list*<*T,bool*> should take the responsibility for the stored objects the *bool* parameter could be set to *true*, in which case the list will delete removed items and will delete all remaining items on destruction. In any case, the *destroy()* member function deletes all items. Note that these two possible versions of *CGAL_In_place_list*<*T,bool*> are not assignable to each other to avoid confusions between the different storage responsibilities.

#include <*CGAL/In_place_list.h*>

Parameters

The full class name is *CGAL_In_place_list*<*T,bool managed = false, T* T::*next = &T::next_link, T* T::*prev = &T::prev_link*>. As long as no default template arguments are supported, only *CGAL_In_place_list*<*T,bool*> is provided.

The parameter T is supposed to have a default constructor, a copy constructor and an assignment operator. The copy constructor and the assignment may not copy the pointers in T for the list management, but they are allowed to. The equality test and the relational order require the operators $==$ and $<$ for T respectively. These operators must not compare the pointers in T.

Types

CGAL_In_place_list<*T,bool*>*:: iterator*
CGAL_In_place_list<*T,bool*>*:: const_iterator*

CGAL_In_place_list<*T,bool*>*:: value_type*
CGAL_In_place_list<*T,bool*>*:: reference*
CGAL_In_place_list<*T,bool*>*:: const_reference*

CGAL_In_place_list<T,bool>:: size_type
CGAL_In_place_list<T,bool>:: difference_type

CGAL_In_place_list<T,bool>:: reverse_iterator
CGAL_In_place_list<T,bool>:: const_reverse_iterator

Creation

CGAL_In_place_list<T,bool> l;	introduces an empty list.
CGAL_In_place_list<T,bool> l(list<T> l1);	copy constructor. Each item in *l1* is copied.
CGAL_In_place_list<T,bool> l(size_type n, T t = T());	introduces a list with n items, all initialized with copies of t.
template <class InputIterator> *CGAL_In_place_list<T,bool> l(InputIterator first, InputIterator last);*	a list with copies from the range [*first*,*last*).
CGAL_In_place_list<T,bool> l(const T first, const T* last);*	non-member-template version.

CGAL_In_place_list<T,bool> &	*l = l1*	assignment. Each item in *l1* is copied. Each item in *l* is deleted if the *bool* parameter is *true*.
void	*l.swap(l1)*	swaps the contents of *l* with *l1*.
void	*l.destroy()*	all items in *l* are deleted regardless of the *bool* parameter.

Comparison Operations

bool	*l == l1*	test for equality: Two lists are equal, iff they have the same size and if their corresponding elements are equal.
bool	*l < l1*	compares in lexicographical order.

Access Member Functions

iterator	*l.begin()*	returns a mutable iterator referring to the first element in *l*.
const_iterator	*l.begin() const*	returns a constant iterator referring to the first element in *l*.

iterator	*l.end()*	returns a mutable iterator which is the past-end-value of *l*.
const_iterator	*l.end() const*	returns a constant iterator which is the past-end-value of *l*.
bool	*l.empty()*	returns *true* if *l* is empty.
size_type	*l.size()*	returns the number of items in list *l*.
size_type	*l.max_size()*	returns the maximum possible size of the list *l*.
T&	*l.front()*	returns the first item in list *l*.
T&	*l.back()*	returns 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)*	
iterator	*l.insert(T* pos, T& t)*	inserts *t* in front of *pos*. The return value points to the inserted item.
void	*l.insert(iterator pos, size_type n, T t = T())*	
void	*l.insert(T* pos, size_type n, T t = T())*	inserts *n* copies of *t* in front of *pos*.

template <*class InputIterator*>
void *l.insert(iterator pos, InputIterator first, InputIterator last)*

template <*class InputIterator*>
void *l.insert(T* pos, InputIterator first, InputIterator last)*

inserts the range [*first, last*) in front of iterator *pos*.

As long as member templates are not supported, member functions using *T** instead of the general *InputIterator* are provided.

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(T* pos)*	removes the item from list *l*, where *pos* refers to.
void	*l.erase(iterator first, iterator last)*	
void	*l.erase(T* first, T* last)*	removes the items in the range [*first, last*) from *l*.

Special List Operations

void *l.splice(iterator pos,* & *x)*
void *l.splice(T* pos,* & *x)* inserts the list x before position *pos* and x becomes empty. It takes constant time.
Precondition: &*l* != &*x*.

void *l.splice(iterator pos,* & *x, iterator i)*
void *l.splice(T* pos,* & *x, T* i)*

inserts an element pointed to by i from list x before position *pos* and removes the element from x. It takes constant time. i is a valid dereferenceable iterator of x. The result is unchanged if $pos == i$ or $pos == ++i$.

void *l.splice(iterator pos,* & *x, iterator first, iterator last)*
void *l.splice(T* pos,* & *x, T* first, T* last)*

inserts elements in the range [*first, last*) before position *pos* and removes the elements from x. It takes constant time if &*x* == &*l*; otherwise, it takes linear time. [*first, last*) is a valid range in x.
Precondition: *pos* is not in the range [*first, last*).

void *l.remove(T value)* erases all elements e in the list l for which $e == value$. It is stable.
Precondition: a suitable *operator*== for the type T.

void *l.unique()* erases all but the first element from every consecutive group of equal elements in the list l.
Precondition: a suitable *operator*== for the type T.

void *l.merge(* & *x)* merges the list x into the list l and x becomes empty. It is stable.
Precondition: Both lists are increasingly sorted. A suitable *operator*< for the type T.

void *l.reverse()* reverses the order of the elements in l in linear time.

void *l.sort()* sorts the list l according to the *operator*< in time $O(n \log n)$ where $n = size()$. It is stable.
Precondition: a suitable *operator*< for the type T.

Example

```
/* in_list_prog.C       */
/* ---------------------- */
#include <CGAL/basic.h>
#include <assert.h>
#include <algobase.h>
#include <CGAL/In_place_list.h>

struct item :  public CGAL_In_place_list_base<item> {
    int key;
    item() {}
    item( const item& i) :  CGAL_In_place_list_base<item>(i), key(i.key) {}
    item( int i) :  key(i) {}
    bool operator== (const item& i) const { return key == i.key;}
    bool operator≠ (const item& i) const { return !  (*this == i);}
    bool operator== (int i) const         { return key == i;}
    bool operator≠ (int i) const         { return !  (*this == i);}
    bool operator<  (const item& i) const { return key < i.key;}
};

main() {
    typedef CGAL_In_place_list<item,true> List;
    List  l;
    item* p = new item(1);
    l.push_back(  *p);
    l.push_back(  *new item(2));
    l.push_front( *new item(3));
    l.push_front( *new item(4));
    l.push_front( *new item(2));
    List::iterator i = l.begin();
    ++i;
    l.insert(i,*new item(5));
    l.insert(p,*new item(5));
    int a[7] = {2,5,4,3,5,1,2};
    assert( equal( l.begin(), l.end(), a));
    #ifndef CGAL_CFG_NO_LAZY_INSTANTIATION
    l.sort();
    l.unique();
    int b[5] = {1,2,3,4,5};
    assert( l.size() == 5);
    assert( equal( l.begin(), l.end(), b));
    #endif
    return 0;
}
```

2.2 Generic Functions

2.2.1 Copy n Items

CGAL_copy_n() copies n items from an input iterator to an output iterator which is useful for possibly infinite sequences of random geometric objects[1].

#include <CGAL/copy_n.h>

template <class InputIterator, class Size, class OutputIterator>
OutputIterator CGAL_copy_n(InputIterator first, Size n, OutputIterator result)

copies the first n items from *first* to *result*. Returns the value of *result* after inserting the n items.

See Also

CGAL_Counting_iterator.

[1]The STL release June 13, 1997, from SGI has a new function *copy_n* which is equivalent with *CGAL_copy_n*.

2.3 Function Objects

Two kinds of function objects are provided: Projections and Creators.

2.3.1 Projection Function Objects

Definition

A *projection* function object o of type O has an unary parentheses operator expecting an argument of type *O::argument_type* as a reference or constant reference, and returns a reference or constant reference of type *O::result_type* respectively.

#include <*CGAL/function_objects.h*>

Operations

O::result_type&	*o.operator()(O::argument_type &) const*
const O::result_type&	*o.operator()(const O::argument_type &) const*

Creation

The following projection function objects are available:

CGAL_Identity<Value> o; — the identity function for projection objects. *argument_type* and *result_type* are equal to *Value*.

CGAL_Compose<Fct1, Fct2> o; — composes two projections: $Fct1 \circ Fct2 \circ x \equiv Fct1()(\ Fct2()(x))$. *argument_type* is equal to *Fct2::argument_type*, *result_type* to *Fct1::result_type*.

CGAL_Dereference<Value> o; — dereferences a pointer. *argument_type* is equal to *Value** and *result_type* is equal to *Value*.

CGAL_Get_address<Value> o; — Get the address for a reference. *argument_type* is equal to *Value* and *result_type* is equal to *Value**.

CGAL_Cast_function_object<Arg, Result> o;

applies a C-style type cast to its argument. *argument_type* is equal to *Arg* and *result_type* is equal to *Result*.

The following function objects are provided with respect to the polyhedral surfaces in the basic library.

CGAL_Project_vertex<Node> o; — calls the member function *vertex()* on an instance of type *Node*. *argument_type* is equal to *Node* and *result_type* is equal to *Node::Vertex*.

CGAL_Project_facet<*Node*> *o*; calls the member function *facet()* on an instance of type *Node*. *argument_type* is equal to *Node* and *result_type* is equal to *Node::Facet*.

CGAL_Project_point<*Node*> *o*; calls the member function *point()* on an instance of type *Node*. *argument_type* is equal to *Node* and *result_type* is equal to *Node::Point*.

CGAL_Project_normal<*Node*> *o*; calls the member function *normal()* on an instance of type *Node*. *argument_type* is equal to *Node* and *result_type* is equal to *Node::Normal*.

CGAL_Project_plane<*Node*> *o*; calls the member function *plane()* on an instance of type *Node*. *argument_type* is equal to *Node* and *result_type* is equal to *Node::Plane*.

CGAL_Project_next<*Node*> *o*; calls the member function *next()* on an instance of type *Node*. *argument_type* and *result_type* are equal to *Node**.

CGAL_Project_prev<*Node*> *o*; calls the member function *prev()* on an instance of type *Node*. *argument_type* and *result_type* are equal to *Node**.

CGAL_Project_next_opposite<*Node*> *o*;

calls the member functions *next()*->*opposite()* on an instance of type *Node*. *argument_type* and *result_type* are equal to *Node**.

CGAL_Project_opposite_prev<*Node*> *o*;

calls the member functions *opposite()*->*prev()* on an instance of type *Node*. *argument_type* and *result_type* are equal to *Node**.

2.3.2 Creator Function Objects

Definition

A *creator* function object *o* of type *O* has a parentheses operator which acts like a constructor. The operator might expect arguments and returns an instance of type *O::result_type*.

#include <*CGAL/function_objects.h*>

Operations

O::result_type *o.operator()(...) const*

Creation

The following creator function objects are available for one to five arguments:

CGAL_Creator_1<*Arg, Result*> *o*; applies the constructor *Result(Arg)*. *result_type* is equal to *Result*.

...

CGAL_Creator_5<*Arg1, Arg2, Arg3, Arg4, Arg5, Result*> *o*;

applies the constructor *Result(Arg1, Arg2, Arg3, Arg4, Arg5)*. *result_type* is equal to *Result*.

The following creator function objects are available for two to nine arguments (one argument is already captured with *CGAL_Creator_1*):

CGAL_Creator_uniform_2<*Arg, Result*> *o*;

applies the constructor *Result(Arg,Arg)*. *result_type* is equal to *Result*.

...

CGAL_Creator_uniform_9<*Arg, Result*> *o*;

applies the constructor for *Result* with nine arguments of type *Arg*. *result_type* is equal to *Result*.

See Also

CGAL_Join_input_iterator_1 ....

2.4 Classes for Composing Function Objects

Although not mentioned in the ANSI draft(CD2), most STL implementations provide two global functions for composing function objects, *compose1* and *compose2*, defined in function.h. Since for both the resulting function is unary, one can only construct unary function objects in this way. This seems to be quite a limitation, since many (also STL) algorithms can be parameterized with binary (e.g. comparison) functions.

In analogy to STL *compose1/2* we define two functions, *CGAL_compose1_2* and *CGAL_compose2_2*, that can be used to construct a binary function object via composition.

2.4.1 Adaptable Unary Function

Definition

A class *AdaptableUnaryFunction* is an adaptable unary function, if it defines the following types and operations.

Types

AdaptableUnaryFunction:: argument_type	the type of f's argument.
AdaptableUnaryFunction:: result_type	the type returned when f is called.

Creation

AdaptableUnaryFunction f;	default constructor.
AdaptableUnaryFunction f(f1);	copy constructor.

Operations

AdaptableUnaryFunction&	*f = f1*	assignment.
result_type	*f(argument_type a)*	function call.

See Also

STL Programmer's Guide.

2.4.2 Adaptable Binary Function

Definition

A class *AdaptableBinaryFunction* is an adaptable binary function, if it defines the following types and operations.

Types

AdaptableBinaryFunction:: first_argument_type	the type of f's first argument.
AdaptableBinaryFunction:: second_argument_type	the type of f's second argument.
AdaptableBinaryFunction:: result_type	the type returned when f is called.

Creation

AdaptableBinaryFunction f;	default constructor.
AdaptableBinaryFunction f(f1);	copy constructor.

Operations

AdaptableBinaryFunction&	*f = f1*	assignment.
result_type	*f (first_argument_type a, second_argument_type b)*	function call.

See Also

STL Programmer's Guide.

2.4.3 Composing Binary into Unary Functions

#include <CGAL/function_objects.h>

template < class Operation1, class Operation2 >
Composition CGAL_compose1_2(Operation1 op1, Operation2 op2)

Operation1 must be an adaptable unary function, *Operation2* an adaptable binary function and *Operation2::result_type* convertible to *Operation1::argument_type*. Then *CGAL_compose1_2* returns an adaptable binary function object c (of some complicated type *Composition*) such that

Operation1::result_type c(Operation2::first_argument_type x,
Operation2::second_argument_type y)

is equal to *Operation1()(Operation2()(x, y))*.

2.4.4 Composing Unary into Binary Functions

#include <CGAL/function_objects.h>

template < class Operation1, class Operation2, class Operation3 >
Composition CGAL_compose2_2(Operation1 op1,
Operation2 op2,
Operation3 op3)

Operation1 must be an adaptable binary function, *Operation2* and *Operation3* adaptable unary functions, *Operation2::result_type* convertible to *Operation1::first_argument_type* and *Operation3::result_type* convertible to *Operation1::second_argument_type*. Then *CGAL_compose2_2* returns an adaptable binary function object *c* (of some complicated type *Composition*) such that

Operation1::result_type *c(Operation2::first_argument_type x,*
Operation2::second_argument_type y)

is equal to *Operation1()(Operation2()(x), Operation3()(y))*.

2.5 Adaptor Classes around Iterators and Circulators

2.5.1 Counting Input Iterator Adaptor

Definition

The iterator adaptor *CGAL_Counting_iterator<Iterator, Value>* adds a counter to the internal iterator of type *Iterator* and defines equality of two instances in terms of this counter. It can be used to create finite sequences of possibly infinite sequences of values from input iterators. It complies itself to the requirements of input iterators.

#include <CGAL/Counting_iterator.h>

Creation

CGAL_Counting_iterator<Iterator, Value> i(Iterator i);

the join of a single iterator i. Applies *Creator* to each item read from i. *value_type* is equal to *Creator::result_type*.

CGAL_Counting_iterator<Iterator, Value> i(size_t n = 0);

initializes the internal counter to n and i has a singular value.

CGAL_Counting_iterator<Iterator, Value> i(Iterator j, size_t n = 0);

initializes the internal counter to n and i to j.

See Also

CGAL_copy_n.

2.5.2 N-Step Iterator or Circulator Adaptor

Definition

The adaptor *CGAL_N_step_adaptor<I,int N,Ref,Ptr,Val,Dst,Ctg>* changes the step width of the iterator or circulator class I to N. It is itself an iterator or circulator respectively. The value type is *Val* and the distance type is *Dst*. The iterator category is *Ctg*. For a mutable iterator, the parameters *Ref* and *Ptr* must be set to *Val&* and *Val** respectively. For a non-mutable iterator the types must be set to *const Val&* and *const Val** respectively. With iterator traits classes *Val*, *Dst* and *Ctg* will be superfluous.

The behavior is undefined if the adaptor is used on a range $[i, j)$ where $j - i$ is not a multiple of n.

Creation

CGAL_N_step_adaptor<I,int N,Ref,Ptr,Val,Dst,Ctg> i(I j); down cast.

Operations

The adaptor conforms to the iterator or circulator category stated with the parameter *Ctg*. *Precondition:* The iterator or circulator I must be at least of this category.

Implementation

For compiler with difficulties with implicit instantiation (the `CGAL_CFG_NO_LAZY_INSTANTIATION` flag) the adaptor is predefined to work only for bidirectional iterators.

2.5.3 Joining Input Iterator Streams

Definition

A join of n input iterators and a creator function object is again an input iterator which reads an object from each stream and applies the creator function object on them whenever it advances.

#include <CGAL/Join_input_iterator.h>

Creation

The following join input iterator objects are available for one to five iterators:

CGAL_Join_input_iterator_1<Iterator, Creator> join(Iterator i);

the join of a single iterator i. Applies *Creator* to each item read from i. *value_type* is equal to *Creator::result_type*.

...

CGAL_Join_input_iterator_5<I1, I2, I3, I4, I5, Creator> join(I1 i1, I2 i2, I3 i3, I4 i4, I5 i5);

the join of five iterators *i1* to *i5*. Applies *Creator* with five arguments **i1* up to **i5*. *value_type* is equal to *Creator::result_type*.

See Also

CGAL_Creator_1 ... and *CGAL_Creator_uniform_2* ....

2.5.4 An Inverse Index for Iterators and Circulators

Definition

The class *CGAL_Inverse_index<IC>* constructs an inverse index for a given range $[i, j)$ of two iterators or circulators of type *IC*. The first element I in the range $[i, j)$ has the index 0. Consecutive elements are numbered incrementally. The inverse index provides a query for a given iterator or circulator k to retrieve its index number. *Precondition:* The iterator or circulator must be either of the random access category or the dereference operator must return stable and distinguishable addresses for the values, e.g. proxies or non-modifiable iterator with opaque values will not work.

Creation

#include <CGAL/Inverse_index.h>

CGAL_Inverse_index<IC>	*inverse;*	invalid index.
CGAL_Inverse_index<IC>	*inverse(IC i);*	empty inverse index initialized to start at i.
CGAL_Inverse_index<IC>	*inverse(IC i, IC j);*	inverse index initialized with range $[i, j)$.

Operations

size_t	*inverse[IC k]*	returns inverse index of k. *Precondition:* k has been stored in the inverse index.
void	*inverse.push_back(IC k)*	adds k at the end of the indices.

Implementation

For random access iterators or circulators, it is done in constant time by subtracting i. For other iterator categories, an STL *map* is used, which results in a $\log j - i$ query time. The comparisons are done using the operator *operator<* for *const T**.

See Also

CGAL_Random_access_adaptor and *CGAL_Random_access_value_adaptor*.

2.5.5 A Random Access Adaptor for Iterators and Circulators

Definition

The class *CGAL_Random_access_adaptor<IC,Dist>* provides a random access for data structures. Either the data structure supports random access iterators or circulators where this class maps function calls to the iterator or circulator, or a STL *vector* is used to provide the random access. The iterator or circulator of the data structure are of type *IC*. Their distance type is *Dist*.

Creation

#include <CGAL/Random_access_adaptor.h>

CGAL_Random_access_adaptor<IC,Dist>	*random_access;*	invalid index.
CGAL_Random_access_adaptor<IC,Dist>	*random_access(IC i);*	empty random access index initialized to start at i.
CGAL_Random_access_adaptor<IC,Dist>	*random_access(IC i, IC j);*	random access index initialized to the range $[i, j)$.
void	*random_access.reserve(Dist r)*	reserve r entries, if a *vector* is used internally.

Operations

IC	*random_access[Dist n]*	returns iterator or circulator to the n-th item. *Precondition*: $n <$ number of items in *random_access*.
void	*random_access.push_back(IC k)*	adds k at the end of the indices.

See Also

CGAL_Inverse_index and *CGAL_Random_access_value_adaptor*.

2.5.6 A Random Access Value Adaptor for Iterators and Circulators

Definition

The class *CGAL_Random_access_value_adaptor<IC,T,Dist>* provides a random access for data structures. It is derived from *CGAL_Random_access_adaptor<IC,Dist>*. Instead of returning iterators from the *operator[]* methods, it returns the dereferenced value of the iterator. The iterator or circulator of the data structure are of type *IC*. Their value type is T and the distance type is *Dist*.

#include <CGAL/Random_access_value_adaptor.h>

Operations

Creation and operations see *CGAL_Random_access_adaptor<IC,Dist>* above, with the exception of:

T&	*random_access[Dist n]*	returns a reference to the n-th item. *Precondition*: $n <$ number of items in *random_access*.

See Also

CGAL_Inverse_index and *CGAL_Random_access_adaptor*.

Chapter 3

Circulators

Circulators are quite similar to iterators in the Standard Template Library STL [C++96, MS96, SL95]. 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.

An introduction to the requirements for circulators is given here. Thereafter a couple of adaptors are presented that convert between iterators and circulators. A few useful functions for circulators follow. Section 3.6 discusses the design decisions taken and Section 3.7 presents the full requirements for circulators which are needed to implement own circulators. A few base classes and circulator tags (like iterator tags [SL95]) are provided. An advanced topic is in Section 3.9 the support for algorithms which work for iterators as well as for circulators.

3.1 Introduction

Iterators in the STL were tailored for linear sequences. The specialization for circular data structures leads to slightly different requirements which we will call *circulators*. The main difference is that a circular data structure has no natural past-the-end value. In consequence, a container supporting circulators will not have an `end()`-member function. The semantic of a circulator range differs: 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 circulator requirements: A comparison c == `NULL` for a circulator c is true for an empty sequence. The following example demonstrates a typical use of circulators. It assumes a CGAL polygon given in the variable `P` of type `Polygon_2` (provided with a `typedef`) and writes all edges to `cout`. Note that the polygon provides iterators over edges as well as the circulator. The benefit of circulators appears in more graph-like data structures as discussed in Section 3.6. Examples are triangulations and polyhedrons.

```
if ( P.edges_circulator() ≠ NULL) {
    Polygon_2::Edge_const_circulator  c = P.edges_circulator();
    do {
        cout ≪ *c;
    } while ( ++c ≠ P.edges_circulator());
}
```

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 from 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, see Chapter 3.9.

Requirements: A class is said to be a circulator if it fulfills a set of requirements. In the following subsections we give a summary for the requirements of the three different circulator categories. The Section 3.7 presents a more formal definition. Note that the stated return values are not required, only a return value that is convertible to the stated type is required.

3.1.1 Forward Circulator (*Circulator*)

Definition

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

Types

Circulator:: value_type	the value type T.
Circulator:: reference	a reference to T.
Circulator:: const_reference	a const reference to T.
Circulator:: pointer	a pointer to T.
Circulator:: const_pointer	a const pointer to T.
Circulator:: size_type	unsigned integral type that can hold the size of the data structure.
Circulator:: difference_type	signed integral type that can hold the distance between two circulators from the same data structure.
Circulator:: iterator_category	the circulator category *CGAL_Forward_circulator_tag*.

Creation

Circulator c;	a circulator equal to *NULL*, i.e. a singular value.
Circulator c(d);	a circulator equal to d.

Operations

bool	$c = d$	Assignment.
bool	$c == NULL$	Test for emptiness.
bool	$c != NULL$	Test for non-emptiness. The result is the same as *!(c == NULL)*.
bool	$c == d$	Test for equality: Two circulators are equal if they refer to the same item.
bool	$c != d$	Test for inequality. The result is the same as *!(c == d)*.
reference	$*c$	Returns the value of the circulator. If *Circulator* is mutable **c = t* is valid. *Precondition*: c is dereferenceable.
pointer	$c ->$	Returns a pointer to the value of the circulator. *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;* .

3.1.2 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.

Types

Circulator:: iterator_category	the circulator category *CGAL_Bidirectional_circulator_tag*.

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;* .

3.1.3 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.

Types

Circulator:: iterator_category	the circulator category *CGAL_Random_access_circulator_tag*.

Operations

Circulator&	$c +=$ *difference_type* n	The result is the same as if the prefix increment operation was applied n times, but it is computed in constant time.
Circulator	$c +$ *difference_type* n	Same as above, but returns a new circulator.
Circulator	*difference_type* $n + c$	Same as above.
Circulator&	$c -=$ *difference_type* n	The result is the same as if the prefix decrement operation was applied n times, but it is computed in constant time.
Circulator	$c -$ *difference_type* n	Same as above, but returns a new circulator.
reference	$c[$ *difference_type* $n]$	Returns **(c + n)*.
difference_type	$c -$ *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 $c - d_{\min}$ to all other circulators c is non-negative.
Circulator	*c.min_circulator()*	Returns the minimal circulator $c_{\min}$ (see previous operation) in constant time. If c has a singular value, a singular value is returned.

There are no comparison operators required.

3.2 Adaptor: Container with Iterators from Circulator

Algorithms working on iterators could not be applied to circulators in full generality, only to subranges (see the warning in Section 3.1). The following adaptors convert circulators to iterators (with the unavoidable space and time penalty) to reestablish this generality.

Definition

The adaptor *CGAL_Forward_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 forward iterators. By 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.

For *CGAL_Forward_container_from_circulator<C>* the circulator has to fulfill at least the requirements for a forward circulator. The similar *CGAL_Bidirectional_container_from_circulator<C>* adaptor requires a bidirectional circulator in order to provide bidirectional iterators and the adaptor *CGAL_Random_access_container_from_circulator<C>* requires a random access circulator to provide random access iterators. In this case the adaptor implements a total ordering relation that is currently not required for random access circulators.

#include <CGAL/circulator.h>

Types

The types shown here are similar for all container-from-circulator adaptors.

typedef C Circulator;
CGAL_Forward_container_from_circulator<C>:: iterator
CGAL_Forward_container_from_circulator<C>:: const_iterator
CGAL_Forward_container_from_circulator<C>:: value_type
CGAL_Forward_container_from_circulator<C>:: reference
CGAL_Forward_container_from_circulator<C>:: const_reference
CGAL_Forward_container_from_circulator<C>:: pointer
CGAL_Forward_container_from_circulator<C>:: const_pointer
CGAL_Forward_container_from_circulator<C>:: size_type
CGAL_Forward_container_from_circulator<C>:: difference_type

Creation

CGAL_Forward_container_from_circulator<C> container;

the resulting iterators will have a singular value.

CGAL_Forward_container_from_circulator<C> container(C c);

the resulting iterators will have a singular value if the circulator c is singular.

The bidirectional and random access container have similar constructors. The default construction is shown here to present the names.

CGAL_Bidirectional_container_from_circulator<C> container;
CGAL_Random_access_container_from_circulator<C> container;

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.

See Also

CGAL_Forward_circulator_from_iterator, *CGAL_Forward_circulator_from_container*.

Example

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

```
Circulator c;  /* c is assumed to be a bidirectional circulator. */
CGAL_Bidirectional_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, which is here still possible in a defined manner. However, for random access iterators it is not yet supported.

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 as well as for the family of the add-operators where a possible wrap around might occur (this feature beyond the iterator requirements is not supported for the moment). 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 3.1.3.

These adaptor cannot be used for non-mutable circulators if the C++ compiler has the `CGAL_CFG_NO_LAZY_INSTANTIATION` bug[1].

[1]The g++ accepts non-mutable circulators with some warnings even though it has this bug.

3.3 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. Adaptors for iterator pairs are described here, adaptors for container classes are described in the next section.

Definition

The adaptor *CGAL_Forward_circulator_from_iterator<I,T,Size,Dist>* converts two iterators, a begin and a past-the-end value, to a forward circulator. The iterators are supposed to be at least forward iterators. The adaptor *CGAL_Bidirectional_circulator_from_iterator< I, T, Size, Dist>* converts bidirectional iterators and *CGAL_Random_access_circulator_from_iterator< I, T, Size, Dist>* random access iterators. Appropriate const circulators are also available.

I is the appropriate iterator type, *T* its value type, *Size* the unsigned integral value to hold the possible number of items in a sequence, and *Dist* is a signed integral value, the distance type between two iterators of the same sequence.

#include <CGAL/circulator.h>

Types

typedef I iterator;

In addition all types required for circulators are provided.

Creation

CGAL_Forward_circulator_from_iterator<I,T,Size,Dist> c;
a circulator *c* with a singular value.

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

CGAL_Forward_circulator_from_iterator<I,T,Size,Dist> c(d, I cur);
a copy of circulator *d* referring to the element **cur*. The circulator *c* contains a singular value if *d* does so.

The bidirectional, random access and const circulators have similar constructors. The default construction is shown here to present the names.

CGAL_Forward_const_circulator_from_iterator< I, T, Size, Dist> c;
CGAL_Bidirectional_circulator_from_iterator< I, T, Size, Dist> c;
CGAL_Bidirectional_const_circulator_from_iterator< I, T, Size, Dist> c;
CGAL_Random_access_circulator_from_iterator< I, T, Size, Dist> c;
CGAL_Random_access_const_circulator_from_iterator< I, T, Size, Dist> c;

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.

See Also

CGAL_Forward_container_from_circulator, *CGAL_Forward_circulator_from_container*.

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.

```
/* circulator_prog1.C     */
/* ------------------- */
#include <CGAL/basic.h>
#include <assert.h>
#include <vector.h>
#include <algo.h>
#include <CGAL/circulator.h>

typedef  vector<int>::iterator         I;
typedef  vector<int>::value_type       V;
typedef  vector<int>::size_type        S;
typedef  vector<int>::difference_type  D;
typedef  CGAL_Random_access_circulator_from_iterator<I,V,S,D> Circulator;
typedef  CGAL_Random_access_container_from_circulator<Circulator> Container;
typedef  Container::iterator Iterator;

int main() {
    vector<int> v;
    v.push_back(5);
    v.push_back(2);
    v.push_back(9);
    Circulator c( v.begin(), v.end());
    Container  container( c);
    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;
}
```

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 *CGAL_Random_access_circulator_from_iterator*`<` *T*, T, size_t, ptrdiff_t*`>` *c(b,e)* is a random circulator *c* over this array.

3.4 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. Adaptors for iterator pairs are described in the previous section, adaptors for container classes are described here.

Definition

The adaptor *CGAL_Forward_circulator_from_container*<*C*> provides a forward circulator for a container C as specified by the STL. The iterator belonging to the container C is supposed to be at least a forward iterator. The adaptor *CGAL_Bidirectional_circulator_from_container*<*C*> converts bidirectional iterators and *CGAL_Random_access_circulator_from_container*<*C*> random access iterators. Appropriate const circulators are also available.

C is the container type. The container 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 *value_type*, *size_type()*, and *difference_type*).

#include <*CGAL/circulator.h*>

Types

typedef C	*Container;*
typedef C::iterator	*iterator;*
typedef C::const_iterator	*const_iterator;*

In addition all types required for circulators are provided.

Creation

CGAL_Forward_circulator_from_container<*C*> *c*;
: a circulator c with a singular value.

CGAL_Forward_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.

CGAL_Forward_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*.

CGAL_Forward_const_circulator_from_container<*C*> *c*;
: a const circulator c with a singular value.

CGAL_Forward_const_circulator_from_container<*C*> *c(const C* container);*
: a const 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.

CGAL_Forward_const_circulator_from_container<*C*> *c(const C* container, C::const_iterator i);*
: a const circulator c initialized to refer to the element **i* in *container*.
Precondition: **i* is dereferenceable and refers to the *container*.

The bidirectional and random access circulators have similar constructors. The default construction is shown here to present the adaptor names.

CGAL_Bidirectional_circulator_from_container<*C*> *c*;
CGAL_Bidirectional_const_circulator_from_container<*C*> *c*;
CGAL_Random_access_circulator_from_container<*C*> *c*;
CGAL_Random_access_const_circulator_from_container<*C*> *c*;

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.

See Also

CGAL_Forward_container_from_circulator, *CGAL_Forward_circulator_from_iterator*.

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.

```
/* circulator_prog2.C      */
/* ------------------------ */
#include <CGAL/basic.h>
#include <assert.h>
#include <vector.h>
#include <algo.h>
#include <CGAL/circulator.h>

typedef CGAL_Random_access_circulator_from_container< vector<int> >  Circulator;
typedef CGAL_Random_access_container_from_circulator<Circulator> Container;
typedef Container::iterator Iterator;

int main() {
    vector<int> v;
    v.push_back(5);
    v.push_back(2);
    v.push_back(9);
    Circulator c( &v);
    Container  container( c);
    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;
}
```

3.5 Functions for Circulators

The size of a circulator is the size of the data structure it refers to. It is zero for a circulator with singular value. The size can be computed in linear time for forward and bidirectional circulators, and in constant time for random access circulators using the minimal circulator. The function *CGAL_circulator_size(c)* returns the circulator size. It uses the *c.min_circulator()* function if c is a random access circulator.

template <*class C*>
C::size_type *CGAL_circulator_size(C c)*

The distance of a circulator c to a circulator d is the number of elements in the range $[c, d)$. It is defined to be zero for a singular circulator and it returns the size of the data structure when applied to a range of the form $[c, c)$.

template <*class C*>
C::difference_type *CGAL_circulator_distance(C c, C d)*

The following function returns the distance between either two iterators or two circulators. The return type is `ptrdiff_t` for compilers not supporting iterator traits yet.

template <*class IC*>
iterator_traits<*IC*>*::difference_type*

CGAL_iterator_distance(IC ic1, IC ic2)

See Also

CGAL_is_empty_range

3.6 Design Rationale

For circular data structures it is not straightforward to provide a *begin(),end()*-pair of STL iterators. There is no natural past-the-end situation. Additional bookkeeping must be implemented to provide an *end()*-iterator or an arbitrary sentinel must be introduced, which is undesirable since it would break the symmetry of otherwise symmetric data structure. Examples in CGAL are triangulations, planar maps and polyhedrons where a natural circular ordering of incident vertices, edges, and faces around a vertex – or around a facet – exists. To overcome the performance and space drawback of the STL iterator requirements, we have introduced *circulators*, and to obtain STL compliance, we provide adaptors between circulators and iterators, see Section 3.2 up to Section 3.4. Hence, circulators are designed as the natural light-weight objects to traverse ring-like data structures, but algorithms working on iterators can also be applied using the adaptors.

The following example shows a template function that accepts a range $[c, d)$ of circulators and process each element in this range:

```
template <class Circulator>
void example( Circulator c, Circulator d) {
    if (c ≠ NULL) { /* at least one element in the range */
        do {
            foo(*c);  /* process element */
        } while (++c ≠ d);
    }
}
```

If the ring-like data structure is known to contain at least one element, the test `if(c != NULL)` can be omitted. This test for an empty data structure has been chosen such that it looks similar to a typical ring implementation using pointers to structs. However, other requirements for circulators, like the *operator++*, will force any implementation of circulators to use classes and operator overloading.

A serious design problem arises due to the fact that the STL template algorithms do not check whether their parameters fulfill the requirements of an iterator or not. Since the circulators are quite similar, they can be used in STL algorithms as iterators, but behave different. A range $[c, c)$ for a circulator C with non singular value denotes the whole sequence, but the iterator algorithm will treat it as an empty sequence. We know four solutions to cope with this problem, neither is fully satisfactory.

One possibility is to choose different signatures for circulators than for iterators. The *operator++* or the *operator==* are the only choices since they are the member functions involved in the semantic of ranges. Both are natural choices and desirable for circulators as well as for iterators. Specific member functions are harder to learn and to remember. The second possibility is to change the semantics of circulators to behave correct in algorithms using iterators, but than they are full-fledged iterators and the idea about light-weight objects and efficiency for circulators is superfluous. The third possibility is to protect the algorithms using iterators against misuse via circulators which is easy to achieve, see the *CGAL_Assert_iterator()* function below in Section 3.8. Nonetheless, this checking has also to be done in the STL library and this is beyond the scope of CGAL. For implementations within CGAL it is recommended. The fourth alternative is to document this design problem and state it as a feature. In fact it is a feature that each range $[c, d)$ of two distinct circulators c and d (a subsequence within a circular sequence) is a valid iterator range where one can apply STL algorithms to. The risk that one applies an algorithm to the whole sequence by accident, where the STL algorithms will see an empty range, is not too likely, since the normal use of iterators from container classes like `sort( A.begin(), A.end())` will fail for circulators due to the missing member function `end()`. We have chosen this (fourth) solution and documented it in the manual where appropriate. Following Bjarne Stroustrup: *Finally, it has been a guideline in the design of C++ that when all is said and done the programmer must be trusted.* [Str97]

3.7 Requirements

Similar to STL iterators, we distinguish between forward, bidirectional, and random access circulators. Input circulators are a contradiction, since any circulator is supposed to return once to itself. Output circulators are not supported since they would be indistinguishable from output iterators. Most requirements for circulators are the same as those for iterators. We present the changes, please refer to [MS96, chapter 18] or [C++96, SL95] for the iterator requirements.

Past-the-end value: There is no past-the-end value. On the other hand, a circulator can denote an empty data structure, but not with a past-the-end value.

Singular values: In addition to iterators, a circulator can denote an empty data structure. In this case it has a singular value.

Dereferenceable values: A circulator with a non-singular value is always dereferenceable.

Reachability: In contrast to iterators, dereferenceable circulators can reach itself with a finite and non-empty sequence of applications of *operator++*.

Ranges: In addition, any range $[c, c)$ of a circulator C is a valid range. If the circulator has a singular value, the range $[c, c)$ denotes the empty range, otherwise the circulator is dereferenceable and the range $[c, c)$ denotes the whole sequence of elements in the data structure. *Remark:* When a circulator is used in a place of an iterator, like with an STL algorithm, it will work as supposed to with the only exception that the range $[c, c)$ denotes always the empty range. This is not a requirement itself, it is a consequence of the requirements stated here and the fact that the STL requirements for iterators do not include a type safety check for the template parameters used in the related STL algorithms.

Types: Since there is no builtin type that can fulfill the requirements for a circulator we can assume that any circulator is implemented as a class and we can use local type declarations to add the useful type information about the value type and others to the circulators. For a circulator class C these are *C::value_type*, *C::reference*, *C::const_reference*, and *pointer* denoting the element type, references, and pointers to them. *C::size_type* is an unsigned integral type that can hold the size of a sequence. *C::difference_type* is a signed integral type that can hold the distance between two circulators of the same sequence (either by counting the elements inbetween or by subtracting two random access circulators).

STL **compliance:** The functions *iterator_category(...)*, *value_type(...)*, and *distance_type(...)* are required for STL compliance [SL95]. Beyond the STL requirements is the additionally required function *CGAL_query_circulator_or_iterator(...)* that distinguishes between circulators and iterators.

For each circulator type `C` we will assume that `a` and `b` denote values of type `C`, `r` denotes a value of `C&` (is assignable), and `t` denotes a value type `T`. Let `D` be the distance type.

Forward Circulators

`C()`	a circulator equal to *NULL*, i.e. a singular value.
`a == NULL`	Returns `true` if `a` has a singular value, `false` otherwise. For simplicity, `NULL == a` is not provided. Implementation issues might fail in detecting comparisons to other values that are not equal to `NULL`. Here, the behavior is undefined, a runtime assertion is recommended.
`a != NULL`	Returns `!(a == NULL)`.
`++r`	Like for forward iterators, but a dereferenceable circulator `r` will always be dereferenceable after `++r` (no past-the-end value). *Precondition:* `r` has

	no singular value.
`r++`	Same as for `++r`.
`C::value_type`	`T`.
`C::reference`	`T&`.
`C::const_reference`	`const T&`.
`C::pointer`	`T*`.
`C::const_pointer`	`const T*`.
`C::size_type`	unsigned integral type.
`C::distance_type`	signed integral type.
`C::iterator_category`	circulator category `CBP_Forward_circulator_tag`.

CGAL_query_circulator_or_iterator(a)	returns *CGAL_Circulator_tag()*.
iterator_category(a)	returns *C::iterator_category()*.
value_type(a)	returns *(T*)(0)*.
distance_type(a)	returns *(D*)(0)*.

Bidirectional Circulators

The same requirements as for the forward circulators hold for bidirectional iterators with the following change:

`C::iterator_category`	circulator category `CBP_Bidirectional_circulator_tag`.

Random Access Circulators

The same requirements as for the bidirectional circulators hold for random access iterators with the following changes and extensions.

The idea of a random access extends naturally to circulators using equivalence classes modulus the length of the sequence. With this in mind, the additional requirements for random access iterators hold also for random access circulators. The single exception is that the random access iterator requires a total order on the sequence, which a circulator cannot provide. One might define the ordering by splitting the circle at a fixed point, e.g. the start circulator provided from the data structure. This is what the adaptor to iterators will do. Nonetheless, we do not require this for circulators.

The difference of two circulators is not unique as for iterators. A reasonable requirement demands that the result is in a certain range $[1 - \mathit{size},\ \mathit{size} - 1]$, where *size* is the size of the data structure, and that whenever a circulator `a` is fixed that the differences with all other circulators of the sequence form a consistent ordering.

For the adaptor to iterators there has to be a minimal circulator $d_{\min}$ for which the difference $c - d_{\min}$ to all other circulators c is non negative.

`C::iterator_category`	circulator category `CBP_Random_access_circulator_tag`.
`b - a`	a limited range and a consistent ordering for a fixed circulator `a` as explained in the paragraph above.
`a.min_circulator()`	returns the minimal circulator from the range $[a, a)$. Its value is singular if `a` has a singular value.

Const Circulators

As with iterators we distinguish between circulators and const circulators. The expression `*a = t` is valid for mutable circulators. It is invalid for const circulators.

Circulators in Container Classes

For a container `x` of class `X` that supports circulators `c` we would like to recommend the following requirements.

`X::Circulator`	the type of the mutable circulator.
`X::Const_circulator`	the type of the const circulator.
`c = x.begin()`	the start point of the data structure. It can be a singular value. It is of type `X::Circulator` for a mutable container or `X::Const_circulator` for a const container. There must not be an `end()` member function.

If a container will support iterators and circulators, the member function `circulator_begin()` is proposed. However, the support of iterators and circulators simultaneously is not recommended, since it would lead to fat interfaces. The natural choice should be supported, the other technique will be available through the adaptors in Section 3.2 up to Section 3.4.

3.8 Compile Time Tags and Base Classes

This section demonstrates how to distinguish in algorithms between different categories of circulators and how to distinguish these from iterators. We use compile time tags for this purpose. They are described as iterator tags in [SL95]. Additionally, a couple of base classes simplify the task of writing own circulators that conform to the mechanism described here which is also part of the requirements: A circulator c has to have an appropriately defined *iterator_category(c)* function and *CGAL_query_circulator_or_iterator(c)* function, see also Section 3.7. All adaptors described in this chapter use these base classes.

#include <*CGAL/circulator.h*>

To use the tags or base classes it is sufficient to include:

#include <*CGAL/circulator_bases.h*>

Compile Time Tags

struct CGAL_Circulator_tag {};	any circulator.
struct CGAL_Iterator_tag {};	any iterator.

Base Classes

template <*class T, class Dist, class Size*> *struct CGAL_Forward_circulator_base* {};
template <*class T, class Dist, class Size*> *struct CGAL_Bidirectional_circulator_base* {};
template <*class T, class Dist, class Size*> *struct CGAL_Random_access_circulator_base* {};

Discriminating Function for Iterator Categories

To distinguish between circulator categories the *iterator_category()* function is sufficient. It is overloaded for the above base classes as follows. (The return values are in fact class derived from the *iterator_tag* types stated here, see the requirements in the previous section. However, to distinguish circulator categories the *iterator_tag* types are sufficient, see also the example below.)

template <class T, class Dst, class Size>
forward_iterator_tag *iterator_category(CGAL_Forward_circulator_base<T,Dst,Size>)*

template <class T, class Dst, class Size>
bidirectional_iterator_tag *iterator_category(CGAL_Bidirectional_circulator_base<T,Dst,Size>)*

template <class T, class Dst, class Size>
random_access_iterator_tag *iterator_category(CGAL_Random_access_circulator_base<T,Dst,Size>)*

Discriminating Function between Circulators and Iterators

The following function distinguishes between circulators and iterators (assuming that the iterators do also conform to the iterator tag description in [SL95]).

template <class T, class Dist, class Size>
CGAL_Circulator_tag *CGAL_query_circulator_or_iterator(*
CGAL_Forward_circulator_base<T,Dist,Size>)

template <class T, class Dist, class Size>
CGAL_Circulator_tag *CGAL_query_circulator_or_iterator(*
CGAL_Bidirectional_circulator_base<T,Dist,Size>)

template <class T, class Dist, class Size>
CGAL_Circulator_tag *CGAL_query_circulator_or_iterator(*
CGAL_Random_access_circulator_base<T,Dist,Size>)

template <class T, class Dist>
CGAL_Iterator_tag *CGAL_query_circulator_or_iterator(input_iterator<T,Dist>)*

CGAL_Iterator_tag *CGAL_query_circulator_or_iterator(output_iterator)*

template <class T, class Dist>
CGAL_Iterator_tag *CGAL_query_circulator_or_iterator(forward_iterator<T,Dist>)*

template <class T, class Dist>
CGAL_Iterator_tag *CGAL_query_circulator_or_iterator(bidirectional_iterator<T,Dist>)*

template <class T, class Dist>
CGAL_Iterator_tag CGAL_query_circulator_or_iterator(random_access_iterator<T,Dist>)

template <class T>
CGAL_Iterator_tag CGAL_query_circulator_or_iterator(const T)*

Value and Distance Type Querying

template <class T, class Dist, class Size>
T value_type(CGAL_Forward_circulator_base<T,Dist,Size>)*
template <class T, class Dist, class Size>
T value_type(CGAL_Bidirectional_circulator_base<T,Dist,Size>)*
template <class T, class Dist, class Size>
T value_type(CGAL_Random_access_circulator_base<T,Dist,Size>)*

template <class T, class Dist, class Size>
Dist distance_type(CGAL_Forward_circulator_base<T,Dist,Size>)*
template <class T, class Dist, class Size>
Dist distance_type(CGAL_Bidirectional_circulator_base<T,Dist,Size>)*
template <class T, class Dist, class Size>
Dist distance_type(CGAL_Random_access_circulator_base<T,Dist,Size>)*

Compile Time Tag Assertions

The following assertions check during the compilation if their argument is of the kind as stated in the assertions name, i.e. a circulator, an iterator, or of a particular category, applicable for an iterator or a circulator. Note that no input or output circulator exists.

template <class C> void CGAL_Assert_circulator(C c)
template <class I> void CGAL_Assert_iterator(I i)
template <class I> void CGAL_Assert_input_category(I i)
template <class I> void CGAL_Assert_output_category(I i)
template <class IC> void CGAL_Assert_forward_category(IC ic)
template <class IC> void CGAL_Assert_bidirectional_category(IC ic)
template <class IC> void CGAL_Assert_random_access_category(IC ic)

Example

The above declarations can be used to distinguish between iterators and circulators and between different circulator categories. The assertions can be used to protect a templatized algorithm against instantiations that do not fulfill the requirements. The following example program demonstrate both. The program is part of the CGAL distribution.

```
/* circulator_prog3.C      */
/* ------------------------ */
#include <CGAL/basic.h>
#include <assert.h>
#include <list.h>
#include <CGAL/circulator.h>

template <class C> inline
int foo( C c, forward_iterator_tag) {
```

```
    CGAL_Assert_circulator( c);
    CGAL_Assert_forward_category( c);
    return 1;
}
template <class C> inline
int foo( C c, random_access_iterator_tag) {
    CGAL_Assert_circulator( c);
    CGAL_Assert_random_access_category( c);
    return 2;
}
template <class I> inline
int foo( I i, CGAL_Iterator_tag) {
    CGAL_Assert_iterator( i);
    return 3;
}

template <class C> inline
int foo( C c, CGAL_Circulator_tag) {
    CGAL_Assert_circulator( c);
    return foo( c, iterator_category(c));
}
template <class IC> inline
int foo( IC ic) {
    return foo( ic, CGAL_query_circulator_or_iterator( ic));
}

int main() {
    typedef CGAL_Forward_circulator_base<int, ptrdiff_t, size_t> F;
    typedef CGAL_Random_access_circulator_base<int, ptrdiff_t, size_t> R;
    F f = F();
    R r = R();
    list<int> l;
    assert( foo( f)         == 1);
    assert( foo( r)         == 2);
#ifndef __GNUG__
    assert( foo( l.begin()) == 3);
#endif
    return 0;
}
```

Implementation

Since not all current compilers can eliminate the space needed for the compile time tags even when deriving from them, we implement a variant for each base class that contains a protected *void** data member called *_ptr*. Here, the allocated space in the derived classes can be reused. The discriminating functions are overloaded for these extensions. The variant base classes are:

template <class T, class Dist, class Size> *class CGAL_Forward_circulator_ptrbase {};*	forward circulator.
template <class T, class Dist, class Size> *class CGAL_Bidirectional_circulator_ptrbase {};*	bidirectional circulator.
template <class T, class Dist, class Size> *class CGAL_Random_access_circulator_ptrbase {};*	random access circulator.

The 1996 release of the STL that comes with the SGI C++ compiler does not derive their iterators from the base classes as described in [SL95]. The *CGAL_query_circulator_or_iterator* function needs to be overloaded for several iterators in the STL explicitly. For that, we must know the classes involved in advance. We do not like to include all STL header files that might be necessary (i.e. `deque.h`, `hashtable.h`, `list.h`, and `tree.h`). Instead, we assume that these header files are already included and define the additional functions if necessary. This makes this scheme dependent on the order of header file inclusions, but we consider it as reasonable to assume that standard header files are included before CGAL header files.

SGI has released newer STL implementations on their web server `http://www.sgi.com/Technology/STL/`. These releases do not longer implement iterators as nested classes which allow us to use forward declarations to stay independent from the actual header file inclusions.

The current scheme is prepared for iterator traits [Mye95, C++96] and will simplify considerably when they are available.

The Gnu g++ 2.7.2 compiler has difficulties with the *iterator_category(i)* function implementation. The workaround for the Gnu library implements the function within the container classes. Since the *CGAL_query_circulator_or_iterator(c)* function relies on the same principle, it could not be implemented without patching the Gnu library. The problem will be solved with the next Gnu g++ release. The *CGAL_Assert_iterator(i)* and *CGAL_Assert_circulator_or_iterator(ic)* wont work either. The involved container are *list* and *tree*. The other container should work.

3.9 Writing Algorithms for Circulators and Iterators Simultaneously

The previous section describes how we can distinguish between circulators and iterators. Now we encapsulate the difference between both in such a way that it is easy to write algorithms that can be parameterized with an interval according to both requirements. The difference we have to consider is whether a loop will be entered the first time or not. For iterators it is the same test as for the rest of the loop, for circulators it is the comparison with `NULL`. So, we provide a test that accepts a range of either two circulators or two iterators and decides whether the range is empty or not.

#include <*CGAL/circulator.h*>

template< *class IC*>

bool *CGAL_is_empty_range(IC i, IC j)* — is *true* if the range $[i, j)$ is empty, *false* otherwise. *Precondition*: *IC* is either a circulator or an iterator type. The range $[i, j)$ is valid.

Using this function we can write an `example()` function that accepts a range $[i, j)$ of iterators or circulators and process each element in this range:

```
template <class IC>
void example( IC i, IC j) {
    // assures that there is at least one element
    if (! CGAL_is_empty_range( i, j)) {
        do {
```

```
            foo(*i);  // process element
        } while (++i != j);
    }
}
```

Again, a protection against misuse with inappropriate template parameters is possible.

template< *class IC*>
void *CGAL_Assert_circulator_or_iterator(IC i)* results in a compile time error when i is neither a circulator nor an iterator.

Loop Macro

A macro *CGAL_For_all(i, j)* simplifies the writing of such simple loops as the one above. i and j can be either iterators or circulators. The macro loops through the range $[i, j)$. It increments i until it reaches j. The implementation looks like:

CGAL_For_all(i,j) ≡
```
for ( bool CGAL__circ_loop_flag = ! CGAL_is_empty_range(i,j);
      CGAL__circ_loop_flag;
      CGAL__circ_loop_flag = ((++i) != (j))
    )
```

Note that the macro behaves like a *for*-loop. It can be used with a single statement or with a statement block. For bidirectional iterators or circulators exist a backwards loop macro *CGAL_For_all_backwards(i, j)* that decrements j until it reaches i.

See Also

CGAL_iterator_distance

Chapter 4

Protected Access to Internal Representations

4.1 Introduction

High level data structures typically maintain integrity of an internal data representation, which they protect from the user. A minimal while complete interface of the data structure allows manipulations in the domain of valid representations. Additional operations might benefit from being allowed to access the internal data representation directly. An example are intermediate steps within an algorithm where the internal representation would be invalid. We present a general method to accomplish access in a safe manner, such that the high level data structures can guarantee validity after the possibly compromising algorithm has finished its work. An example are polyhedral surfaces in the Basic Library, where a construction process like for a file scanner could be performed more efficiently on the internal halfedge data structure than by using the high-level Euler operators of the polyhedron.

The solution provided here follows the known Strategy pattern [GHJV95], see Figure 4.1. The abstract base class *CGAL_Modifier_base<R>* declares a pure virtual member function *operator()* that accepts a single reference parameter of the internal representation type. The member function *delegate()* of the

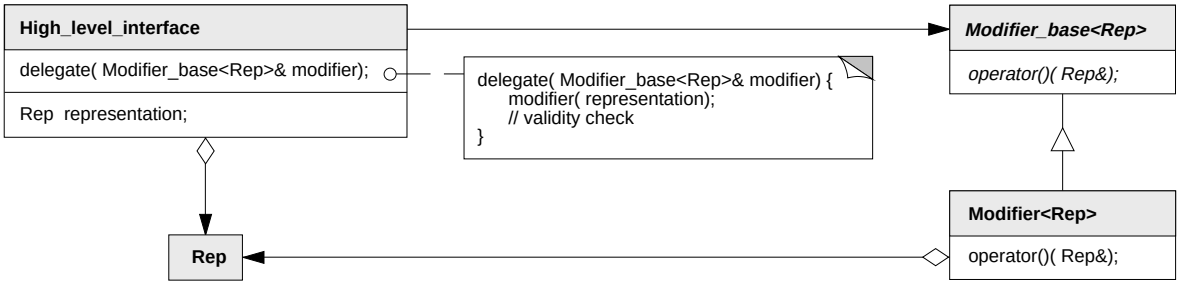

Figure 4.1: Class diagram for the modifier following the Strategy pattern [GHJV95]. It illustrates the safe access to an internal representation through an high-level interface.

high-level interface calls this *operator()* with its internal representation. An actual modifier implements this virtual function, thus gaining access to the internal representation. Once, the modifier has finished its work, the member function *delegate()* is back in control and can check the validity of the internal representation. Summarizing, a user can implement and apply arbitrary functions based on the internal representation and keeps the benefit if a protected high-level interface. User provided modifiers must in any case return a valid internal representation or the checker in the high-level interface is allowed (and supposed) to abort the program. The indirection via the virtual function invocation is negligible for operations that consists of more than a pointer update or integer addition.

4.2 Abstract Basis for Modifiers (*CGAL_Modifier_base<R>*)

Definition

CGAL_Modifier_base<R> is an abstract base class providing the interface for any modifier. A modifier is a function object derived from *CGAL_Modifier_base<R>* that implements the pure virtual member function *operator()*, which accepts a single reference parameter *R&* on which the modifier is allowed to work. *R* is the type of the internal representation that is to be modified.

#include <CGAL/Modifier_base.h>

Types

typedef R	*Representation;*	the internal representation type.

Operations

virtual void	*modifier.operator()(R& rep)*	*Postcondition*: *rep* is a valid representation.

Example

The following fragment defines a class `A` with an internal representation `i` of type `int`. It provides a member function `delegate()`, which gives a modifier access to the internal variable and checks validity thereafter. The example modifier sets the internal variable to 42. The example function applies the modifier to an instance of class `A`.

```
class A {
    int i;  // protected internal representation
public:
    void delegate( CGAL_Modifier_base<int>& modifier) {
        modifier(i);
        CGAL_postcondition( i > 0);  // check validity
    }
};

struct Modifier : public CGAL_Modifier_base<int> {
    void operator()( int& rep) { rep = 42;}
};
```

```
void use_it() {
    A a;
    Modifier m;
    a.delegate(m);  // a.i == 42 and A has checked that A::i > 0.
}
```

Chapter 5

Random Sources and Geometric Object Generators

A variety of generators for random numbers and geometric objects is provided in CGAL. They are useful as synthetic test data sets, e.g. for testing algorithms on degenerate object sets and for performance analysis.

The first section describes the random number source used for random generators. The second section provides useful generic functions related to random numbers like *CGAL_random_selection()*. The third section documents generators for two-dimensional point sets, the fourth section for three-dimensional point sets. The fifth section presents examples using functions from Section 2.2 to generate composed objects like segments. Note that the STL algorithm *random_shuffle* is useful in this context to achieve random permutations for otherwise regular generators (e.g. points on a grid or segment).

5.1 Random Numbers Generator (*CGAL_Random*)

Definition

An instance of type *CGAL_Random* is a random numbers generator. It allows to generate uniformly distributed random *bool*s, *int*s and *double*s. It can be used as the random number generating function object in the STL algorithm *random_shuffle*.

Instances of *CGAL_Random* can be seen as input streams. Different streams are *independent* of each other, i.e. the sequence of numbers from one stream does *not* depend upon how many numbers were extracted from the other streams.

It can be very useful, e.g. for debugging, to reproduce a sequence of random numbers. This can be done by either initializing deterministically or using the seed functions as described below.

#include <*CGAL/Random.h*>

Global Variables

CGAL_Random	*CGAL_random;*	This global variable is used as the default random numbers generator.

Types

CGAL_Random:: Seed	Seed type.

Creation

CGAL_Random random;	introduces a variable *random* of type *CGAL_Random*.
CGAL_Random random(Seed seed);	introduces a variable *random* of type *CGAL_Random* and initializes its internal seed with *seed*.
CGAL_Random random(long init);	introduces a variable *random* of type *CGAL_Random* and initializes its internal seed using *init*. Equal values for *init* result in equal sequences of random numbers.

Operations

bool	*random.get_bool()*	returns a random *bool*.
int	*random.get_int(int lower, int upper)*	returns a random *int* from the interval [*lower*,*upper*).
double	*random.get_double(double lower = 0.0, double upper = 1.0)*	returns a random *double* from the interval [*lower*,*upper*).
int	*random(int upper)*	returns *random.get_int(0, upper)*.

Seed Functions

void	*random.save_seed(Seed& seed)*	saves the current internal seed in *seed*.
void	*random.restore_seed(Seed seed)*	restores the internal seed from *seed*.

Equality Test

bool	*random == random2*	returns *true*, iff *random* and *random2* have equal internal seeds.

Implementation

We use the C library function *erand48* to generate the random numbers, i.e. the sequence of numbers depends on the implementation of *erand48* on your specific platform.

5.2 Support Functions for Generators

5.2.1 *CGAL_random_selection()*

CGAL_random_selection chooses n items at random from a random access iterator range which is useful to produce degenerate input data sets with multiple entries of identical items.

#include <*CGAL/random_selection.h*>

```
template <class RandomAccessIterator, class Size, class OutputIterator, class Random>
OutputIterator    CGAL_random_selection( RandomAccessIterator first,
                                         RandomAccessIterator last,
                                         Size n,
                                         OutputIterator result,
                                         Random& rnd = CGAL_random)
```

chooses a random item from the range $[first, last)$ and writes it to *result*, each item from the range with equal probability, and repeats this n times, thus writing n items to *result*. A single random number is needed from *rnd* for each item. Returns the value of *result* after inserting the n items.
Precondition: *Random* is a random number generator type as provided by the STL or by *CGAL_Random*.

5.3 2D Point Generators

Two kind of point generators are provided: First, random point generators and second deterministic point generators. Most random point generators and a few deterministic point generators are provided as input iterators. The input iterators model an infinite sequence of points. The function *CGAL_copy_n()* could be used to copy a finite sequence, see Section 2.2.1. The iterator adaptor *CGAL_Counting_iterator* can be used to create finite iterator ranges, see Section 2.5.1. Other generators are provided as functions writing to an output iterator. Further functions add degeneracies or random perturbations.

5.3.1 Point Generators as Input Iterators

Definition

Input iterators are provided for random points uniformly distributed over a two-dimensional domain (square or disc) or a one-dimensional domain (boundary of a square, circle, or segment). Another input iterator generates equally spaced points from a segment.

All iterators are parameterized with the point type P and all with the exception of the class *CGAL_Points_on_segment_2* have a second template argument *Creator* which defaults to the class *CGAL_Creator_uniform_2*<*double,P*>[1]. The *Creator* must be a function object accepting two *double* values x and y and returning an initialized point *(x,y)* of type P. Predefined implementations for these creators like the default can be found in Section 2.3.2. They simply assume an appropriate constructor for type P.

All generators know a range within which the coordinates of the generated points will lie.

#include <*CGAL/point_generators_2.h*>

Types

The generators comply to the requirements of input iterators which includes local type declarations including *value_type* which denotes P here.

Creation

CGAL_Random_points_in_disc_2<*P,Creator*> *g(double r,*
CGAL_Random& rnd = CGAL_random)

g is an input iterator creating points of type P uniformly distributed in the open disc with radius r, i.e. $|*g| < r$. Two random numbers are needed from *rnd* for each point.

CGAL_Random_points_on_circle_2<*P,Creator*> *g(double r,*
CGAL_Random& rnd = CGAL_random)

g is an input iterator creating points of type P uniformly distributed on the circle with radius r, i.e. $|*g| == r$. A single random number is needed from *rnd* for each point.

[1]For compilers not supporting these kind of default arguments, both template arguments must be provided when using these generators.

CGAL_Random_points_in_square_2<P,Creator> g(double a,
CGAL_Random& rnd = CGAL_random)

g is an input iterator creating points of type P uniformly distributed in the half-open square with side length $2a$, centered at the origin, i.e. $\forall p = *g : -a \leq p.x() < a$ and $-a \leq p.y() < a$. Two random numbers are needed from *rnd* for each point.

CGAL_Random_points_on_square_2<P,Creator> g(double a,
CGAL_Random& rnd = CGAL_random)

g is an input iterator creating points of type P uniformly distributed on the boundary of the square with side length $2a$, centered at the origin, i.e. $\forall p = *g$: one coordinate is either a or $-a$ and for the other coordinate c holds $-a \leq c < a$. A single random number is needed from *rnd* for each point.

CGAL_Random_points_on_segment_2<P,Creator> g(P p,
P q,
CGAL_Random& rnd = CGAL_random)

g is an input iterator creating points of type P uniformly distributed on the segment from p to q (excluding q), i.e. $*g == (1-\lambda)\,p + \lambda q$ where $0 \leq \lambda < 1$. A single random number is needed from *rnd* for each point.
Precondition: The expressions *CGAL_to_double(p.x())* and *CGAL_to_double(p.y())* must result in the respective *double* representation of the coordinates and similar for q.

CGAL_Points_on_segment_2<P> g(P p, P q, size_t n, size_t i = 0);

g is an input iterator creating points of type P equally spaced on the segment from p to q. n points are placed on the segment including p and q. The iterator denoted the point i where p has the index 0 and q the index n.
Precondition: The expressions *CGAL_to_double(p.x())* and *CGAL_to_double(p.y())* must result in the respective *double* representation of the coordinates and similar for q.

Operations

double	*g.range()*	returns the range in which the point coordinates lie, i.e. $\forall x : \lvert x\rvert \leq$*range()* and $\forall y : \lvert y\rvert \leq$*range()*.

The generators *CGAL_Random_points_on_segment_2* and *CGAL_Points_on_segment_2* have to additional methods.

P	*g.source()*	returns the source point of the segment.
P	*g.target()*	returns the target point of the segment.

See Also

stl::random_shuffle, *CGAL_random_selection*, *CGAL_copy_n*, *CGAL_Counting_iterator*, *CGAL_Join_input_iterator_1*.

5.3.2 Point Generators as Functions

Grid Points

Grid points are generated by functions writing to an output iterator.

template <*class OutputIterator, Creator creator*>
OutputIterator *CGAL_points_on_square_grid_2(*
double a,
size_t n,
OutputIterator o,
Creator creator = CGAL_Creator_uniform_2<*double,P*>*)*

creates the n first points on the regular $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$ grid within the square $[-a,a] \times [-a,a]$. Returns the value of o after inserting the n points.
Precondition: *Creator* must be a function object accepting two *double* values x and y and returning an initialized point *(x,y)* of type P. Predefined implementations for these creators like the default can be found in Section 2.3.2. The *OutputIterator* must accept values of type P. If the *OutputIterator* has a *value_type* the default initializer of the *creator* can be used. P is set to the *value_type* in this case.

template <*class P, class OutputIterator*>
OutputIterator *CGAL_points_on_segment_2(P p, P q, size_t n, OutputIterator o)*

creates n points equally spaced on the segment from p to q, i.e. $\forall i : 0 \leq i < n : o[i] := \frac{n-i-1}{n-1}\, p + \frac{i}{n-1}\, q$. Returns the value of o after inserting the n points.

Random Perturbations

Degenerate input sets like grid points can be randomly perturbed by a small amount to produce *quasi*-degenerate test sets. This challenges numerical stability of algorithms using inexact arithmetic and exact predicates to compute the sign of expressions slightly off from zero.

template <*class ForwardIterator*>
void *CGAL_perturb_points_2(ForwardIterator first,*
ForwardIterator last,
double xeps,
double yeps = xeps,
CGAL_Random& *rnd = CGAL_random,*

Creator creator = CGAL_Creator_uniform_2<double,P>)

perturbs the points in the range [*first*, *last*) by replacing each point with a random point from the rectangle *xeps* × *yeps* centered at the original point. Two random numbers are needed from *rnd* for each point.
Precondition: *Creator* must be a function object accepting two *double* values x and y and returning an initialized point *(x,y)* of type P. Predefined implementations for these creators like the default can be found in Section 2.3.2. The *value_type* of the *ForwardIterator* must be assignable to P. P is equal to the *value_type* of the *ForwardIterator* when using the default initializer. The expressions *CGAL_to_double((*first).x())* and *CGAL_to_double((*first).y())* must result in the respective coordinate values.

Adding Degeneracies

For a given point set certain kinds of degeneracies can be produced adding new points. The *CGAL_random_selection()* function is useful to generate multiple entries of identical points, see Section 5.2.1. The *CGAL_random_collinear_points_2()* function adds collinearities to a point set.

template <class RandomAccessIterator, class OutputIterator>
OutputIterator CGAL_random_collinear_points_2(
RandomAccessIterator first,
RandomAccessIterator last,
size_t n,
OutputIterator first2,
CGAL_Random& rnd = CGAL_random,
Creator creator = CGAL_Creator_uniform_2<double,P>)

randomly chooses two points from the range [*first*, *last*), creates a random third point on the segment connecting this two points, writes it to *first2*, and repeats this n times, thus writing n points to *first2* that are collinear with points in the range [*first*, *last*). Three random numbers are needed from *rnd* for each point. Returns the value of *first2* after inserting the n points.
Precondition: *Creator* must be a function object accepting two *double* values x and y and returning an initialized point *(x,y)* of type P. Predefined implementations for these creators like the default can be found in Section 2.3.2. The *value_type* of the *RandomAccessIterator* must be assignable to P. P is equal to the *value_type* of the *RandomAccessIterator* when using the default initializer. The expressions *CGAL_to_double((*first).x())* and *CGAL_to_double((*first).y())* must result in the respective coordinate values.

See Also

stl::random_shuffle, *CGAL_random_selection*.

Example

We want to generate a test set of 1000 points, where 60% are chosen randomly in a small disc, 20% are from a larger grid, 10% duplicates are added, and 10% collinearities added. A random shuffle removes the construction order from the test set. See Figure 5.1 for the example output.

```
/* generators_prog1.C      */
/* ------------------------*/
/* CGAL example program for point generators. */

#include <CGAL/basic.h>
#include <assert.h>
#include <vector.h>
#include <algo.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/copy_n.h>
#include <CGAL/random_selection.h>
#include <CGAL/IO/Window_stream.h>   /* only used for visualization */

typedef CGAL_Cartesian<double>                R;
typedef CGAL_Point_2<R>                       Point;
typedef CGAL_Creator_uniform_2<double,Point>  Creator;

int main()
{
    /* Create test point set. Prepare a vector for 1000 points. */
    vector<Point> points;
    points.reserve(1000);

    /* Create 600 points within a disc of radius 150. */
    CGAL_Random_points_in_disc_2<Point,Creator> g( 150.0);
    CGAL_copy_n( g, 600, back_inserter( points));

    /* Create 200 points from a 15 x 15 grid. */
    CGAL_points_on_square_grid_2( 250.0, 200, back_inserter(points),Creator());

    /* Select 100 points randomly and append them at the end of */
    /* the current vector of points. */
    CGAL_random_selection( points.begin(), points.end(), 100,
               back_inserter( points));

    /* Create 100 points that are collinear to two randomly chosen */
    /* points and append them to the current vector of points. */
    CGAL_random_collinear_points_2( points.begin(), points.end(), 100,
                    back_inserter( points));

    /* Check that we have really created 1000 points. */
    assert( points.size() == 1000);

    /* Use a random permutation to hide the creation history */
    /* of the point set. */
    random_shuffle( points.begin(), points.end(), CGAL_random);

    /* Visualize point set. Can be omitted, see example programs */
    /* in the CGAL source code distribution. */
    CGAL_Window_stream W(512, 512);
    W.init(-256.0, 255.0, -256.0);
    W << CGAL_BLACK;
    for( vector<Point>::iterator i = points.begin(); i != points.end(); i++)
```

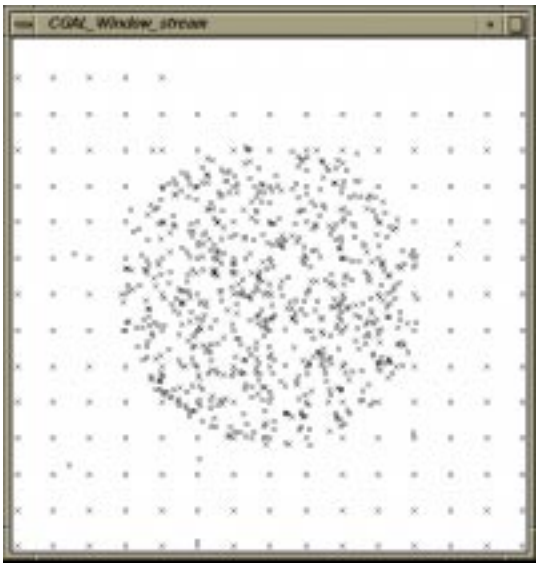

Figure 5.1: Output of example program for point generators.

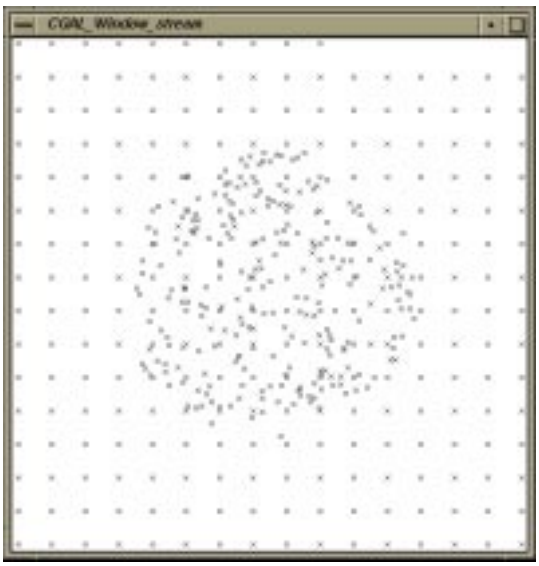

Figure 5.2: Output of example program for point generators working on integer points.

```
    W ≪ *i;

    /* Wait for mouse click in window. */
    Point p;
    W ≫ p;

    return 0;
}
```

The second example demonstrates the point generators with integer points. Arithmetic with *double*'s is sufficient to produce regular integer grids. See Figure 5.2 for the example output.

```
/* generators_prog2.C      */
/* ————————————————— */
/* CGAL example program for point generators creating integer points. */

#include <CGAL/basic.h>
#include <assert.h>
#include <vector.h>
#include <algo.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/copy_n.h>
#include <CGAL/IO/Window_stream.h>   /* only for visualization used */

typedef CGAL_Cartesian<double>                R;
typedef CGAL_Point_2<R>                       Point;
typedef CGAL_Creator_uniform_2<double,Point>  Creator;

int main()
{
    /* Create test point set. Prepare a vector for 400 points. */
    vector<Point> points;
    points.reserve(400);

    /* Create 250 points from a 16 x 16 grid. Note that the double */
    /* arithmetic _is_ sufficient to produce exact integer grid points. */
    /* The distance between neighbors is 34 pixel = 510 / 15. */
    CGAL_points_on_square_grid_2( 255.0, 250, back_inserter(points),Creator());

    /* Lower, left corner. */
    assert( points[0].x() == -255);
    assert( points[0].y() == -255);

    /* Upper, right corner. Note that 6 points are missing to fill the grid. */
    assert( points[249].x() == 255 - 6 * 34);
    assert( points[249].y() == 255);

    /* Create 250 points within a disc of radius 150. */
    CGAL_Random_points_in_disc_2<Point,Creator> g( 150.0);
    CGAL_copy_n( g, 250, back_inserter( points));

    /* Check that we have really created 500 points. */
    assert( points.size() == 500);

    /* Visualize point set. Can be omitted, see example programs */
    /* in the CGAL source code distribution. */
    CGAL_Window_stream W(524, 524);
    W.init(-262.0, 261.0, -262.0);
    W << CGAL_BLACK;
    for( vector<Point>::iterator i = points.begin(); i != points.end(); i++)
    W << *i;
```

```
    /* Wait for mouse click in window. */
    Point p;
    W ≫ p;

    return 0;
}
```

5.4 3D Point Generators

One kind of point generators is currently provided: Random point generators implemented as input iterators. The input iterators model an infinite sequence of points. The function *CGAL_copy_n()* could be used to copy a finite sequence, see Section 2.2.1. The iterator adaptor *CGAL_Counting_iterator* can be used to create finite iterator ranges, see Section 2.5.1.

5.4.1 Point Generators as Input Iterators

Definition

Input iterators are provided for random points uniformly distributed in a three-dimensional volume (sphere or cube) or a two-dimensional surface (boundary of a sphere).

All iterators are parameterized with the point type P and a second template argument *Creator* which defaults to *CGAL_Creator_uniform_3<double,P>*[2]. The *Creator* must be a function object accepting three *double* values x, y and z and returning an initialized point *(x,y,z)* of type P. Predefined implementations for these creators like the default can be found in Section 2.3.2. They simply assume an appropriate constructor for type P.

All generators know a range within which the coordinates of the generated points will lie.

#include <CGAL/point_generators_3.h>

Types

The generators comply to the requirements of input iterators which includes local type declarations including *value_type* which denotes P here.

Creation

CGAL_Random_points_in_sphere_3<P,Creator> g(double r,
CGAL_Random& rnd = CGAL_random)

g is an input iterator creating points of type P uniformly distributed in the open sphere with radius r, i.e. $|*g| < r$.

CGAL_Random_points_on_sphere_3<P,Creator> g(double r,
CGAL_Random& rnd = CGAL_random)

g is an input iterator creating points of type P uniformly distributed on the boundary of a sphere with radius r, i.e. $|*g| == r$. Two random numbers are needed from *rnd* for each point.

[2]For compilers not supporting these kind of default arguments, both template arguments must be provided when using these generators.

CGAL_Random_points_in_cube_3<P,Creator> g(double a,
CGAL_Random& rnd = CGAL_random)

g is an input iterator creating points of type P uniformly distributed in the half-open cube with side length $2a$, centered at the origin, i.e. $\forall p = *g : -a \leq p.x(), p.y(), p.z() < a$. Three random numbers are needed from *rnd* for each point.

See Also

stl::random_shuffle, *CGAL_random_selection*, *CGAL_copy_n*, *CGAL_Counting_iterator*, *CGAL_Join_input_iterator_1*.

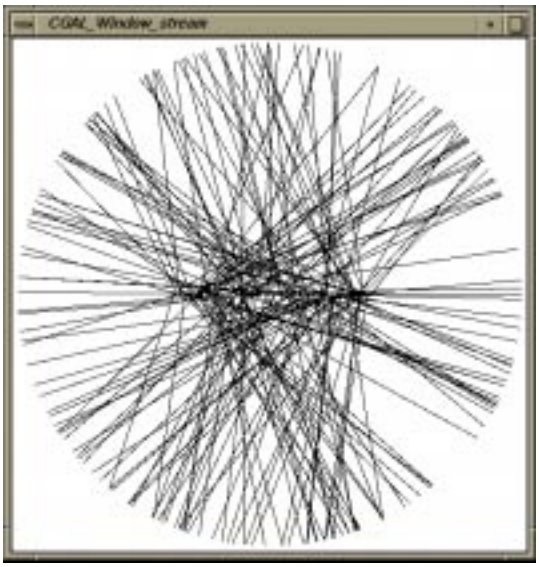

Figure 5.3: Output of the first example program for the generic generator.

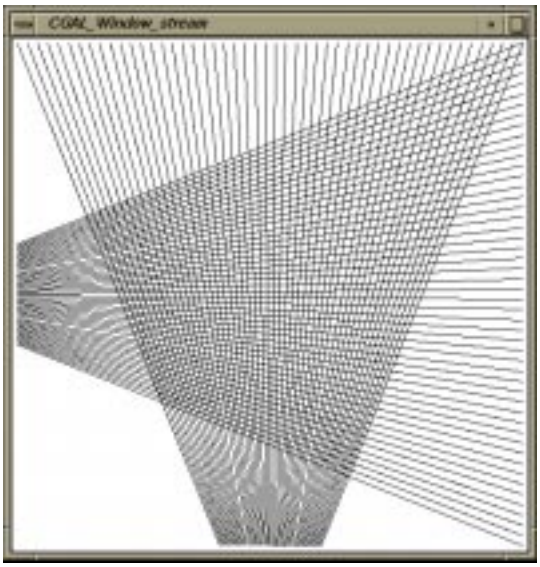

Figure 5.4: Output of the second example program for the generic generator without using intermediate storage.

5.5 Examples Generating Segments

The following two examples illustrate the use of the generic functions from Section 2.2 like *CGAL_Join_input_iterator* to generate composed objects from other generators – here two-dimensional segments from two point generators.

We want to generate a test set of 200 segments, where one endpoint is chosen randomly from a horizontal segment of length 200, and the other endpoint is chosen randomly from a circle of radius 250. See Figure 5.3 for the example output.

```
/* Segment_generator_prog1.C */
/* ------------------------- */
/* CGAL example program for the generic segment generator. */

#include <CGAL/basic.h>
#include <assert.h>
#include <vector.h>
#include <algo.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/Segment_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/function_objects.h>
#include <CGAL/Join_input_iterator.h>
#include <CGAL/copy_n.h>
#include <CGAL/IO/Window_stream.h>   /* only for visualization used */

typedef CGAL_Cartesian<double>                R;
typedef CGAL_Point_2<R>                       Point;
typedef CGAL_Creator_uniform_2<double,Point>  Pt_creator;
typedef CGAL_Segment_2<R>                     Segment;
```

```
int main()
{
    /* Create test segment set. Prepare a vector for 200 segments. */
    vector<Segment> segs;
    segs.reserve(200);

    /* Prepare point generator for the horizontal segment, length 200. */
    typedef  CGAL_Random_points_on_segment_2<Point,Pt_creator>  P1;
    P1 p1( Point(-100,0), Point(100,0));

    /* Prepare point generator for random points on circle, radius 250. */
    typedef  CGAL_Random_points_on_circle_2<Point,Pt_creator>  P2;
    P2 p2( 250);

    /* Create 200 segments. */
    typedef CGAL_Creator_uniform_2< Point, Segment> Seg_creator;
    typedef CGAL_Join_input_iterator_2< P1, P2, Seg_creator> Seg_iterator;
    Seg_iterator g( p1, p2);
    CGAL_copy_n( g, 200, back_inserter( segs));

    /* Visualize segments. Can be omitted, see example programs */
    /* in the CGAL source code distribution. */
    CGAL_Window_stream W(512, 512);
    W.init(-256.0, 255.0, -256.0);
    W << CGAL_BLACK;
    for( vector<Segment>::iterator i = segs.begin(); i != segs.end(); i++)
    W << *i;

    /*  Wait for mouse click in window. */
    Point p;
    W >> p;

    return 0;
}
```

The second example generates a regular structure of 100 segments, see Figure 5.4 for the example output. It uses the *CGAL_Points_on_segment_2* iterator, *CGAL_Join_input_iterator_2* and *CGAL_Counting_iterator* to avoid any intermediate storage of the generated objects until they are used, in this example copied to a window stream.

```
/*  Segment_generator_prog2.C  */
/*  ---------------------- */
/*  CGAL example program generating a regular segment pattern. */

#include <CGAL/basic.h>
#include <algo.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/Segment_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/function_objects.h>
#include <CGAL/Join_input_iterator.h>
#include <CGAL/Counting_iterator.h>
#include <CGAL/IO/Ostream_iterator.h>
#include <CGAL/IO/Window_stream.h>
```

```
typedef CGAL_Cartesian<double>                              R;
typedef CGAL_Point_2<R>                                     Point;
typedef CGAL_Segment_2<R>                                   Segment;
typedef CGAL_Points_on_segment_2<Point>                     PG;
typedef CGAL_Creator_uniform_2< Point, Segment>             Creator;
typedef CGAL_Join_input_iterator_2< PG, PG, Creator>        Segm_iterator;
typedef CGAL_Counting_iterator<Segm_iterator,Segment>       Count_iterator;


int main()
{
    /* Open window. */
    CGAL_Window_stream W(512, 512);
    W.init(-256.0, 255.0, -256.0);
    W << CGAL_BLACK;

    /* A horizontal like fan. */
    PG p1( Point(-250, -50), Point(-250, 50),50);      /* Point generator. */
    PG p2( Point( 250,-250), Point( 250,250),50);
    Segm_iterator  t1( p1, p2);                        /* Segment generator. */
    Count_iterator t1_begin( t1);                      /* Finite range. */
    Count_iterator t1_end( 50);
    copy( t1_begin, t1_end,
      CGAL_Ostream_iterator<Segment,CGAL_Window_stream>(W));

    /* A vertical like fan. */
    PG p3( Point( -50,-250), Point(  50,-250),50);
    PG p4( Point(-250, 250), Point( 250, 250),50);
    Segm_iterator  t2( p3, p4);
    Count_iterator t2_begin( t2);
    Count_iterator t2_end( 50);
    copy( t2_begin, t2_end,
      CGAL_Ostream_iterator<Segment,CGAL_Window_stream>(W));

    /*  Wait for mouse click in window. */
    Point p;
    W >> p;
    return 0;
}
```

5.6 Building Random Convex Sets

This section describes a function to compute a random convex planar point set of given size where the points are drawn from a specific domain.

#include <*CGAL/random_convex_set_2.h*>

template < *class OutputIterator, class Point_generator, class Traits* >
OutputIterator *CGAL_random_convex_set_2(*
int n,
OutputIterator o,
Point_generator pg,
Traits t = Default_traits)

computes a random convex n-gon by writing its vertices (oriented counterclockwise) to o. The resulting polygon is scaled such that it fits into the bounding box as specified by pg. Therefore we cannot easily describe the resulting distribution.

Precondition

1. *Point_generator* satisfies the requirements stated in section 5.6.1,
2. If *Traits* is specified, it has to satisfy the requirements stated in section 5.6.2 and *Traits::Point_2* must be the same as *Point_generator::value_type*,
3. if *Traits* is not specified, *Point_generator::value_type* must be *CGAL_Point_2*< R > for some representation class R,
4. *OutputIterator* accepts *Point_generator::value_type* as value type *and*
5. $n \geq 3$.

See Also

CGAL_Random_points_in_square_2 and *CGAL_Random_points_in_disc_2*.

Implementation

The implementation uses the centroid method described in [Sch96] and has a worst case running time of $O(r \cdot n + n \cdot \log n)$, where r is the time needed by pg to generate a random point.

Example

The following program displays a random convex 500-gon where the points are drawn uniformly from the unit square centered at the origin.

```
#include <CGAL/Cartesian.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>
#include <CGAL/IO/Window_stream.h>
#include <CGAL/IO/Ostream_iterator.h>

int main() {
  typedef CGAL_Cartesian< double >   R;
  typedef CGAL_Point_2< R >          Point_2;
  typedef CGAL_Random_points_in_square_2<
    Point_2,
    CGAL_Creator_uniform_2< double, Point_2 > >
  Point_generator;

  // create 500-gon and write it into a window:
  CGAL_Window_stream W;
  W.init( -0.55, 0.55, -0.55);
  CGAL_random_convex_set_2(
    500,
    CGAL_Ostream_iterator< Point_2, CGAL_Window_stream >( W),
    Point_generator( 0.5));

  // wait for mouse-click:
  Point_2 p;
  W >> p;
}
```

5.6.1 Requirements for Point Generator Classes (*Point_generator*)

Definition

A class *Point_generator* satisfying input iterator requirements has to provide the following additional types and operations in order to qualify as a point generator class. The point generators described in section 5.3 fulfill these requirements.

Types

Point_generator:: value_type	point class.
Point_generator:: FT	class used for doing computations on point coordinates (has to fulfill field type requirements).

Operations

FT	*pg.range() const*	return an absolute bound for the coordinates of all generated points.

5.6.2 Requirements for Random Convex Sets Traits Classes

Definition

A class *Traits* has to provide the following types and operations in order to qualify as a traits class for *CGAL_random_convex_set_2*.

Types

Traits:: Point_2	point class.
Traits:: FT	class used for doing computations on point and vector coordinates (has to fulfill field type requirements).
Traits:: Sum	AdaptableBinaryFunction class: *Point_2* $\times$ *Point_2* $\rightarrow$ *Point_2*. It returns the point that results from adding the vectors corresponding to both arguments.
Traits:: Scale	AdaptableBinaryFunction class: *Point_2* $\times$ *FT* $\rightarrow$ *Point_2*. *Scale(p,k)* returns the point that results from scaling the vector corresponding to p by a factor of k.
Traits:: Max_coordinate	AdaptableUnaryFunction class: *Point_2* $\rightarrow$ *FT*. *Max_coordinate(p)* returns the coordinate of p with largest absolute value.
Traits:: Angle_less	AdaptableBinaryFunction class: *Point_2* $\times$ *Point_2* $\rightarrow$ *bool*. It returns *true*, iff the angle of the direction corresponding to the first argument with respect to the positive x-axis is less than the angle of the direction corresponding to the second argument.

Operations

Point_2	*t.origin() const*	return origin (neutral element for the *Sum* operation).

Chapter 6

Timer

6.1 A Timer for User-Process Time (*CGAL_Timer*)

Definition

A timer t of type *CGAL_Timer* is an object with a state. It is either *running* or it is *stopped*. The state is controlled with *t.start()* and *t.stop()* . The timer counts the time elapsed since its creation or last reset. It counts only the time where it is in the running state. The time information is given in seconds.

#include <*CGAL/Timer.h*>

Creation

CGAL_Timer	*t*;	state is *stopped*.

Operations

void	*t.start()*	*Precondition*: state is *stopped*.
void	*t.stop()*	*Precondition*: state is *running*.
void	*t.reset()*	reset timer to zero. The state is unaffected.
bool	*t.is_running()*	*true* if the current state is running.
double	*t.time()*	user process time in seconds.
int	*t.intervals()*	number of start/stop-intervals since the last reset.
double	*t.precision()*	smallest possible time step in seconds.
double	*t.max()*	maximal representable time in seconds.

Implementation

The timer class is based in the C function *clock(...)* which measures the user and system times of the current process and its subprocesses. The time granularity is reasonable fine with currently 10 ms on IRIX and Solaris. However the counter wraps around after only about 36 minutes. On Solaris machines the man page states that the timer could fail. In that case an error message is printed and the program aborted.

6.2 A Timer Measuring Real-Time (*CGAL_Real_timer*)

Definition

A timer t of type *CGAL_Real_timer* is an object with a state. It is either *running* or it is *stopped*. The state is controlled with *t.start()* and *t.stop()* . The timer counts the time elapsed since its creation or last reset. It counts only the time where it is in the running state. The time information is given in seconds.

#include <*CGAL/Real_timer.h*>

Creation

CGAL_Real_timer	*t*;	state is *stopped*.

Operations

void	*t.start()*	*Precondition*: state is *stopped*.
void	*t.stop()*	*Precondition*: state is *running*.
void	*t.reset()*	reset timer to zero. The state is unaffected.
bool	*t.is_running()*	*true* if the current state is running.
double	*t.time()*	real time in seconds.
int	*t.intervals()*	number of start/stop-intervals since the last reset.
double	*t.precision()*	smallest possible time step in seconds.
double	*t.max()*	maximal representable time in seconds.

Implementation

The timer class is based in the C function *gettimeofday(...)*.

Chapter 7

Stream Support

7.1 Stream Iterator (*CGAL_Ostream_iterator<T,Stream>*)

Definition

The class *CGAL_Ostream_iterator<T,Stream>* is an output iterator adaptor for the output stream class *Stream* and value type T.

#include <CGAL/IO/Ostream_iterator.h>

Creation

CGAL_Ostream_iterator<T,Stream> o(Stream& s);

creates an output iterator o writing to s.

Operations

o fulfills the requirements for an output iterator.

Implementation

The *operator*()* in class *CGAL_Ostream_iterator<T,Stream>* uses a proxy class.

7.2 Stream Iterator (*CGAL_Istream_iterator<T,Stream>*)

Definition

The class *CGAL_Istream_iterator<T,Stream>* is an input iterator adaptor for the input stream class *Stream* and value type T.

#include <CGAL/IO/Istream_iterator.h>

Creation

CGAL_Istream_iterator<T,Stream> i(Stream& s);

creates an input iterator i writing to s.

Operations

i fulfills the requirements for an input iterator.

Implementation

The *operator*()* in class *CGAL_Istream_iterator<T,Stream>* uses a proxy class.

7.3 3D Objects with Window Stream

3D objects can be used with *CGAL_Window_stream*. The stream output operator << projects them to the xy-plane. The stream input operator >> assumes zero for the z coordinate.

#include <CGAL/IO/window_stream_xy_3.h>

7.4 Verbose ostream (*CGAL_Verbose_ostream*)

Definition

The class *CGAL_Verbose_ostream* can be used as an output stream. The stream output operator << is defined for the builtin types. The class *CGAL_Verbose_ostream* stores in an internal state a stream and whether the output is active or not. If the state is active, all stream output operator << forward their output to the stream. Otherwise, no output happens.

#include <CGAL/IO/Verbose_ostream.h>

Creation

CGAL_Verbose_ostream verr(bool active = false, ostream& out = cerr);

creates an output stream with state set to *active* that writes to the stream *out*.

Operations

CGAL_Verbose_ostream&	*verr << char c*
CGAL_Verbose_ostream&	*verr << const char* s*
CGAL_Verbose_ostream&	*verr << int a*
CGAL_Verbose_ostream&	*verr << long l*
CGAL_Verbose_ostream&	*verr << double d*
CGAL_Verbose_ostream&	*verr << float f*
CGAL_Verbose_ostream&	*verr << unsigned int a*
CGAL_Verbose_ostream&	*verr << unsigned long l*
CGAL_Verbose_ostream&	*verr << long long ll*
CGAL_Verbose_ostream&	*verr << unsigned long long ll*
CGAL_Verbose_ostream&	*verr << void* p*
CGAL_Verbose_ostream&	*verr << short i*
CGAL_Verbose_ostream&	*verr << unsigned short i*
CGAL_Verbose_ostream&	*verr << ostream& (*f)(ostream&)*
CGAL_Verbose_ostream&	*verr << ios& (*f)(ios&)*
CGAL_Verbose_ostream&	*verr.flush()*
CGAL_Verbose_ostream&	*verr.put(char c)*
CGAL_Verbose_ostream&	*verr.write(const char* s, int n)*

Example

The class *CGAL_Verbose_ostream* can be conveniently used to implement for example the *is_valid()* member function for triangulations or other complex data structures.

```
bool is_valid( bool verbose = false, int level = 0) {
    CGAL_Verbose_ostream verr( verbose);
    verr ≪ "Triangulation::is_valid( level = " ≪ level ≪ ')' ≪ endl;
    verr ≪ "    Number of vertices = " ≪ size_of_vertices() ≪ endl;
    ...
}
```

Chapter 8

Geomview

8.1 Introduction

This chapter presents the CGAL interface to geomview, which is a viewer for three-dimensional objects, developed at the Geometry Center in Minneapolis[1].

Important: The last line in the startup file *.geomview* must be *(echo "started")*.

Definition

An object of the class *CGAL_Geomview_stream* is a stream in which geometric objects can be inserted and where geometric objects can be extracted from. The constructor starts geomview either on the local either on a remote machine.

#include <*CGAL/IO/Geomview_stream.h*>

Creation

CGAL_Geomview_stream *G(CGAL_Bbox_3 bbox = CGAL_Bbox_3(0,0,0, 1,1,1),*
*const char *machine = NULL,*
*const char *login = NULL)*

Introduces a geomview stream G with a camera that sees the bounding box. If machine and login are non-null, geomview is started on the remote machine.

Operations

Output Operators for CGAL Kernel Classes

At the moment not all classes of the CGAL kernel have output operators. 2D objects are embedded in the xy-plane.

template <*class R*>
CGAL_Geomview_stream& *CGAL_Geomview_stream& G << CGAL_Point_2*<*R*> *p*

Inserts the point p into the stream G.

template <*class R*>
CGAL_Geomview_stream& *CGAL_Geomview_stream& G << CGAL_Point_3*<*R*> *p*

Inserts the point p into the stream G.

[1] `http://www.geom.umn.edu/`

template <*class R*>
CGAL_Geomview_stream& *CGAL_Geomview_stream*& *G* << *CGAL_Segment_2*<*R*> *s*

Inserts the segment s into the stream G.

template <*class R*>
CGAL_Geomview_stream& *CGAL_Geomview_stream*& *G* << *CGAL_Segment_3*<*R*> *s*

Inserts the segment s into the stream G.

template <*class R*>
CGAL_Geomview_stream& *CGAL_Geomview_stream*& *G* << *CGAL_Triangle_2*<*R*> *t*

Inserts the triangle t into the stream G.

template <*class R*>
CGAL_Geomview_stream& *CGAL_Geomview_stream*& *G* << *CGAL_Triangle_3*<*R*> *t*

Inserts the triangle t into the stream G.

template <*class R*>
CGAL_Geomview_stream& *CGAL_Geomview_stream*& *G* << *CGAL_Tetrahedron_3*<*R*> *t*

Inserts the tetrahedron t into the stream G.

CGAL_Geomview_stream& *G* << *CGAL_Bbox_2 b*

Inserts the bounding box b into the stream G.

CGAL_Geomview_stream& *G* << *CGAL_Bbox_3 b*

Inserts the bounding box b into the stream G.

Input Operators for CGAL Kernel Classes

At the moment input is only provided for points. The user has to select a point on the *pick plane* with the right mouse button. The pick plane can be moved anywhere with the left mouse button, before a point is entered.

template <*class R*>
CGAL_Geomview_stream& *CGAL_Geomview_stream*& *G* >> *CGAL_Point_2*<*R*>& *p*

Extracts the point p from the stream G. The point is echoed.

IO Operators for CGAL Basic Library Classes

template <*class Traits*>
CGAL_Geomview_stream& *operator*<<(*CGAL_Geomview_stream*& *G*,
CGAL_Tetrahedralization_3<*Traits*> *T*)

Inserts the tetrahedralization T into the stream G.

For geometric structures as a tetrahedalization only one geomview object is created. If you want to select a single simplex you have to insert the simplices individually with the following operator.

template <*class V*>
CGAL_Geomview_stream& *operator*<<(*CGAL_Geomview_stream*& *G*,
const CGAL_Tetrahedralization_simplex<*V*>* *s*)

Inserts the tetrahedralization simplex s into the stream G.

template < *class V* >
CGAL_Geomview_stream& *operator*>>(*CGAL_Geomview_stream* &*G*,
CGAL_Tetrahedralization_simplex<*V*>*& *s*)

Assigns the pointer to the tetrahedralization simplex the user selected with the right mouse button to variable s.
Precondition: It is in the responsibility of the user to guarantee that the pointer still points to memory allocated for a simplex.

template <*class P*>
CGAL_Geomview_stream& *operator*<<(*CGAL_Geomview_stream*& *G*,
const CGAL_Tetrahedralization_vertex<*P*>* *v*)

Inserts the tetrahedralization vertex v into the stream G. Vertices are drawn as a cube with twice *G.get_vertex_radius* as side length. This value is 1/100'th of the x-length of the initial bounding box.

template < *class P*>
CGAL_Geomview_stream& *operator*>>(*CGAL_Geomview_stream* &*G*,
CGAL_Tetrahedralization_vertex<*V*>*& *v*)

Assigns the pointer to the tetrahedralization vertex the user selected with the right mouse button to variable v.
Precondition: It is in the responsibility of the user to guarantee that the pointer still points to memory allocated for a vertex.

Colors

geomview distinguishes between edge and face colors. The edge color is at the same time the color of vertices.

CGAL_Geomview_stream& *G* << *CGAL_Color c*

Makes c the color of vertices, edges and faces in subsequent IO operations.

CGAL_Color *G.set_bg_color(CGAL_Color c)*

Changes the background color. Returns the old value.

CGAL_Color *G.set_vertex_color(CGAL_Color c)*

Changes the vertex color. Returns the old value.

CGAL_Color *G.set_edge_color(CGAL_Color c)*

Changes the edge color. Returns the old value.

CGAL_Color *G.set_face_color(CGAL_Color c)*

Changes the face color. Returns the old value.

Miscellaneous

void — *G.clear()*

Deletes all objects.

void — *G.look_recenter()*

Positions the camera in a way that all objects can be seen.

int — *G.get_line_width()*

Returns the line width.

int — *G.set_line_width(int w)*

Sets the line width to w. Returns the previous value.

double — *G.get_vertex_radius()*

Returns the radius of vertices.

double — *G.set_vertex_radius(double r)*

Sets the radius of vertices to d. Returns the previous value.

bool — *G.set_trace(bool b)*

Sets tracing on. The data that are sent to *geomview* are also sent to *cerr*. Returns the previous value. By default tracing is off.

bool — *G.get_trace()*

Returns *true* iff tracing is on.

Advanced and Developers Features

The following functions are helpful if you develop your own insert and extract functions. The following functions allow to pass a string from geomview and to read data sent back by geomview.

CGAL_Geomview_stream& — *G << const char* s*

Inserts string s into the stream.

CGAL_Geomview_stream& — *G >> char* s*

Extracts a string s from the stream.
Precondition: You have to allocate enough memory.

CGAL_Geomview_stream& — *G << int i*

Inserts integer i into the stream. Puts whitespace around if the stream is in ascii mode.

CGAL_Geomview_stream& — *G << double d*

Inserts double d into the stream. Puts whitespace around if the stream is in ascii mode.

bool *G.in_binary_mode()*

Returns *true* iff G is in binary mode.

bool *G.in_ascii_mode()*

Returns *true* iff G is in binary mode.

void *G.set_binary_mode()*

void *G.set_ascii_mode()*

For convenience we offer the manipulators *ascii* and *binary* that can be inserted in the stream.

CGAL_Geomview_stream& $G <<$ *ascii*

Sets the stream in ascii mode.

CGAL_Geomview_stream& $G <<$ *binary*

Sets the stream in binary mode.

Implementation

The constructor forks a process and establishes two pipes between the processes. The forked process is then overlayed with geomview. The file descriptors *stdin* and *stdout* of geomview are hooked on the two pipes.

All insert operators construct expressions in *gcl*, the geomview comand language, which is a subset of LISP. These expressions are sent to geomview via the pipe. The extract operators notify *interest* for a certain kind of events. When such an event happens geomview sends a description of the event in gcl and the extract operator has to parse this expression.

In order to implement further insert and extract operators you should take a look at the implementation [Fab97] and at the geomview manual [Phi96].

Bibliography

[C++96] Working paper for draft proposed international standard for information systems – programming language C++. Doc. No. X3J16/96 - 0225 - WG21/N1043, December 1996. `http://www.maths.warwick.ac.uk/c++/pub/`.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Fab97] A. Fabri. How to Implement Geomview Stream IO Operators. CGAL report, July 1997.

[Gra96] T. Granlund. *GNU MP, The GNU Multiple Precision Arithmetic Library*, 2.0.2 edition, June 1996.

[MNU97] K. Mehlhorn, S. Näher, and C. Uhrig. *The LEDA User Manual, Version 3.5*. Max-Planck-Insitut für Informatik, 66123 Saarbrücken, Germany, 1997. `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.

[Mye95] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[Phi96] M. Phillips et al. *Geomview Manual*. Version 1.6.1 for Unix Workstations. December 10, 1996.

[Sch96] M. Schutte. Zufällige konvexe mengen. Master's thesis, Freie Universität Berlin, Germany, 1996.

[SVH89] B. Serpette, J. Vuillemin, and J.C. Hervé. BigNum, a portable and efficient package for arbitrary-precision arithmetic. Technical Report Research Report 2, Digital Paris Research Laboratory, 1989.

[SL95] A. Stepanov and M. Lee. The standard template library. `http://www.cs.rpi.edu/~musser/doc.ps`, October 1995.

[Str97] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.