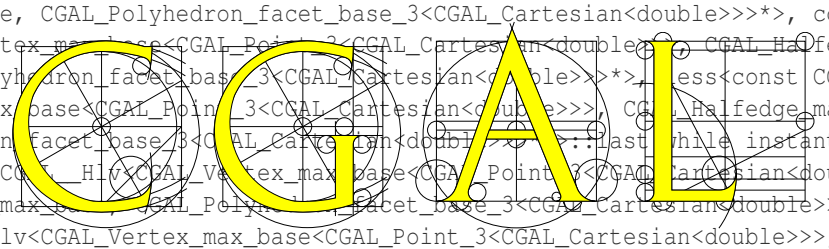
Warning: last hides rb_tree<const CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_C
artesian<double>>>, CGAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<CGAL_Cartes
ian<double>>>*, pair<const CGAL__Hlv<CGAL_Vertex_max_base<CGAL_Point_3<CGAL_Cartesia
n<double>>>, CGAL_Halfedge_max_base, CGAL_Polyhedron_facet_base_3<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 3.2-1

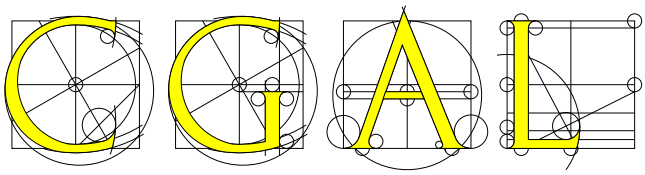
13 July 2006

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 Manual organization

This manual is meant to be a resource for developers who wish to contribute to the CGAL library either by designing new packages or maintaining or enhancing existing ones. The manual is organized roughly in the order in which a developer will need the information in order to produce a package for the library. We begin in this chapter with a description of the design goals of CGAL and the overall design, which should be kept in mind during all stages of development. The remaining chapters describe in more concrete terms the requirements and recommendations for documentation, code writing, and testing that are derived from these goals. We also describe a number of tools that have been created to help in the development process and give pointers to other sources of information.

A description of how the specification for a package should be documented is provided along with information about the tools available to help produce the documentation in Chapter 2. Chapter 3 discusses the SVN server on which all CGAL source code is kept. Chapter 4 describes the directory structure required for a package and Chapter 5 describes a set of tools that may be used to create or modify various files required within this directory structure. Chapters 6 through 15 discuss issues related to the writing of code that is in keeping with the goals of CGAL. Chapter 16 describes issues related to the configuration of CGAL and discusses portability

issues for various platforms. Chapter 17 describes the requirements and behaviour of the test suite for a package and Chapter 18 consists of various hints for debugging. Guidelines for the development of example and demo programs are given in Chapter 19. Information about how to submit a package's specification to the editorial board for approval and the implementation (and documentation) for inclusion in internal releases is presented in Chapter 20. Chapter 21 gives information about the creation of the internal, public, and bug-fix releases. Chapter 22 provides information about the CGAL web site and what developers should do to assure that it is kept up to date. Chapters 23 and Chapter 24 provide information about mailing lists and other information resources developers might find useful.

1.2 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 15 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 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.

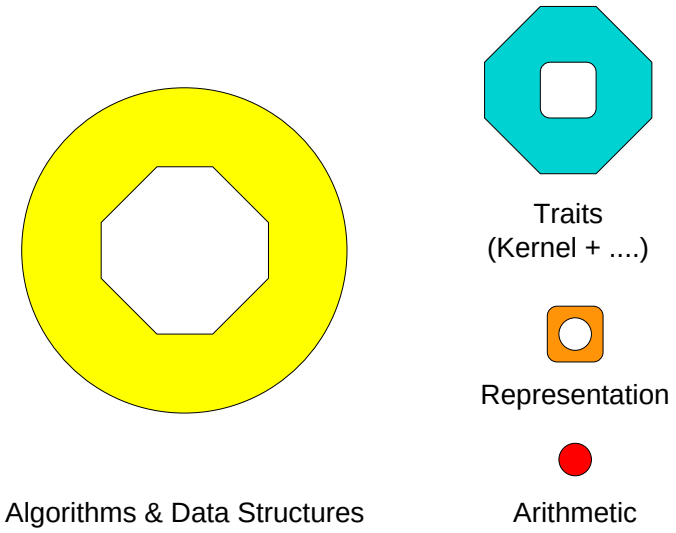


Figure 1.1: The generic design of CGAL.

1.3 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 order to avoid long parameter lists, the parameter types are collected into a single class, called the traits class in CGAL (Chapter 8.) 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 7 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.

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

Chapter 2

Specification Documentation

*Susan Hert (*hert@mpi-sb.mpg.de*)*
*Lutz Kettner (*kettner@mpi-sb.mpg.de*)*

> There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.
>
> *C. A. R. Hoar*

CGAL is physically structured into packages, which is reflected in the SVN package structure, see Chapter 3. Generally, each package in the library will contribute some chapters to the manuals, some might only contribute a section to some larger chapter.

The following is the current plan and in the process of realization

CGAL is logically presented to the users in modules. A module consists of one or several packages. A package can be part of several modules. An example for a module would be the CGAL Kernel, and also the whole library.

Generally, a manual is created for a particular module. Such a manual consists of a wrapper file that will include the manual parts from the individual packages that comprise the module. However, manuals can be created for individual packages, mostly for editing and reviewing, and also only reasonably if the package contributes a whole chapter.

The division into different manuals is currently under revision. We merge several manuals into one manual with different parts.

The official CGAL documentation is currently divided into six separate manuals:

- Installation Guide
- Kernel
- Basic Library
- Support Library
- Use of STL and STL Extensions in CGAL

Not surprisingly, the Installation Guide describes how to install CGAL and the Use of STL manual describes features of STL that are used and extended in CGAL.

Documentation of new packages and features will likely be included in the Kernel, Basic Library, and Support Library documentation. See Section 1.3 for a description of what these three parts of the library contain. Each of these manuals is further subdivided into two parts: a Users' Manual and a Reference Manual. Sections 2.3 and 2.4 below describe the contents of these two parts.

The manuals are produced from LaTeX input files in PostScript, PDF, and HTML format. We provide a set of style files with numerous commands that are used to format the manuals in a (more or less) uniform fashion. The conversion from LaTeX input to HTML output is done with our own `latex_to_html` converter program that in particular understands our manual style files. Section 2.1 describes how to organize your documentation files and what is required for submitting documentations. Section 2.2 provides the basic information about these tools and style files that you will need to produce the documentation for your package. Sections 2.4 and 2.3 describe the contents and the macros provided in our style files for the Users' Manual and the Reference Manual, respectively. Sections 2.6 and 2.7 give information about including figures in your documentation and inserting indexing commands, respectively. Section 2.8 describes the manual test suite run for each internal release of the library. Section 2.9 discusses common problems and solutions. Section 2.10 concludes with the requirements and recommendations for the specification documentation.

2.1 File and directory organization

We start with the organization of the documentation of a package. After that, we describe the organization of a module documentation.

Generally, each package in the library will have its own chapter in both the reference manual and the users' manual, some might only contribute a section to some larger chapter. So, in principle, the documentation of a package starts with the `\chapter` command. Note, that each chapter, in particular the `\chapter` command, needs to be in a different file. A wrapper file may then include all chapters.

The `Polyhedron` package can serve as an example for the manual organization.

It is required that the documentation submitted with a package be organized according to a specific directory structure, which is described more completely in Section 4.2. In summary, all `.tex` source files and corresponding figures are kept in a `doc_tex` subdirectory of the package. The following rules describe the details:

- The programs creating the manuals, i.e., `latex`, `pdflatex` etc., are called in the `doc_tex` directory. Since `latex`, and consequently `latex_to_html` too, do not maintain a current working directory, all filenames, such as in include statements, figures, etc., must be given relative to this `doc_tex` directory.
- The input search paths of `latex` and `latex_to_html` will be enriched with the paths for example and demo programs, such that they can be referenced starting from their package subdirectory name, e.g., `\ccIncludeExampleCode{Polyhedron/simple.C}`.
- The `.tex` files for the users' manual should be placed in a directory `doc_tex/`<*Package*>`/`. Among them, the file `main.tex` must exist that inputs all other users' manual source files using the `\input` command (NOT the `\include` command)..
- The `.tex` files for the reference manual should be placed in the directory `doc_tex/`<*Package*>`_ref/`. Also among them, the file `main.tex` must exist that inputs all other reference manual source files. For the purposes of the LaTeX-to-HTML converter, each item documented in the reference manual should be put in its own file. (The script `cc_ref_wizard` can help in creating these files.) The script `cc_make_ref_pages`

will create a `main.tex` file given the name of the directory that contains the separate files for each reference manual item. The items will be input in alphabetical order, with a file `intro.tex`, if it exists, included first. The `intro.tex` file should start with the `\chapter` command, contain a short introductory section, give an overview of the reference pages contained in this chapter, and mention optionally with a non-numbered section labeled **Assertions** the string used in the package-specific assertion macros (Section 9.3).

This is all that is needed for a package manual and the `cgal_manual` program mentioned in Section 2.2.1 will create the necessary wrapper file automatically on the fly to create PostScript, PDF, or HTML manuals for an individual package.

For modules, an additional wrapper file is needed. The following rules describe the details, and Section 2.2.5 lists an example wrapper file:

- To be detected automatically by the tools in Section 2.2 the wrapper filename must match the pattern `*_manual.tex`.
- Wrapper files must be placed in the `doc_tex` subdirectory. Wrapper files that do not have a good package where they would belong to can be kept in the `Manual` SVN package, for example, `cgal_manual.tex` containing the manual for whole CGAL.

2.2 The manual tools

As a package author, it suffices to learn from this section that you need to install two SVN packages, how to use the `cgal_manual` program, and what style files are available in the automatically generated wrapper for packages. Occasionally, you might need to add a new entry in the globally maintained BibTEX file.

For module authors, we list an example wrapper file and document the commands available in particular for wrapper files and for handling dependencies between packages. The example wrapper file might also be helpful for package authors to understand the context in which the package chapter is formatted.

All programs, style files, and other supporting files for the creation of the manuals are contained in two SVN packages:

Manual_tools contains the style files for C++ manual page formatting and the LATEX to HTML converter program. These style files and the converter can be used independently from CGAL. Stable snapshots are also released at `http://www.cgal.org/Members/Manual_tools/`.

Manual contains the CGAL specific LATEX wrapper files with the customization for the CGAL manuals and a top-level driver script to run LATEX and the `latex_to_html` converter consistently in the manual file structure of CGAL manuals. It contains also the additional auxiliary files for creating the CGAL manuals that are not particular to individual packages. These are figures like the CGAL logo or the bibliography file `doc_tex/Manual/cgal_manual.bib` for BibTEX.

In a nutshell, it is mandatory to install the `Manual_tools` package properly on your system. From the `Manual` package one needs only the program `Manual/developer_scripts/cgal_manual` in the execution path and the package itself should be placed side-by-side with your own package(s) that you develop (the `cgal_manual` script finds then automatically the other files in the `Manual` package). Side-by-side means that the package directory `Manual` is in the same directory as the package directory `Package` that you develop.

2.2.1 Program `cgal_manual` in the `Manual` package

The `cgal_manual` program is a `bash` script residing in the `developer_scripts/` directory. It is called in the `doc_tex/` directory of a CGAL package or of an internal release.

This script is the driver program for creating CGAL manuals. It makes use of LaTeX, PDFLaTeX, BibTeX, `makeindex`, `latex_to_html`, and other tools to create the PostScript, PDF, and HTML manuals for individual *SVN Packages* as well as custom *Modules* and its *Sub-Modules*. It encodes the conventions of how CGAL manuals are organized and how the general purpose tools need to be called to create the manuals. We specify these conventions here in the developers manual. Note that a brief documentation is also kept in the script itself, these documentations need to be kept synchronized!

SVN Package

Development unit in CGAL hosted on our SVN server. A package has a fixed directory structure. Let's assume the package is called `Geom`, then the documentation must reside in a directory `doc_tex/` within the package `Geom`. Here, individual subdirectories, typically `Geom/` and `Geom_ref/`, contain the users' and the reference manual respectively. The individual subdirectories contain a `main.tex` file that contains the manual chapter, possibly several and possibly using several other input files, but all included with relative paths from the `doc_tex/` directory, where the tools and this script will run.

In general, all individual subdirectories that contain a `main.tex` file are processed. If they exist in pairs, we assume that `doc_tex/*/main.tex` and `doc_tex/*_ref/main.tex` are corresponding users' and reference manual entries in the table of contents.

The `main.tex` files are not stand-alone LaTeX files. They are chapters, i.e., do not contain `\begin{document}` commands etc. This script provides the necessary LaTeX wrapper file.

Modules

Presentation unit of modularity towards the user, assembled from SVN Packages. For a module of name `Algo` it is assumed that in the `doc_tex/` directory the necessary LaTeX driver file `Algo.tex` exists. This driver is a complete LaTeX file including the `\begin{document}` command and such. It includes the various chapters from the packages.

Sub-Modules

Another presentation unit on top of modules. A sub-module only contains a subset of the packages in a module. The numbering of pages, chapters, equations et cetera is inherited from the module, while the table of context and the index is shrink-wrapped to the sub-module.

Generally, for some module `m.tex`, a sub-module `sub` is defined in a file `m__sub.only.tex` that contains LaTeX commands to be executed before the document starts. The most obvious command therein is probably `\includeonly`. Note: packages split into user manual and reference manual need two entries in the
`includeonly` command. Sub-modules have to adhere to the filenaming scheme from above, in order to be found by the `cgal_manual` script. There can be multiple sub-modules for each module. Sub-modules of sub-modules are not supported.

For example, consider a module stored in `cgal_manual.tex`. A sub-module `Geom` should be built, restricting the module to user and reference manual of the package `Geom`. It could be stored in a file called `cgal_manual__Geom.only.tex`, only containing the LaTeX command `\includeonly{Geom/main,Geom_ref/main}`.

When building a module with the `cgal_manual` script, the default is to produce no sub-modules of it. The desired sub-modules can be specified by `-sub-modules=mod1,mod2..` where sub-modules are given as a comma-separated list. Only the sub-module name has to be given, not the filename under which it is stored. A special sub-module is `all`, which matches all sub-modules present in the current directory.

This script can create individual package documentation given the name of the individual subdirectories in doc_tex/. It creates module documentations given the name of the module LaTeX driver file.

The default is to create package manuals for all */main.tex files, where */main.tex and corresponding *_ref/main.tex are kept in one manual, and to create manuals for all *_[Mm]anual.tex, which are representing modules.

This script can be used in three different environments of CGAL sources. It decides automatically in which situation it is and adapts the necessary search paths for style files and bibliographies.

1. CGAL **Internal Release:** The style and bibliography files are relative from the doc_tex directory in the subdirectory Manual/.
2. **Individual** CGAL **Packages +** CGAL **Package Manual:** The style and bibliography files are relative from the doc_tex directory reachable with the path ../../Manual/doc_tex/Manual/.
3. **All other environments:** The style and bibliography files need to be installed properly from the SVN package Manual, such that the tools can find them, for example, through the search paths defined in the environment variables TEXINPUTS, BIBINPUTS, and LATEX_CONV_INPUTS. The program checks if the files are actually in the search paths and issues an error message otherwise.

This script properly adds entries in TEXINPUTS, BIBINPUTS, and LATEX_CONV_INPUTS so that bib and style files are found. It also adds ../examples:../demo to these paths so that example source codes are properly found.

The result of run of cgal_manual are the specified manuals in the corresponding ../doc_ps, ../doc_pdf, and ../doc_html directories. A logfile with suffix .cgallog is created with a logfile summary from the individual logfiles .log, .blg, .ilg, .pdflg, and .hlg. The logfiles are kept in case of warnings and error messages.

The screen output reports the progress and result status of each module (abbreviated as Mod) and package (abbreviated as Pck) for each manual type, i.e., PostScript, PDF, and HTML.

This script also serves for the test-suite. It can be configured to create the test-suite result tables also at your local configuration (and without sending the email notification to cgal-develop ;-). The default settings are encoded and documented at the beginning of the script in a clearly marked region. For individual customizations, the script reads the individual resource file ${HOME}/.cgalmanualrc after the default initialization.

When the -testsuite option is used the script will copy all the manuals and logfiles to ${TestSuiteResultPath}/CGAL-${CgalVersion}/ and creates an HTML summary page index.html in that subdirectory. The latest result is also always accessible at ${TestSuiteResultPath}/LAST/index.html. Furthermore, the script will cleanup old results. For the most recent number ${TestSuiteFullHistory} of test suites the full results including the manuals are kept. Older test suites will have their manuals deleted to save space. In total only ${TestSuiteHistory} many test suites are kept. The history of test suites is managed in a shift register like fashion using files of defined names History.<i> that contain the name of the *i*-th test suite subdirectory. The 1st is the most recent test suite and corresponds to the LAST directory. If the test suite is repeated for the same internal release number, the new results will overwrite the old results.

```
Usage: cgal_manual [<options>] [<module-files...>] [<package-dirs...>]
Options:
    -ps                 PostScript manuals.
    -pdf                PDF manuals.
    -html               HTML manuals (incl. a LaTeX run).
    -wrapper            creates the LaTeX wrapper files only.
    -testsuite          runs testsuite, installs results and sends email. Use
                        only after reading the config section of this script.
```

```
-h                  help.
-V                  version.
-v                  verbose: repeats logfiles on stderr.
-k                  keep logfiles (default: delete after a clean run).
-n                  no logfile: delete logfiles always.
-s                  create a summary logfile .sum .
-quiet              no progress messages.
-realquiet          suppresses also error messages.

-cmdlog             create a logfile "cmd_log" containing all commands
                    that were issued during execution

-sub-modules=mod1,mod2...
                    build specified sub-modules, given as a comma-
                    separated list.
```

2.2.2 SVN package `Manual_tools`

The SVN package `Manual_tools` and the release tar ball from `http://www.cgal.org/Members/Manual_tools/` keep all files in the subdirectory called `Manual_tools`, which itself has the following subdirectories:

`doc/` – directory containing the documentation for the style files and the converter program

`example/` – directory containing example documentation

`format/` – directory containing the LaTeX style files used for formatting the manual

`scripts/` – directory containing various support scripts

`src/` – directory containing the style files and program source code for the LaTeX to HTML converter.

There are also `README` and `CHANGES` files provided for information. It is mandatory to install this package properly on your system following the description in the `INSTALLATION` file. Linux and Sun Solaris are supported platforms.

2.2.3 Style files in the `Manual_tools` package

The manuals are written with macros provided in the following style files provided in the `Manual_tools` package. When properly installed, these style files are in the search paths of LaTeX and `latex_to_html`. The most prominent macros are introduced in this chapter. For the full documentation please see to the files with corresponding names in the `Manual_tools/doc` directory.

- `cc_manual.sty` provides the commands to structure C++ reference manuals and to format C++ declarations nicely.
- `cc_manual_index.sty` provides the commands for creating an index.
- `latex_converter.sty` provides commands useful for the conversion to HTML.

2.2.4 Programs in the `Manual_tools` package

The `latex_to_html` program is the main program provided by the `Manual_tools` package. It is in fact a `bash`-shell script and uses itself a number of other programs, which are not detailed here. In addition, a couple of programs come with this package that can be of independent interest.

- `latex_to_html` – script that does the conversion from LaTeX to HTML. See `Manual_tools/doc/latex_to_html.ps.gz`.
- `cc_extract_images` – program used by `latex_to_html` that extracts inline images (`.gif|.png|.jpg` files) from an HTML stream given on *stdin*.
- `cc_extract_include` – script to extract `\ccInclude` statements from a specification file and write these converted to C preprocessor includes to standard output.
- `cc_make_ref_pages` – script that creates a `main.tex` file in a directory, the name of which is supplied as an argument. This `main.tex` will be the driver file for the reference manual chapter for the directory named in the argument. See `Manual_tools/doc/cc_manual.ps.gz` and the script itself for further explanation.
- `cc_ref_wizard` – script that creates a reference page skeleton given the category of the item and the item name. See `Manual_tools/doc/cc_manual.ps.gz` and the script itself for further explanation.
- `add_part_num` – script used to prepend a part number to the page numbers in a `.idx` file produced by the indexing commands in a LaTeX document. See `Manual_tools/doc/cc_manual_index.ps.gz`.
- `index_fix` – script used to postprocess an `.ind` file produced by `makeindex` to make the index fit the specifications detailed in `Manual_tools/doc/cc_manual_index.ps.gz`.

2.2.5 Example wrapper file for a module manual

The `cgal_manual` program can be used with the option `-wrapper` to create the default wrapper file for a particular package. This can be used as a good (more minimal) starting point for a new module. Larger examples can be studied looking at existing modules, such as the `cgal_manual.tex` in the `Manual/doc_tex/` directory that contains the whole CGAL manual. The following is a short wrapper file that came out of the call `cgal_manual -wrapper Polyhedron Polyhedron_ref`. Note that improvements in these tools might change the actual outcome. For the custom `\cgal...` commands see the next Section 2.2.6 for their definition.

```
% +------------------------------------------------------------------------+
% | Polyhedron.tex
% | LaTeX Wrapper File for CGAL User and Reference Manual
% | Automatically generated by 'cgal_manual' for CGAL Packages
% +------------------------------------------------------------------------+

\documentclass{book}
\usepackage{cgal_manual}

\makeindex

% Make the table of contents use two columns
\lcHtml{\lcTwoColumnToc}
```

```
\begin{document}

\cgaltitlepage{Polyhedron Package}

\cgalchapters{
    %\entryleftright{\part{User Manual}}{\part{Reference Manual}}
    \lcTex{\entryright{\listofrefpages}}
    \packageleftright{Polyhedron}{Polyhedron_ref}
}
\bibliographystyle{alpha}
\bibliography{cgal_manual,geom}

\printindex

\end{document}
```

Uncommenting the line with the `\part` commands would add parts to the manual as they are used for large modules, but they would be wasteful for individual package manuals.

2.2.6 Style file `cgal_manual.sty` in the `Manual` package

The style of the CGAL manuals is encoded in the style file

```
Manual/cgal_manual.sty
```

which is located at `Manual/doc_tex/Manual/cgal_manual.sty`. It determines the page size, pulls in several other style files, and defines CGAL specific macros. The style files currently pulled in are:

`cc_manual`	`alltt`	`graphicx`	`psfrag`
`cc_manual_index`	`ifthen`	`epsfig`	`rotating`
`latex_to_html`	`makeidx`	`ipe`	`longtable`
`path`	`amssymb`	`pslatex`	

The additional macros defined are:

`\cgalversion`

Expands to the current CGAL version. This is either the version number, e.g., **Release 3.1-I-58**, that is contained in the `version.tex` file of an internal release if it exists, or otherwise the dummy text **Separate Build**.

`\cgalversiondate`

Expands to the date of the current CGAL version. This is either the date, e.g., **18 January 2004**, that is contained in the `version.tex` file of an internal release if it exists, or otherwise the current date from the LaTeX macro `\today`.

\cgaltitlepage{*quoted-package-name***}**

Creates a title page for the CGAL package named *quoted-package-name*. Note that underscores in the *quoted-package-name* need to be quoted, i.e., written as _. This macro will be used by the `cgal_manual` program to generate automatic title pages for manuals of individual packages.

\cgaldeclarepackage{*package-name***}**

Remembers the package name, such that it can be tested later with the `\cgalifpackage{`*package-name*`}` macro. The *package-name* can be given literally without quoting underscores and such. This macro will be used in the `\cgalchapters` expansion to declare all packages used in a module before they are included. So packages can check for the existence of other packages in a manual even if thus package is included later.

\cgalifpackage{*package-name***}{***yes-text***}{***no-text***}**

Tests if *package-name* was previously declared with `\cgaldeclarepackage{`*package-name*`}`, and if so expands to the *yes-text*, and otherwise expands to the *no-text*.

\cgalreinit

Re-initializes some style parameters of the C++ manual layout. It is typically used before each package chapter to give a package a clean start.

\cgalchapters{*body***}**

In a wrapper file for a CGAL manual the different user manual and reference manual chapters are grouped in this *body* using the macros `\entryleft`, `\entryright`, `\entryleftright`, `\packageleft`, `\packageright`, and `\packageleftright` explained below. The *body* is evaluated several times, let us call them the *declaration pass*, the *left pass*, and the *right pass*. The *declaration pass* applies the `\cgaldeclarepackage` macro to all arguments of all `\package`... macros, so that all packages are declared in the following passes.

The next passes are tailored for the idea in the HTML output to have a table-of-contents with two columns; one on the left for the user manual chapters, and one on the right for the reference manual chapters. This two-columns layout is optional. If it is activated (the default for CGAL manuals) only another, second evaluation of *body* is necessary and each macro is evaluated as if it would be the *left pass* and then immediately afterwards as if it would be the *right pass*. For the LaTeX processing and for a single-column table-of-contents in the HTML manual the *left pass* and the *right pass* are two separate evaluations of the *body*, effectively placing the reference manual behind the user manual.

The two-columns layout for the HTML table-of-contents organizes the table-of-contents in blocks on the left and right side. Entries on the same side are aggregated to a block until the side is switched. A switch from left to right closes the left block and opens a block on the right that is aligned with this left block. A switch from right to left closes the right block and opens a new left block below all other blocks. A block can be explicitly closed with the `\lcTocSync` command that effectively starting the next block below all others.

The following macros can be used in the *body*, and if suitable, also combined with `\lcTex{...}` and `\lcHtml{...}` macros. See Section 2.2.5 for an example.

\entryleft{*left-text***}**

Expands *left-text* when evaluated for the left pass of `\cgalchapters`. Can be used, for example, to place `\part{...}` commands in the *left-text*.

\entryright{*right-text***}**

Expands *right-text* when evaluated for the right pass of `\cgalchapters`.

\entryleftright{*left-text***}{***right-text***}**

For HTML output, synchronizes the block structure of the table-of-contents with `\lcTocSync` before and after expanding the texts. Expands *left-text* when evaluated for the left pass and expands *right-text* when evaluated for the right pass of `\cgalchapters`.

\packageleft{*left-package-name***}**

Expands to \cgaldeclarepackage{*left-package-name*} in the declaration pass of \cgalchapters. Expands to \cgalreinit\include{*left-package-name*/main} when evaluated for the left pass. It is typically used to include a package documentation in the manual with its own chapter, for example, with \packageleft{Kernel_23}. The \cgalreinit command re-initializes some style parameters to give the chapter a clean start.

\packageright{*right-package-name***}**

Same as \packageleft, but expanding for the right pass instead for the left pass of \cgalchapters.

\packageleftright{*left-package-name***}{***right-package-name***}**

Same as \packageleft{*left-package-name*} followed by \packageright{*right-package-name*} and enclosed by \lcTocSync to synchronizes the block structure of the table-of-contents for HTML output.

2.3 Users' manual

In contrast to the reference manual, the users' manual is intended to be read by users who are first considering whether to use part of the library. Thus the descriptions provided here should give sufficient information for a user to understand the functionality provided by the packages, but this does not mean, for example, that all functions and classes need to be described in great detail. Descriptions should be given in a more general way than in the reference manual. For example, one might mention that there is a function called *ch_jarvis* that implements the Jarvis march convex hull algorithm, but one need not explicitly state what the arguments of this function are, what the traits class requirements are, or the template parameters. The users' manual should also provide examples that are more lengthy than the ones provided in the reference manual. The examples should naturally be accompanied by sufficient explanation of the code in order for them to be understandable, and one should aim for examples that illustrate the most interesting features of a package and not necessarily the most complicated ones.

You should mention the precise name of the class(es) or function(s) being described (although NOT in the table of contents) so if the user wants the functionality being described it will be easy to find the thing that provides it. If the precise name is given (*i.e.*, the name given at the top of the reference page), then a hyperlink can be created automatically.

As a rule, each chapter should include:

- definitions necessary to describe the functionality (usually at the beginning of the chapter);
- a section describing the software design, where applicable. When there are no real design issues to explain, then a section isn't really warranted but at the very least the interface to the class or function should be briefly explained (*e.g.*, "the class *CGAL::Cone<T, DS>* has two template parameters; the first represents the traits class and the second the data structure", or "the function *flavor(b, e, r)* computes the value 42 from an iterator range *[b, e)* over a set of points. The computed value is stored in the output iterator *r*");
- one or more examples, which are included in the examples or demo directory of the package and inserted into the documentation using the \ccIncludeExampleCode command.
- a section (or paragraph) with information about the implementation, which includes:
  - references to the papers describing the algorithms/DS's implemented;
  - comments on the running times;
  - and perhaps a short description of how the algorithm works or the data structure is built.

- pictures of resulting output are good too.

The table of contents reflects the organization of the manual, and it should be a concise but informative listing. These specific guidelines regarding the organization are provided:

- There should be an introductory section that actually introduces the chapter, perhaps providing definitions of relevant terms. It should not be followed immediately by examples, as this implies that the entire description was contained in the introduction.
- The section describing software design should be labeled (you guessed it) "Software Design."
- Example programs should have entries in the table of contents and the user should be able to figure out quite easily what this example illustrated from the table of contents. This means, examples should be in sections of their own and the sections should have descriptive names (*i.e.*, "Example Constructing a Vanilla Cone" instead of just "Example", unless this is a subsection of a section entitled "Vanilla Cone").
- The examples should appear near the things of which they are examples. So for chapters describing more than one class (such as Triangulations) or one class that achieves different things by using different traits classes (such as Arrangements) or more than one global function, the examples should appear in the sections corresponding to each class. For example, if you had a section labeled, say, "Vanilla Cones" and then you do some explanations about what vanilla cones are and finally show an example of constructing one, the expected organization would be

  ```
  \section{Vanilla Cones}
  \subsection{Definition}
  \subsection{...}
  \subsection{Example}
  OR
  \subsection{Example with Extra Stuff}
  ```

  For chapters describing a single class (such as HalfedgeDS), the examples can appear in a section of their own.
- Examples should generally not span more than a page. Advanced examples are a possible exception.

In general, one should describe things at a level of detail that gives people enough information to determine if they want to refer to the reference page for the full details and one should use the precise name of a reference manual item so that cross-linking (by humans or the HTML converter) is possible. For example, you might say:

> The fact that points are required to be chocolate in order to compute a Hundink cone means that the traits class for a Hundink cone must provide flavored predicates. The precise description of the requirements is given by the concept HundinkConeTraits_6. The class *Hundink_cone_traits_6* is a model of this concept.

2.4 Reference manual

The reference manual is meant to be a place where programmers already familiar with the library can look to find information about specific parts of the library they want to use. It should provide detailed descriptions of the functions, classes, concepts, *etc.* provided in the library and should not be weighted down with lengthy explanations of the design philosophy or complicated examples; these things belong, if anywhere, in the users'

manual. Because the reference manual and users' manual are meant to be independent documents, there is going to be some overlap between the two (*e.g.*, text that introduces a package chapter), but this overlap should be kept to a minimum.

There are currently nine categories of reference pages that are supported by the manual tools. These are: class, function object class, concept, function object concept, enum, function, macro, constant, and variable. For each of these categories there is an environment that takes a single argument, which is the identifier plus an optional list of template arguments (for classes). Function parameters, function template declarations and macro parameters are not given here. The result of entering one of these environments is the production of a section heading that has the category followed by the identifier provided as the argument. For all categories except concepts and function object concepts, this identifier is prefixed by the global scope name that has been defined using `\ccDefGlobalScope`. By default, this scope is empty. For example the following commands

```
\ccDefGlobalScope{CGAL::}
\begin{ccRefClass}{Some_Class<T>} ... \end{ccRefClass}
```

produce the heading

2.5 Alphabetical List of Reference Pages

CGAL::Some_Class<T>

Class

If \ccNewRefManualStyle has been set to \ccTrue, each new reference manual environment will also produce a new page and will decorate the first page of each environment with a tab in the side margin. The macro \ccRefPageBreak can be used to turn off page breaks for very short reference pages (*e.g.*, for constants).

Below we briefly describe what should be documented in each of the ccRef* environments and list some of the most useful commands for each section. See Manual_tools/doc/cc_manual.ps.gz for a full description of these environments and the other commands available.

2.5.1 Section headings

The following commands defined in cc_manual.sty produce non-numbered section headings under which various aspects of an item should be documented. The commands are listed in the order in which they should be used. Sections that are empty for a particular item should, obviously, be left out.

\ccDefinition	**Definition**	including template parameters
\ccInheritsFrom	**Inherits From**	base classes
\ccRefines	**Refines**	concept names
\ccHasModels	**Has Models**	models for concepts
\ccIsModel	**Is Model for the Concept**	concept names
\ccTypes	**Types**	local type definitions
\ccConstants	**Constants**	constant values
\ccCreation	**Creation**	constructors, assignment
\ccOperations	**Operations**	functions and operators
\ccAccessFunctions	**Access Functions**	member access
\ccQueryFunctions	**Query Functions**	query functions
\ccPredicates	**Predicates**	predicates
\ccModifiers	**Modifiers**	insert, delete, update
\ccSeeAlso	**See Also**	other classes, functions
\ccImplementation	**Implementation**	running time, memory
\ccExample	**Example**	program examples

2.5.2 Classes

Classes are described using the ccRefClass environment. The types provided by the class should be described using\ccTypedef followed by a description of each member function, its parameters, and pre- and postconditions (using either \ccMethod, \ccMemberFunction, or \ccConstructor). The types and functions provided by a class should be documented in the order indicated by the ordering of the section commands listed in Section 2.5.1. Just after the definition section, a command \ccInclude should be used to indicate in which file the class is defined.

Note that it is NOT generally sufficient to say simply that this class provides all types and operations required by a concept it models. The point is to document the implementation details here so people can find out, for example, exactly what type of point is being used in the computation.

2.5.3 Concepts

Concepts should be described using the ccRefConcept environment. Concepts are abstractions that are defined by a set of syntactical and semantical requirements. These requirements include data, types, and functions. You

should describe the concept under a \ccDefinition heading and describe the requirements using as many of the headings listed in Section 2.5.1 as appropriate. In particular, under the heading produced by \ccHasModels you should list the classes in the library that are models for this concept.

2.5.4 Constants

Global constants should be described using the ccRefConstant environment. The name of the constant should be provided as the argument to the environment. Under a \ccDefinition heading, the meaning of the constant should be described. Just after the definition, a command \ccInclude should be used to indicate in which file the constant is defined. Following this, the complete declaration of the constant should be formatted using \ccGlobalVariable. Another possible section heading here is \ccSeeAlso.

2.5.5 Enums

Global enums should be described using the ccRefEnum environment. The name of the enum is provided as an argument to this environment. Within this environment the \ccGlobalEnum macro should be used to format the complete enum declaration. You should describe the enum and its values under a \ccDefinition heading and use as many of the other section headings listed in Section 2.5.1 as appropriate (*e.g.*, \ccSeeAlso, or \ccExample).

2.5.6 Functions

Global functions should be described using the ccRefFunction environment. The name of the function, excluding template declarations and parameters, is provided as the argument to this environment. Under a \ccDefinition heading, the purpose of the function should be described. Just after the definition, a command \ccInclude should be used to indicate in which file the function is defined. Following this, the macro \ccGlobalFunction should be used to format the function together with its template declarations and parameters. Preconditions and postconditions of the function should be documented using \ccPrecond and \ccPostcond. Other possible section headings here are \ccSeeAlso, \ccImplementation, and \ccExample.

2.5.7 Function Object Classes

Function object classes should be documented using the ccRefFunctionObjectClass environment. Just after the definition section, a command \ccInclude should be used to indicate in which file the class is defined. Following this, the concepts that this class is a model for should be listed under the heading \ccIsModel and then the list of member functions (a set of *operator()* functions) should be listed. Another possible heading here is \ccSeeAlso.

2.5.8 Function Object Concepts

Function object classes should be documented using the ccRefFunctionObjectConcept environment. The name of the concept is provided as the argument to this environment. Under the \ccDefinition heading, the concept should be described followed by the set of required functions (one or more *operator()* methods). Under the heading \ccRefines you should list concepts that this one "inherits" from and under \ccHasModels list the classes that are models of this concept.

2.5.9 Macros

Global macros should be describe using the `ccRefMacro` environment. The name of the macro, excluding its arguments, is provided as the argument to this environment. Under a `\ccDefinition` heading, the purpose of the macro should be described. Just after the definition, a command `\ccInclude` should be used to indicate in which file the macro is defined. Following this, the macro together with its arguments should be formatted using `\ccc`.

2.5.10 Variables

Global variables should be described using the `ccRefVariable` environment. The name of the variable should be provided as the argument to the environment. Under a `\ccDefinition` heading, the meaning of the variable should be described. Just after the definition, a command `\ccInclude` should be used to indicate in which file the variable is defined. Following this, the complete declaration of the variable should be formatted using `\ccGlobalVariable`. Another possible section heading here is `\ccSeeAlso`.

2.6 Figures

When including pictures in your documentation, you must provide versions of the pictures for the PostScript, PDF, and HTML versions of the manuals. This generally means providing both a `.ps` or `.eps` file for PostScript manuals, a `.pdf` or `.png` file for PDF manuals, and a `.gif`, `.jpg`, or `.png` file for HTML manuals. You should take care that the figures are readable in both formats and that they are neither too large nor too small. Also, all figures used in the HTML documentation should have transparent backgrounds. You can achieve this using the `giftrans` program distributed with LaTeX or by using one of the scripts available at `http://www.cgal.org/Members/Manual_tools` for converting from LaTeX to gif, Encapsulated PostScript to gif, PostScript to gif, and from ipe to gif.

We use currently PDFLaTeX to create the PDF manuals. The `\includegraphics` macro can be used conveniently for the PostScript images as well as the PDF images. One has just to omit the file suffix, and the `\includegraphics` macro uses automatically an `.eps` or `.ps` file (whatever exits) for the LaTeX run, and uses a `.pdf` file or `.png` file for the PDFLaTeX run.

To convert an existing `.eps` file to the corresponding `.pdf` file one can use the `epstopdf` script (e.g. available under Linux), which has the benefit of respecting the bounding box. (The `pstopdf` script ignores the bounding box.) Ipe users should save their files as `.eps` files and also use the `epstopdf` script to get a proper conversion respecting the bounding box and placement on paper. Recent Ipe versions have a CropBox option which also allows to respect the bounding box.

The program that converts to HTML does not currently support the commands that input PostScript figures into a document (since the figures for the HTML manual will be in a different format). Thus for each figure you must provide the raw HTML command that inserts the `.gif`, `.jpg`, or `.png` file in the document. For example:

```
\begin{figure}[htbp]
\begin{ccTexOnly}
\begin{center}
\includegraphics{orient} % omit suffix .eps to supprt PS and PDF
\end{center}
\end{ccTexOnly}
\caption{Orientation of a cell (3-dimensional case)
```

```
\label{Triangulation3-fig-orient}}

\begin{ccHtmlOnly}
<CENTER>
<IMG BORDER=0 SRC="./orient.gif" ALIGN=center ALT="Orientation of a cell
(3-dimensional case)">
</CENTER>
\end{ccHtmlOnly}
\end{figure}
```

Note above that only the parts that include the two different figure files are enclosed in the `ccTexOnly` and `ccHtmlOnly` environments. The `figure`, `caption`, and `label` commands will be processed for both LaTeX and `latex_to_html`. Note also that the caption and label are placed **above** the HTML picture. This is done so references to this figure will go to the top of the picture instead of the bottom (and thus be visible in the browser).

2.7 Indexing

In order to be truly useful, a manual needs to have an index. The `latex_to_html` program produces an index for the HTML manual automatically. The style file `cc_manual_index.sty` was developed in order to provide a means for producing an index for the PostScript version of the manual. Though there is also some automatic indexing done by the `cc_manual_index.sty` file in conjunction with `cc_manual.sty`, this indexing is not sufficient as it can index only things that are given as arguments to the formatting macros. When writing your documentation, you should add other indexing commands in the text in order to achieve the following goals:

- All key words, phrases, topics, and concepts should be indexed.
- All CGAL C++ identifiers described in the manual should be indexed.
- There should be listings in the index for any pages on which an item is introduced, defined, or described as well as any pages where key uses for or instances of these things are described.
- There should be sufficient cross referencing to allow users to find things starting from many different points.
- There should NOT be entries for every mention of every item in the manual.

You should also take care that the index entries produced by the automatic indexing are in keeping with these goals, and, when not, use the commands provided in `cc_manual_index.sty` to turn off the automatic indexing. See the file `Manual_tools/doc/cc_manual_index.ps.gz` for a description of the commands available and a full description of what should and should not be indexed.

2.8 Test suite

With each internal release of the library, a test suite is run on the manuals to make sure that the documentation submitted makes it through all the programs and scripts without a problem. Each package and each module manual is tested with both LaTeX (and its companion programs `makeindex` and `bibtex`) and `latex_to_html`. The results of the test suite are currently available at

`http://www.cgal.org/Members/Manual_test/LAST/`

It is a given, of course, that developers should **test** that their packages' documentation works with **both** LaTeX and `latex_to_html` **before** submitting. The test suite is meant simply to assure that all the parts fit together nicely and all references to other parts of the manual get resolved correctly. Each time you submit a package with modified documentation, you should check these test results to make sure your documentation did not cause any problems.

2.9 Common problems and solutions

Problem — Undesired Links in HTML Manual

Cause: Any occurrence of the name of a class, function, etc. that is documented in the reference manual is automatically crosslinked to the definition of that entity.

Solution: Sometimes, the preferred solution is to chosse a better name for the entity under consideration. For example, a traits class for a planar map should be called *Planar_map_traits* rather than *Traits*. Another possibility is to add the command `\ccHtmlNoClassLinks` before the `\begin{ccClass}{Traits}` command to globally turn off linking of the word *Traits*. To disable links from one word only, use the command `\ccHtmlNoLinksFrom`. For example, `\ccHtmlNoLinksFrom{Traits}` will not create links for this particular occurrence of the word *Traits*.

Problem — Nontransparent backgrounds for HTML figures

Cause: The `.gif` file was produced without a transparent background.

Solution: Use either the `giftrans` program distributed with LaTeX or one of the scripts `ps2gif`, `ipe2gif`, or `latex2gif` that are available from `http://www.cgal.org/Members/Manual_tools` (which use `giftrans` and work for figures with white backgrounds) to make the background transparent.

Problem — Unresolved figure references in HTML

Cause: This problem is generally caused by the absence of a `\label` command inside the HTML figure environment.

Solution: The easiest way to solve this problem is to put the `\label` command in the text that is processed by the HTML converter, as the following example illustrates:

```
\begin{figure}[htbp]
\begin{ccTexOnly}
\begin{center}
\includegraphics{orient} % omit suffix .eps to supprt PS and PDF
\end{center}
\caption{Orientation of a cell (3-dimensional case)
\label{Triangulation3-fig-orient}}
\end{ccTexOnly}
\lcHtml{\label{Triangulation3-fig-orient}}
\begin{ccHtmlOnly}
<CENTER>
```

```
<IMG BORDER=0 SRC="./orient.gif" ALIGN=center ALT="Orientation of a cell
(3-dimensional case)">
</CENTER>
\end{ccHtmlOnly}
\end{figure}
```

This will produce a centered figure without a caption in the HTML manual. References to the label `Triangulation3-fig-orient` will produce a link that refers to the top of this figure. If you want the caption with the HTML figure as well, then simply move the `\end{ccTexOnly}` command above the `\caption` command and remove the `\lcHtml` command. See also Section 2.6.

Problem — Raw LaTeX commands in HTML manual

Cause: This problem is generally caused by the use of commands that are not supported by the HTML converter. See `latex_to_html.ps.gz` for a list of these unsupported commands.

Solution: Either remove the offending command altogether or enclose it in a `\lcTex` command or an `lcTexBlock` environment (which are both defined in `latex_converter.sty`).

Problem — Included files cannot be found

Cause: This problem is generally caused by a wrong relative path given in the `\input` command (or its derivative) or the absence of a directory from the appropriate environment variable.

Solution: Paths to source files in other directories must generally be provided relative to the place where LaTeX and `latex_to_html` are run, not relative to the place where the file containing the command is. When producing the manual, these programs are run in the `doc_tex` directory, so paths should be relative to this directory.

An exception is for example and demo programs included in the documentation. . These should always be programs that are in the distribution (and thus in the test suite) and should be included using the command `ccIncludeExampleCode`. For programs you need only specify the path relative to the directory `CGAL_ROOT/examples` or `CGAL_ROOT/demo` as the environment variables `TEXINPUTS` and `LATEX_CONV_INPUTS` are set so these directories are searched when the manuals are created with the `cgal_manual` program from Section 2.2.1. (See `Manual_tools/doc/latex_to_html.ps.gz` for more details.) For example, you would use the following command

```
\ccIncludeExampleCode{Polygon/PolygonDemo.C}
```

to include the source file `CGAL_ROOT/demo/Polygon/PolygonDemo.C`. Obviously, you could have problems if you have an example program and a demo program with the same name in the same package, so try to avoid this.

2.10 Requirements and recommendations

Requirements:

- Documentation must be written using the `cc_manual.sty` and `cc_manual_index.sty` style files.

- There must be a `main.tex` file for both the users' manual and reference manual parts of the documentation.
- Files must be organized according to the directory structure outlined in Section 2.1 and further detailed in Section 4.2.
- Indexing commands must be placed in the text to achieve the goals listed in Section 2.7.
- Example programs included in the manual should also be distributed in the `examples` or `demo` directories and should be incorporated in the documentation using the `\ccIncludeExampleCode` command.

Recommendations:

- Documentation of items in the reference manual should use as many of the headings listed in Section 2.5.1 as appropriate and in the indicated order.
- The users' manual should include illustrative examples of a package's functionality.
- Document each item in the reference manual in its own file.

Chapter 3

SVN Server

*Sylvain Pion (*Sylvain.Pion@sophia.inria.fr*)*

SVN (Subversion) is a popular version control system that keeps track of the complete history of changes for a set of source files. SVN is of great help when several people work on the same project.

You can find the SVN manual at http://svnbook.red-bean.com. Lots of documentation, including tutorials, can easily be found on the net.

The InriaGForge site http://gforge.inria.fr hosts a SVN server for CGAL. The complete documentation to access the CGAL project on InriaGForge is at http://www.cgal.org/Members/svn_cgal/. In a nutshell:

3.1 Structure of the repository

The CGAL repository, which is backed up daily, is organized in the following way:

- There is one folder under /svn/cgal/trunk/ per CGAL package.
- Such a folder has the same structure as the current CGAL packages (Chapter 4).

  As a general rule, try to not put under SVN control any files that are automatically generated. Note that you can put some files under SVN control that you don't want to be included in releases (*e.g.*, literate programming web files). You can do that by adding their names in a dont_submit file (at the top of the module), using the syntax explained in the tar manual (see the --exclude-from option in the doc at http://www.gnu.org/manual/tar/). By default, the files TODO and wrapper.tex are excluded from submissions.

The repository tree has the following structure:

```
/svn/cgal
        |
        +---- trunk/
        |         |
        |         +--- Convex_hull_2/
        |         |     |
        |         |     |-- maintainer
```

```
|          |     |
|          |     |-- changes.txt
|          |     |
|          |     |-- TODO
|          |     |
|          |     +-- include/CGAL/
|          |     |
|          |     +-- src/
|          |     |
|          |     +-- doc_tex/
|          |     |
|          |     +-- test/
|          |     |
|          |     +--  ...
|          |
|          +--- Number_types/
|          |
|          +--- Cartesian_kernel/
|          |
|          +--- ...
|
+---- branches/
|          |
|          +--- CGAL-1-1-patches/
|          |
|          +--- ...
|
+---- tags/
           |
           +--- CGAL-1-0/
           |
           +--- ...
```

3.2 Access to the repository

Here are the access rules:

- Everybody who has a login and is part of the CGAL project on InriaGForge has read/write access to the CGAL repository via SVN. There is also read-only access for members via the web at `https://gforge.inria.fr/plugins/scmsvn/viewcvs.php/?root=cgal`.

- Each package has a set (which can be only one person) of people having write access to the corresponding folder. It's up to the maintainer of the package to decide who is given write access, and how you should coordinate.
  This rule is not currently enforced by SVN (each CGAL developer has write access to the whole repository from SVN's point of view). CGAL developers must voluntary enforce this rule.

- Currently, there is no anonymous read-only access via SVN. Only the read/write access for members via SVN over SSH is available.

3.3 How to use it

- The first step is to register to InriaGForge at `http://gforge.inria.fr`.
- Step 2 is to create a pair of public/private SSH 2 keys, and to upload the public one to InriaGForge.
- To be added to CGAL project, send your login name to `Sylvain.Pion@sophia.inria.fr` or `Laurent.Saboret@sophia.inria.fr`. Also, indicate for which packages you need write permission.
- If your SSH key contains a passphrase, you have to enter it in `ssh-agent`. Type:
  `ssh-add`
  and enter the passphrase corresponding to your key.
- At this point, it's handy to store the repository URL in a shell variable:
  `export svncgal=svn+ssh://scm.gforge.inria.fr/svn/cgal`
- To get the list of existing modules:
  `svn ls $svncgal/trunk`
- To check out a module, say *Convex_hull_2*:
  `svn co $svncgal/trunk/Convex_hull_2`
- Once you have checked out a module, you don't need to specify the `$svncgal` URL anymore for the usual commands like `svn commit`...
- You may check out all packages by:
  `svn co $svncgal/trunk`

After each commit, an automatic mail is sent to the `cgal-commits@lists.gforge.inria.fr` mailing-list which records the log message as well as the URLs to the corresponding diffs within the ViewCVS interface. Maintainers and developpers can subscribe to it in order to check what gets commited to their packages.

Chapter 4

Directory Structure for Packages

*Geert-Jan Giezeman (*geert@cs.uu.nl*)*
*Susan Hert (*hert@mpi-sb.mpg.de*)*
*Sylvain Pion (*Sylvain.Pion@sophia.inria.fr*)*

In order for code, demos, documentation, *etc.* developed for CGAL to be included in an automated way into the internal (and public) releases of the library, the files must be organized in a specific directory structure, which we describe here. We describe the entire directory structure for a package. Not every package will have all the parts described here. Only the files `maintainer`, `changes.txt` and `description.txt` are obligatory. Submissions should not contain files in other places than described here. Packages may be rejected if they do, and the contents won't go into releases.

The directory structure of a submission should be as follows:

```
+--- include/CGAL
|
+--- src/
|
+--- test/<testdir>
|
+--- doc_tex/
|
+--- examples/<exampledir>
|
+--- demo/<demodir>
|
+--- auxiliary/
|
+--- scripts/
|
+--- developer_scripts/
|
|- changes.txt
|
|- description.txt
|
|- dont_submit
|
|- long_description.txt
```

```
|
|- maintainer
```

`include/CGAL` contains the `.h` files for the package.

`src` contains the `.C` files (if any) for the package.

`test` contains the test suite code for the package. See Section 4.1 for a detailed description.

`doc_tex` contains the documentation for the reference manuals. These are the files used to produce the manuals for public releases and to run the internal release reference manual test suite. See Section 4.2 for a detailed description of this subdirectory. See Chapter 2 for guidelines for producing the documentation.

`examples` contains the example programs for the package. See Section 4.3 for a detailed description of this subdirectory.

`demo` contains the demo programs for the package. Contrary to example programs, demo programs are not expected to be usable on all platforms. Demo programs may depend on platform-specific software and may require user interaction. They are compiled but not run in the test suite. See Section 4.4 for a more detailed description of this subdirectory.

`auxiliary` contains auxiliary software for CGAL. For instance, GMP goes here. The average package won't have this directory.

`scripts` contains scripts that are of interest to CGAL users.

`developer_scripts` contains scripts that are of interest to CGAL developers. This directory is included in internal releases only, not in public releases.

`changes.txt` used to document important changes between subsequent versions. Each entry must provide the date of the change and the name of the person who committed it, as in the following example:

> 2 Feb 2004 Sylvain Pion
> - Fix a memory-leak in file.h.
> - Optimized function A::f().

`description.txt` should give a very short description of the contents of the package.

`dont_submit` specifies files and directories that are to be excluded from the release, for example:

> TODO
> include/CGAL/my_internal_file.h
> Benchmark

`long_description.txt` may be added with more detailed information about the submission.

`maintainer` should be used to names the maintainers of the package. The file should have for each maintainer a line with the following format:

> *name* <*email address*>

For example:

> *Susan Hert* <hert@mpi-sb.mpg.de>
> *Sylvain Pion* <Sylvain.Pion@sophia.inria.fr>

4.1 `test` subdirectory

This directory contains the test suite for the package. Here we just briefly list the files contained in such a directory. For more detailed information about testsuites for CGAL packages refer to Chapter 17.

```
test/<testdir>
         +--- data/
         |
         +--- include/
         |
         |- cgal_test
         |
         |- makefile
         |
         |- *.C
         |
         |- *.cin
```

where the directory `<testdir>` has a name that corresponds to the package name.

`data/` is a subdirectory that contains local data files for the test suite.

`include/` is a subdirectory that contains local include files for the test suite.

`cgal_test` is the script that will be called when the entire test suite is run. As this file is created automatically during the release process, submitting it is error-prone and thus **strongly discouraged**.

For testing purposes, such a script can be created using the `create_cgal_test` script (Section 5.5).

`makefile` is the makefile for the test programs. It is created automatically, just like `cgal_test`. Submitting it is **discouraged even more strongly**, because it is quite difficult to produce a makefile that works correctly on all supported platforms.

For testing purposes, such a makefile can be created using the script `cgal_create_makefile` with the argument `-t` (Section 5.4).

`*.C` source code for the test programs.

When a test program runs correctly, it should return the value zero, or, more precisely, the value `std::EXIT_WITH_SUCCESS` defined in `<cstdlib>`. Test programs may not be graphical and they may not require any user interaction.

`*.cin` files containing command-line input for test programs. These files are necessary for only those programs that require command-line input (which can usually be avoided). If a file `program.cin` is present in the test directory, then, when the test suite is run, `program` will be executed using the command

```
./program < program.cin
```

advanced

Custom cgal_test script. In special cases, you may want to provided a custom `cgal_test` script to fit special needs. The script should rely on four environment variables:

`$CGAL_MAKEFILE` (an included makefile, which **must** include the full path name!!)

`$PLATFORM` (the extension of this makefile, that will be used as an extension to the output files),

`$TESTSUITE_CXXFLAGS` (additional compiler flags)

`$TESTSUITE_LDFLAGS` (additional linker flags)

The latter two flags must be passed to the makefile and they should precede all other compiler and linker flags. The script then performs all tests using the makefile `$CGAL_MAKEFILE`.

To indicate whether the tests are successful or not, the script writes two one-line messages to a file called `error.txt` for each target. If something goes wrong during compilation or during execution, an error message is written that starts with the keyword `ERROR:`; otherwise a message indicating successful compilation or execution is written to this file. Running the script `cgal_test` must not require any user interaction and the script cleans up after itself by removing object files and executables (usually by using the command `make clean`).

Custom makefile. In special cases, you may want to provided a custom `makefile` to fit special needs. Use the two variables `$TESTSUITE_CXXFLAGS` and `$TESTSUITE_LDFLAGS` as the first compiler and linker flags, respectively.

advanced

4.2 `doc_tex` subdirectory

As described in Chapter 2, the CGAL documentation is organized in subdirectories for each package and the different manuals are assembled from these packages. Contained in these subdirectories are the files required for producing a package's contributions to the different reference and users' manuals. The users' manual input files are located in the package's directory; the reference manual files are be located in a directory named $<$ *Package* $>$`_ref` . For both the users' manual and reference manual parts, the input can be split into more than one file (In fact, this is necessary for the reference manual in order to support conversion to HTML; see Section 2.4.), but there must be a file called `main.tex` in both the user and reference manual directories that inputs all the other files for that manual part. (**Note**: You should use the `\input` command and NOT the `\include` command to input other source files in this file, and they have to include their files using relative paths starting in the `doc_tex` directory.)

For example, the optimisation package of the basic library can have the following documentation.

```
doc_tex/Optimisation
           |--- main.tex
           |
           |--- *.tex
           |
           +--- *.((ps)|(eps)|(pdf)|(gif)|(jpg)|(png))

doc_tex/Optimisation_ref/
           |--- main.tex
           |
           |--- intro.tex
           |
           |--- *.tex
           |
           +--- *.((ps)|(eps)|(pdf)|(gif)|(jpg)|(png))
```

`main.tex` must contain one chapter. It must NOT contain a preamble (so no `documentclass`, `usepackage`, `\begin{document}` or `\end{document}` commands). If you want more than one chapter in the documentation of this package you have to put each chapter in its own LaTeX file, and include those files in `main.tex` using the `\input` macro.

`intro.tex` is not mandatory but common for reference manual chapter. It contains a brief introduction to the package (one paragraph) and lists the different concepts, classes, functions, etc. that are contained in this package in a systematic way.

`*.tex` – the source files for the Users' and Reference Manual that are input by `main.tex`

`*.((ps)|(eps))` – the PostScript pictures included in the PostScript documentation.

`*.pdf` – the PDF pictures included in the PDF documentation.

`*.((gif)|(jpg)|(png))` – the raster images included in the HTML documentation.

Historical Notes. Once upon a time a file called `htmlfiles` has been used to indicate files (e.g., images) to be used for the html conversion. This file is no longer needed nor is it evaluated anymore. The functionality is now provided by an mechanism in the `latex_to_html` converter to find and copy the included raster images automatically. Also the `cgal.bib` files are not used anymore. They are replaced by a common file `doc_tex/Manual/cgal_manual.bib` in the `Manual` package.

4.3 `examples` subdirectory

Example programs (Chapter 19) for a package should be placed in a subdirectory of the directory `examples`. The subdirectory name, `<exampledir>`, will usually correspond to the package name, but this is not a requirement. To make sure that the examples will be tested, a directory with examples should be submitted in exactly the same way as a test directory (Section 4.1).

The structure of an example directory should be as follows:

```
examples/<exampledir>
             +--- data/
             |
             +--- include/
             |
             |- README
             |
             |- cgal_test
             |
             |- makefile
             |
             |- *.C
```

The file `README` should contain information about what the programs do and how to compile them. See the rules for a test directory for an explanation of the other files and subdirectories.

4.4 `demo` subdirectory

The `demo` directory (Chapter 19) contains programs with graphical interfaces or programs requiring user input. These programs will be compiled but not run by the test suite. The structure of this directory should be as follows:

```
demo/<demodir>
      +--- data/
      |
      +--- include/
      |
      |- README
      |
      |- makefile
      |
      |- *.C
```

where `<demodir>` is a name that corresponds (at least in part) to the package for which it is a demo.

The file `README` should contain information about what the program does, and how to compile it (*i.e.*, what graphical libraries are needed, *etc.*). Note that, in contrast to example programs and test suite programs, for demo programs is it necessary to submit a `makefile` since different demos will require different libraries and thus the makefiles for these programs will be somewhat dissimilar.

4.5 Requirements and recommendations

Requirements:

- The directory structure outlined here must be followed exactly.

Recommendations:

- Do not submit `makefile`s for example programs and test suite programs.
- Do not submit the script `cgal_test` used to compile and test your test suite programs.

Chapter 5

Scripts and Other Tools

*Geert-Jan Giezeman (*geert@cs.uu.nl*)*

There are a number of tools that automate some of the tedious and error-prone tasks that are unavoidable when developing for CGAL. Most of these scripts can be in at least two places.

- in every internal release (directory `developer_scripts` or `scripts`)
- in the package Scripts on the SVN server (`https://gforge.inria.fr/plugins/scmsvn/viewcvs.php/trunk/Scripts/?root=cgal`) .

5.1 `create_assertions.sh`

The shell script `create_assertions.sh` generates assertion header files. The command

```
create_assertions.sh minkowski
```

creates a file `minkowski_assertions.h`. This file is usually placed in directory `include/CGAL` and included in files that implement the minkowski package. The file contains several macros for adding checks (preconditions, postconditions, *etc*.) that can be switched on or off with compile flags. You can read more about the use of these package-level checks in Section 9.3.

5.2 `remove_line_directives`

CGAL source files should not contain line directives, even if they are made with the help of a literate programming tool like funnelweb. The script `remove_line_directives` can be used to remove lines starting with `#line`.

5.3 `rename_clib_calls`

According to standard C++, when you include a header from the standard C library in the modern way (so, `#include <cmath>` instead of `#include <math.h>`), the C functions should be in the namespace *std*. Not all

compilers conform to this. That's why CGAL has a macro, CGAL_CLIB_STD, that is set to *std* (for a conforming compiler) or to the empty string (for a non-conforming compiler). This macro should be prepended to all calls of functions from the C library. So, you should call *CGAL_CLIB_STD::sin(x)*, not *std::sin(x)* or *sin(x)*.

The perl script `rename_clib_calls` does this prepending. The script is idempotent, so applying it twice in a row does no harm. The script is not perfect, though. Especially, it recognizes complete words, but it does not discriminate between a function call and a variable or a word in a comment. You should check the differences of the modified file with the original file (which will be saved, with a `.bak` extension), for example with `diff` or `xdiff`.

Before using this script, you should edit the first line. This line states where your perl interpreter is located. The command `which perl` can help you to find out what to put there.

5.4 cgal_create_makefile

The shell script `cgal_create_makefile` creates a CGAL makefile in the current working directory. The makefile will have rules to make an executable from every `.C` file in the directory. The following options are available to indicate the kind of makefile to create:

-d create a default CGAL makefile

-t create a makefile for the test suite

-w create a makefile with flags for LEDA windows

-g create a makefile with flags for GEOWIN

-q create a makefile with flags for QT

You should use this script to create makefiles for testing before submitting your packages, but you should **avoid** customizing the makefile for your package and then submitting the makefiles for your test suite and example programs. Makefiles get created automatically when a release is made. This ensures that the most up-to-date makefiles are created. Experience has shown that makefiles need to be updated from time to time, especially if new compilers are supported. When makefiles are submitted as part of a package, they tend to get outdated. Moreover, creating a makefile that works for a diverse range of compilers is difficult.

The makefiles created by this script have two hooks for customization. If the directory where the makefile is created contains a directory named `include`, that directory will be on the header search path before any other files. This provides a means to override header files. Furthermore, the makefiles in the test suite (created with the `-t` option), contain the variable `EXTRA_FLAGS`. This variable can be set, *e.g.*, as an environment variable in `cgal_test`, to add `-I` or `-D` options.

5.5 create_cgal_test

In every subdirectory of `test` there must be a shell script called `cgal_test`. This script is called from the `run_testsuite` script when the test suite is run. The script `create_cgal_test` will create the shell script `cgal_test` in the current directory.

As is the case with makefiles, this file will be generated automatically for each subdirectory of `test` if it is not supplied by the package. This is also the preferred way. So you should use `create_cgal_test` to test your

package before submitting but generally should avoid submitting the file cgal_test. But sometimes you may want to do something extra in this script (*e.g.*, set the EXTRA_FLAGS environment variable that is used in the makefile). In this case, you can run create_cgal_test yourself and edit the result. The generated file contains a header that lists the requirements a cgal_test program should fulfill. These are also listed in Section 4.1 with the description of the cgal_test script.

5.6 autotest_cgal

The shell script autotest_cgal is used to run automated test suites for the internal releases of the library. The script takes no command-line argument, but it is configurable using a .autocgalrc file stored in $HOME. It is meant to be run regularly by a cron job run by the testers.

This script allows you to run the testsuite on multiple hosts (remotely or locally) and multiple compiler-platforms. It is also possible to specify how many processors the host has and the script will use this information to run the testsuite process on several processors at the same time. By default, the number of processors assigned to every host is one.

In a couple of words, the script first downloads a file named LATEST that contains the current release file name. It compares it with the file RELEASE_NR in the $CGAL_ROOT directory and if it is different it starts the testsuite process : it downloads the internal release, unzips, untars, copies the configuration files, and starts the testsuite. When everything is done, it puts the results on some ftp server.

How to set up your local testing

The first step is to create a directory where you will store the releases. This directory is known in the script as $CGAL_ROOT. By default $CGAL_ROOT is the directory where you put the script autotest_cgal, but you may change it in .autocgalrc.

The next step is to download an internal release of your choice, say CGAL-3.1-I-xx, and install it on the various platforms you want it to be installed using the usual procedure with the install_cgal script. This will create the configuration files in the directory CGAL-3.1-I-xx/config/install/.

Then, create a symbolic link to this release using ln -s CGAL-3.1-I-xx CGAL-I. This link will be updated by autotest_cgal to point to the last installed release. You may remove older releases at some point.

The script uses wget to download the release. The different options you may pass to wget you should put in the WGET variable in .autocgalrc. By default, WGET=''wget''. We noticed that you must use a recent version of wget (1.9.1 is fine, but 1.8.2 is not). You must also create a file in your HOME called .wgetrc (alternatively, it is also possible to use another file by setting the WGETRC variable), that contains the following lines:

```
--http-user=member1
--http-passwd=xxxxxxx
```

Alternatively, you can use curl instead of wget. In order to do so, you need to add USE_CURL=''y'' to your .autocgalrc file. curl is then also used as a replacement for ftp for uploading the test results. You also have to add the following line in a file called HOME/.curlrc:

```
--user member1:xxxxxxx
```

The next step is to specify the name of the hosts. To do that, you must define the variable BUILD_HOSTS in the file .autocgalrc. For every host *h* listed in BUILD_HOSTS you must also specify the variable COMPILERS_*h* as a list of OS-compiler descriptors for which the testsuite is to be run on *h*.

Optionally, you can also specify a variable BUILD_ON_*h* as a list of OS-compiler descriptors for which the CGAL libraries are to be built on *h*. If BUILD_ON_*h* is not defined, it defaults to the value of COMPILERS_*h*. For obvious reasons, it is a good idea to keep the compilers in COMPILERS_*h* a subset of the compilers listed in BUILD_ON_*h*. Setting BUILD_ON_*h* to all, invokes the install script with --rebuild-all and, therefore, builds CGAL using all compilers for which a matching config/install file is found.

Here is an example:

```
BUILD_HOSTS="papillon electra winmachine localhost"
COMPILERS_papillon="mips_IRIX64-6.5_CC-n32-7.30 mips_IRIX-6.5_g++-fsquangle-2.95.2"
COMPILERS_electra="sparc_SunOS-5.6_g++-2.95.2"
BUILD_ON_electra="sparc_SunOS-5.6_g++-2.95.2 sparc_SunOS-5.6_CC-5.80"
COMPILERS_winmachine="i686_CYGWINNT-5.0-1.3.22_CL.EXE-1300"
COMPILERS_localhost="i686_Linux-2.4.17_g++-3.4.0"
PROCESSORS_papillon="3"
BUILD_ON_papillon="all"
```

You should take those names from the configuration files mentioned earlier. The PROCESSORS_hostname must be set if you have more than one processor on that host and of course if you want to give all of them to the testsuite. If you want to run the testsuite locally and not remotely, name the host as localhost and the script will run the testsuite locally for that host.

List of configurable variables

MYSHELL is a variable that must be defined in .autocgalrc, otherwise the script will not run. You can use it to pass environment variables along to remote hosts. Here is an example of MYSHELL variable :

```
MYSHELL="TERM=vt100 PATH=$PATH:/bin:/usr/local/bin:/usr/ccs/bin:/opt/SUNWspro/bin \
QTDIR=/usr/local/qt2 PARASOFT=/usr/local/parasoft /usr/local/bin/zsh -c\"
```

CGAL_URL is the URL where internal releases are available. The LATEST file is comming from the same location. If this will change, you may change either CGAL_URL, or LATEST_LOCATION in .autocgalrc.

TAR, GZIP, GUNZIP are the variables for compression and decompression tools. They are defaulted to tar, gzip, gunzip found in $PATH.

SENDMAIL has the default value ''mail''.

CGAL_TESTER has the default value `whoami`. It is used to identify the tester. You should specify a kind of login name for this variable.

CGAL_TESTER_NAME has the default value `whoami`. It is used to identify the tester. You should specify your full name for this variable.

CGAL_TESTER_ADDRESS has the default value `whoami`. It is used to identify the tester. You should specify your email address for this variable.

MAIL_ADDRESS has no value by default. In case you want to set it in .autocgalrc, the script will send an additional email to everyone mentioned in this variable.

LOGS_DIR has the default value $CGAL_ROOT/AUTOTEST_LOGS. This is used to keep local log files.

CONSOLE_OUTPUT has the default value "y". If you want that the script also shows messages to the console, the value is "y" otherwise the value should be "".

TEST_DIR_ROOT specifies where the test-suite will be copied before being run. For example, this can be set to */tmp*, instead of the default which is $CGAL. This directory can be local to the testing machine.

USE_CURL specifies if curl should be used as a replacement for wget and ftp. Set it to ''y'' to use curl.

NICE_OPTIONS specifies the command line options to pass to the nice command, which is used to prevent the test-suite from keeping all CPU time for itself. The default value is ''-19''.

5.7 create_internal_module

The perl script create_internal_module is used to generate independent modules from CGAL packages. This script was adapted from the script create_internal_release that is used to build CGAL internal releases. It generates modules from CGAL SVN packages.

To see a short list of possible parameters you may want to set, just run the script with no arguments.

You must have the packages you want to include in the module in one directory. You must create one file modules that contains the packages list that will go into the module itself.

Here is an example of using the script:

```
$ ./create_internal_module -a /cygdrive/d/CGALSVN -p modules
-m Triangulation_module -s /cygdrive/d/CGALSVN/Scripts/scripts/
-r CGAL-3.1-I-76 -n 1003001076
```

The -a parameter is used to specify the directory that contains the packages. The -p parameter is used to specify the file containing the packages list. The -m parameter is used to specify the name of the module. If there is no m parameter, the default module name is NEW_MODULE. The parameter -r is used to specify the module version. The parameter -n is used to specify the module version number. If you don't use the -n parameter, the version number will be taken from the module version. The module version and version number will be used in the files version.h and version.tex. The -s parameter is used to specify the directory containing the scripts cgal_create_makefile, and all the other scripts that CGAL made public for users. This is not the package Scripts, but the scripts directory in the Scripts package.

You must have wininst package in the same directory with Scripts package because the script is looking for C2vcproj script used to generate the projects in examples for VC++.

The script also generates the makefiles in examples, which was done also by the mother script create_internal_release. This is done only if there is no makefile called makefile in the directory. The script behavior concerning the src dir of the module is as follows:

- when there is a src dir in the module it generates the makefile_lib file that is the makefile to build the CGAL library (only for the C files in the src dir).

- when there is no src directory it simply does nothing.

For the moment the script does not produce the tar.gz version of the module, this will be done in the future if it will be required.

5.8 create_modules

This script is used to trigger the previous script create_internal_module. It looks in the $MODULES_DIRECTORY directory for all the files containing the packages list that should generate the module. For example if the $MODULES_DIRECTORY is "Modules", than it looks in the Modules directory for all the files. Each file in this directory represents a module to be created, the name of the file becoming the name of the module. Each file contains a list of packages, one package per line.

For each file in Modules directory, for each package enumerated in the file, it checks out or updates the package in the $SVNDIR. $SVNDIR represents the directory where you store the packages. It is the same as the "All packages dir" from the previous script.

Once the update process is done, it calls create_internal_module script with the right parameters.

The version and version number are preset and the way to determine those will be changed in the near future. I am just looking for the best way to do it.

5.9 check_licenses

This script can be used to check all files in an internal or external release for proper license notices. It reports all files without a proper notice. Observe that the check is fairly simple, we just grep through the files looking for a fixed string. Additionally, there might be provisions in the top-level LICENSE file that are not taken into account by this script.

Note that there might be license errors that are not detected by this script. For example, we do not check that files under LGPL and QPL are not mixed in one library.

Chapter 6

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 6.2) that should be adhered to, rules for the code format (Section 6.3), and a mandatory heading for each source file (Section 6.4)

6.1 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 14). 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 ' '.

6.2 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.

6.3 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*.
- Remove line pragmas, *i.e.*, `#line...`.(See Section 5.2.)

6.4 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.C`) 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.

QPL version

Here follows what this gives for a file under the QPL :

```
// Copyright (c) 1999,2000,2001,2002  INRIA Sophia-Antipolis (France).
// All rights reserved.
//
```

```
// This file is part of CGAL (www.cgal.org); you may redistribute it under
// the terms of the Q Public License version 1.0.
// See the file LICENSE.QPL distributed with CGAL.
//
// Licensees holding a valid commercial license may use this file in
// accordance with the commercial license agreement provided with the software.
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//
// $URL: svn+ssh://scm.gforge.inria.fr/svn/cgal/branches/CGAL-3.2-branch/Developers_manual/doc_tex/Develo
// $Id: code_format_internal.tex 28567 2006-02-16 14:30:13Z lsaboret $
//
//
// Author(s)     : Monique Teillaud <Monique.Teillaud@sophia.inria.fr>
//                 Sylvain Pion <Sylvain.Pion@sophia.inria.fr>
```

LGPL version

Here follows what this gives for a file under the LGPL :

```
// Copyright (c) 2000,2001,2002,2003  Utrecht University (The Netherlands),
// ETH Zurich (Switzerland), Freie Universitaet Berlin (Germany),
// INRIA Sophia-Antipolis (France), Martin-Luther-University Halle-Wittenberg
// (Germany), Max-Planck-Institute Saarbruecken (Germany), RISC Linz (Austria),
// and Tel-Aviv University (Israel).  All rights reserved.
//
// Licensees holding a valid commercial license may use this file in
// accordance with the commercial license agreement provided with the software.
//
// This file is part of CGAL (www.cgal.org); you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public License as
// published by the Free Software Foundation; version 2.1 of the License.
// See the file LICENSE.LGPL distributed with CGAL.
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//
// $URL: svn+ssh://scm.gforge.inria.fr/svn/cgal/branches/CGAL-3.2-branch/Developers_manual/doc_tex/Develo
// $Id: code_format_internal.tex 28567 2006-02-16 14:30:13Z lsaboret $
//
//
// Author(s)     : Herve Bronnimann, Sylvain Pion
```

6.5 Requirements and recommendations

Requirements:

- Follow the naming schemes outlined above.
- Provide typedefs of template arguments as necessary to make the template parameters accessible elsewhere.
- Label member function and parameters with *const* where appropriate
- Use C++-style type casts.
- Protect header files from multiple inclusions.
- Obey the code format rules outlined above.
- Provide a header for each submitted file in the proper format.

Chapter 7

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 8. If you need primitives not provided by the kernel (yet), please read Section 7.6 below.

7.1 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,

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

.

7.2 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

7.3 Available kernels

At present, there are two homogeneous and two Cartesian kernels, one with reference counting (Chapter 10) 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.

7.4 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.

7.5 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.

7.6 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 7.2 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 8

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.

8.1 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.

8.2 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 8.4), 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.

8.3 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.

8.3.1 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:

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.

8.3.2 CGAL-provided traits classes

As mentioned in Section 8.1, 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.

8.4 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 6.1) should be used to make the library more consistent and thus easier to use.

8.5 Requirements and recommendations

This section condenses the previous material into a few guidelines you have to observe when you design a traits class yourself.

- Keep it small and simple. In particular, avoid redundant functionality in the traits class requirements. For example, if you require *Less_xy_2*, there is no reason to require *Greater_xy_2*, because the latter can be constructed from the former. In general, designing a good traits class requires a deep understanding of the algorithm it is made for. Finding the "right" set of geometric primitives required by the algorithm can be a nontrivial task. However, spending effort on that task decreases the effort needed later to implement traits classes and increases the ease of use of the algorithm.
- Obey the naming conventions (Section 6.1).
- Use functors instead of member functions for the predicates required. This is not only necessary for the kernel traits, it also gives the benefit of more flexibility. For each type you must provide a member function to get the actual functor and thus this seems to increase the size of the traits class. However, if you follow the naming scheme, the signatures of these functions are obvious and obtainable mechanically.
- Provide at least one model (which should normally be the kernel traits class) for every traits concept.
- Define and document a default traits class so the user need not provide a traits class argument if customization of the algorithm is not needed.

Chapter 9

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 9.1), the usage of checks (Section 9.2), and a more selective means of controlling checks (Section 9.3). Finally, a statement about exception handling is given (Section 9.4).

9.1 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.

9.2 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) .

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.

9.3 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.

For a new package you will first have to create a suitable header file with all macro definitions. This is done with the shell script `create_assertions.sh` (Section 5.1). The following command will create a file `optimisation_assertions.h`.

```
sh create_assertions.sh optimisation
```

You should place the generated file in the proper directory (and possibly rename it). Then you can use the checks in the following fashion.

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

void optimisation_foo( int i)
{
   CGAL_optimisation_precondition_msg( i == 42, "Only 42 allowed!");
   // ...
}
```

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.

9.4 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.

9.5 Requirements and recommendations

Requirements:

- Write pre- and postcondition checkers for your functions wherever possible.
- Use the CGAL preprocessor macros (Sections 9.2 and 9.3) exclusively throughout your code (instead of, for example, the *assert* macro) for all checks to assure that all CGAL invariant tests can be handled in a uniform way.

Chapter 10

Reference Counting and Handle Types

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

10.1 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 10.1.

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.

10.2 Handle & Rep

The two classes *Handle* and *Rep* provide reference counting functionality; see Figure 10.2. 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

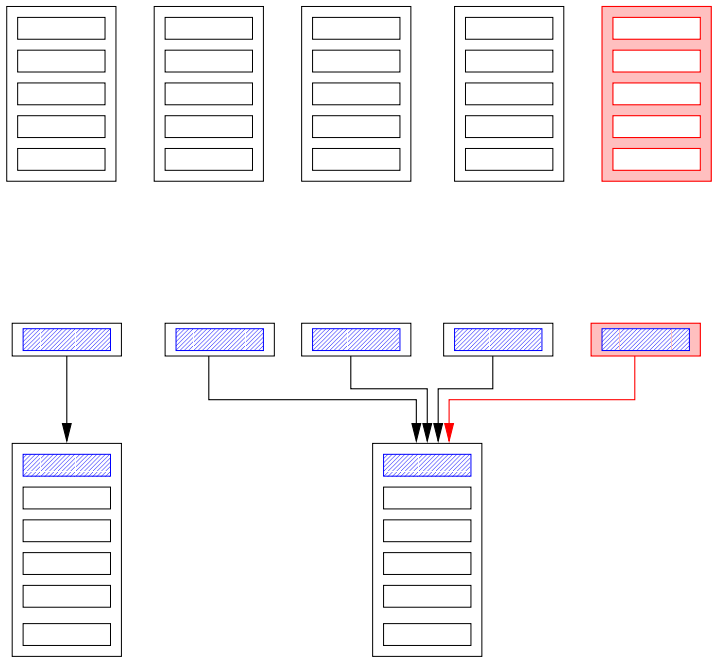


Figure 10.1: Objects using reference counting (bottom) share common representation; copying creates a new handle (drawn at the right) pointing to the same representation as the object copied. Without reference counting (top) all data are copied to the new object (drawn at the right);

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;
```

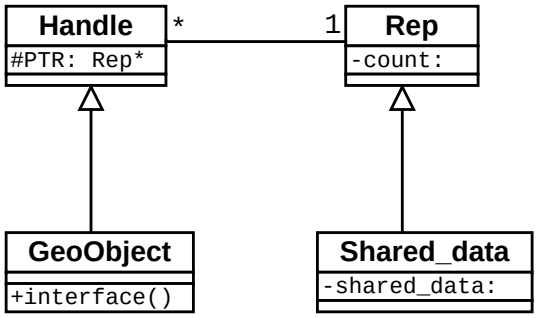


Figure 10.2: UML class diagram for Handle & Rep scheme.

10.3 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.

10.4 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 10.3. The reference counting functionality and the required interface can be inherited from class *Ref_counted*.

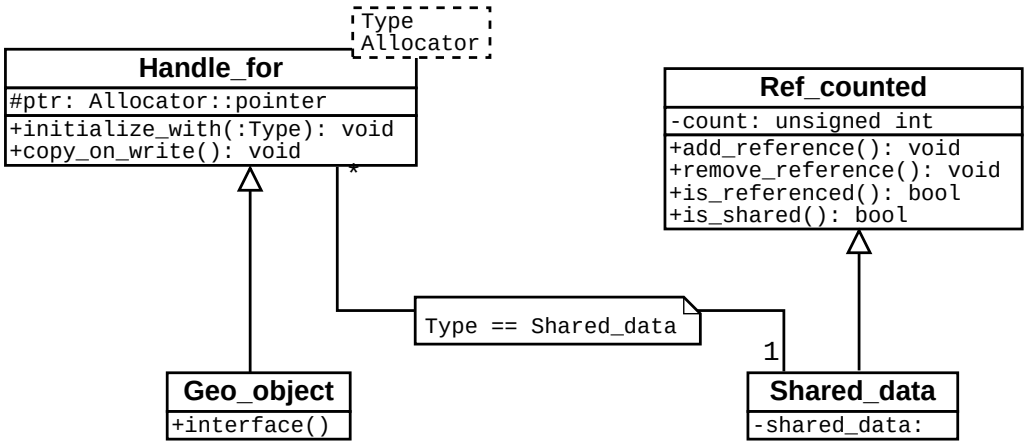


Figure 10.3: UML diagram for templated handles.

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

10.5 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*>

10.6 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 11 for more information.

Chapter 11

Memory Management

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

One of the design goals of CGAL (Section 1.2) 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.

11.1 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.

11.2 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>
```

11.3 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;
}
```

11.4 Requirements and recommendations

Recommendations:

- Use an allocator template parameter (which defaults to *CGAL_ALLOCATOR*) for data structures for which an optimization with regard to allocation and deallocation is beneficial.

Chapter 12

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*.

12.1 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.

12.2 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)
```

12.3 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.)

12.4 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.

12.4.1 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. CGAL checks for Koenig-lookup. If this is not available, the flag *CGAL_CFG_NO_KOENIG_LOOKUP* is set. As a workaround, you can add matching function(s) with the same name in the using namespace; most likely, this will be the global scope. Here is an example:

```
namespace A {

class Cls {};

bool
foo(const Cls&)
{ return true; }

} // namespace A

bool
foo(int)
{ return false; }

int
main()
{
  A::Cls c;
  foo( c );   // Koenig lookup finds A::foo(const Cls&)
  return 0;
}
```

While `gcc` doesn't have problems with correctly compiling the program above, the MipsPro compiler on IRIX (up till version 7.30) does not implement Koenig lookup, so it fails:

```
bash-2.03$ CC -64 kl.C
cc-1387 CC: ERROR File = kl.C, Line = 19
  No suitable conversion function from "A::Cls" to "int" exists.

    foo( c );
         ^

1 error detected in the compilation of "kl.C".
```

Workaround for missing Koenig lookup: Define

```
bool
foo( const A::Cls& cls)
{ return A::foo( cls); }
```

in global scope, too.

12.4.2 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.C
cc-1282 CC: ERROR File = poi.C, 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.C".
```

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.

12.5 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.

12.5.1 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

  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

[4] `http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_toc.html`
[5] The use of the macro eases future changes in our policy.

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*.

12.6 Requirements and recommendations

Requirements:

- all names defined by CGAL are in namespace *CGAL* (including namespaces nested in namespace *CGAL*).
- qualify calls of *is_zero*, *is_one*, *is_negative*, *is_positive*, *sign*, *abs*, *compare*, *square* by `CGAL_NTS`.
- use `CGAL_CLIB_STD` with C-library functions.

Chapter 13

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 13.1.

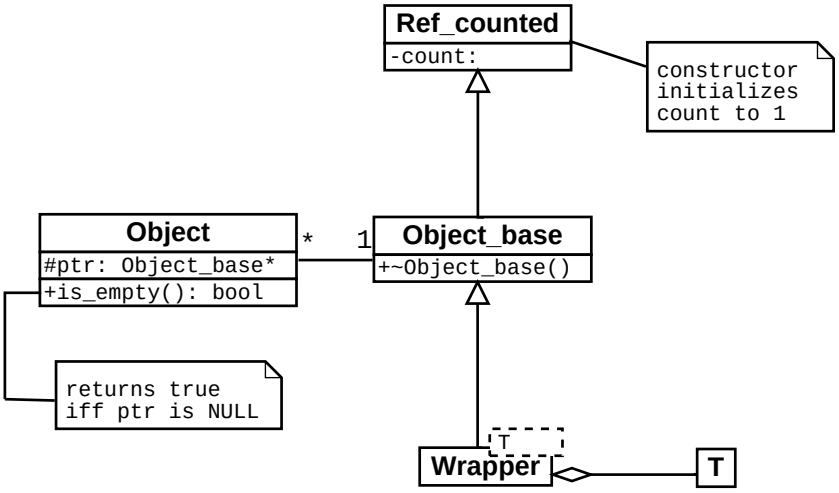


Figure 13.1: UML class diagram for faked object hierarchies (since 2.2-I-4).

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 14

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 14.3.1 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.

14.1 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;
};
```

14.2 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 ));
```

14.3 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.

14.3.1 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.

14.3.2 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.

14.3.3 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 6.1 for more details about the naming conventions for iterators and their access functions.

14.3.4 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 6.1).

14.4 Requirements and recommendations

Requirements:

- All container classes should provide iterator types and access functions.
- All container classes that represent circular data structures should provide circulator types and access functions.
- Take care that decrement of the past-the-end value is, in accordance with the standard, a legal operation for a bidirectionnal iterator. This can, for example, be used to get the last element of a sequence.

Recommendations:

- Be aware that postincrement (respectively, postdecrement) is more expensive than preincrement (predecrement) since the iterator or circulator value must be copied in the former case.
- Remember that iterators and circulators are intended to be lightweight objects. That is, copying them should require only constant time.
- When writing a container-like structure, provide *push_back*, *push_front*, and *insert* member functions so all insert iterators can be used with your container.

Chapter 15

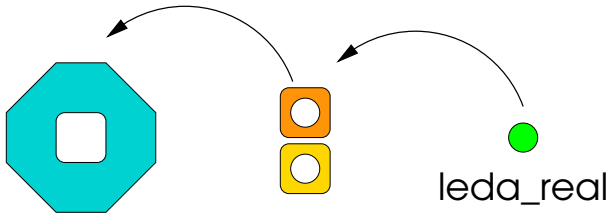
Robustness Issues

*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.



15.1 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.

15.2 Requirements and recommendations

Recommendations:

- Encapsulate basic arithmetic in predicates and constructions.
- Use kernel primitives whenever possible. This allows you to use the kernel as a traits class for your algorithm or data structure.
- If no appropriate kernel primitives are available, have a look at Chapter 7 on how to proceed.

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

Chapter 16

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 16.1)
- What version of CGAL am I running? (Section 16.3)
- Which compiler is this? (Section 16.5)
- Does the compiler support Koenig lookup? (Section 16.6.1)

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 16.6.1 to get an idea of which "features" are still present.

16.1 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.

16.2 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` ?)

16.3 Identifying CGAL and LEDA versions

Every release of CGAL defines (in <CGAL/version.h>) two macros:

CGAL_VERSION – a textual description of the current release (*e.g.*, or 1.2 or 2.2-I-1 or 3.2.1-I-15), and

CGAL_VERSION_NR , – a numerical description of the current release such that more recent releases have higher number.

More precisely, it is defined as 1MMmmbiiii, where MM is the major release number (e.g. 03), mm is the minor release number (e.g. 02), b is the bug-fix release number (e.g. 0), and iiii is the internal release number (e.g. 0001). For public releases, the latter is defined as 1000. Examples: for the public release 3.2.4 this number is 1030241000; for internal release 3.2-I-1, it is 1030200001. Note that the scheme was modified around 3.2-I-30.

LEDA defines a macro __LEDA__ in <LEDA/basic.h> that provides a numerical description of the current LEDA version. Examples: for LEDA-3.8, it is 380, for LEDA-4.1 it is 410.

16.4 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
```

16.5 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 2.95	`__GNUC__`	2
GNU 2.95	`__GNUC_MINOR__`	95
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`

16.6 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.

16.6.1 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 16.5 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.

Which compilers passed which tests can be determined by looking at the test suite results page (`http://cgal.inria.fr/CGAL/Members/testsuite/`) for the package `Configuration`.

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 `Configuration/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_AUTOMATIC_TEMPLATE_INCLUSION

When template implementation files are not included in the source files, a compiler may attempt to find the unincluded template bodies automatically. For example, suppose that the following conditions are all true:

- template entity `ABC::f` is declared in file `xyz.h`
- an instantiation of `ABC::f` is required in a compilation
- no definition of `ABC::f` appears in the source code processed by the compilation

In this case, the compiler may look to see if the source file `xyz.n` exists, where `n` is `.c`, `.C`, `.cpp`, `.CPP`, `.cxx`, `.CXX`, or `.cc`. The flag is set if this feature is missing.

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.

16.6.2 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.

16.6.3 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);
```

16.7 Requirements and recommendations

Recommendations:

- Workarounds for a compiler bug or a missing feature should not be treated on a per-compiler basis. When you detect a deficiency, you should rather write a short test program that triggers the setting of a flag for this deficiency during configuration.
- Avoid classes having friend functions and member functions with the same name. `g++ 2.95.*` does not like that. This holds also for operators, especially *operator-()*.

Chapter 17

Testing

*Matthias Bäsken (*baesken@infsn2.informatik.uni-halle.de*)*

The CGAL test suite is a way to test the compilation and execution of CGAL programs automatically (*i.e.*, without user interaction) on a number of different platforms. Developers should, of course, thoroughly test their code on their own development platform(s) **before** submitting it. The test suite serves as a way to test on additional platforms not available to the developer.

17.1 What a test suite for a package should contain

The test suite helps the developer(s) of a package to

- detect compilation problems on the various platforms
- detect runtime problems
- check the correctness of the algorithms in the package

That does not mean that the test suite is a platform for initial testing of code. New code should be tested on different platforms by the developer before submission.

It is strongly recommended for a test suite of a package to

- Cover the complete code of the package; every (member) function should be called at least once. (See Section 17.2 for a description of a tool you can use to test for code coverage.)
- Use more than one instantiation of templated functions or classes.
- A lot of classes in CGAL can be parametrized by traits classes, so that they are usable with different kernels. In such cases more than one kernel should be used for testing.
- Use pre- and postcondition checkers wherever it is possible.

17.2 Using the code coverage tool gcov

The tool gcov can be used together with the GNU C++ compiler to test for code coverage in your programs and may be helpful when you create your CGAL test suite programs. You can find a complete guide to this tool in the GNU on-line documentation at http://gcc.gnu.org/onlinedocs/gcc-2.95.2/gcc_6.html. If you want to use the code coverage tool gcov, you have to compile your programs with the options -fprofile-arcs and -ftest-coverage. Here is a simple example:

```
#include<iostream>

using namespace std;

void fu(int val)
{
 int w,v=0;
 if (val==0) {
  cout << "val == 0!\n";
  for(w=0;w<100;w++) v=v+w;
 }
 else {
  cout << "val != 0!\n";
  for(w=0;w<10;w++) v=v+w;
 }

 cout << "v:" << v << "\n";
}

int main()
{
  fu(0);
  return 0;
}
```

First you have to compile the example program test.C with the special options. Then you have to execute it, and, after this, gcov can be used.

```
g++ -fprofile-arcs -ftest-coverage -o test test.C
test
gcov test.C
```

gcov will create a file test.C.gcov containing output from gcov:

```
               #include<iostream>

               using namespace std;

               void fu(int val)
        1      {
        1       int w,v=0;
        1       if (val==0) {
```

```
        1       cout << "val == 0!\n";
        1       for(w=0;w<100;w++) v=v+w;
                }
   ######       else {
   ######        cout << "val != 0!\n";
   ######        for(w=0;w<10;w++) v=v+w;
               }

        1      cout << "v:" << v << "\n";
               }

               int main()
        1      {
        1        fu(0);
        1        return 0;
               }
```

The lines that were not executed will be marked with `######`, so you will see what should be added in the (CGAL) test suite programs.

17.3 Test suite directory

The test suite is located in the directory `test` of the internal releases of CGAL. This directory is not part of external releases. The directory `test` contains:

- a script `run_testsuite` that is (not surprisingly) used to run the test suite.
- a subdirectory for every package included in the internal release. These subdirectories are created from the `test` directories of the packages by copying the source, include, and input files from these directories and adding makefiles and `cgal_test` scripts where needed. See Section 4.1 for more information about the proper structure of the `test` directory for a package.
- a subdirectory with a name that ends in `_Examples` for every package that was submitted with an `examples` directory (Section 4.3)
- a subdirectory with a name that ends in `_Demo` for every package that was submitted with a `demo` directory (Section 4.4)

The test suite will attempt to compile all the programs in the subdirectories of `test` and to run all except the demo programs (which usually require user interaction) by using the `cgal_test` scripts (Sections 4.1 and 5.5) and will save the results in files in the package subdirectories (Section 17.6). Even if a program fails to compile or run, the test suite will continue.

17.4 Test suite input

Input to programs in the test suite can be supplied in three different ways:

data files in the `data` directory As described in Section 4.1, a package's `test` directory may contain a subdirectory `data` that contains input files for the test programs.

`*.cin` **files** If a test program `program.C` requires input from standard input (*i.e.*, `cin`), you should put a file called `program.cin` in the test directory. The test suite will then execute the program using the command

```
./program < program.cin
```

command-line arguments supplied in the `cgal_test` **script** *You are discouraged from using this option to give input values to your programs* since it requires you to edit and submit a `cgal_test` script; see Section 5.5.

However, if a test program `program.C` absolutely requires command-line parameters, you should do the following. Use `create_cgal_test` to create the script `cgal_test`. This file contains an entry of the form

```
compile_and_run program
```

Just put the command-line parameters for `program` at the end of this line:

```
compile_and_run program  arg1 arg2 ..
```

The test suite will then execute the program using the command

```
./program <arg1> <arg2> ...
```

17.5 Running the test suite

The test suite is run using the `run_testsuite` script that is distributed with every internal release in the `test` directory. There are several ways you can customize this script to meet you needs:

- Add additional compiler and linker flags by setting the variables `TESTSUITE_CXXFLAGS` and `TESTSUITE_LDFLAGS` at the top of the script. These variables are prepended to `CXX_FLAGS` and `LDFLAGS`, respectively, in the test suite makefiles. So, for example, if you have a directory `experimental/include/CGAL` containing new or experimental CGAL files, you can do the following:

  ```
  TESTSUITE_CXXFLAGS="-Iexperimental/include"
  ```

  and in this way test with your new files without overwriting the originals.

- Export additional environment variables by ading lines to the `run_testsuite` script. As an example, it will be demonstrated how to export the `LD_LIBRARY_PATH` by editing `run_testsuite`.

  1. Add the line

     ```
     LD_LIBRARY_PATH=<your library path>
     ```

     to the script.

  2. Append `LD_LIBRARY_PATH` to the line

     ```
     export PLATFORM CGAL_MAKEFILE TESTSUITE_CXXFLAGS TESTSUITE_LDFLAGS
     ```

     in the script.

After this, the programs from the test suite will be run using the LD_LIBRARY_PATH that was specified in step 1.

- Run the test suite on more than one platform by adding a line at the bottom of the script of the form

```
run_testsuite <include makefile>
```

for every platform that you wish to test. Just substitute for <include makefile> the appropriate include makefiles that were generated during installation. (Don't forgot to use the full path name for the makefile!) By default, the last line in the file is

```
run_testsuite $CGAL_MAKEFILE
```

so you need not make any changes if you run the testsuite on only one platform and have set the CGAL_MAKEFILE environment variable properly.

After these steps you are ready to run the test suite. It can be run in two different ways:

```
./run_testsuite
```

The test suite will run the tests from all test directories. This may take a considerable amount of time.

```
./run_testsuite <dir1> <dir2> ...
```

The test suite will run only the test programs in the test directories

```
<dir1> <dir2> ...
```

To run an entire CGAL test suite automatically, including downloading of an internal release, configuration, and installation of the library, you can use the autotest_cgal script described in Section 5.6.

17.6 Files generated by the test suite

The testsuite will generate the following output files:

- <testdir>/ErrorOutput_<platform>

  This file contains two lines for every program that was tested on platform <platform> in the test directory <testdir>. The first line tells if the compilation was successful and the second line tells if the execution was successful (*i.e.*, the program returned the value 0). (See Section 4.1 for more details.)

- <testdir>/ProgramOutput.<program>.<platform>

  This file contains the console output from the test program <program.C> run on platform <platform>.

- <testdir>/CompilerOutput_<platform>

  This file contains the compiler output from platform <platform> for all programs.

- error.txt

  This is just a concatenation of all the ErrorOutput files that were generated during the last run of the test suite.

17.7 Test suite results

The results of test suites run on the various supported or soon-to-be-supported platforms are posted on the test suite results page (`http://cgal.inria.fr/CGAL/Members/testsuite/`).

17.8 Requirements and recommendations

Requirements:

- Test your code thoroughly **before** submitting it.
- Obey the directory structure detailed in Section 4.1
- Check the test suite results for your package regularly.

Recommendations:

- Write test suite programs that use more than one instantiation of templated functions and classes, call every member function at least once, and use more than one kernel.
- Use pre- and postcondition checkers.
- Use `gcov` to test your code for coverage.
- Don't submit a makefile for your test suite unless you need to do something very special to compile or link your program. If you find you want to do something very special in your makefile, think long and hard about whether it's really necessary or not.
- Don't submit the script `cgal_test` with your package.

Chapter 18

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 18.1), why and how to check your geometric predicates (Section 18.2), and what to do in order to evaluate handles and iterators during the debugging phase (Section 18.3).

18.1 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.

18.2 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.

18.3 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 7), 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.

18.4 Requirements and recommendations

Requirements:

- Test and debug your code **before** submitting it.

Recommendations:

- Don’t write buggy code `:-)`.

Chapter 19

Example and Demo Programs

The best way to illustrate the functionality provided by the library is through programs that users can compile, run, copy and modify to their hearts' content. Thus every package should contain some of these programs. In CGAL we distinguish between two types of programs: those that provided graphical output (demos) and those that do not (examples).

In this chapter we provide guidelines for the development of these programs and their inclusion in the documentation. See Sections 4.3 and 4.4 for a description of the directory structure required for example and demo programs, respectively. Note in particular that each directory should contain a `README` file that explains what the programs do and how one interacts with them.

19.1 Coding conventions

Remember that these programs are likely to be a user's first introduction to the library, so you should be careful to follow our coding conventions (Chapter 6) and good programming practice in these programs. In particular:

- Include a comment that gives the name of the file relative to the `CGALROOT` directory, such as:

```
//
// file : examples/Generator/random_polygon_ex.C
//
```

- Do **not** use the commands

```
using namespace CGAL;
using namespace std;
```

  We discourage the use of these as they introduce more names than are necessary and may lead to more conflicts than are necessary.

- As of release 2.3, you can include only the kernel include file (*e.g.*, `Cartesian.h` or `Homogeneous.h`) to get all kernel classes as well as the `basic.h` file. All example and demo programs should do this. For exmaple, you should have simply:

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

instead of:

```
#include <basic.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/Triangle_2.h>
#include <CGAL/Segment_3.h>
// etc.
```

- Types should be declared using the following syntax:

```
typedef CGAL::Cartesian<double>     Kernel;
typedef Kernel::Point_2             Point_2;
typedef Kernel::Triangle_3          Triangle_3;
```

instead of this syntax:

```
typedef CGAL::Cartesian<double>     Kernel;
typedef Point_2<Kernel>             Point_2;
typedef Triangle_3<Kernel>          Triangle_3;
```

Although both will work, the former is to be preferred since it reflects that the types are actually part of the kernel and also reflects the new (as of release 2.3) kernel design that allows types to be easily exchanged and adapted.

Note also that the typedef used above is

```
typedef CGAL::Cartesian<double>     Kernel;
```

instead of

```
typedef CGAL::Cartesian<double>     R; // for representation
```

This also reflects the new design, where the kernel classes are kernels containing objects and predicates not just representation classes for the objects.

19.2 The Programs

The following guidelines should be followed to the greatest extent possible when writing the example and demo programs.

- Provide simple programs with which a beginner can get started.
- Provide more involved programs that illustrate the power of the software.

- Provide programs that truly exercise the data structure. Though you may have some canned programs that work on fixed data sets to illustrate specific things, you should also have one (or more) programs that work on randomly generated or user-generated data. This illustrates confidence in the software (and can also build confidence by highlighting bugs).
- Take some care to design a good interface to the program; the packaging of a product does make a difference.

19.3 Including programs in documentation

All programs included in the documentation should be included in either the `examples` or the `demo` directory to ensure that:

(a) the programs will be included in the test suite and

(b) the user can more easily use the example program

The macro to do this inclusion is `\ccIncludeExampleCode`. See Chapter 2 and the documentation of the Manual tools for more details.

The reverse statement, that all programs in the `examples` and `demo` directory should be included in the documentation, is not necessarily true. Ideally the programs included in the documentation will be a proper (non-empty) subset of the programs included in the distribution.

19.4 Demo programs on the web

We maintain a page of demo programs on the CGAL web site (`http://www.cgal.org/demos.html`). For each demo on the page, we have a screen shot of the demo in action, a link to the source code, and a precompiled executable for the Windows platform.

If you have other demos you would like to add to the page (and the more here the better), you should do the following:

1. Compile the demo and create a screen shot (as a `.gif` or `.jpg` file) and then create a smaller version of this (approx 235×280) that can be used as a "thumbnail" (admittedly for a rather large thumb) on the page. These names of the image files should be as follows:
   <*program*>`_[big|small].[gif|jpg]` ,
   where <*prog_name*> is the name of the program from which the screen shots were made (an nice mnemonic name like *alpha_shapes_2* or *hull_of_int_points_2*).
2. Create the `.exe` file for use under Windows (probably with the Borland compiler with optimization flags turned on)
3. `gzip` the `.exe` file
4. Package these together with the source code using the following structure:

```
images/demo/<program>_big.[gif|jpg]
demo/<program>.exe
demo/<subdir>/<program>.C
```

where <*subdir*> is the name of the subdirectory in which the program is (or will be) found in the distribution. <*program*> should be a name that is self-explanatory (*e.g.*, `demo.C` is a bad name here).

5. Send the package to the web page maintainer together with a one-paragraph description similar to the ones on the page already.

19.5 Requirements and recommendations

Requirements:

- Follow the coding conventions outlined in Section 19.1.
- Include all example and demo programs from the documentation in the `examples` or `demo` directories.

Recommendations:

- Place a demo of your package on the web site.

Chapter 20

Submitting Packages

*Geert-Jan Giezeman (*geert@cs.uu.nl*)*
*Susan Hert (*hert@mpi-sb.mpg.de*)*

20.1 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 6);
- carefully and efficiently treat robustness issues (Chapter 15);
- 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.

The discussion of specific packages are logged on the Editorial Committee web page (http://www.cgal.org/Members/Editorial). Reading the feedback given on other packages can be quite instructive as a means of learning what the editorial committee is looking for.

One should write a specification for a new package (Chapter 2) 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 as detailed in Section 20.2 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	Bernd Gärtner
Efi Fogel	Michael Hoffmann
Lutz Kettner	Monique Teillaud
Remco Veltkamp	Mariette Yvinec
Sylvain Pion	Menelaos Karavelas
Ron Wein	

20.2 Electronic submission

Whether you produce library code, demos, documentation or something else, if you want it to become part of CGAL, you'll have to submit it in the form of a package which has to be a folder under SVN trunk. The directory structure required for a package is described in Chapter 4.

Here we focus on how to submit a package. This section is mostly obsolete since we have gotten rid of the mail-based system for package submission in 2004.

A package has a name, which identifies it. This name should obey the same rules as for C identifiers: it consists of letters, digits and underscores and it does not start with a digit. Choose a name that is descriptive, yet not too long (under 25 characters). If a package deals with objects of a particular dimension, then use the suffixes _2, _3, and _d, especially if there exists (or may exist in the future) a package with similar functionality in different dimensions. Examples of good package names are `Triangulation_2` for a package dealing with triangulations of points in the plane and `Min_ellipse_2`, which contains an algorithm that finds the minimal enclosing ellipse of a set of points in the plane. The package names `pm` and `arr` are a bit too terse. `Planar_map` and `Arrangement` (or `Arrangement_2`) are better.

20.3 When something goes wrong

There are several reasons why a submission may not succeed. In most cases the confirmation message will state that an error occured. In some cases, you will not get a confirmation message at all.

The following is a list of reasons why a submission might fail.

- The submitted file must obey the naming convention mentioned above, otherwise the submission is rejected.
- The submitted package must not contain any file whose name is the same as a file in some other submitted package, otherwise the submission is rejected.
- The package must contain a valid maintainer file.
- It may be impossible to get the package from the SVN server. This could be due to server failures, wrong permissions, misspellings, etc.
- Some resources may be exhausted at INRIA (disk space).

20.4 Requirements and recommendations

Requirements:

- Submit specifications to the editorial committee before submitting packages for internal releases.
- Obey the directory structure outlined in Chapter 4.

Recommendations:

- Wait for approval from the editorial committee before submitting packages for internal releases.

Chapter 21

Making Releases

21.1 Internal Releases

Internal releases are currently created 2 or 3 times a week from a script run at INRIA. This script packages together the current versions of all packages into a tar file and then sends a mail to `cgal-develop`. People responsible for running the test suite can pick it up automatically using the `autotest_cgal` script (Section 5.6).

21.2 Public Releases

A public release can be created from an internal release by following the steps detailed in the `README` file in the package `Release` on the SVN server. This file describes how to create the code, the documentation and the updated web pages for a new release.

Each public release is tagged on the SVN server with a tag in the following format `CGAL_N_release`, where `N` is the release number with all '.'s replaced by '_'s (*e.g.*, for release 3.0, the tag is `CGAL_3_0_release`. There is also a branch created for each release with the name `CGAL_N_branch`, where `N` is the release number as before. This facilitates the creation of bug fix releases.

Chapter 22

The CGAL Web Site

The web site for CGAL is `http://www.cgal.org`. It is currently hosted by the Max-Planck-Institut für Informatik (MPI) and is maintained by Andreas Meyer.

22.1 SVN cgal-web Project and Maintenance

Most files that are visible on the website are contained in a seperate SVN project called

cgal-web

Exceptions are large binaries and, for example, the Manual, which is generated (and large) and is thus not suitable to be placed in SVN. One can see the distinction in the URL's. Everything starting with

pub/
Members/pub/
Manual/

is not in SVN. If you want something changed or added here, you have to contact Andreas Meyer (or some other CGAL person at MPI). (For the manual an exception for the above exception are some HTML files, for example the frameset, that are actually available in the SVN package `Manual`.)

Several developers have write access to the new `cgal-web` project, or might get it whenever need arises. Note that the SVN project

is not automatically synchronized

with the actual web pages. It is a manual `svn up` call that has to be initiated at MPI. This is also intended as a final quality control before updates show up on our web pages. It allows us to have the set of people with write access to the SVN project not to be too restrictive.

How do you get your changes finally on the web pages of CGAL?

The SVN server actually informs the maintainer at MPI (see above) automatically by email about committed changes in the `cgal-web` project. So, either just commit small changes, the maintainer verifies them and integrates them in the next update of the web pages. If you plan something bigger, please let the maintainer know, so that updates can be suspended until you are done. Or, if you plan to have the updates done with a particular deadline, inform the maintainer well ahead, e.g., a week in advance, since:

- The maintainer does not necessarily read email all the time (maybe not even daily ;-).
- The maintainer might not consider your request as important as other tasks, especially since
- almost no request for CGAL's web site has the importance that needs 24/7 service, and most things can also wait a week.

If you plan updates that are not obviously consensus, you should discuss them beforehand on the usual mailing lists. The maintainer might block an update if its status is not clear.

22.2 Regular Tasks

Some aspects of the web pages need constant attention to keep them up-to-date. If you can please do the necessary updates yourself, notify the corresponding maintainer, or the maintainer mentioned above if you know about updates for the following:

- new publications related to CGAL, maintained by Monique Teillaud;
- updates for the work in progress page, which includes work that is finished and thus no longer in progress and new work.
- new demo programs to be included on the web site. (See Section 19.4 for the precise specification of how to submit these.)
- new people involved in the project
- new projects using CGAL by people other than developers.

And, of course, please tell the maintainer about any problems you have with the web site.

Chapter 23

Mailing Lists and Addresses

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

There are a number of mailing lists and email addresses that have been set up to communicate with various subsets of the CGAL community. The addresses that should be of interest to CGAL developers are as follows:

cgal-develop-l@postino.mpi-sb.mpg.de (a.k.a. cgal-develop@cgal.org) This list is intended for discussing detailed matters about implementation and development issues. All developers should subscribe to this list by sending a message to ameyer@mpi-sb.mpg.de. An archive of this mailing list is maintained at http://postino.mpi-sb.mpg.de/cgi-bin/lyris.pl?enter=cgal-develop-l . To enter this archive, use your email address and password (if you have one) or member1@cgal.org and the usual password.

cgal-commits@lists.gforge.inria.fr This list is read-only. It collects all automatic mails sent by the SVN server, one per commit, with the log messages and the URLS pointing to the corresponding diffs. Developers can subscribe to this list from CGAL mailing lists page. An archive of this mailing list is maintained at http://lists.gforge.inria.fr/cgi-bin/mailman/private/cgal-commits . To enter this archive, use your email address and InriaGForge password.

cgal-discuss-l@postino.mpi-sb.mpg.de Users of CGAL post questions and discuss issues related to CGAL on this list. All developers should subscribe to this list so they can answer questions relevant to their packages and monitor the input from the users provided here. To subscribe to the list, send a message to cgal-discuss-l-request@postino.mpi-sb.mpg.de with the content "subscribe cgal-discuss-l". An archive of this mailing list is maintained at http://postino.mpi-sb.mpg.de/cgi-bin/lyris.pl?enter=cgal-discuss-l .

cgal-announce-l@postino.mpi-sb.mpg.de Announcements of new releases, bug fixes, special courses, etc. are posted to this list. The list is moderated (meaning every message posted needs to be approved) to keep the traffic low. To subscribe to the list, send a message to cgal-announce-l-request@postino.mpi-sb.mpg.de with the content "subscribe cgal-announce-l". An archive of this mailing list is maintained at http://postino.mpi-sb.mpg.de/cgi-bin/lyris.pl?enter=cgal-announce-l .

cgal-editors-list@lists.gforge.inria.fr This is the email address to which you should send proposed design specifications (Chapter 2) for approval by the editorial board (Section 20.1). Anyone can do that at any time about any CGAL design issues.

This list is also archived at http://lists.gforge.inria.fr/cgi-bin/mailman/listinfo/cgal-editors-list, and members of the list should use their own email addresses and passwords (if you have one) to enter the archive.

This list was previously hosted at postino, where it was archived at `http://postino.mpi-sb.mpg.de/cgi-bin/lyris.pl?enter=cgal-editor-l`, and members of the list should use their own email addresses and passwords (if you have one) to enter the archive.

The mailing lists at `lists.gforge.inria.fr` are currently maintained by Sylvain Pion. The mailing lists at `postino.mpi-sb.mpg.de` are currently maintained by Andreas Meyer.

Chapter 24

Sources of Information

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

24.1 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.

24.2 Web documents and pages

Here we provide a list of the documents and web pages that contain information you might find useful. Most of these documents are protected by a password, which you can obtain by asking one of the other developers.

- The CGAL home page — http://www.cgal.org
- Manuals of the latest public release — http://www.cgal.org/Manual/doc_html
- The CGAL developers' pages — http://www.cgal.org/Members/
- Manuals of the latest internal release — http://www.cgal.org/Members/Manual_test
- Links to Local CGAL pages — http://www.cgal.org/Members/local_links.html
- The Manual Tools — http://www.cgal.org/Members/Manual_tools

- Browse SVN Repository — https://gforge.inria.fr/plugins/scmsvn/viewcvs.php/?root=cgal
- Documentation of CGAL SVN hosted by InriaGForge — http://www.cgal.org/Members/svn_cgal/
- Internal release page — http://cgal.inria.fr/CGAL/Members/Releases/
- Test suite results — http://cgal.inria.fr/CGAL/Members/testsuite/
- Archive of cgal-develop — http://postino.mpi-sb.mpg.de/cgi-bin/lyris.pl?enter=cgal-develop-l
- Archive of cgal-discuss-l — http://postino.mpi-sb.mpg.de/cgi-bin/lyris.pl?enter=cgal-discuss-l

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**.