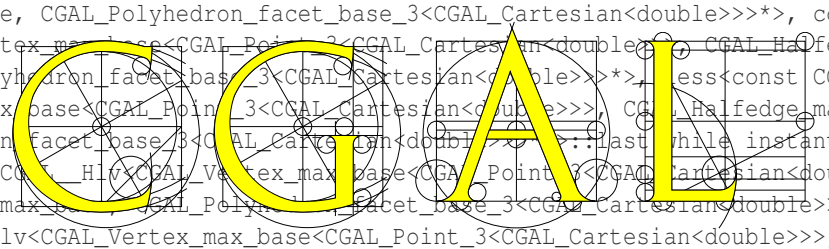
Warning: last hides rb_tree<const CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_C
artesian<double>>>, CGAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartes
ian<double>>>*, pair<const CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesia
n<double>>>, CGAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<dou
ble>>>* const, CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>,
CGAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>*>, sel
ect1st<pair<const CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>
>>, CGAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>* c
onst, CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>, CGAL_Hal
fedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>*>, const CGAL_
_Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>, CGAL_Halfedge_max_b
ase, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>*>, less<const CGAL__Hlv<C
GAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>, CGAL_Halfedge_max_base, C
GAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>*>>::last while instantiating "r
b_tree<const CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>, C
GAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>*, pair<
const CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>, CGAL_Hal
fedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>* const, CGAL__
Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>, CGAL_Halfedge_max_ba
se, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>*>, select1st<pair<const CG
AL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>, CGAL_Halfedge_ma
x_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>* const, CGAL__Hlv<CGAL
_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>, CGAL_Halfedge_max_base, CGAL
_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>*>, const CGAL__Hlv<CGAL_Vertex_max
_base<CGAL_Point_3<CGAL_Cartesian<double>>>, CGAL_Halfedge_max_base, CGAL_Polyhedron
_facet_base_3<CGAL_Cartesian<double>>>*>, less<const CGAL__Hlv<CGAL_Vertex_max_base<
CGAL_Point_3<CGAL_Cartesian<double>>>, CGAL_Halfedge_max_base, CGAL_Polyhedron_facet
_base_3<CGAL_Cartesian<double>>>*>>::erase(rb_tree<const CGAL__Hlv<CGAL_Vertex_max_b
ase<CGAL_Point_3<CGAL_Cartesian<double>>>, CGAL_Halfedge_max_base, CGAL_Polyhedron_f
acet_base_3<CGAL_Cartesian<double>>>*, pair<const CGAL__Hlv<CGAL_Vertex_max_base<CGA
L_Point_3<CGAL_Cartesian<double>>>, CGAL_Halfedge_max_base, CGAL_Polyhedron_facet_ba
se_3<CGAL_Cartesian<double>>>* const, CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CG
AL_Cartesian<double>>>, CGAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Ca
rtesian<double>>>*>, select1st<pair<const CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_
3<CGAL_Cartesian<double>>>, CGAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGA
L_Cartesian<double>>>* const, CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Carte
sian<double>>>, CGAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<
double>>>*>, const CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double
>>>, CGAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>*>
, less<const CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>, C
GAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>*>>::ite
rator(local), rb_tree<const CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesi
an<double>>>, CGAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<do
uble>>>*, pair<const CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<doub
le>>>, CGAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>
* const, CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>, CGAL_
Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>*>, select1s
t<pair<const CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>, C
GAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>* const,
 CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>, CGAL_Halfedge
_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>*>,const CGAL__Hlv<C
GAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>, CGAL_Halfedge_max_base, C
GAL_Polyhedron_facet_base_3<CGAL_Cartesian<double>>>*>, less<const CGAL__Hlv<CGAL_Ve
rtex_max_base<CGAL_Point_3<CGAL_Cartesian<double>>>, CGAL_Halfedge_max_base, CGAL_Po



Developers' Manual

Release 3.3-I

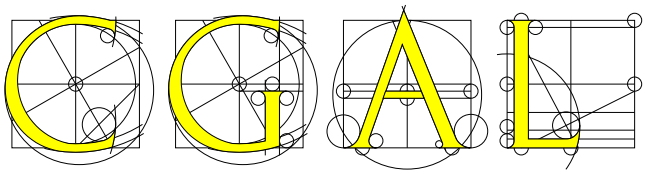
25 August 2007

Contents

Chapter 1

Introduction

*Susan Hert (*hert@mpi-sb.mpg.de*)*
*Stefan Schirra (*stschirr@mpi-sb.mpg.de*)*



COMPUTATIONAL GEOMETRY ALGORITHMS LIBRARY

The goal of CGAL *is to make available to users in industry and academia the most important efficient solutions to basic geometric problems developed in the area of computational geometry in a C++ software library.*

Work on CGAL has been supported by ESPRIT IV projects 21957 (CGAL) and 28155 (GALIA).

1.1 Primary design goals

The primary design goals of CGAL are [FGK+00]:

Correctness

A library component is correct if it behaves according to its specification. Basically, correctness is therefore a matter of verification that documentation and implementation coincide. In a modularized program the correctness of a module is determined by its own correctness and the correctness of all the modules it depends on. Clearly, in order to get correct results, correct algorithms and data structures must be used.

Exactness should not be confused with correctness in the sense of reliability. There is nothing wrong with algorithms computing approximate solutions instead of exact solutions, as long as their behavior is clearly documented and they do behave as specified. Also, an algorithm handling only non-degenerate cases can be correct with respect to its specification, although in CGAL we would like to provide algorithms handling degeneracies.

Robustness

A design goal particularly relevant for the implementation of geometric algorithms is robustness. Many implementations of geometric algorithms lack robustness because of precision problems; see Chapter 11 for a discussion of robustness issues within CGAL.

Flexibility

The different needs of the potential application areas demand flexibility in the library. Four sub-issues of flexibility can be identified.

Modularity. A clear structuring of CGAL into modules with as few dependencies as possible helps a user in learning and using CGAL, since the overall structure can be grasped more easily and the focus can be narrowed to those modules that are actually of interest.

Adaptability. CGAL might be used in an already established environment with geometric classes and algorithms in which case the modules will most probably need adaptation before they can be used.

Extensibility. Not all wishes can be fulfilled with CGAL. Users may want to extend the library. It should be possible to integrate new classes and algorithms into CGAL.

Openness. CGAL should be open to coexist with other libraries, or better, to work together with other libraries and programs. The C++ Standard [C++98] defines with the C++ Standard Library a common foundation for all C++ platforms. So it is easy and natural to gain openness by following this standard. There are important libraries outside the standard, and CGAL should be easily adaptable to them as well.

Ease of Use

Many different qualities can contribute to the ease of use of a library. Which qualities are most important differs according to the experience of the user. The above-mentioned correctness and robustness issues are among these qualities. Of general importance is the length of time required before the library becomes useful. Another issue is the number of new concepts and exceptions to general rules that must be learned and remembered.

Ease of use tends to conflict with flexibility, but in many situations a solution can be found. The flexibility of CGAL should not distract a novice who takes the first steps with CGAL.

Uniformity. A uniform look and feel of the design in CGAL will help in learning and memorizing. A concept once learned should be applicable in all places where one would wish to apply it. A function name once learned for a specific class should not be named differently for another class.

CGAL is based in many places on concepts borrowed from STL (Standard Template Library) or the other parts of the C++ Standard Library. An example is the use of streams and stream operators in CGAL. Another example is the use of container classes and algorithms from the STL. So these concepts should be used uniformly.

Complete and Minimal Interfaces. A goal with similar implications as uniformity is a design with complete and minimal interfaces, see for example Item 18 in Ref. [Mey97]. An object or module should be complete in its functionality, but should not provide additional decorating functionality. Even if a certain function might look like it contributes to the ease of use for a certain class, in a more global picture it might hinder the understanding of similarities and differences among classes, and make it harder to learn and memorize.

Rich and Complete Functionality. We aim for a useful and rich collection of geometric classes, data structures and algorithms. CGAL is supposed to be a foundation for algorithmic research in computational geometry and

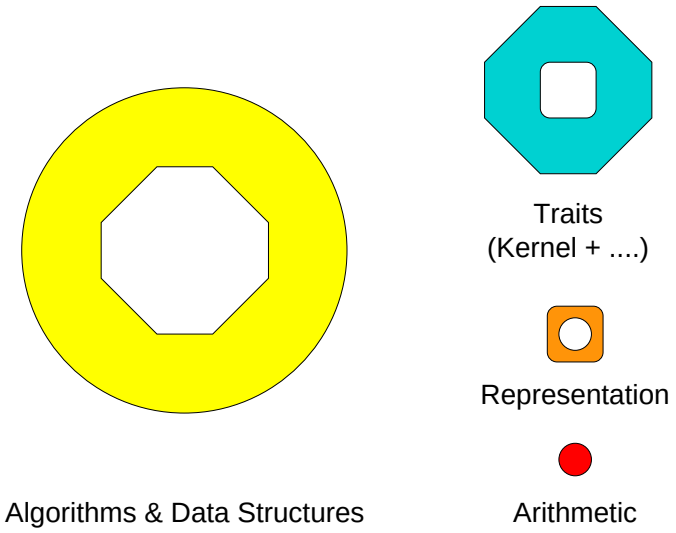


Figure 1.1: The generic design of CGAL.

therefore needs a certain breadth and depth. The standard techniques in the field are supposed to appear in CGAL.

Completeness is also related to robustness. We aim for general-purpose solutions that are, for example, not restricted by assumptions on general positions. Algorithms in CGAL should be able to handle special cases and degeneracies. In those cases where handling of degeneracies turns out to be inefficient, special variants that are more efficient but less general should be provided in the library in addition to the general algorithms handling all degeneracies. Of course, it needs to be clearly documented which degeneracies are handled and which are not.

Efficiency

For most geometric algorithms theoretical results for the time and space complexity are known. Also, the theoretic interest in efficiency for realistic inputs, as opposed to worst-case situations, is growing [Vle97]. For practical purposes, insight into the constant factors hidden in the O-notation is necessary, especially if there are several competing algorithms. Therefore, different implementations should be supplied if there is not one best solution, as, for example, when there is a tradeoff between time and space or a more efficient implementation when there are no or few degeneracies.

1.2 The overall design

The design goals, especially flexibility and efficient robust computation, have led us to opt for the generic programming paradigm using templates in C++.[1] In the overall design of CGAL two major layers can be identified, the layer of algorithms and data structures and the kernel layer (Figure 1.1).

Algorithms and data structures in CGAL are parameterized by the types of objects and operations they use. They work with any concrete template arguments that fulfill certain syntactical as well as semantic requirements. In

[1] In appropriate places, however, CGAL does and should make use of object-oriented solutions and design patterns, as well.

order to avoid long parameter lists, the parameter types are collected into a single class, called the traits class in CGAL (Chapter 4.) A *concept* is an abstraction of a type defined by a set of requirements. Any concrete type is called a *model* for a concept if it fulfills the set of requirements corresponding to the concept. Using this terminology, we can say a CGAL algorithm or data structure comes with a traits concept and can be used with any concrete traits model for this concept. Further contributions to CGAL should continue the current high level of genericity.

CGAL defines the concept of a geometry kernel. Ideally, any model for this concept can be used with any CGAL algorithm. This holds, of course, only if the requirements of an algorithm or data structure on its traits class are subsumed by the kernel concepts, *i.e.*, if an algorithm or data structure has no special requirements not covered in the definition of the kernel concept.

CGAL currently provides four models for the CGAL 2D and 3D kernel concept. Those are again parameterized and differ in their representation of the geometric objects and the exchangeability of the types involved. In all cases the kernels are parameterized by a number type, which is used to store coordinates and which determines the basic arithmetic of the kernel primitives. See Chapter 3 for more details.

There are further complementary layers in CGAL. The most basic layer is the configuration layer. This layer takes care of setting configuration flags according to the outcome of tests run during installation. The *support library* layer is documented in the Support Library Reference Manual and contains packages that deal with things such as visualization, number types, streams, and STL extensions in CGAL.

Chapter 2

Coding Conventions

*Sven Schönherr (*sven@inf.ethz.ch*)*

We do not want to impose very strict coding rules on the developers. What is most important is to follow the CGAL naming scheme described in the next section. However, there are some programming conventions (Section 2.2) that should be adhered to, rules for the code format (Section 2.3), and a mandatory heading for each source file (Section 2.4)

2.1 Naming scheme

The CGAL naming scheme is intended to help the user use the library and the developer develop the library. The rules are simple and easy to remember. Where appropriate, they aim for similarity with the names used in the STL. Deviations from the given rules should be avoided; however, exceptions are possible if there are *convincing* reasons.

General rules

- Words in the names of everything except concepts should be separated by underscores. For example, one would use *function_name* and *Class_name* instead of *functionName* or *Classname*.
- Words in the names of concepts (*e.g.*, template parameters) should be separated using capital letters. The only use of underscores in concept names is before the dimension suffix. For example, one should use a name such as ConvexHullTraits_2 for the concept in contrast to *Convex_hull_traits_2* for the name of the class that is a model of this concept. This different naming scheme for concepts and classes was adopted mainly for two reasons (a) it is consistent with the STL (cf. InputIterator) and (b) it avoids name clashes between concepts and classes that would require one or the other to have a rather contrived name.
- Abbreviations of words are to be avoided (*e.g.*, use *Triangulation* instead of *Tri*). The only exceptions might be standard geometric abbreviations (such as "CH" for "convex hull") and standard data structure abbreviations (such as "DS" for "data structure"). Unfortunately, the long names that result from the absence of abbreviations are known to cause problems with some compilers.
- Names of constants are uppercase (*e.g.*, *ORIGIN*).
- The first word of a class or enumeration name should be capitalized (*e.g.*, *Quotient*, *Orientation*).
- Function names are in lowercase (*e.g.*, *is_zero*).

- Boolean function names should begin with a verb. For example, use *is_empty* instead of simply *empty* and *has_on_bounded_side* instead of *on_bounded_side*.
- Names of macros should begin with the prefix *CGAL_*.

Geometric objects

- All geometric objects have the dimension as a suffix (*e.g.*, *Vector_2* and *Plane_3*). **Exception:** For *d*-dimensional objects there may be a dynamic and a static version. The former has the suffix *_d* (*e.g.*, *Point_d*), while the latter has the dimension as the first template parameter (*e.g.*, *Point<d>*).

Geometric data structures and algorithms

- Names for geometric data structures and algorithms should follow the "spirit" of the rules given so far, *e.g.* a data structure for triangulations in the plane is named *Triangulation_2*, and a convex hull algorithm in 3-space is named *convex_hull_3*.
- Member functions realizing predicates should start with *is_* or *has_*, *e.g.* the data structure *Min_ellipse_2* has member functions *is_empty* and *has_on_bounded_side*.
- Access to data structures is given via iterators and circulators (Chapter 10). For iterators and functions returning them we extend the STL names with a prefix describing the items to be accessed. For example, the functions *vertices_begin* and *vertices_end* return a *Vertex_iterator*. (Note that we use the name of the items in singular for the iterator type name and in plural for the functions returning the iterator.) Names related to circulators are handled similarly, using the suffix *_circulator*. For example, the function *edges_circulator* returns an *Edge_circulator*.

Kernel traits

All types in the kernel concept are functor types. We distinguish the following four categories:

1. **generalized predicates**, that is, standard predicates returning a Boolean value as well as functions such as *orientation()* that return an enumeration type (values from a finite discrete set).
2. **construction objects**, which replace constructors in the kernel,
3. **constructive procedures** and
4. **functors replacing operators**.

As detailed below, the names of these functors reflect the actions performed by calls to *operator()*. This naming scheme was chosen instead of one in which the computed object determines the name because this latter naming scheme is more natural for functions that compute values where the function call can be seen as a name for the object returned instead of functors that simply maintain data associated with the computation. It was also noted that the naming of functions and functors is not consistent, either in CGAL or in the STL (In some cases the action performed by a function determines its name (*e.g.*, *multiply()*) while in others the result of the action determines the name (*e.g.*, *product()*).), so achieving complete consistency with an existing naming scheme is not possible.

Here are the naming rules:

- All names in the kernel traits have the dimension as a suffix. This is necessary because in some cases (*e.g.*, the *Orientation_2* object) the absence of the suffix would cause a name conflict with an existing type or class (*e.g.*, the enumeration type *Orientation*).
- The names of generalized predicates are determined by their results. Furthermore, names are as much as possible consistent with the current kernel and the STL. So, for example, we have objects like *Has_on_bounded_side_2*, *Is_degenerate_2*, and *Is_horizontal_2*. According to the current kernel we also have *Left_turn_2*. For reasons of consistency with STL, all "less-than"-objects start with *Less_*, *e.g.*, *Less_xy_2*. Further examples are *Less_distance_to_point_2* and *Less_distance_to_line_2*. However, we have *Equal_2*, whereas the corresponding STL functor is called *equal_to*. Here, we give our dimension rule higher priority.
- The names of construction objects (category 2 above) follow the pattern *Construct_type_d*, where `type_d` is the type constructed, *e.g.*, *Construct_point_2* and *Construct_line_2*. The *operator()* of these functor classes is overloaded. This reduces the number of names to remember drastically, while replacing one of the constructions gets more complicated, but is still possible.
- Functors in category 3 above can be further subdivided into two classes:
  - constructive procedures that construct objects whose types are known *a priori*
  - procedures that construct objects whose types are not known *a priori*

  The names of functors in the first class also start with *Construct_* whenever a geometric object is constructed, otherwise they start with *Compute_*. Here, real numbers are not considered to be 1-dimensional geometric objects. For example, on the one hand we have *Construct_perpendicular_vector_2*, *Construct_midpoint_3*, *Construct_circumcenter_d*, *Construct_bisector_2*, and *Construct_point_on_3*, while on the other hand there are *Compute_area_2* and *Compute_squared_length_3*.

  For the second class, the names of the objects describe the (generic) action, *e.g. Intersect_2*.
- The equality operator (*operator==()*) is replaced by function objects with names of the form *Equal_k*, where `k` is the dimension number (*i.e.*, 2, 3, or *d*). For replacing arithmetic operators, we might also provide *Add_2*, *Subtract_2*, *Multiply_2*, and *Divide_2*. (As mentioned above, the action determines the name, not the result.) We think that these objects are not really needed. They are likely to be part of primitive operations that have a corresponding function object in the traits.

In addition, for each functor the kernel traits class has a member function that returns an instance of this functor. The name of this function should be the (uncapitalized) name of the functor followed by the suffix *_object*.For example, the function that retuns an instance of the *Less_xy_2* functor is called *less_xy_2_object*.

File names

- File names should be chosen in the "spirit" of the naming rules given above.
- If a single geometric object, data structure, or algorithm is provided in a single file, its name (and its capitalization) should be used for the file name. For example, the file *Triangulation_2.h* contains the data structure *Triangulation_2*.
- If a file does not contain a single class, its name should not begin with a capital letter.
- No two files should have the same names even when capitalization is ignored. This is to prevent overwriting of files on operating systems where file names are not case sensitive. A package that contains file names that are the same as files already in the release will be rejected by the submission script.
- The names of files should not contain any characters not allowed by all the platforms the library supports. In particular, it should not contain the characters ':', '*', or ' '.

2.2 Programming conventions

The first list of items are meant as rules, *i.e.*, you should follow them.

- Give typedefs for all template arguments of a class that may be accessed later from outside the class.

  The template parameter is a concept and should follow the concept naming scheme outlines in the previous section. As a general rule, the typedef should identify the template parameter with a type of the same name that follows the naming convention of types. For example

```
template < class GeometricTraits_2 >
class Something {
  public:
    typedef GeometricTraits_2 Geometric_traits_2;
};
```

  For one-word template arguments, the template parameter name should be followed by an underscore. (Note that using a preceding underscore is not allowed according to the C++ standard; all such names are reserved.)

```
template < class Arg_ >
class Something {
  public:
    typedef  Arg_  Arg;
};
```

- Use *const* when a call by reference argument is not modified, e.g. *int f(const A& a)*.
- Use *const* to indicate that a member function does not modify the object to which it is applied, *e.g.*, *class A { int f(void) const; };*. This should also be done when it is only conceptually *const*. This means that the member function *f()* is *const* as seen from the outside, but internally it may modify some data members that are declared *mutable*. An example is the caching of results from expensive computations. For more information about conceptually *const* functions and mutable data members see [Mey97].
- Prefer C++-style to C-style casts, *e.g.*, use *static_cast<double>(i)* instead of *(double)i*.
- Protect header files against multiple inclusion, *e.g.* the file `This_is_an_example.h` should begin/end with

```
#ifndef CGAL_THIS_IS_AN_EXAMPLE_H
#define CGAL_THIS_IS_AN_EXAMPLE_H
...
#endif // CGAL_THIS_IS_AN_EXAMPLE_H
```

The following items can be seen as recommendations in contrast to the rules of previous paragraph.

- Use `#define` sparingly.
- Do not rename the base types of C++ using *typedef*.
- When using an overloaded call, always give the exact match. Use explicit casting if necessary.
- Do not call global functions unqualified, that is, always specify explicitly the namespace where the function is defined.

2.3 Code format

- Lines should not exceed 80 characters (the SVN server warns about that when committing)
- Use indentation with at least two spaces extra per level.
- Write only one statement per line.
- Use C++-style comments, *e.g.*, *// some comment*.

2.4 File header

Each CGAL source file must start with a heading that allows for an easy identification of the file. The file header contains:

- a copyright notice, specifying all the years during which the file has been written or modified, as well as the owner(s) (typically the institutions employing the authors) of this work,
- the corresponding license (at the moment, only LGPL v2.1 and QPL v1.0 are allowed in CGAL), and a pointer to the file containing its text in the CGAL distribution,
- a disclaimer notice,
- then, there are 2 keywords, which are automatically expanded by SVN (there are options in SVN to suppress these expansions if you need):
  - $URL: $: the name of the source file in the repository, it also helps figuring out which package the file comes from,
  - $Id: $: the SVN revision number of the file, the date of this revision, and the author of the last commit.
- Then the authors of (non-negligible parts of) this file are listed, with optional affiliation or e-mail address.

For example and demo programs, the inclusion of the copyright notice is not necessary as this will get in the way if the program is included in the documentation. However, these files should always contain the name of the file realtive to the `CGAL_HOME` directory (*e.g.*, `examples/Convex_hull_3/convex_hull_3.cpp`) so the file can be located when seen out of context (*e.g.*, in the documentation or from the demos web page).

For the test-suite and the documentation source, these are not distributed at the moment, so there is no policy for now.

Chapter 3

Geometry Kernels

*Stefan Schirra (*stschirr@mpi-sb.mpg.de*)*

The layer of geometry kernels provides basic geometric entities of constant size[1] and primitive operations on them. Each entity is provided as both a stand-alone class, which is parameterized by a kernel class, and as a type in the kernel class. Each operation in the kernel is provided via a functor class[2] in the kernel class and also as either a member function or a global function. See [HHK+01] for more details about this design. Ideally, if the kernel provides all the primitives required, you can use any kernel as a traits class directly with your algorithm or data structure; see also Chapter 4. If you need primitives not provided by the kernel (yet), please read Section 3.6 below.

3.1 Cartesian and homogeneous representation

There are two different coordinate representations available in the kernel at present: Cartesian representation and homogeneous representation. Cartesian representation is the one you are familiar with. A point in the plane is given by its x- and its y-coordinates;[3] a point in space by its x-, y- and z-coordinates.[4] Homogeneous representation can be seen as a divison-free representation of Cartesian coordinates. There is an additional coordinate, sometimes called the *homogenizing* coordinate. So with homogeneous representation in 2-space[5], we have coordinates (x,y,w), where w is the homogenizing coordinate, and in 3-space[6], we have homogeneous coordinates (x,y,z,w). Since CGAL uses homogeneous representation for affine geometry (not for projective geometry), we assume $w \neq 0$. The Cartesian representation corresponding to $(x_0,x_1,\ldots,x_d,w)$ is $(x_0/w,x_1/w,\ldots,x_d/w)$. Hence, homogeneous representation is not unique; $(\alpha x,\alpha y,\alpha z,\alpha w)$ is an alternative representation to (x,y,z,w) for any $\alpha \neq 0$. Internally, CGAL always maintains a non-negative homogenizing coordinate.

With the homogeneous representation, division operations can be avoided. Homogeneous representation is advantageous over Cartesian representation whenever systems of linear equations with integral coefficients are to be solved. By Cramer's rule, the rational solutions all have the same denominator D. The Cartesian representation would be

$$(N_0/D,N_1/D,\ldots,N_{d-1}/D)$$

[1] In dimension d, an entity of size $O(d)$ is considered to be of constant size.
[2] A class which defines a member *operator()*.
[3] http://www.geom.uiuc.edu/docs/reference/CRC-formulas/node4.html
[4] http://www.geom.uiuc.edu/docs/reference/CRC-formulas/node39.html
[5] http://www.geom.uiuc.edu/docs/reference/CRC-formulas/node6.html
[6] http://www.geom.uiuc.edu/docs/reference/CRC-formulas/node43.html

while a corresponding less space-consuming homogeneous representation is

$$(N_0, N_1, \ldots, N_{d-1}, D)$$

For example, computing the Cartesian coordinates (x,y) of the intersection point of lines with equations $a_1X + b_1Y + c_1 = 0$ and $a_2X + b_2Y + c_2 = 0$ gives

$$(x,y) = \left(\frac{\begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}, -\frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} \right)$$

while with homogeneous representation we have

$$(x,y,w) = \left(\begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}, -\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}, \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} \right)$$

In general, however, homogeneous representation is not more space-efficient. Naive conversion from a rational Cartesian represenatation

$$(N_0/D_0, N_1/D_1, \ldots, N_{d-1}/D_{d-1})$$

to a homogeneous representation will lead to much bigger numbers, namely,

$$(N_0 \prod D_i/D_0, N_1 \prod D_i/D_1, \ldots, N_{d-1} \prod D_i/D_{d-1}, \prod D_i)$$

.

3.2 Cartesian versus homogeneous computation

Homogeneous representation has the disadvantage that predicates become more complicated. Testing equality of points is more complicated because the homogeneous representation is not unique. In the orientation predicate, the sign of a 3x3 determinant must be computed:

$$\text{sign} \begin{vmatrix} x_1 & y_1 & w_1 \\ x_2 & y_2 & w_2 \\ x_3 & y_3 & w_3 \end{vmatrix}$$

With Cartesian representation, it is essentially a 2x2 determinant only:

$$\text{sign} \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = \text{sign} \begin{vmatrix} x_1 - x_3 & y_1 - y_3 & 0 \\ x_2 - x_3 & y_2 - y_3 & 0 \\ x_3 & y_3 & 1 \end{vmatrix}$$

With homogeneous coordinates all formulas are homogeneous polynomials in the coordinates, *i.e.*, all monomials of homogeneous coordinates have the same total degree.[7] For a sign test for some polynomial expression over Cartesian coordinates you get a corresponding sign test with an expression over homogeneous coordinates by replacing each Cartesian coordinate x_{ij} by hx_{ij}/hw_j and then multiplying by the least common multiple of the denominators, *i.e.*, some multiple of the hw_j. Here, x_{ij} denotes the i-th Cartesian coordinate, hx_{ij} denotes the i-th homogeneous coordinate, and hw_j denotes the homogenizing coordinate of point j. Since all hw_j are positive in CGAL, sign is not affected by this multiplication.

[7] the sum of the degrees of each variable in the monomial

3.3 Available kernels

At present, there are two homogeneous and two Cartesian kernels, one with reference counting (Chapter 6) and one without. The corresponding kernel classes are

```
CGAL::Cartesian< FieldNumberType >
CGAL::Homogeneous< RingNumberType >
CGAL::Simple_cartesian< FieldNumberType >
CGAL::Simple_homogeneous< RingNumberType >
```

These are all parameterized by a number type, which is used for storing the coordinates and the arithmetic in the corresponding primitives and predicates. Actually, parameterization of *Homogeneous<>* involves two number types. While in the internal homogeneous representation, an integral number type is sufficient, rational numbers must sometimes be used outside the internal representation, for example, when the squared length of a vector is computed. To represent such rational values, there is a second number type whose default value is `CGAL::Quotient< RingNumberType >`. The type of this number type can be accessed as *Homogeneous< >::FT*. The internally used number type can be accessed as *Homogeneous<>::RT*. For the sake of a uniform interface for both representations, the Cartesian kernels provide such types as well. For Cartesian representation, both *FT* and *RT* map to same number type used everywhere in the implementation of the Cartesian kernel.

Whenever one wants to ensure that predicates are evaluated correctly, an exact number type is usually necessary. As exact computations are time consuming, filtering techniques have been developed. The predicate is first evaluated using efficient but inexact floating point arithmetic. At the same time an error bound is computed that is used to judge the quality of the computed solution. If the solution is not guaranteed to be correct, the predicate is re-evaluated using exact arithmetic. The classes

```
CGAL::Filtered_kernel< CK >
CGAL::Filtered_kernel_adaptor< CK >
```

are adaptors that add such a filtering mechanism to the predicates of a given Kernel *CK*. Note that only predicates are affected by these adaptors, the constructions are still the same as in the original kernel *CK*.

The difference between *Filtered_kernel* and *Filtered_kernel_adaptor* is that the latter affects the predicate functors in the kernel only while the former also affecs the behaviour of predicates that are implemented as global functions.

3.4 Kernel design and conventions

Each kernel object is provided as both a stand-alone class, which is parameterized by a kernel class (*Geo_object_D<K>*), and as a type in the kernel class (*K::Geo_object_D*). While the former use may be more natural for users not interested in the flexibility of the kernel (and is compatable with the original kernel design [FGK+00]), the latter syntax should be used in all code distributed with the library as it allows types in the kernel to be easily exchanged and modified. Similarly, each operation and construction in the kernel is provided via a function object class in the kernel class and also as either a member function or a global function; developers should use the function object classes to gain access to the functionality. See [HHK+01] for more details about this design and how it is accomplished.

The classes for the geometric objects in the kernel have a standardized interface.

- All classes have a *bbox()* member function computing a bounding box. oxLEDA

- All classes have a *transform(Aff_transformation_d t)* member function to compute the object transformed by *t*.
- Oriented $d-1$ dimensional objects[8] provide member functions *has_on_positive_side(Point_d)*, *has_on_boundary(Point_d)*, and *has_on_negative_side(Point_d)*. Furthermore, there is a member function *oriented_side(Point_d)* returning an object of type *CGAL::Oriented_side*.
- Full-dimensional bounded objects provide member functions *has_on_bounded_side(Point_d)*, *has_on_boundary(Point_d)*, and *has_on_unbounded_side(Point_d)*. Furthermore, there is a member function *bounded_side(Point_d)* returning an object of type *CGAL::Bounded_side*.
- Oriented objects have a member function *opposite()* returning the same object with opposite orientation.

3.5 Number-type based predicates

For a number of predicates, there are versions that operate on the coordinates directly, not on the geometric objects. These number-type based predicates ease re-use with non-CGAL types.

3.6 Missing functionality

Kernel traits do not provide redundant functionality. In particular, they do not provide a right turn predicate, since a left turn predicate exists. A right turn functor can be created from the left turn functor using the function *CGAL::swap_1*.

Whenever you need a predicate that is not present in the current kernel traits, you should first try to re-use the available predicates (you might rewrite the code or implement the new predicate using existing ones). If this is not feasible (especially for efficiency reasons), we have to decide on adding the new predicate to the kernel traits. If the new predicate is not too special, it will be added. Otherwise you cannot use the kernel as a traits class, but have to use additional traits.

See Section 3.2 on how to derive the homogeneous version of a predicate from the Cartesian version.

[8] Note that the dimension of an object might depend on its use. A line in the plane has dimension $d-1$. As a halfspace, it has dimension d.

Chapter 4

Traits Classes

*Bernd Gärtner (*gaertner@inf.ethz.ch*)*

The concept of a traits class is central to CGAL. The name "traits class" comes from a standard C++ design pattern [Mye95]; you may have heard about iterator traits which follow this design pattern. In CGAL, traits classes are something different, although the philosophy is similar in a certain sense.

4.1 What are traits classes in CGAL?

The algorithms in CGAL's basic library are implemented as function templates or class templates, usually having a template parameter whose name contains the word *Traits*. This template parameter represents a concept and so has a corresponding set of requirements that define the interface between the algorithm and the geometric (or numeric) primitives it uses. Any concrete class that serves as a model for this concept is a traits class for the given algorithm or data structure.

4.2 Why are traits classes in CGAL?

Using traits concepts as template parameters allows for customization of the behavior of algorithms without changing implementations. At least one model for each traits concept should be provided in CGAL (in the simplest case, the kernel models fit; see Section 4.5), but often more than one are provided in order to supply certain customizations that users may want. The user is also free to supply his or her own class as a model of the traits concept when the desired tailoring is not present in the library.

Traits classes allow for tailoring of algorithms not only at compile time but also at run time. Some primitive operations that appear in the traits class (in the form of functor types) may need additional data that are not known at compile time. A standard example is the following: we have three-dimensional points, but we want the convex hull of the two-dimensional points that arise after projecting along some direction in space, which is computed as the program runs. How does the algorithm get to know about this direction? If there is a traits class object as a parameter, the information can be provided to the proper primitives through a proper initialization of the traits class object. For this reason, traits class objects are passed as parameters to functions.

4.3 An example – planar convex hulls

Consider convex hulls in the plane. What are the geometric primitives a typical convex hull algorithm uses? Of course, this depends on the algorithm, so let us consider what is probably the simplest efficient algorithm, the so-called Graham Scan. This algorithm first sorts the points from left to right, and then builds the convex hull incrementally by adding one point after another from the sorted list. To do this, it must at least know about some point type, it should have some idea how to sort those points, and it must be able to evaluate the orientation of a triple of points. The signature of the Graham Scan algorithm in CGAL (actually a variation due to Andrews) is as follows:

template <class InputIterator, class OutputIterator, class Traits>
OutputIterator ch_graham_andrew(InputIterator first,
InputIterator beyond,
OutputIterator result,
Traits ch_traits)

TRAITS: operates on *Traits::Point_2* using *Traits::Left_turn_2* and *Traits::Less_xy_2*.

You notice that there is a template parameter named *Traits*, and you also see a comment that mentions three identifiers (*Point_2*, *Left_turn_2* and *Less_xy_2*) that have to be defined in the scope of the traits class in order for the algorithm to work. As you can guess, *Left_turn_2* is responsible for the orientation test, while *Less_xy_2* does the sorting. So, obviously, the traits class must provide these three identifiers. The requirements it has to satisfy beyond that are documented in full with the concept ConvexHullTraits_2.

4.3.1 Traits class requirements

Whenever you write a function or class that is parameterized with a traits class, you must provide the requirements that class has to fulfill. These requirements should be documented as a concept. For the example above, if you look in the manual at the description of the concept ConvexHullTraits_2, you will find that the traits class itself and the identifiers that are mentioned have to meet the following specifications:

4.4 Alphabetical List of Reference Pages

ConvexHullTraits_2

Concept

Types

ConvexHullTraits_2:: Point_2	The point type on which the convex hull functions operate.
ConvexHullTraits_2:: Less_xy_2	Binary predicate object type comparing *Point_2*s lexicographically. Must provide *bool operator()(Point_2 p, Point_2 q)* where *true* is returned iff $p <_{xy} q$. We have $p <_{xy} q$, iff $p_x < q_x$ or $p_x = q_x$ and $p_y < q_y$, where p_x and p_y denote x and y coordinate of point p, respectively.
ConvexHullTraits_2:: Left_turn_2	Predicate object type that must provide *bool operator()(Point_2 p,Point_2 q,Point_2 r)*, which returns *true* iff r lies to the left of the oriented line through p and q.

Creation

Only a copy constructor is required.

ConvexHullTraits_2 traits(& t);

Operations

The following member functions to create instances of the above predicate object types must exist.

Less_xy_2	*traits.less_xy_2_object()*
Left_turn_2	*traits.left_turn_2_object()*

This ends the copied manual text. Some comments are in order here. You might have expected *Less_xy_2* and *Left_turn_2* to be simply member functions of the traits class. Instead, they are functor types, and there are member functions generating instances of these types, *i.e.*, the actual functors. Reasons for this are the following.

- CGAL is designed to have an STL-like look-and-feel. All algorithms in the STL that depend on computational primitives (like a sorting algorithm depending on a comparison operator), receive those primitives via parameters which are functors. (The only way to pass an actual function as a parameter would be via function pointers.)
- More flexibility. In contrast to member functions, functors can carry data. For example, repeated calls to a function with only slightly different parameters might be handled efficiently by storing intermediate results. Functors are the natural framework here. See [HHK+01] for more exposition.

If you really look up the documentation of the concept in the manual, you will find a larger list of requirements. A traits class fulfilling this complete list of requirements can be used for all of the 2-dimensional convex hull algorithms provided in CGAL. For example, there are also algorithms that require a sorting of points by angle, and a traits class for that algorithm has to supply appropriate predicates for that. Still, to use the Graham Scan, a traits class meeting only the specifications listed above is sufficient.

4.4.1 CGAL-provided traits classes

As mentioned in Section 4.1, the traits class requirements define a concept. An actual traits class that complies with these requirements is a model for that concept. At least one such model must be provided for all CGAL algorithms. Often this is called the default traits class. Default traits classes are very easy to use, especially when they are invoked via default arguments. Look at the function *ch_graham_andrews* again. The signature does not tell the whole story. In reality, the third template parameter defaults to the default traits class, and the last function parameter defaults to a default instance of the default traits class. Of course, such behavior must be specified in the description of the function.

The implication is that a user can call *ch_graham_andrews* with just three parameters, which delimit the iterator range to be handled and supply the iterator for the result. The types and primitives used by the algorithm in this case are the ones from the CGAL 2D and 3D kernel.

In many cases, there are more than one traits classes provided by CGAL. In the case of convex hulls, for example, there are traits classes that interface the algorithms with the geometry kernel of LEDA. Though the user who has a third-party geometric kernel will not be able to profit from the CGAL or LEDA traits, he or she can still provide own traits classes, which meet the specified requirements.

4.5 Kernel as traits

Most default traits classes in CGAL are written in terms of the types and classes provided in the CGAL kernel. So one may wonder why it is not possible to plug the kernel in as a traits class directly. Ideally, it provides all the primitives an algorithm needs. However, some algorithms and data structures require specialized predicates that would not be appropriate to add to a general-purpose kernel. The traits classes for these algorithms and data structures should use kernel primitives wherever possible, and for those primitives not provided by the kernel the fixed naming scheme for predicates and constructions (Section 2.1) should be used to make the library more consistent and thus easier to use.

Chapter 5

Checks: Pre- and Postconditions, Assertions, and Warnings

*Sven Schönherr (*sven@inf.ethz.ch*)*

Much of the CGAL code contains checks. Some are there to check if the code behaves correctly, others check if the user calls routines in an acceptable manner. We describe the different categories of checks (Section 5.1), the usage of checks (Section 5.2), and a more selective means of controlling checks (Section 5.3). Finally, a statement about exception handling is given (Section 5.4).

5.1 Categories of checks

There are four types of checks.

- **Preconditions** check if a routine has been called in a proper fashion. If a precondition fails it is the responsibility of the caller (usually the user of the library) to fix the problem.
- **Postconditions** check if a routine does what it promises to do. If a postcondition fails it is the fault of this routine, so the author of the code is responsible.
- **Assertions** are other checks that do not fit in the above two categories, *e.g.* they can be used to check invariants.
- **Warnings** are checks for which it is not so severe if they fail.

Failures of the first three types are errors and lead to a halt of the program, failures of the last one only lead to a warning. Checks of all four categories can be marked with one or both of the following attributes.

- *Expensive* checks take considerable time to compute. "Considerable" is an imprecise phrase. Checks that add less than 10 percent to the execution time of their routine are not expensive. Checks that can double the execution time are. Somewhere in between lies the border line.
- *Exactness* checks rely on exact arithmetic. For example, if the intersection of two lines is computed, the postcondition of this routine may state that the intersection point lies on both lines. However, if the computation is done with *double*s as the number type, this may not be the case, due to roundoff errors.

By default, all standard checks (without any attribute) are enabled, while expensive and exactness checks are disabled. How this can be changed and how checks are actually used in the code are described in the next section.

5.2 Using checks

The checks are implemented as preprocessor macros; *i.e.*, *CGAL_<check_type>(<Cond>)* realizes a check of type *<check_type>* that asserts the condition *<Cond>*. For example,

```
CGAL_precondition( first != last);
```

checks the precondition that a given iterator range is not empty. If the check fails, an error message similar to

```
CGAL error: precondition violation!
Expr: first != last
File: <file name>
Line: <source code line>
```

is written to the standard error stream and the program is aborted. If an additional explanantion should be given to the user, macros *CGAL_<check_type>_msg(<Cond>,<Msg>)* can be used. The text in *<Msg>* is just appended to the failure message given above.

In case a check is more complicated and the computation does not fit into a single statement, the additional code can be encapsulated using *CGAL_<check_type>_code(<Code>)*. This has the advantage that the computation is not done if the corresponding category is disabled. For an example, suppose an algorithm computes a convex polygon. Thus we want to check the postcondition that the polygon is indeed convex, which we consider an expensive check. The code would look like this.

```
CGAL_expensive_postcondition_code( bool is_convex; )
CGAL_expensive_postcondition_code( /* compute convexity */ )
CGAL_expensive_postcondition_code( /* ... */ )
CGAL_expensive_postcondition_msg ( is_convex, \
                                   "The computed polygon is NOT convex!" );
```

As already mentioned above, the standard checks are enabled by default. This can be changed through the use of compile-time flags. By setting the flag *CGAL_NO_<CHECK_TYPE>* all checks of type *<CHECK_TYPE>* are disabled, *e.g.* adding *-DCGAL_NO_ASSERTIONS* to the compiler call switches off all checks for assertions. To disable all checks in the library, the flag *NDEBUG* can be set (which also switches off the *assert* macro from the C-library) . The flag *CGAL_NDEBUG* disables all checks in CGAL but does not affect the standard *assert* macro.

To enable expensive and exactness checks, respectively, the compile-time flags *CGAL_CHECK_EXPENSIVE* and *CGAL_CHECK_EXACTNESS* have to be supplied. However, exactness checks should only be turned on if the computation is done with some exact number type.

5.3 Controlling checks at a finer granularity

The macros and related compile-time flags described so far all operate on the whole library. Sometimes the user may want to have a more selective control. CGAL offers the possibility to turn checks on and off on a

per-package basis. Therefore a package-specific term is inserted in the macro names directly after the CGAL prefix, *e.g.*, *CGAL_kernel_assertion(<Cond>)*. Similarly, the uppercase term is used for the compile-time flags; *e.g.*, *CGAL_KERNEL_NO_WARNINGS* switches off the warnings in *only* the kernel. Other packages have their own specific terms as documented in the corresponding chapters of the reference manual.

The documentation of your new package has to name the term chosen to be part of the package-specific macros in order to enable the user to selectively turn off and on the checks of your package. For example, in the documentation of the optimisation package you can find a sentence similar to the following.

> The optimisation code uses the term OPTIMISATION for the checks; *e.g.*, setting the compile time flag *CGAL_OPTIMISATION_NO_PRECONDITIONS* switches off precondition checking in the optimisation code.

5.4 Exception handling

Some parts of the library, *e.g.*, the interval-arithmetic package, use exceptions, but there is no general policy concerning exception handling in CGAL.

Chapter 6

Reference Counting and Handle Types

*Stefan Schirra (*stschirr@mpi-sb.mpg.de*)*

6.1 Reference counting

As of release 2.1, a reference counting scheme is used for the kernel objects in the kernels *Cartesian* and *Homogeneous*. All copies of an object share a common representation object storing the data associated with a kernel object; see Figure 6.1.

The motivation is to save space by avoiding storing the same data more than once and to save time in the copying procedure. Of course, whether we actually save time and space depends on the size of the data that we would have to copy without sharing representations. The drawback is an indirection in accessing the data. Such an indirection is bad in terms of cache efficiency. Thus there are also non-reference-counting kernels available *Simple_cartesian* and *Simple_homogeneous*.

The reference counting in the kernel objects is not visible to a user and does not affect the interface of the objects. The representation object is often called the *body*. The object possibly sharing its representation with others is called a *handle*, because its data consists of a pointer to its representation object only. If the implementation of the member functions is located with the representation object and the functions in the handle just forward calls to the body, the scheme implements the *bridge* design pattern, which is used to separate an interface from its implementation. The intent of this design pattern is to allow for exchanging implementations of member functions hidden to the user, especially at runtime.

6.2 Handle & Rep

The two classes *Handle* and *Rep* provide reference counting functionality; see Figure 6.2. By deriving from these classes, the reference counting functionality is inherited. The class *Rep* provides a counter; representation classes derive from this class. The class *Handle* takes care of all the reference-counting related stuff. In particular, it provides appropriate implementations of copy constructor, copy assignment, and destructor. These functions take care of the counter in the common representation. Classes sharing reference-counted representation objects (of a class derived from *Rep*) do not have to worry about the reference counting, with the exception of non-copy-constructors. There a new representation object must be created and the pointer to the representation object must be set.

If *CGAL_USE_LEDA* is defined and *CGAL_NO_LEDA_HANDLE* is not defined, the types *Handle* and *Rep* are set to the LEDA types *handle_base* and *handle_rep*, respectively (yes, without a *leda_*-prefix). Use of the LEDA

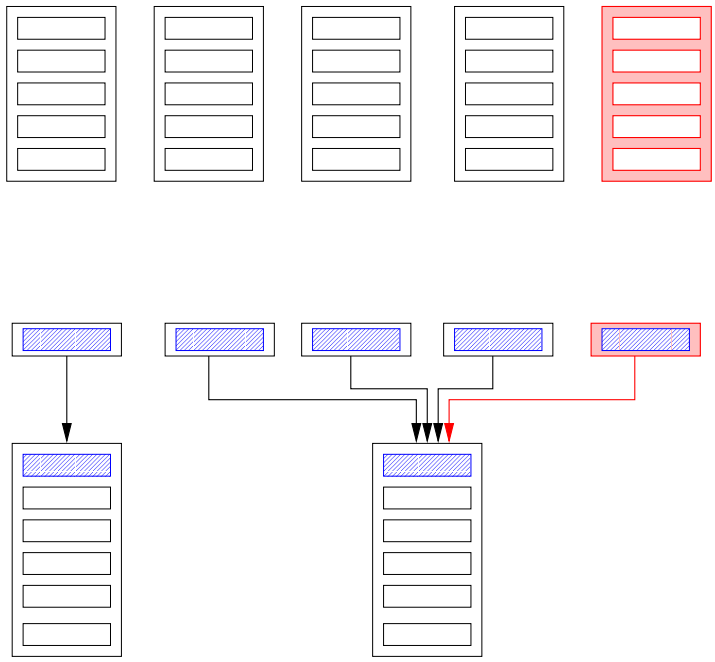


Figure 6.1: Objects using reference counting (bottom) share common representation; copying creates a new handle (drawn at the right) pointing to the same representation as the object copied. Without reference counting (top) all data are copied to the new object (drawn at the right);

class *handle_rep* implies that LEDA memory management is used for the representation types.

```
typedef handle_base      Handle;
typedef handle_rep       Rep;
```

Scavenging LEDA, we provide the identical functionality in the CGAL classes *Leda_like_handle* and *Leda_like_rep*. If LEDA is not available or *CGAL_NO_LEDA_HANDLE* is set, *Handle* and *Rep* correspond to these types.

```
typedef Leda_like_handle Handle;
typedef Leda_like_rep    Rep;
```

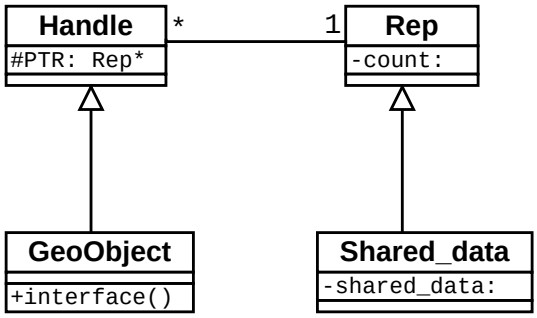


Figure 6.2: UML class diagram for Handle & Rep scheme.

6.3 Using Handle & Rep

In order to make use of the reference counting provided by the classes *Handle* and *Rep*, your interface class (the class providing the interface of the geometric object) must be derived from *Handle* and your representation class (the class containing the data to be shared) must be derived from *Rep*:

```
class My_rep : public Rep { /*...*/ };
class My_geo_object : public Handle
{
  public:
    My_geo_object(const My_geo_object& m);

    My_geo_object(Arg1, Arg2);
};
```

The class *My_geo_object* is responsible for allocating and constructing the *My_rep* object "on the heap". Typically, a constructor call is forwarded to a corresponding constructor of *My_rep*. The address of the new *My_rep* is assigned to *PTR* inherited from *Handle*, e.g.:

```
My_geo_object::My_geo_object(Arg1 a1, Arg2 a2)
{ PTR = new My_rep(a1, a2); }
```

The default constructor of *Handle* is called automatically by the compiler and the reference counting is initialized. You always have to define a copy constructor for *My_geo_object* and to call the copy constructor of *Handle* there:

```
My_geo_object::My_geo_object(const My_geo_object& m)
 : Handle( m)
{}
```

That's it! There is no need to define a copy assignment operator nor is there a need to define a destructor for the derived class *My_geo_object*! Handle & Rep does the rest for you! You get this functionality by

#include <CGAL/Handle.h>

It is common practice to add a (protected) member function *ptr()* to the class *My_geo_object*, which casts the *PTR* pointer from *Rep** to the actual type *My_rep**.

Note that this scheme is meant for non-modifiable types. You are not allowed to modify data in the representation object, because the data are possibly shared with other *My_geo_object* objects.

6.4 Templated handles

Factoring out the common functionality in base classes enables re-use of the code, but there is also a major drawback. The *Handle* class does not know the type of the representation object. It maintains a *Rep** pointer. Therefore, this pointer must be cast to a pointer to the actual type in the classes derived from *Handle*. Moreover, since the *Handle* calls the destructor for the representation through a *Rep**, the desctructor of *Rep* must be

virtual. Finally, debugging is difficult, again, because the *Handle* class does not know the type of the representation object. Making the actual type of the representation object a template parameter of the handle solves these problems. This is implemented in class template *Handle_for*. This class assumes that the reference-counted class provides the following member functions to manage its internal reference counting:

- *add_reference*
- *remove_reference*
- *bool is_referenced*
- *bool is_shared*

See the UML class diagram in Figure 6.3. The reference counting functionality and the required interface can be inherited from class *Ref_counted*.

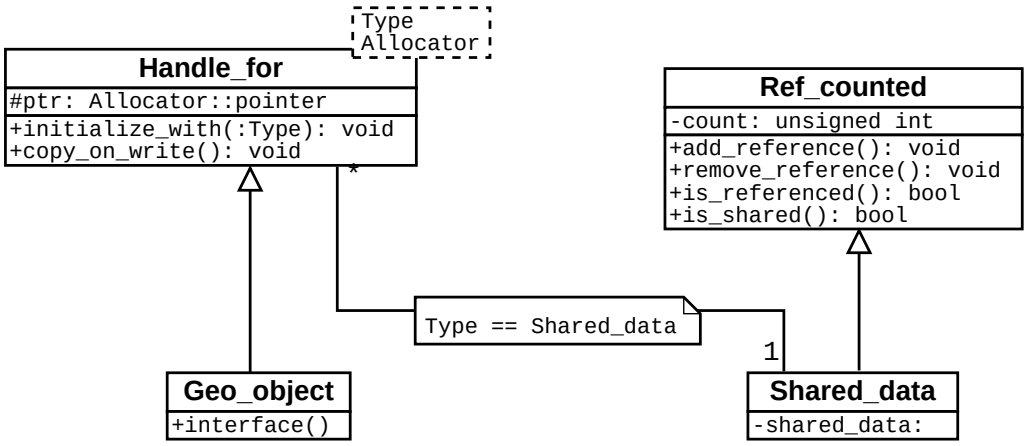


Figure 6.3: UML diagram for templated handles.

Kernel objects have used such a handle/rep scheme since release 2.2.

6.5 Using templated handles

In order to make use of the reference counting provided by the classes *Handle_for* and *Ref_counted*, your representation class, let's say *My_rep* (the class containing the data to be shared), must provide the interface described above (*e.g.*, by deriving from *Ref_counted*), and your interface class (the class providing the interface of the geometric object) must be derived from *Handle_for<My_rep>*. It is assumed that the default constructor of *My_rep* sets the counter to 1 (the default constructor of *Ref_counted* does this, of course):

```
class My_rep : public Ref_counted { /*...*/ };

class My_geo_object : public Handle_for<My_rep>
{
  public:
    My_geo_object(const My_geo_object& m);

    My_geo_object(Arg1, Arg2);
};
```

You should also define a copy constructor for *My_rep* as this may be used in in-place creation in the allocator scheme. The constructors of class *My_geo_object* are responsible for constructing the *My_rep* object. Typically, a corresponding constructor call of *My_rep* is forwarded to *Handle_for*.

```
My_geo_object::My_geo_object(Arg1 a1, Arg2 a2)
 : Handle_for<My_rep>( My_rep(a1, a2)) {}
```

Sometimes, you have to do some calculation first before you can create a representation object in a constructor of *My_rep*. Then you can use the *initialize_with()* member function of *Handle_for*.

In both cases, *Handle_for* takes care of allocating space for the new object.

If you define a copy constructor for *My_geo_object* you have to call the copy constructor of *Handle_for* there:

```
My_geo_object::My_geo_object(const My_geo_object& m)
 : Handle_for<My_rep>( m)
{}
```

That's it! Again, there is no need to define a copy assignment operator nor is there a need to define a destructor for the derived class *My_geo_object*! *Handle_for* does the rest for you!

Handle_for provides you with an option to modify the data. There is a *copy_on_write()* that should be called right before every modification of the data. It ensures that only your data are overwritten:

```
void
My_geo_object::set_x(double d)
{
  copy_on_write();
  ptr->x = d;
}
```

You get this functionality by

#include <CGAL/Handle_for.h>

6.6 Allocation

Class *Handle_for* has two template parameters. Besides the type of the stored object, there is also a parameter specifying an allocator. Any concrete argument must be a model for the *Allocator* concept defined in the C++ standard. There is a default value for the second parameter defined as *CGAL_ALLOCATOR(T)*. But you can also choose your own, for example

```
class My_geo_object : public Handle_for<My_rep, leda_allocator<My_rep> >
{ /* ... */ };
```

The default allocator is defined in

#include <CGAL/memory.h>

See Chapter 7 for more information.

Chapter 7

Memory Management

*Michael Seel (*seel@mpi-sb.mpg.de*)*

One of the design goals of CGAL (Section 1.1) is efficiency, and this means not only implementing efficient algorithms but also implementing them efficiently. One way to improve the efficiency of an implementation is through efficient memory management. Here we describe one way to address this using the allocator interface.

7.1 The C++ standard allocator interface

We first give a short presentation of the memory allocator interface. Objects of type *allocator<T>* can be used to obtain small, typed chunks of memory to be used, for example, as static members of a class. This is especially interesting with classes of a constant size that are frequently allocated and deallocated (geometric objects, etc.), since a memory allocator can maintain the corresponding memory chunks in local blocks and thus can answer allocation and deallocation calls much faster than the corresponding system calls. We first recapitulate the interface of an allocator:

Class *CGAL::allocator<T>*

Definition

An instance *A* of the data type *allocator<T>* is a memory allocator according to the C++standard.

Types

Local types are *size_type*, *difference_type*, *value_type*, *pointer*, *reference*, *const_pointer*, and *const_reference*.

allocator<T>:: template <class T1> rebind

allows the construction of a derived allocator:
allocator<T>::template rebind<T1>::other
is the type *allocator<T1>*.

Creation

allocator<T> A; introduces a variable *A* of type *allocator<T>*.

Operations

pointer *A.allocate(size_type n, const_pointer = 0)*

returns a pointer to a newly allocated memory range of size *n * sizeof(T)*.

void *A.deallocate(pointer p, size_type n)*

deallocates a memory range of *n * sizeof(T)* starting at *p*.
Precondition: the memory range was obtained via *allocate(n)*.

pointer *A.address(reference r)*

returns `&`*r*.

const_pointer *A.address(const_reference r)*

returns `&`*r*.

void *A.construct(pointer p, const_reference r)*

copies the object referenced by *r* to **p*. (Technically this is achieved by an inplace new *new((void*)p) T(r)*).

void *A.destroy(pointer p)*

destroys the object referenced via *p* by calling *p->~T()*.

size_type *A.max_size()*

the largest value *n* for which the call *allocate(n,0)* might succeed.

7.2 The allocator macro

The macro *CGAL_ALLOCATOR* is defined in the file <*CGAL/memory.h*> to be the standard allocator from <*memory*>. However, the user can redefine it, for example, if LEDA is present, he can define it (before including any CGAL header file) this way :

```
#include <LEDA/allocator.h>
#define CGAL_ALLOCATOR(t) leda_allocator<t>
```

7.3 Using the allocator

How should a data structure use the allocator mechanism? Just make the allocator one of the template arguments of the data structure. Then use a static member object to allocate items on the heap that you want to keep optimized regarding allocation and deallocation. We show an example using a trivial list structure:

```
#include <CGAL/memory.h>

template <typename T>
class dlink
{ T some_member; };

template < typename T, typename Alloc = CGAL_ALLOCATOR(dlink<T>) >
class list
{
public:
  typedef dlink<T>* dlink_ptr;
  typedef Alloc list_allocator;

  static list_allocator M;

list() {
  p = M.allocate(1);          // allocation of space for one dlink
  M.construct(p,dlink<T>());  // inplace construction of object
}

~list() {
  M.destroy(p);      // destroy object
  M.deallocate(p,1); // deallocate memory
}

private:
  dlink_ptr p;
};

// init static member allocator object:
template <typename T, typename Alloc>
typename list<T,Alloc>::list_allocator list<T,Alloc>::M =
             typename list<T,Alloc>::list_allocator();


int main()
{
  list<int> L;
  return 0;
}
```

Chapter 8

Namespaces

*Stefan Schirra (*stschirr@mpi-sb.mpg.de*)*

Names, in particular (member) function names and class names should be descriptive and easily remembered. So it is not surprising that different libraries or packages choose the same name for corresponding or similar classes and functions. A common approach to solving the naming problem is to add a prefix, for example, OpenGL adds *gl* and FLTK adds *fl*. LEDA uses prefix *leda_* to some extent, but you have to tell LEDA not to make the corresponding unprefixed names available as well.[1] Initially, CGAL used prefix *CGAL_*. At the beginning of 1999, it was decided to drop prefix *CGAL_* and to introduce namespace *CGAL*.

8.1 What are namespaces

A namespace is a scope with a name.[2] Inside the namespace, *i.e.*, the named scope, all names defined in that scope (and made known to the compiler) can be used directly. Outside a namespace, a name defined in the namespace has to be qualified[3] by the namespace name, for example *CGAL::Object*, or they have to be made usable without qualification by a so-called *using declaration*,

```
using CGAL::Object;
Object obj;  // name is now known
```

There is also a statement to make all names from a namespace available in another scope, but this is a bad idea. Actually, in order not to set a bad example, we recommend not to use this in CGAL's example and demo programs.

8.2 Namespace std

The names form the standard C++ library, especially those from the standard template library, are (supposed to be) in namespace *std*. This subsumes the I/O-library and also the C-library functions. You have to qualify streams and so by *std::*, too. That is:

[1] CGAL's makefile does this by setting *-DLEDA_PREFIX*.
[2] There are also unnamed namespaces. They are intented to replace file scope.
[3] This is a somewhat simplified view; read further for more details.

```
std::cout << "Hello CGAL" << std::endl;
```

or you have to add *using* declarations for the names you want to use without *std::* qualification. Whenever a platform does not put names into namespace *std*, CGAL adds the names it needs to namespace *std*. This is done by the configuration tools.

As for the C-library functions, you should use the macro `CGAL_CLIB_STD` instead of `std`:

```
CGAL_CLIB_STD::isspace(c)
```

8.3 Namespace CGAL

All names introduced by CGAL should be in namespace *CGAL*, *e.g.*:

```
#include <something>

namespace CGAL {

class My_new_cgal_class {};

My_new_cgal_class
my_new_function( My_new_cgal_class& );

} // namespace CGAL
```

Make sure not to have include statements nested between `namespace CGAL {` and `} // namespace CGAL`. Otherwise all names defined in the file included will be added to namespace *CGAL*. (Some people use the macros *CGAL_BEGIN_NAMESPACE* and *CGAL_END_NAMESPACE* in place of the `namespace CGAL {` and `} // namespace CGAL`, respectively, for better readability.)

8.4 Name lookup

The process of searching for a definition of a name detected in some scope is called name lookup. Simply speaking, name lookup looks up names in the scope where the name is used, and if not found, the lookup proceeds in successively enclosing scope until the name is found. It terminates as soon as the name is found, no matter whether the name found fits or not. If a name is qualified by a namespace name, that namespace is searched for the name.

8.4.1 Argument-dependent name lookup

Unqualified name lookup, *i.e.*, lookup when there is no namespace or class name and no scope resolution operator `::`, proceeds like qualified name lookup, but name lookup for a function call is supposed to search also the namespaces of the argument types for a matching function name. This is sometimes called Koenig-lookup.

8.4.2 Point of instantiation of a template

Name lookup in a template is slightly more complicated. Names that do not depend on template parameters are looked up at the point of definition of the template (so they must be known at the point of definition). Names depending on the template parameters are looked up at the point of instantiation of the template. The name is searched for in the scope of the point of instantiation and the scope of the point of definition. Here is a small example:

```
namespace A {

template <class T>
const T&
mix(const T& a1, const T& a2)
{ return a1 < a2 ? a1 : a2; }

template <class T>
const T&
use(const T& a1, const T& a2)
{ return mix( a1, a2); }

} // namespace A

namespace B {

template <class T>
const T&
mix(const T& a1, const T& a2)
{ return a2 < a1 ? a1 : a2; }

double
use_use( const double& t1, const double& t2)
{ return A::use(t1,t2); }

} // namespace B

int
main()
{
  B::use_use( 0.0, 1.0);
  return 0;
}
```

There is a ambiguity, because both the scope enclosing the point of instantiation and the scope enclosing the point of definition contain a *mix* function. The mips compiler on IRIX complains about this ambiguity:

```
bash-2.03$ CC -64 poi.cpp
cc-1282 CC: ERROR File = poi.cpp, Line = 11
  More than one instance of overloaded function "mix" matches the argument list.

            Function symbol function template "B::mix(const T &, const T &)"
                      is ambiguous by inheritance.
            Function symbol function template "A::mix(const T &, const T &)"
                      is ambiguous by inheritance.
```

```
           The argument types are:  (const double, const double).
  { return mix( a1, a2); }
           ^
         detected during instantiation of
                   "const double &A::use(const double &, const double &)"

1 error detected in the compilation of "poi.cpp".
```

There wouldn't be any problems, if `B::use_use()` would be a template function as well, because this would move the point of instantiation into global scope.

By the way, `gcc 2.95` uses the function defined at the point of definition.

8.5 Namespace CGAL::NTS

Note: *This section will be revised once the forthcoming revision of the C++-standard gets into a more definite state. The standard library has similar problems, e.g. for swap(), see*[4] *issues 225, 226, and 229. Currently, CGAL::NTS does not exist anymore, and the CGAL_NTS macro boils down to CGAL::. As the future interface is not yet fixed, people should still follow the guidelines given below.*

What are the conclusions from the previous subsection. If *A* plays the role of *std* and *B* the role of CGAL, we can conclude that CGAL should not define template functions that are already defined in namespace *std*, especially *min* and *max*. Also, letting CGAL be namespace *A*, we should not define templates in CGAL that are likely to conflict with templates in other namespaces (playing the role of *B*): Assume that both CGAL and some other namespace define an *is_zero()* template. Then an instantiation of some CGAL template using *is_zero()* that takes place inside the other namespace causes trouble. For this reason, we have another namespace *NTS* nested in *CGAL*, which contains potentially conflicting template functions.

8.5.1 Which function calls should be qualified in CGAL code?

Out current policy is:

- *max* should be used without qualification
- *min* should be used without qualification
- For the following functions, templates are provided in nested namespace *NTS*:

  abs
  compare
  gcd
  is_negative
  is_positive
  is_one
  is_zero
  sign
  square

[4] `http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_toc.html`

Calls of the above functions should be qualified using macro *CGAL_NTS*, which maps to *CGAL::NTS::*[5]: For example,

```
if ( CGAL_NTS is_zero(0) ) { /* ... */ }
```

Qualification with *CGAL* does not work.

- The following functions can be qualified by *CGAL_NTS* as well:

  to_double

  is_valid

  is_finite

  sqrt

  div

  Whenever the argument of *sqrt* is a concrete type, *i.e.*, it does not depend on a template parameter, you should qualify the call of *sqrt*, for example `CGAL_CLIB_STD::sqrt(2.0)`

  Here, qualification with *CGAL* works as well.

Summarizing, you can always qualify functions on number types with *CGAL_NTS* besides *min* and *max*.

[5] The use of the macro eases future changes in our policy.

Chapter 9

Polymorphic Return Types

*Stefan Schirra (*stschirr@mpi-sb.mpg.de*)*

For some geometric operations, the type of the result of the operation is not fixed a priori, but depends on the input. Intersection computation is a prime example. The standard object-oriented approach to this is defining a common base class for all possible result types and returning a reference or a pointer to an object of the result type by a reference or pointer to the base class. Then all the virtual member functions in the interface of the base class can be applied to the result object and the implementation corresponding to the actual result type is called. It is hard to define approriate base class interface functions (besides *draw()*).

CGAL has chosen a different approach, since CGAL wants to avoid large class hierarchies. With the CGAL class *Object*, you can fake a common base class, see Figure 9.1.

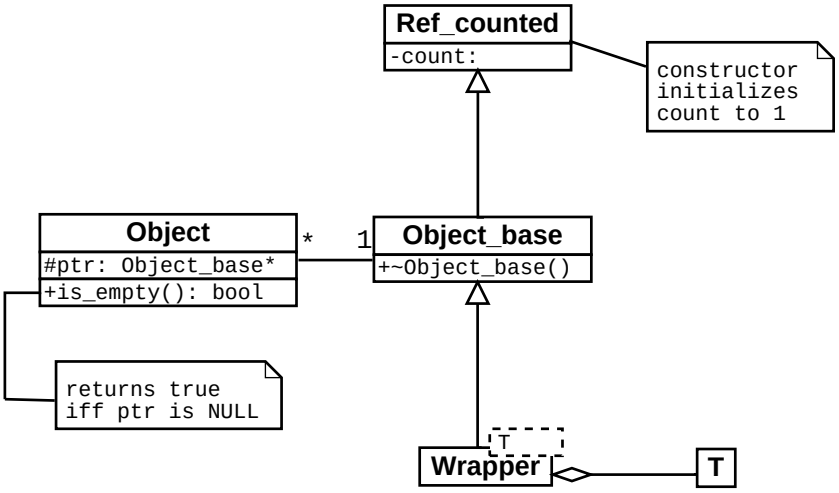


Figure 9.1: UML class diagram for faked object hierarchies (since 2.2-I-4).

Functions having a polymorphic return type create an object of the actual result type and wrap it into an object of type *Object*. You can use the *make_object()* function to do this.

```
template <class R>
Object
intersection(const Plane_3<R>& plane1, const Plane_3<R>& plane2);
```

The following piece of code is taken from the above CGAL intersection routine:

```
if (det != zero)
{
    is_pt = Point_3<R>(zero, c*s-d*r, d*q-b*s, det);
    is_dir = Direction_3<R>(det, c*p-a*r, a*q-b*p);
    return make_object(Line_3<R>(is_pt, is_dir));
}
```

There is only one member operation defined for *Object*, a test for an empty object. In order to make use of the returned object, you can try to assign it to the possible return types using *CGAL::assign(Candidate_return_type, Object)* function; see also the definition of the class *Object* in the CGAL kernel manual.

For functions potentially computing more than one polymorphic objects, some CGAL functions use *std::list<CGAL::Object>* as return value.

```
std::list<CGAL::Object>
fct_might_return_several_objects_of_different_types(...);
```

A more generic approach is the use of *OutputIterators*, just like the STL does:

```
template <typename OutputIterator>
OutputIterator
fct_might_return_several_objects_of_different_types(..., OutputIterator result);
```

where *iterator_traits<OutputIterator>::value_type* must be *Object*. The sequence of objects returned starts with the output iterator passed to the function. The output iterator returned is the past-the-end iterator of the constructed sequence of objects.

Chapter 10

Iterators and Circulators (and Handles)

*Mariette Yvinec (*yvinec@sophia.inria.fr*)*

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

The concept of an iterator is one of the major tools of the genericity in STL. Iterators are used almost everywhere in the STL to achieve the communication between containers and algorithms. Iterators are widely used in CGAL too. CGAL extends the idea of the iterator, which works for linear data structures, to circular data structures by defining the concept of a circulator. Circulators are quite similar to iterators, with the major difference being the absence of a past-the-end position in a sequence. Note that circulators are NOT part of the STL, but of CGAL.

In CGAL, we also define the concept of a handle, which bahaves roughly like a pointer to an object without an increment or decrement operation. More details about handles and their requirements can be found in the CGAL Support Library Reference Manual. Section 10.3.1 below discusses when handles should be used in your code.

The concept of iterators is relatively well described in textbooks such as Stroustrup's *The C++ Programming Langage* [Str97] and Austern's *Generic Programming and the* STL [Aus98] and in the Use of STL and STL Extensions in CGAL manual (which also discusses the circulator concept). Thus we will not give a full description of this concept here but only a few hints about how to use and write iterators (and circulators) in CGAL. Developers should consult the above-mentioned references to become familiar with the iterator and circulator concepts in general and, in particular, iterator and circulator ranges, dereferencable and past-the-end values, mutable and constant iterators and circulators, and the different categories (forward, bidirectional, random-access, etc.) of iterators and circulators.

10.1 Iterator and circulator traits

The algorithms working with iterators and/or circulators often need to refer to types associated with the iterator or circulator (*e.g.*, the type of the object referred to by the iterator or the type of the distance between two circulators). These types are usually declared in the iterator or circulator classes. However, for pointer classes, which can be valid models of the iterator and circulator concepts, this not possible. Thus iterator traits have been introduced to resolve this problem. An algorithm using an iterator of the type *Iter* will find the relevant types in an instantiation of a small templated class *iterator_traits*.

There is a general templated version of *iterator_traits* that looks like:

```
template <class Iter>
struct iterator_traits {
  typedef typename Iter::iterator_category  iterator_category ;
  typedef typename Iter::value_type         value_type;
  typedef typename Iter::difference_type    difference_type;
  typedef typename Iter::pointer            pointer;
  typedef typename Iter::reference          reference;
};
```

and a partial specialization of *iterator_traits* classes for pointers:

```
template <class T>
struct iterator_traits<T*> {
  typedef random_access_iterator            iterator_category ;
  typedef T                                 value_type;
  typedef ptrdiff_t                         difference_type;
  typedef T*                                pointer;
  typedef T&                                reference;
};
```

10.2 Input and output iterators

Operator * for input and output iterators

The operator * of input and output iterators has a restricted semantics. Input iterators are designed for input operations, and it is not required that the value type *T* of an input iterator *it* be assignable. Thus, while assignments of the type *t = *it* are the usual way to get values from the input iterators, statements like **it = ...* are likely to be illegal. On the other hand, output iterators are designed for write operations, and the only legal use of the operator * of an output iterator *it* is in the assignement **it = ...* The code of a standard copy function of the STL provides an example of both of these operations:

```
template< class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first,
                    InputIterator last,
                    OutputIterator result) {
    while (first != last)
    {
       *result = *first;
       ++first;
       ++result;
    }
    return result;
}
```

The first two arguments of *copy* are of type *InputIterator* (meaning any type that fulfills the requirements for an input iteraor) while the third one is of type *OutputIterator*. If these types were exchanged, then the statement **result = *first;* might not be valid.

Stream iterators

STL provides a special type of input iterator called *istream_iterator*, which is designed to be bound to an object of the class *istream* and provides a way to read a sequence of values from the input stream to which it is bound. For example, the following code reads numbers of type *double* from the standard input stream *cin* and computes their sum.

```
istream_iterator<double> it(cin);
istream_iterator<double> end();

double sum=0.0;
while(it != end) {
   sum += *it;
   ++it;
}
cout << sum << endl;
```

In a similar fashion, STL provides the type *ostream_iterator*, which is designed to be bound to an object of the class *ostream* and used to output values to the output stream to which it is bound.

CGAL provides extensions of the classes *istream_iterator* and *ostream_iterator*. The class *CGAL::Ostream_iterator<T,Stream>* is an output iterator adaptator for the stream class *Stream* and value type *T*. It provides output iterators that can be used to ouput values of type *T* to objects of the class *Stream*. For example, the following code fragment inserts a list of segments into a window stream (*i.e.*, it draws the segments in the window) using the standard copy function:

```
typedef CGAL::Cartesian<double>   K;
typedef K::Segment_2              Segment;

std::vector<Segment>       segments;
CGAL::Window_stream        W( 400, 400);

int main (int argc, char** argv)
{
  // initialize segments
  std::copy( segments.begin(),
             segments.end(),
             CGAL::Ostream_iterator< Segment, CGAL::Window_stream>( W));
}
```

Likewise, the class *CGAL::Istream_iterator<T,Stream>* is an input iterator adaptor for the stream class *Stream* and value type *T*. These adaptors are particularly useful for stream classes that are similar to but not derived from *std::istream* and *std::ostream*. The only requirements of the stream classes are that they define *operator>>* (for *Istream_iterator*) and *operator<<* (for *Ostream_iterator*).

Insert iterators

Insert iterators are output iterators that can be used to insert items into containers. With regular iterator classes, the code given above for the *copy* function of STL, causes the range *[first,last)* to be copied into an existing range starting with *result*. No memory allocation is involved and the existing range is overwritten. With an insert iterator supplied as the third argument, the same code will cause elements to be inserted into the container with which the output iterator is associated. That is, new memory may be allocated for these inserted elements.

The STL provides three kinds of insert iterators: *insert_iterator*s, *back_insert_iterator*s and *front_insert_iterator*s. The *back_inserter_iterator*s are used to insert elements at the end of a container by using the *push_*

back member function of the container. Similarly, *front_insert_iterator*s are used to insert elements at the beginning of a container by using the container's *push_front* function. The general *insert_iterator* is used to insert elements at any point in a container, by using the container's *insert* member function and a provided location of the insertion.

For convenience, STL provides the templated functions (or adaptors) *front_inserter*, *back_inserter* and *inserter* to get inserters from containers.

```
template<class Container, class Iterator>
insert_iterators<Container> inserter(Container& c, Iterator it);

template<class Container>
back_insert_iterators<Container> back_inserter(Container& c);

template<class Container>
front_insert_iterators<Container> front_inserter(Container& c);
```

Thus, the *inserter* adaptor can be called for any container that has an *insert* member function, and *back_inserter* (resp. *front_inserter*) can be called for any container that has a *push_back* (resp. *push_front*) member function. Some versions of STL (in particular, the one of KCC and Borland) also require that containers define a *value_type* and a *const_reference* type.

The following code will insert 200 copies of the value 7 at the end of *vec*.

```
void g(vector<int>& vec)
{
    fill_n(std::back_inserter(vec),200,7);
}
```

and this code will insert the points contained in the vector *vertices* into a Delaunay triangulation data structure:

```
  typedef CGAL::Cartesian<double>                          K;
  typedef CGAL::Triangulation_euclidean_traits_2<K>        Gt;
  typedef CGAL::Triangulation_vertex_base_2<Gt>            Vb;
  typedef CGAL::Triangulation_face_base_2<Gt>              Fb;
  typedef CGAL::Triangulation_default_data_structure_2<Gt,Vb,Fb>
                                                           Tds;
  typedef CGAL::Delaunay_triangulation_2<Gt,Tds>           DT;

  DT triangulation;

  std::copy( vertices.begin(),
             vertices.end(),
             std::back_inserter( triangulation ));
```

10.3 Writing code with and for iterators, circulators, and handles

Because you should write generic code for CGAL, algorithms that require a sequence of data for input should be written to take an iterator (or circulator) range as input instead of, say, a particular container. Similarly,

algorithms that compute a sequence of data as output should place the output data into an output iterator range. Both of these points are illustrated by the prototype of the following function that computes the convex hull of a set of points in two dimensions:

template <class InputIterator, class OutputIterator>
OutputIterator convex_hull_points_2(InputIterator first,
InputIterator beyond,
OutputIterator result,
Traits ch_traits)

generates the counterclockwise sequence of extreme points of the points in the range [*first*,*beyond*). The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. It is not specified at which point the cyclic sequence of extreme points is cut into a linear sequence.

Also, when writing container classes, you should be sure to provide iterators and/or circulators for the containers and design the interfaces so they can be used with generic algorithms from the STL and other CGAL algorithm. Here we give a few more details about how to accomplish these goals.

10.3.1 Handle, iterator, or circulator?

Handles are indirect references that do not move, so whenever you need a pointer-like reference to a single element of a data structure, and it is not necessary to iterate (or circulate), use a handle. In contrast, iterators should be used when you want to move (that is, iterate) over a linear sequences of elements. When the sequence is circular, prefer a circulator over an iterator.

10.3.2 Writing functions for iterators AND circulators

To make your code as generic as possible, you should, where appropriate, write functions that can accept either a circulator range or an iterator range to delimit the input values. Since empty circulator ranges are represented differently than empty iterator ranges, the following function is defined in `<CGAL/circulator.h>` so the test for an empty range can be done generically:

template< class IC>
bool 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.

One would use this function in conjunction with a *do-while* loop as follows:

```
if ( ! CGAL::is_empty_range( i, j) )
{
  do
  {
    // ...
  } while ( ++i != j )
}
```

The following two macros are also defined as a generic means for iterating over either a linear or circular sequence:

CGAL_For_all(ic1, ic2)

CGAL_For_all_backwards(ic1, ic2)

See the Circulator documentation in the Support Library Reference Manual for more information and examples.

10.3.3 Writing an iterator for your container

Every container class in CGAL should strive to be a model for the STL concept of a container. As for all concepts, this means that certain types and functions are provided, as detailed, for example in [Aus98]. For the purposes of this discussion, the relevant types are:

iterator	type of iterator
const_iterator	iterator type for container with constant elements

and the relevant functions are:

iterator	*begin()*	beginning of container
const_iterator	*begin()*	beginning of container with constant elements
iterator	*end()*	past-the-end value for container
const_iterator	*end()*	past-the-end value for container with constant elements

Variations on the above names are possible when, for example, the container contains more than one thing that can be iterated over. See Section 2.1 for more details about the naming conventions for iterators and their access functions.

10.3.4 Writing a circulator for your container

When a container represents a circular data structure (*i.e.*, one without a defined beginning or end), one should provide circulators for the data elements in addition to (or, where appropriate, instead of) the iterators. This means that the following types should be defined:

circulator	type of circulator
const_circulator	circulator type for container with constant elements

as well as two access functions, one for each of the two types, with names that end in the suffix *_circulator* (Section 2.1).

Chapter 11

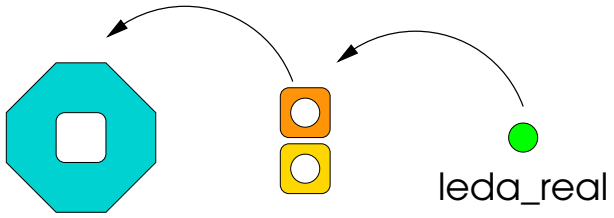
Robustness Issues

*Stefan Schirra (*stschirr@mpi-sb.mpg.de*)*

Design and correctness proofs of geometric algorithms usually assume exact arithmetic. Since imprecise calculations can cause wrong or, much worse, mutually contradictory decisions in the control flow of an algorithm, many implementations crash, or at best, compute garbage for some inputs. For some applications the fraction of bad inputs compared to all possible inputs is small, but for other applications this fraction is large.

CGAL has a layered design. The correctness of some components depends on the correctness of the components that are used. Correctness of a component means behaving according to its (mathematical) specification. Simply speaking, the source of the robustness problem is that the default hardware-supported arithmetic does not really fulfill the requirements of the algorithm, since it does not implement arithmetic on the real numbers.

Nevertheless, the generic implementation of the kernel primitives that are parameterized by the arithmetic (more precisely, by a number type) assumes that the arithmetic plugged in does behave as real arithmetic. The generic code does not and should not (otherwise it would slow down "exact" number types) deal with any potential imprecision. There are a number of (third-party provided) "exact" number types available for use with CGAL, where "exact" means that all decisions (comparison operations) are correct and that the representation of the numbers allows for refinement to an arbitrary precision, if needed. Most notably, *leda_real*s provide easy-to-use adaptive "exact" arithmetic for the basic operations and $\sqrt[k]{}$ operations.



11.1 The role of predicates and constructions

CGAL favors encapsulation of the basic arithmetic operations, the lowest level in geometric computing, into units on a higher level, namely, the level of geometric primitives, i.e., predicates and constructions. Here predicates are used in a generalized sense, i.e., not only primitives returning a Boolean value, but also primitives returning a value of some enumeration type, e.g. *CGAL::Sign*. So the value computed by a predicate does not involve any numerical data. Basic constructions construct new primitive geometric objects that may involve

newly computed numerical data, i.e. that is not part of the input to the constructions. An example of such a basic constructions is computing the midpoint of the straight line segment between two given points. A special kind of constructions is selections. For selections, all the data in the constructed objects was already part of the input. An example is computing the lexicographically smaller point for two given points.

CGAL provides generic implementations of geometric primitives. These assume "exact computation". This may or may not work, depending on the actual numerical input data. CGAL also provides[1] specialisation of the primitives that (are still fairly generic and) guarantee exact predicate results and much higher efficiency than exact number types like arbitrary precision integers or rationals. The efficiency relies on the use of speedy floating-point arithmetic in order to filter out reliable floating-point computations. Interval arithmetic is largely used in such filter steps.

[1] at present, for the Cartesian kernel(s) only. The homogeneous counterpart still needs revision.

Chapter 12

Portability Issues

*Michael Hoffmann (*hoffmann@inf.ethz.ch*)*
*Stefan Schirra (*stschirr@mpi-sb.mpg.de*)*
*Sylvain Pion (*Sylvain.Pion@sophia.inria.fr*)*

This chapter gives an overview of issues related to the configuration of CGAL that allow you to answer such questions as:

- Is LEDA/GMP there? (Section 12.1)
- What version of CGAL am I running? (Section **??**)
- Which compiler is this? (Section 12.4)
- Does the compiler support Koenig lookup? (Section 12.5.1)

Also addressed here are issues related to writing code for non-standard-compliant compilers. Compilers have made a lot of progress toward the C++-standard recently. But still they do not fully implement it. There are a few features you may assume; others you may not assume. Especially you may assume that the compiler

- supports namespaces
- supports member templates
- support for `std::iterator_traits`.

Still, there are many bugs (sometimes known as "features") left in the compilers. Have a look at the list of (non-obsolete) workarounds in Section 12.5.1 to get an idea of which "features" are still present.

12.1 Checking for LEDA or GMP support

In the makefiles included for the compilation of every CGAL program (*i.e.*, those to which the environment variable `CGAL_MAKEFILE` refers), we define command line switches that set the flags

```
CGAL_USE_LEDA, CGAL_USE_GMP
```

iff CGAL is configured with LEDA or GMP support, respectively.

12.2 Using Boost

CGAL code can rely on Boost libraries to some extent.

Boost was installed with CGAL Release 3.1, and is no longer installed, as it is already distributed with Linux and cygwin.

Since portability and backward compatibility are a concern in CGAL, we have decided that the list of Boost libraries usable in CGAL will be decided by the CGAL editorial board. The requirements are higher when it appears in the user visible interface than when Boost code is used only internally. Requirements are even lower for code that is not released such as the test-suite. Boost libraries already accepted in the C++ Standard Library Technical Report will be the first easy candidates (these are marked `[TR1]` in the list below).

Finally, the policy is that if a better alternative exists in Boost and is allowed, then CGAL code must use it instead of a CGAL version (which probably must be deprecated and phased out), trying not to break backward compatibility too much.

Here follows a list of Boost libraries allowed for use in CGAL (those with question marks are not decided yet) :

- Operators — Templates ease arithmetic classes and iterators
- ? Any — Safe, generic container for single values of different value types (what's the relation with `variant`)
- Concept check — Tools for generic programming
- ? Bind `[TR1]` and mem_fn — Generalized binders for function/object/pointers and member functions (overlaps with STL Extensions)
- ? Graph — Generic graph components and algorithms
- ? Interval — Extends the usual arithmetic functions to mathematical intervals (overlaps with CGAL::Interval_nt)
- Iterators — Iterator construction framework, adaptors, concepts, and more
- MPL — Template metaprogramming framework of compile-time algorithms, sequences and metafunction classes
- Optional — Discriminated-union wrapper for optional values
- Property map — Concepts defining interfaces which map key objects to value objects
- ? Random `[TR1]` — A complete system for random number generation (overlaps with support library)
- ? Rational — A rational number class (overlaps with CGAL::Quotient)
- ? Ref `[TR1]` — A utility library for passing references to generic functions
- ? Smart Pointers `[TR1]` — Five smart pointer class templates
- ? Static assertions — Static assertions (compile time assertions)
- Tuple `[TR1]` — Ease definition of functions returning multiple values, and more
- ? Type traits `[TR1]` — Templates for fundamental properties of types
- ? Variant — Safe, generic, stack-based discriminated union container (how does it relate to `any` ?)

12.3 Using the version-number and configuration macros and flags

Here is a short example on how these macros can be used. Assume you have some piece of code that depends on whether you have LEDA-4.0 or later.

```
#ifdef CGAL_USE_LEDA
#include <LEDA/basic.h>
#endif

#if defined(CGAL_USE_LEDA) && __LEDA__ >= 400
... put your code for LEDA 4.0 or later ...
#else
... put your code for the other case ...
#endif
```

12.4 Identifying compilers and architectures

Every compiler defines some macros that allow you to identify it; see the following table.

Borland 5.4	`__BORLANDC__`	0x540
Borland 5.5	`__BORLANDC__`	0x550
Borland 5.5.1	`__BORLANDC__`	0x551
GNU 3.2.1	`__GNUC__`	3
GNU 3.2.1	`__GNUC_MINOR__`	2
GNU 3.2.1	`__GNUC_PATCHLEVEL__`	1
Microsoft VC7.1	`_MSC_VER`	1310
Microsoft VC8.0	`_MSC_VER`	1400
Intel 7.0	`__INTEL_COMPILER`	???
SGI 7.3	`_COMPILER_VERSION`	730
SUN 5.0	`__SUNPRO_CC`	0x500
SUN 5.3	`__SUNPRO_CC`	0x530

There are also flags to identify the architecture.

SGI	`__sgi`
SUN	`__sun`
Linux	`__linux`

12.5 Known problems and workarounds

For (good) reasons that will not be discussed here, it was decided to use C++ for the development of CGAL. An international standard for C++ has been sanctioned in 1998 [C++98] and the level of compliance varies widely between different compilers, let alone bugs.

12.5.1 Workaround flags

In order to provide a uniform development environment for CGAL that looks more standard compliant than what the compilers provide, a number of workaround flags and macros have been created. Some of the workaround macros are set in <CGAL/config.h> using the macros listed in Section 12.4 to identify the compiler. But most of them are set in the platform-specific configuration files

<CGAL/config/*os-compiler*/CGAL/compiler_config.h>

where *os-compiler* refers to a string describing your operating system and compiler that is defined as follows.

<arch>_<os>-<os-version>_<comp>-<comp-version>

<**arch**> is the system architecture as defined by "uname -p" or "uname -m",

<**os**> is the operating system as defined by "uname -s",

<**os-version**> is the operating system version as defined by "uname -r",

<**comp**> is the basename of the compiler executable (if it contains spaces, these are replaced by "-"), and

<**comp-version**> is the compiler's version number (which unfortunately can not be derived in a uniform manner, since it is quite compiler specific).

Examples are mips_IRIX64-6.5_CC-n32-7.30 or sparc_SunOS-5.6_g++-2.95. For more information, see the CGAL installation guide.

This platform-specific configuration file is created during installation by the script install_cgal. The flags listed below are set according to the results of test programs that are compiled and run. These test programs reside in the directory

$(CGAL_ROOT)/config/testfiles

where $(CGAL_ROOT) represents the installation directory for the library. The names of all testfiles, which correspond to the names of the flags, start with "CGAL_CFG_" followed by

- *either* a description of a bug ending with "_BUG"
- *or* a description of a feature starting with "NO_".

For any of these files a corresponding flag is set in the platform-specific configuration file, iff either compilation or execution fails. The reasoning behind this sort of negative scheme is that on standard-compliant platforms there should be no flags at all.

Currently (CGAL-3.1-I-33), we have the following configuration test files (and flags). The short descriptions that are given in the files are included here. In some cases, it is probably necessary to have a look at the actual files to understand what the flag is for. This list is just to give an overview. See the section on troubleshooting in the installation guide for more explanation of some of these problems and known workarounds. Be sure to have a look at Installation/config/testfiles/ to have an uptodate version of this list.

CGAL_CFG_CCTYPE_MACRO_BUG

This flag is set if a compiler defines the standard C library functions in `cctype` (`isdigit` etc.) as macros. According to the standard they have to be functions.

CGAL_CFG_LONGNAME_BUG

This flag is set if a compiler (or assembler or linker) has problems with long symbol names.

CGAL_CFG_MATCHING_BUG_3

This flag is set, if the compiler does not match function arguments of pointer type correctly, when the return type depends on the parameter's type (*e.g.*, sun C++ 5.3).

CGAL_CFG_MATCHING_BUG_4

This flag is set, if a compiler cannot distinguish the signature of overloaded function templates, which have arguments whose type depends on the template parameter. This bug appears for example on Sunpro 5.3 and 5.4.

CGAL_CFG_NET2003_MATCHING_BUG

This flag is set, if the compiler does not match a member definition to an existing declaration. This bug shows up on VC 7.1 Beta (*cl1310*).

CGAL_CFG_NO_BIG_ENDIAN

The byte order of a machine architecture distinguishes into big-endian and little-endian machines. This flag is set if it is a little-endian machine.

CGAL_CFG_NO_KOENIG_LOOKUP

This flag is set if the compiler does not support the operator Koenig lookup. That is, it does not search in the namespace of the arguments for the function.

CGAL_CFG_NO_LIMITS

This flag is set if a compiler does not know the limits.

CGAL_CFG_NO_LOCALE

This flag is set if a compiler does not know the locale classic.

CGAL_CFG_NO_LONG_LONG

The *long long* built-in integral type is not part of the ISO C++ standard, but many compilers support it nevertheless, since it is part of the ISO C standard. This flag is set if it is supported.

CGAL_CFG_NO_STDC_NAMESPACE

This flag is set if a compiler does not put the parts of the standard library inherited from the standard C library in namespace *std* (only tests for the symbols used in CGAL).

CGAL_CFG_NO_TMPL_IN_TMPL_DEPENDING_FUNCTION_PARAM

This flag is set of a compiler does not support member functions that have parameter types that are dependent on the template parameter list of the class and are implemented outside of the class body (*e.g.*, g++ 2.95.2).

CGAL_CFG_NO_TMPL_IN_TMPL_PARAM

Nested templates in template parameter, such as "`template < template <class T> class A>`" are not supported by any compiler. This flag is set if they are not supported.

CGAL_CFG_OUTOFLINE_TEMPLATE_MEMBER_DEFINITION_BUG

This flag is set, if a compiler does not support the definition of member templates out of line, *i.e.*, outside class scope. The solution is to put the definition inside the class. This is a feature of SunPRO 5.5.

12.5.2 Macros connected to workarounds/compilers

Some macros are defined according to certain workaround flags. This is done to avoid some `#ifdef`s in our actual code.

`CGAL_CLIB_STD` set to `std`, if `CGAL_CFG_NO_STDC_NAMESPACE` is not set and empty, otherwise.

`CGAL_LITTLE_ENDIAN` set, iff `CGAL_CFG_NO_BIG_ENDIAN` is set.

`CGAL_BIG_ENDIAN` set, iff `CGAL_CFG_NO_BIG_ENDIAN` is not set.

12.5.3 Various other problems and solutions

Templated member functions For SunPRO C++ member function templates with dependent return type must be defined in the body of the class.

Function parameter matching The function parameter matching capacities of Visual C++are rather limited. Failures occur when your function *bar* is like

```
bar(std::some_iterator<std::some_container<T>>....) ...
...
bar(std::some_iterator<std::some_other_container<T>>....) ...
```

VC++ fails to distinguish that these parameters have different types. A workaround is to add some dummy parameters that are defaulted to certain values, and this affects only the places where the functions are defined, not the places where they are called. This may not be true anymore for recent VC++ versions.

typedefs of derived classes Microsoft VC++ does not like the following sorts of typedefs that are standard

```
class A : public B::C {
        typedef B::C C;
};
```

It says that the typedef is ”redefinition”. So such typedefs should be enclosed by

```
#ifndef _MSC_VER

#endif
```

This may not be true anymore for recent VC++ versions.

parse error in constructions The following program will produce a parse error with g++ 3.1.

```
#include <CGAL/Segment_circle_2.h>

typedef CGAL::Segment_circle_2<double> Curve;
typedef Curve::Segment                 Segment;
typedef Curve::Point                   Point;

int main()
{
  Segment s1(Point(0,0), Point(1,1));
  Curve curve(Segment(Point(0,0), Point(1,1))); // parse error

  // ...
  return 0;
}
```

This is a well-known bug in the Gnu compiler (see http://gcc.gnu.org/bugs.html#parsing). The workaround is to split :

```
Curve curve(Segment(Point(0,0), Point(1,1)));
```

into, *e*.g., :

```
Segment s (Point(0,0), Point(1,1));
Curve rude_curve(s);
```

Chapter 13

Debugging Tips

*Oren Nechushtan (*theoren@math.tau.ac.il *)*

Efficient debugging techniques can become an asset when writing geometric libraries such as CGAL. This chapter discusses debugging-related issues, like how to use the demo as a powerful debugger (Section 13.1), why and how to check your geometric predicates (Section 13.2), and what to do in order to evaluate handles and iterators during the debugging phase (Section 13.3).

13.1 Graphical debugging

CGAL packages usually provide a graphical demo that demostrates the functionality in the package. Many times this demo is simply a fancier version of a program that was used in the early stages of development as a (graphical) debugging tool. In many cases, the output of a geometric algorithm is much easier to interpret in graphical form than numeric form. Thus you should use the powerful graphical output capabilities of CGAL (see the Support Library documentation) to develop

- programs that can be used for debugging the internal workings of your package (*i.e.*, things a user may not have access to)
- interesting and informative demos that highlight the features and, at the same time, the absence or presence of bugs in your package. Other demo/debugging programs can be found in the `demo` directory of every internal release and CGAL installation.

13.2 Cross-checkers

A cross-checker is a powerful means to allow for efficient maintainence of your code. A cross-checker for a given concept is a model of that concept that is constructed from another model or models (one of which is the one you wish to check). In order to implement the functionality required by the concept, the cross-checker will use functions from the models upon which it is built and perform tests for validity, etc. on them. If the tests succeed, the cross-checker returns the expected result. Otherwise, the cross-checker can generate an assertion violation or a warning, depending on the severity of the offense.

For example, if you have a version of an algorithm, traits class, or kernel that you know works, you can easily use this as an oracle for another version of the algorithm, traits class, or kernel that you wish to test. This is

easily done because the code in CGAL is highly templatized. The cross-checker would simply plug in the two different versions of, say, your traits class, as the relevant template parameters for two different instantiations of a class, say, and then compare the results from using the two different instantiations.

An example: Traits class binary cross-checker

As a more concrete example, assume that you have a traits class concept that requires a nested type *X_curve* and a function

bool *curve_is_vertical(X_curve cv)*

A binary cross-checker for this concept might look like

```
template <class Traits1,class Traits2,class Adapter>
class Binary_traits_checker{

Traits1 tr1;
Traits1 tr2;
Adapter P;

public:

typedef typename Traits1::X_curve X_curve;

Traits_binary_checker(Traits1 tr1_,Traits2 tr2_,Adapter P_) :
tr1(tr1_),tr2(tr2_),P(P_){};

bool curve_is_vertical(const X_curve & cv) const;

}
```

and possibly be implemented as

```
bool curve_is_vertical(const X_curve & cv) const
{
    CGAL_assertion(tr1.curve_is_vertical(cv)==tr2.curve_is_vertical(P(cv)));
    return tr1.curve_is_vertical(cv);
}
```

Notice that the class *Binary_triats_checker* has template parameters named *Traits1* and *Traits2*, and a third parameter named *Adapter*. One of the traits classes is the one to be tested and the other is (presumably) a traits class that always gives the right answer. The *Adapter* is needed since the *X_curve* types for *Traits1* and *Traits2* might be different. This cross-checker does nothing other then asserting that the two traits classes return the same values by calling the the counterparts in the member traits classes (*tr1*,*tr2*) and comparing the results.

13.3 Examining the values of variables

When using an interactive debugger, one often wishes to see the value of a variable, such as the y-value of a segment's source point. Thus one would naturally issue a command such as

```
print segment.source().y()
```

This most often produces disappointingly unrevealing results, *e.g.*, an error message saying the value cannot be evaluated because functions may be inlined.

We recommend the following approaches to work around (or avoid) this and similar problems:

- Use the *Simple_cartesian* kernel (Chapter 3), which does not do reference counting and uses no handles so data member values can be inspected directly.
- Print the values by following the pointers in the handles used to represent objects. For example, for the segment above, the statement

  ```
  print s.ptr->start->ptr->e1
  ```

  is likely to work. This technique can also work for non-kernel handles, such as *Halfedge_handle* and *Vertex_handle*. One must know, of course, the right names for the data members, but this you can find out by printing the things that pointers point to. For example,

  ```
  print *s.ptr
  ```

  In the case of the planar map package, these handles are actually polyhedron iterators. If h is a halfedge of a planar map and you want to know the curve associated with it, then if

  ```
  print h->curve()
  ```

  fails, try using

  ```
  print h.nt.node->cv
  ```

  instead.

  For a vertex v of a planar map, if

  ```
  print v->point()
  ```

  fails, use

  ```
  print v.nt.node->p
  ```

  instead.

Note: You can also use watches to continously examine such values during execution.

Chapter 14

Editorial Committee

The editorial committee is in charge of approving the inclusion of new packages in the library. This means that they assure that new contributions

- are in keeping with the philosophy of CGAL (Chapter 1);
- are generic and fit seamlessly with other parts of the library;
- satisfy the coding conventions of CGAL (Chapter 2);
- carefully and efficiently treat robustness issues (Chapter 11);
- are designed in a flexible, extensible, and easy-to-use fashion;
- and are designed to be technically feasible for the platforms supported by CGAL.

Software specifications and implementations should be submitted to the editorial committee for approval. This can be done by sending mail to the committee (`editor@cgal.org`) indicating where the (PostScript) documentation and code can be found. After some reasonable amount of time, you should receive feedback from the committee about the specification and what, if anything, needs to be changed. The usual procedure is that someone from the committee is assigned to be (or volunteers to be) the primary reviewer and sends comments on the submitted package to the committee and to the authors of the package. Discussion then proceeds among the committee members and the authors until a consensus is reached about how the package should be modified before being accepted. When the package has been modified, the authors should again notify the editorial committee to let them know what has changed so a decision about acceptance of the package can be taken.

One should write a specification for a new package and submit it to the editorial committee for approval before submitting the package for inclusion in the internal releases (and ideally before implementation of the package). This assures that time is not wasted in fixing code that may later be changed due to the recommendations of the committee. However, since it can take some time for the committee to process submissions, packages that are to become part of the library (as opposed to being listed as CGAL Extension Packages) can be submitted before approval. Inclusion in an internal release does not ensure inclusion in a public release. Only after approval by the committee will packages be included in new public releases and then only if they pass the test suite, of course.

The current members of the editorial committee are:

Andreas Fabri
Efi Fogel
Lutz Kettner
Remco Veltkamp
Sylvain Pion
Ron Wein

Bernd Gärtner
Michael Hoffmann
Monique Teillaud
Mariette Yvinec
Menelaos Karavelas

Chapter 15

Recommended Reading

The following books and papers are recommended as references:

[Aus98] – Mathew Austern's introduction to the STL using the concept/model style of presentation. Austern wrote the WWW STL documenataion at SGI.

[Str97] – Bjarne Stroustrup's introduction to C++ and the STL for those who already know some C++. Stroustrup is the designer and original implementor of C++.

[LL98] – Stanley Lippman and Josee Lajoie's C++ primer.

[Mey97] – Scott Meyers's book on ways to improve your C++ programs. Items 21 and 29 discuss the concept of const-correctness.

[FGK+00] – The CGAL design paper.

[HHK+01] – The new CGAL kernel design paper.

Bibliography

[Aus98] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1998.

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

[FGK+00] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Softw. – Pract. Exp.*, 30(11):1167–1202, 2000.

[HHK+01] Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Michael Seel. An adaptable and extensible geometry kernel. In *Proc. Workshop on Algorithm Engineering*, volume 2141 of *Lecture Notes Comput. Sci.*, pages 79–90. Springer-Verlag, 2001.

[LL98] Stanley B. Lippman and Josee Lajoie. *C++ Primer*. Addison-Wesley, 3rd edition, 1998.

[Mey97] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 2nd edition, 1997.

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

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

[Vle97] J. Vleugels. *On Fatness and Fitness — Realistic Input Models for Geometric Algorithms*. Ph.D. thesis, Dept. Comput. Sci., Univ. Utrecht, Utrecht, The Netherlands, 1997.

Index

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