

# **CGAL Reference Manual**

## Part 0: General Introduction

Release 1.0, April 1998



# Preface

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

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

## Editors

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

## Acknowledgement

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



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Organization . . . . .	1
1.2	Naming conventions . . . . .	1
<b>2</b>	<b>Checks</b>	<b>3</b>
2.1	Altering the failure behaviour . . . . .	4
2.2	Control at a finer granularity . . . . .	4
2.3	Customising how errors are reported . . . . .	4
<b>3</b>	<b>Operators for IO Streams</b>	<b>7</b>
3.1	Output Streams (ostream) . . . . .	8
3.2	Input Streams (istream) . . . . .	9
<b>4</b>	<b>Colors</b>	<b>11</b>
4.1	Colors (CGAL_Color) . . . . .	11
<b>5</b>	<b>Window Stream</b>	<b>13</b>
5.1	Window Stream (CGAL_Window_stream) . . . . .	13



# Chapter 1

## Introduction

CGAL, the *Computational Geometry Algorithms Library*, is written in C++ and consists of three parts. The first part is the kernel, which consists of constant size non-modifiable geometric primitive objects and operations on these objects. The objects are parameterized by a representation class, which specifies the coordinate representation and the underlying number types used for calculations. The second part is a collection of basic geometric data structures and algorithms, which are parameterized by traits classes that define the interface between the data structure or algorithm, and the primitives they use. The third part consists of non-geometric support facilities, such as ‘circulators’, random sources, I/O support for debugging and for interfacing CGAL to various visualization tools.

### 1.1 Organization

This part of the Reference Manual gives information that is relevant to all three parts of the library. Part 1 of this Reference Manual covers the kernel, part 2 the basic library, and part 3 the support library.

Additional documents accompanying the CGAL distribution are ‘The Use of STL and STL Extensions in CGAL’, and ‘Getting Started with CGAL’. The former document gives a manual style introduction to STL constructs such as iterators and containers, as well an extension, called circulator, used in many places in CGAL. The latter document gives an introduction in CGAL programming for the novice user.

Some functionality is considered more advanced. Such functionality is described in sections like this one, bounded by horizontal brackets.

### 1.2 Naming conventions

The use of representation classes not only avoids problems, it also makes all CGAL classes very uniform. They **always** consist of:

1. The *name space prefix* *CGAL\_*. This avoids name clashes. It will be dropped as soon as C++ compilers support the concept of name spaces as a feature of the programming language.
2. The *capitalized base name* of the geometric object, such as *Point*, *Segment*, *Triangle*.
3. An *underscore* followed by the *dimension* of the object, for example *\_2*, *\_3* or *\_d*.
4. A *representation class* as parameter, which itself is parameterized with a number type, such as *CGAL\_Cartesian<double>* or *CGAL\_Homogeneous<leda\_integer>*.



# Chapter 2

## Checks

Much of the CGAL code contains checks. For example, all checks used in the kernel code are prefixed by *CGAL\_KERNEL*. Other packages have their own prefixes, as documented in the corresponding chapters. Some are there to check if the kernel behaves correctly, others are there to check if the user calls kernel routines in an acceptable manner.

There are four types of checks. The first three are errors and lead to a halt of the program if they fail. The last only leads to a warning.

**Preconditions** check if the caller of a routine has called it in a proper fashion. If such a check fails it is the responsibility of the caller (usually the user of the library).

**Postconditions** check if a routine does what it promises to do. If such a check fails it is the fault of this routine, so of the library.

**Assertions** are other checks that do not fit in the above two categories.

**Warnings** are checks for which it is not so severe if they fail.

By default, all of these checks are performed. It is however possible to turn them off through the use of compile time switches. For example, for the checks in the kernel code these are *CGAL\_KERNEL\_NO\_PRECONDITIONS*, *CGAL\_KERNEL\_NO\_POSTCONDITIONS*, *CGAL\_KERNEL\_NO\_ASSERTIONS* and *CGAL\_KERNEL\_NO\_WARNINGS*. So, in order to compile the file `foo.C` with the postcondition checks off, you should do:  
`CC -DCGAL_KERNEL_NO_POSTCONDITIONS foo.C`

Not all checks are on by default. All four types of checks can be marked as expensive or exactness checks (or both). These checks need to be turned on explicitly by supplying one or both of the compile time switches *CGAL\_KERNEL\_CHECK\_EXPENSIVE* and *CGAL\_KERNEL\_CHECK\_EXACTNESS*.

Expensive checks are, as the word says, checks that take a considerable time to compute. Considerable is an imprecise phrase. Checks that add less than 10 percent to the execution time of the routine they are in are not expensive. Checks that can double the execution time are. Somewhere in between lies the border line. Exactness checks are checks that 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 doubles as number type, this may not be the case, due to round off errors. So, exactness checks should only be turned on if the computation is done with some exact number type.

## 2.1 Altering the failure behaviour

As stated above, if a postcondition, precondition or assertion is violated, the program will abort (stop and produce a core dump). This behaviour can be changed by means of the following function.

```
#include <CGAL/assertions.h>
CGAL_Failure_behaviour    CGAL_set_error_behaviour( CGAL_Failure_behaviour eb)
```

The parameter should have one of the following values.

```
enum CGAL_Failure_behaviour { CGAL_ABORT, CGAL_EXIT, CGAL_CONTINUE};
```

The first value is the default. If the *CGAL\_EXIT* value is set, the program will stop, but not dump the core. The last value tells the checks to go on after diagnosing the error. The value that is returned is the value that was in use before.

For warnings there is a separate routine, which works in the same way. The only difference is that for warnings the default value is *CGAL\_CONTINUE*.

```
CGAL_Failure_behaviour    CGAL_set_warning_behaviour( CGAL_Failure_behaviour eb)
```

## 2.2 Control at a finer granularity

The compile time flags as described up to now all operate on the whole library. Sometimes you may want to have a finer control. CGAL offers the possibility to turn checks on and off with a bit finer granularity, namely the module in which the routines are defined. The name of the module is to be appended directly after the CGAL prefix. So, the flag *CGAL\_KERNEL\_NO\_ASSERTIONS* switches off assertions in the kernel only, the flag *CGAL\_CH\_CHECK\_EXPENSIVE* turns on expensive checks in the convex hull module. The name of a particular module is documented with that module.

---

## 2.3 Customising how errors are reported

Normally, error messages are written to the standard error output. It is possible to do something different with them. To that end you can register your own handler. This function should be declared as follows.

```
void                                my_failure_function(
                                   const char *type,
                                   const char *expression,
                                   const char *file,
                                   int line,
                                   const char *explanation)
```

Your failure function will be called with the following parameters. *type* is a string that contains one of the words precondition, postcondition, assertion or warning. The parameter *expression* contains the expression that was violated. *file* and *line* contain the place where the check was made. The

*explanation* parameter contains an explanation of what was checked. It can be *NULL*, in which case the *expression* is thought to be descriptive enough.

There are several things that you can do with your own handler. You can display a diagnostic message in a different way, for instance in a pop up window or to a log file (or a combination). You can also implement a different policy on what to do after an error. For instance, you can throw an exception or ask the user in a dialog whether to abort or to continue. If you do this, it is best to set the error behaviour to *CGAL\_CONTINUE*, so that it does not interfere with your policy.

You can register two handlers, one for warnings and one for errors. Of course, you can use the same function for both if you want. When you set a handler, the previous handler is returned, so you can restore it if you want.


```
#include <CGAL/assertions.h>
CGAL_Failure_function    CGAL_set_error_handler( CGAL_Failure_function handler)
CGAL_Failure_function    CGAL_set_warning_handler( CGAL_Failure_function handler)
```

### Example

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

void my_failure_handler(
    const char *type,
    const char *expr,
    const char* file,
    int line,
    const char* msg)
{
    /* report the error in some way. */
}

void foo()
{
    CGAL_Failure_function prev;
    prev = CGAL_set_error_handler(my_failure_handler);
    /* call some routines. */
    CGAL_set_error_handler(prev);
}
```





## Chapter 3

# Operators for IO Streams

All classes in the CGAL kernel provide input and output operators for IOStreams. The basic task of such an operator is to produce a representation of an object that can be written as a sequence of characters on devices as a console, a file, or a pipe. In CGAL we distinguish between a raw ASCII, a raw binary and a pretty printing format.

*CGAL\_IO::Mode* = { *ASCII* = 0, *BINARY*, *PRETTY* };

The first one just writes number, e.g. the coordinates of a point or the coefficients of a line in a machine independent format. The second one writes the data in a binary format, e.g. a double is represented as a sequence of four byte. The format depends on the machine. The last one serves mainly for debugging as the type of the geometric object is written, as well as the data defining the object. For example for a point at the origin with Cartesian double coordinates, the output would be *CGAL\_PointC2(0.0, 0.0)*. At the moment CGAL does not provide input operations for pretty printed data. By default a stream is in ASCII mode.

CGAL provides the following functions to modify the mode of an IO stream.

*CGAL\_IO::Mode*    *CGAL\_set\_mode( ios& s, CGAL\_IO::Mode m)*

*CGAL\_IO::Mode*    *CGAL\_set\_ascii\_mode( ios& s)*

*CGAL\_IO::Mode*    *CGAL\_set\_binary\_mode( ios& s)*

*CGAL\_IO::Mode*    *CGAL\_set\_pretty\_mode( ios& s)*

The following functions allow to test whether a stream is in a certain mode.

*CGAL\_IO::Mode*    *CGAL\_get\_mode( ios& s)*

*bool*                *CGAL\_is\_ascii( ios& s)*

*bool*                *CGAL\_is\_binary( ios& s)*

*bool*                *CGAL\_is\_pretty( ios& s)*

## 3.1 Output Streams (*ostream*)

### Definition

CGAL defines output operators for classes that are derived from the class *ostream*. This allows to write to ostream as *cout* or *cerr*, as well as to strstreams and fstreams. Let *os* be an output stream.

### Operations

The output operator is defined for all classes in the CGAL kernel.

*ostream*&                    *os* << *CGAL\_Class* *c*                    Inserts object *c* in the stream.

### Example

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

#include <iostream.h>
#include <fstream.h>

#include <CGAL/Segment_2.h>

typedef CGAL_Point_2< CGAL_Cartesian<double> >      Point;
typedef CGAL_Segment_2< CGAL_Cartesian<double> >    Segment;

int main()
{
    Point p(0,1), q(2,2);
    Segment s(p,q);

    CGAL_set_pretty_mode(cout);
    cout << p << endl << q << endl;

    ofstream f("data.txt");
    CGAL_set_binary_mode(f);
    f << s << p ;

    return 1;
}
```

## 3.2 Input Streams (*istream*)

### Definition

CGAL defines input operators for classes that are derived from the class *istream*. This allows to read from istreams as *cin*, as well as from strstreams and fstreams. Let *is* be an input stream.

### Operations

The input operator is defined for all classes in the CGAL kernel.

*istream*&                    *is* >> *CGAL\_Class c*                    Extracts object *c* from the stream.

### Example

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

#include <iostream.h>
#include <fstream.h>

#include <CGAL/Segment_2.h>

typedef CGAL_Point_2< CGAL_Cartesian<double> >      Point;
typedef CGAL_Segment_2< CGAL_Cartesian<double> >      Segment;

int
main()
{
    Point p, q;
    Segment s;

    CGAL_set_ascii_mode(cin);
    cout >> p >> q;

    ofstream f("data.txt");
    CGAL_set_binary_mode(f);
    f >> s >> p;

    return 1;
}
```





# Colors

### Definition

The following symbolic constants are predefined: *CGAL\_BLACK*, *CGAL\_WHITE*, *CGAL\_RED*, *CGAL\_GREEN*, *CGAL\_BLUE*, *CGAL\_VIOLET* and *CGAL\_ORANGE*.

## Creation

*CGAL\_Color* *c*( *int red*, *int green*, *int blue*); creates a color with rgb-value (*red,green,blue*).

<i>bool</i>	<i>c</i> == <i>q</i>	Test for equality: Two colors are equal, iff their rgb-values are equal.
<i>bool</i>	<i>c</i> != <i>q</i>	Test for inequality.
<i>int</i>	<i>c.red()</i>	returns the red component of <i>c</i> .
<i>int</i>	<i>c.green()</i>	returns the green component of <i>c</i> .
<i>int</i>	<i>c.blue()</i>	returns the blue component of <i>c</i> .



# Chapter 5

## Window Stream

### 5.1 Window Stream (*CGAL\_Window\_stream*)

#### Definition

An object of type *CGAL\_Window\_stream* is a two-dimensional window for graphical IO. The input and output operations performs a mapping from object coordinates to window coordinates. The class *CGAL\_Window\_stream* provides input and output operators for the classes which are defined in the CGAL kernel.

The class *CGAL\_Window\_stream* is derived from the LEDA class *window* and inherits its rich functionality.

```
#include <CGAL/IO/Window_stream.h>
```

#### Creation

```
CGAL_Window_stream W ( int width, int height );
```

creates a window *W* of *width*  $\times$  *height* pixels.

#### Operations

```
void W.init( CGAL_Bbox_2 b = CGAL_Bbox_2(0, 0, 1, 1))
```

Defines the part of the object space that is mapped to the physical window.

```
void W.init( double xmin, double xmax, double ymin)
```

Defines the part of the object space that is mapped to the physical window.

## Output Operators

The output operator is defined for all geometric classes in the CGAL kernel.

```
CGAL_Window_stream& CGAL_Window_stream &W << CGAL_Class c
```

inserts object *c* in the stream *W*.

```
CGAL_Window_stream& CGAL_Window_stream &W << CGAL_Color c
```

changes the foreground color for the next objects that will be inserted in *W*.

## Input Operators

The input operator is defined for all geometric classes in the CGAL kernel. The coordinates of the geometric objects are in object space. Data are entered with the left mouse button.

```
CGAL_Window_stream& CGAL_Window_stream &W >> CGAL_Class &c
```

extracts object *c* from the stream *W*.

## Example

It is important that the window stream header file gets included *after* the inclusion of the header files of geometric classes that get inserted to or extracted from the window stream.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Segment_2.h>
#include <CGAL/IO/Window_stream.h>

typedef CGAL_Point_2< CGAL_Cartesian<double> >    Point;
typedef CGAL_Segment_2< CGAL_Cartesian<double> >    Segment;

int main()
{
    Point p(0,1), q(2,2);
    Segment s(p,q);

    CGAL_Window_stream W(100,100);
    W.init(0,10,10);

    W << CGAL_RED << s << CGAL_BLACK << p << q ;
    W >> s >> p;
    return 1;
}
```