Getting Started

with

CGAL

Release 2.1, December 1999

Preface

CGAL is the Computational Geometry Algorithms Library, written in C++. It is developped by a consortium of seven sites: Utrecht University (The Netherlands), ETH Zürich (Switzerland), Free University of Berlin (Germany), INRIA Sophia-Antipolis (France), Martin-Luther-Universität Halle-Wittenberg (Germany), Max-Planck Institute for Computer Science, Saarbrücken (Germany), RISC Linz (Austria) and Tel-Aviv University (Israel). More information about the project can be found on the CGAL home page at URL `http://www.cs.uu.nl/CGAL/`.

This document is acompanied with a number of example source files. The example files mentioned in the text refer to these source files. They can be found in the CGAL distribution in the directory *examples/Getting_started*.

Authors

Geert-Jan Giezeman, Remco Veltkamp, Wieger Wesselink
Department of Computer Science
Utrecht University, The Netherlands

Acknowledgement

This work is supported by the Esprit IV Project No. 21957 (CGAL) and by the Esprit IV Project No. 28155 (GALIA).

Contents

Chapter 1

Introduction

The CGAL library is a C++ library that contains primitives, data structures, and algorithms for computational geometry. The goal of this document is to teach you how to use the CGAL library. It contains information about how to use primitives, datastructures, and algorithms, and contains also example programs. This document should be used together with the CGAL Reference Manual. For downloading and installing CGAL, see the CGAL Installation Guide.

This chapter gives a short overview of the CGAL library. Besides mentioning the purpose and the intended users, this chapter will give you some background information about the project behind CGAL. Chapters 2 to 8 introduce several aspects of the CGAL library.

This document was written with the assumption that you are familiar with the C++ language. To assist the C-programmer, we have tried to explain some typical C++ features in Appendix A. Some C++ notions are explained whenever they are encountered. This is done to such an extent that it should be possible to use the library without relying on a C++ textbook. To really learn the C++ language we recommend an elementary book, for example [Lippman 98].

1.1 Overview of CGAL

Geometric algorithms are used in many application domains. People in areas like computer graphics, robotics, geographic information systems and computer vision are more and more realizing that concepts and algorithms from computational geometry can be of importance for their work. However, implementing these algorithms isn't easy. As a result, many useful geometric algorithms haven't found their way into practice yet. The most common problems are the dissimilarity between fast floating-point arithmetic normally used in practice and exact arithmetic over the real numbers assumed in theoretical papers, the lack of explicit handling of degenerate cases in these papers, and the inherent complexity of many efficient solutions. Therefore, the computational geometry community itself has started to develop a well-designed library: CGAL, the Computational Geometry Algorithms Library. This library is developed by seven institutions: Utrecht University (The Netherlands), ETH Zurich (Switzerland), Free University Berlin (Germany), INRIA Sophia-Antipolis (France), Max Planck Institute Saarbrucken (Germany), RISC Linz (Austria), and Tel Aviv University (Israel).

The CGAL library contains a number of different parts. The elementary part of the library (the kernel) consists of primitive, constant-size geometric objects (points, lines, spheres, etc.) and predicates on them (orientation test for points, intersection tests, etc.). The next part of the library contains a number of standard geometric algorithms and data structures such as convex hull, smallest enclosing circle, and triangulation. The last part of the library consists of a support library for example for I/O, visualization, and random generators. Currently, the library contains mainly 2 and 3-dimensional objects, but in the future there will also be support for objects of arbitrary dimension.

CGAL is developed for different groups of users. There are the researchers working in computational geometry itself who want to use the library to more easily implement and test their own algorithms. There are researchers with knowledge of computational geometry who want to use geometric algorithms in application research areas. There are developers working in other research areas and companies who want to use CGAL in, possibly commercial, applications. All these groups of users have rather different demands. To please all of them, the CGAL library has to fulfill a number of design goals. The most important of these are robustness, generality, efficiency and ease of use. Of course it isn't easy to combine these in one library. Below we will describe briefly what has been done to achieve these goals.

1.2 Robustness

Especially in the field of computational geometry, robustness of a software library is of vital importance. In geometric algorithms many decisions are based on geometric predicates. If these predicates are not computed exactly (for example due to round-off errors), the algorithm may easily give incorrect results. For some algorithms, strategies exist to deal with inexact predicates. However, in general this is very difficult to achieve. Therefore, in CGAL we use the following rule: a correct result of an algorithm can only be guaranteed if geometric predicates are evaluated exactly. The most natural way of obtaining exact predicates is to choose an appropriate number type for doing computations. As a consequence of this, in CGAL there is a strong emphasis on the specification of algorithms. It should always be clear for which inputs and for which number types a correct result is guaranteed. Of course the user is always free to use fast but imprecise number types like floats or doubles. This should not cause the algorithm to break down, although it could occasionally lead to incorrect results. The above discussion should be seen separate from dealing with degenerate cases.

1.3 Generality

The applications of the CGAL library will be very heterogeneous, with very different requirements. To make the library as general as possible, C++ templates (parameterized data types and functions) are heavily used. This enables the user to choose an appropriate number type for doing computations. For example, if speed is important, computations can be done with floats or doubles. On the other hand, if reliability is more important, computations can be done with arbitrary precision rational numbers. Furthermore, the user can choose the representation type of points (i.e. Cartesian or homogeneous coordinates). And to some extent it is even possible to replace a CGAL data type with a user defined one (see Chapter 8).

1.4 Efficiency

A computational geometry library must be efficient to be really useful. Whenever possible the most efficient version of an algorithm is used. Clearly, a library algorithm cannot be the best solution for every application. Therefore, sometimes multiple versions of an algorithm are supplied. For example, this will be the case if dealing with degenerate cases is expensive, or when for a specific number type a more efficient algorithm exists (in which case it will be implemented as a C++ template specialization). Another (C++ level) decision that has been made in favor of efficiency is that geometric objects do not share a common base class with virtual methods. However, this can be simulated through the use of `CGAL_Object`.

1.5 Ease of use

Generality and ease of use are not always easy to combine. The abundant use of templates seems to make the library difficult to use for people who just want to do something simple with it. This problem can be mostly solved by using appropriate C++ typedefs. Through these typedefs, the use of templates can be effectively hidden to the novice user. In the examples of this document we use a header file containing typedefs for the most commonly used number types and representation types. It is not possible to guarantee that the user will never see templates at all (for example the templates will sometimes become visible in error messages of the compiler or during low level debugging), but the absence of templates on the source code level makes it definitely easier to start using CGAL. Developing computational geometry applications is in general very difficult because of problems with inaccuracies and degeneracies. In CGAL these problems are largely overcome by the strong support for computing with exact number types and the existence of algorithms that can deal with degeneracies. Furthermore, the algorithms in the library contain many pre and postcondition checks. These checks are performed by default, which can be a great help when debugging an application. By setting a compiler flag these checks can be turned off to gain execution speed. To facilitate using CGAL with existing code, CGAL types and algorithms are placed in namespace *CGAL*. All macro names, which can not be put in a namespace, are prefixed with *CGAL_*. Another point where the library can make the user's life easier is the consistent use of the iterator concept of the C++ Standard Template Library (see also Appendix A).

1.6 Other design goals

Apart from the above mentioned design goals, there are several others that apply to a library like CGAL such as openness and modularity. More on this can be found in [Fabri & al. 96], [Overmars 96], [Schirra 96], and [Fabri & al. 98].

Chapter 2

Elementaries

2.1 Points and Vectors

Example file: examples/Getting_started/basic.C

Points and vectors are about the most basic elements in geometry. The ”Hello, Geo” program of this document shows some operations that can be done on them.

```
 1  #include "tutorial.h"
 2  #include <CGAL/Point_2.h>
 3  #include <CGAL/Vector_2.h>
 4  #include <iostream>
 5
 6  main()
 7  {
 8      Point p1(1.0, -1.0), p2(4.0, 3.0), p3;
 9      Vector v1(-1, 10);
10      Vector v2(p2-p1);
11      v1 = v1 + v2;
12      p3 = p2 + v1*2;
13      std::cout << "Vector v2 has coordinates: ("
14           << v2.x() <<", "<<v2.y() <<")\n";
15      std::cout << "Point p3 has coordinates: ("
16           << p3.x() <<", "<<p3.y() <<")\n";
17  }
```

When we compile and run the program, the output is:

```
Vector v2 has coordinates: (3, 4)
Point p3 has coordinates: (8, 31)
```

Before we have a look at the operations on points and vectors, we consider the structure of the program. First there are include files. The first include file is `tutorial.h`. This header file contains some definitions that make our example programs easier to read. In Section 3.2 we will show what it contains. Next we include some header files that define the CGAL points and vectors. As we want to do some output, we also include the standard C++ IO. The order of inclusion sometimes is important in CGAL. We always include `tutorial.h` as the first file. When we explain the contents of this file, we'll explain why.

In line 8, three two-dimensional points are declared. The first two are initialised with x and y values. The third is not initialised. In the next two lines, two vectors are declared. The first vector is initialised in the same way as the points. The second vector is initialised with the difference of two points.

In lines 11 and 12 we see some operations on vectors and points. Two vectors can be added, and a new vector results. A vector can be multiplied with a number. A vector can be added to a point, resulting in another point.

In lines 13 to 16 we print the coordinates of the computed vector and point to standard output. The x and y coordinates are doubles.

2.1.1 The difference between points and vectors

Points and vectors seem to be very similar. They both have an x and y coordinate. In what way do they differ?

A point is a geometrical object. It has a position in the two dimensional space, or in a higher dimension, if we have higher dimensional points. A point can lie on a line, or inside a triangle. It has a distance to another point and to other geometric objects.

Vectors are not geometric objects in this sense. A vector can be thought of as the difference between two points. Vectors can be added and subtracted. Every vector *vec* has an inverse $-vec$. Finally, vectors can be multiplied by a number, which multiplies all coordinates.

The two concepts should be well separated. This is enforced by typing. Trying to add two points to each other or taking the distance from a vector to a point will lead to compilation errors.

CGAL also knows the concept of an origin. It can be used in cases where the concepts of vector and point become mingled. In 2D, the origin is a point with coordinates (0,0). However, we use a separate type (*Origin*) that has a single value: *ORIGIN*. This constant can be used to convert between vectors and points in an efficient way.

Remember that we can subtract two points from each other, in which case we get a vector, and can add a vector to a point, resulting in a point. In the same way it is possible to subtract the origin from a point, resulting in a vector with the same coordinates as the point, and we can add a vector to the origin, resulting in a point with the same coordinates as the vector. The value *ORIGIN* can be used as the origin in all dimensions.

2.2 Predicates

As we have access to the coordinates, we can do anything with points what we could possibly want to do. But usually we don't want to work on such a low level. CGAL provides predicates that work on a higher level.

2.2.1 Orientation of points

Example file: examples/Getting_started/orientation.C

An important predicate is about the orientation of points. Three points may make a left or a right turn or they may lie on a line. Figure 2.1 shows the possible orientations of three points, p1, p2 and p3.

In the following program the user is repeatedly prompted to give 3 points. The predicate is used to decode in what orientation they are.

```
#include "tutorial.h"
#include <CGAL/Point_2.h>
#include <CGAL/predicates_on_points_2.h>
#include <iostream>

using std::cout;
using std::cin;

main()
{
```

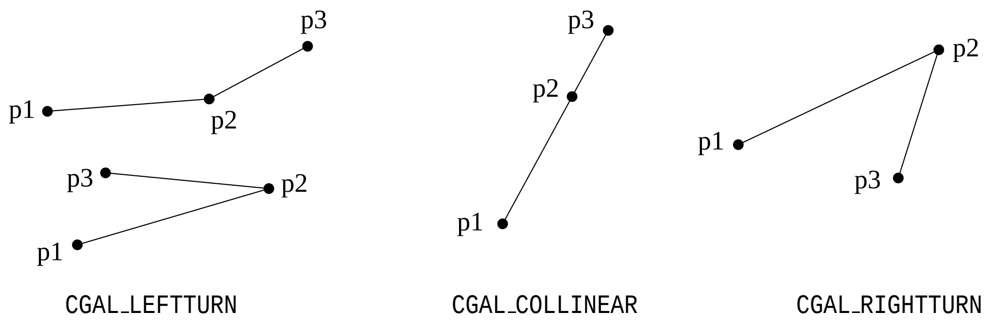


Figure 2.1: Orientation of points.

```
    double x1, x2, x3, y1, y2, y3;
    do {
        cout << "Give three points (6 coordinates, separated by spaces).\n";
        cout << ">> " << std::flush;
        cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;
        if (!cin)
            break;
        Point p1(x1,y1), p2(x2,y2), p3(x3,y3);
        switch (CGAL::orientation(p1,p2,p3)) {
          case CGAL::LEFTTURN:
            cout << "Left turn.";
            break;
          case CGAL::RIGHTTURN:
            cout << "Right turn.";
            break;
          case CGAL::COLLINEAR:
            cout << "The three points lie on a line.";
            break;
        }
        cout << "\n\n";
    } while (1);
}
```

A session is shown below. We give four times normal input. The last time we type ’q’, which ends the program.

```
Give three points (6 coordinates, separated by spaces).
>> 0 0  1 0  2 1
Left turn.

Give three points (6 coordinates, separated by spaces).
>> 0 0  1 0 2 -0.1
Right turn.

Give three points (6 coordinates, separated by spaces).
>> 0 0  1 0  2 0
The three points lie on a line.

Give three points (6 coordinates, separated by spaces).
>> 1 0 1.3 1.7 1.9 5.1
Left turn.
```

```
Give three points (6 coordinates, separated by spaces).
>> q
```

Note the last result. Although the three points lie on a line, the program tells us that they make a left turn. This is due to round-off errors, either during computation of the predicate or during the conversion of the decimal input to the binary internal representation. The coordinates of the points are represented as doubles, and so round-off errors are to be expected. This is an important fact to keep in mind when implementing geometric algorithms. When a round-off error will occur is hard to predict, especially if the predicates are treated as black boxes.

It is possible to do exact computations in CGAL. The next chapter tells more about this topic. The inexactness here is a consequence of the choices that were made in the header file `tutorial.h`.

2.2.2 Inside circle

Example file: examples/Getting_started/incircle.C

Another predicate decides if a test point lies inside a circle going through three points. If three points do not lie on a line, there is exactly one circle that passes through them. Figure 2.2 shows a circle through three points p1, p2 and p3 and three test points.

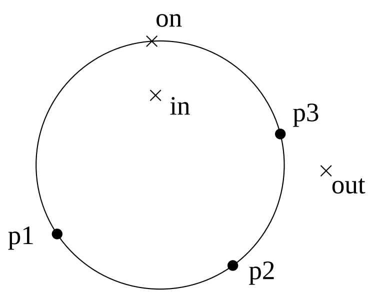


Figure 2.2: Inside or outside a circle.

The function *side_of_bounded_circle()* tests on which side of a circle a query point lies. The circle itself is defined by three points through which it should pass. The result is a value of type *Bounded_side*. This is an enumeration type with the values *ON_BOUNDED_SIDE*, *ON_UNBOUNDED_SIDE* and *ON_BOUNDARY*.

The following listing shows how the predicate is used. The program does not output anything. The assert macros are used to check if a boolean expression is true. If no core dump is produced, all assertions are true.

```
#include "tutorial.h"
#include <CGAL/Point_2.h>
#include <CGAL/predicates_on_points_2.h>
#include <assert.h>

main()
{
    Point p1(0, -5), p2(3, -4), p3(4, 3), in(-1, 4), out(5, -1), on(0, 5);
    CGAL::Bounded_side inside, onside, outside;
    inside = CGAL::side_of_bounded_circle(p1, p2, p3, in);
    outside = CGAL::side_of_bounded_circle(p1, p2, p3, out);
    onside = CGAL::side_of_bounded_circle(p1, p2, p3, on);
    assert(inside == CGAL::ON_BOUNDED_SIDE);
    assert(outside == CGAL::ON_UNBOUNDED_SIDE);
    assert(onside == CGAL::ON_BOUNDARY);
}
```

The names of the predicate and of the constants may seem long. This is due to the fact that there is another predicate which does almost the same thing, but considers the circle as oriented (see the reference manual for details). In order to avoid confusion between those two predicates and their return values, shorter names like *inside* or *which side* were not used. CGAL places clarity before brevity in its naming.

2.3 Example: centre of mass

Example file: examples/Getting_started/centre_of_mass.C

In this section we will compute the centre of mass of a number of point masses. This can be done by taking the weighted sum of a number of vectors:

$$\frac{\sum_{i=1}^{n} m_i \vec{v}_i}{\sum_{i=1}^{n} m_i} \tag{2.1}$$

This formula describes the position of a point mass in terms of a vector. We would expect that it is given as a point. The vector gives the position relative to a fixed, chosen point: the origin.[1]

In the program below we use conversion between points and vectors in both ways. This occurs in the function *centre_of_mass*. The program starts with the inclusion of header files.

```
#include "tutorial.h"
#include <iostream.h>
#include <CGAL/Point_2.h>
#include <CGAL/Vector_2.h>
```

A struct is defined that defines a point mass which has fields for the position and the mass. One constructor is defined, which initialises the position and the mass. This is done by the code after the colon.

```
struct Point_mass {
    Point_2 pos;
    double mass;
    Point_mass(const Point_2 & p, double m): pos(p), mass(m) {}
};
```

The actual computation is done in the function *centre_of_mass*. In a loop we compute the numerator and denominator of equation (2.1). Those sums are collected in the variables *sumv* and *sumw*. As the formula is written in terms of vector additions and multiplications (with a number), we have to convert the data points to vectors first, which is done by subtracting the origin. Finally, we convert the resulting vector *sumv/sumw* back to a point by adding the origin to it.

```
Point_2 centre_of_mass(Point_mass *cur, Point_mass *beyond)
{
    Vector_2 sumv(0.0, 0.0);
    double sumw = 0.0;
    for ( ; cur != beyond; ++cur) {
        sumv = sumv + (cur->pos - ORIGIN) * cur->mass;
        sumw += cur->mass;
    }
    return ORIGIN + sumv/sumw;
}
```

The main procedure creates an array of point masses, calls the routine to compute the centre of mass and writes the result to standard output.

[1]The laws of nature can normally be written by means of vectors because the choice of the origin is arbitrary. They are invariant under translation.

```
main()
{
    const int N = 4;
    Point_mass points[N] = {
            Point_mass(Point_2(3,4), 1),
            Point_mass(Point_2(-3,5), 1),
            Point_mass(Point_2(2.1,0), 10),
            Point_mass(Point_2(7,-12), 1)
            };
    Point_2 centre = centre_of_mass(points, points+N);
    cout << "The centre of mass is: ("
         << centre.x() <<", "<< centre.y() <<")\n";
}
```

This program passes parameters in a way that may seem strange at first sight. Here we pass two pointers, one to the start of the array and one pointing just after the array. A more common way is to give the number of arguments as second parameter. This way of passing parameters is more in line with the practice of the standard template library (STL). We will tell more about STL in Section 4.1.1, where we will also come back to the example above.

2.4 Naming

In order to make it easier to remember what kind of entity a particular name refers to, CGAL has a naming convention.

- All globally visible names are in namespace CGAL. This means that, for instance, you will have to refer to *CGAL::ORIGIN*, not to *ORIGIN*. In the text, we will omit the namespace, but in the code, we will use it. The few macros that exist in CGAL all start with a *CGAL_* prefix.
- If a name is made of several words, those words are separated by underscores. For example, *side_of_bounded_circle*.
- All types (classes and enums) start with one uppercase letter and are all lowercase for the rest. Examples are *Bounded_side* and *Point_2*.
- Functions, whether member functions or global funcions, are all lowercase. Examples are *side_of_bounded_circle(...)* and *Point_2::x()*.
- Constants and enum values are all uppercase. For instance *ON_BOUNDED_SIDE* and *Triangulation_2::EDGE*.

Chapter 3

Arithmetics

3.1 Number types and exact arithmetic

Until now everything was based on doubles. Coordinates were stored as doubles and computations were done on doubles. On page 8 we saw how this can lead to problems. Round-off errors may cause a wrong decision to be made.

The CGAL library itself does not favour doubles over other number types. The decision to use doubles is not taken in the CGAL library, but in the header file `tutorial.h`. In CGAL all geometric classes are parameterised by number type.

The problem with floating point types is that their operations are inexact. The C++ language also has number types like *int* and *long* where computations are done exact (as long as there is no overflow). Alas, those integer number types have their drawbacks, one of which is that they have no division operator with the nice property that $a*(b/a) \approx b$ for all a and b. For example, $100*(99/100)$ equals 0. Still, we can use integer types as a basis for exact computation, by representing numbers as rationals with an integer numerator and denominator.

There can be two good reasons for choosing this representation. Your application may use a number type where division is an expensive operation (compared to multiplication). Or you may want to use exact arithmetic based on integers. In this case, the most common choice is to use an integer class that can deal with arbitrarily large numbers, since types like *long* are bound to overflow. An example of such a class is the LEDA type *integer* [Mehlhorn & al. 98]. This type can be used in CGAL programs as *leda_integer*, by including the header file `CGAL/leda_integer.h`, which adapts the LEDA integers to the requirements that CGAL imposes on its number types. The precise requirements for using a number type as a parameter are described in the CGAL Reference Manual. Another class for arbitrary precision integer arithmetic is the *gmp_z* type (Gnu Multiple Precision $\mathbb{Z}$) [Granlund 96]. The type *Gmpz* is a wrapper class around this type. Note that CGAL only provides wrappers for LEDA and Gnu number types. If you want to use them, you need to have LEDA or GMP installed on your system.

3.2 Coordinate representation

Let's have a look at points. How can we represent the point $(5.2, 3.18)$ using integers? We can introduce a third value which is supposed to divide the other values. So, we describe this point by the three-tuple $(520, 318, 100)$. This third value is known as the homogenising coordinate. The trick we apply here is like switching from integers to rationals. Once the issue of representation is solved, computation is not difficult. Here we can use familiar rules for rationals:

$$\frac{a}{b}\Big/\frac{c}{d} = \frac{a*d}{b*d}$$

$$\frac{a}{b} < \frac{c}{d} \equiv a*d < b*c \text{ (for positive } b \text{ and } d)$$

The example with the points shows a peculiarity when we switch from one number type to another. When we have a number type that supports division, the most natural representation of a point uses two numbers (Cartesian representation).[1] But when we have an integer number type, we need three numbers (homogeneous representation). This fact, that the representation of a geometric object depends on the underlying number type, occurs frequently. It is the reason why the parameterisation by number type takes place in two stages.

First, there is the representation class. This is a class that decides which representation is chosen by the different geometric objects. Currently there are only two possibilities; either *Cartesian* or *Homogeneous*.

These classes are parameterised by number types. In the case of *Cartesian* this number type should provide a division operator that behaves in the appropriate way. The language-defined number types float and double are commonly used. But there are also libraries that provide number types that can be used here. For example, LEDA [Mehlhorn & al. 98] supplies the number types rational and real.

The number type is a template parameter of the representation class, and the representation class is a template parameter of the geometric object class. Readers not familiar with the C++ concept of templates can just follow the examples below. Here is how we can declare points based on C++ doubles, LEDA rationals and LEDA reals:

```
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/leda_rational.h>
#include <CGAL/leda_real.h>

using CGAL::Cartesian;
using CGAL::Point_2;

Point_2< Cartesian <double> > pd1;
Point_2< Cartesian <leda_rational> > pd2;
Point_2< Cartesian <leda_real> > pd3;
```

And here is how we can declare a point based on integers. We define a point with the built-in long, a point with LEDA's integer, one with a Gmpz's integer and one with *double* as number type.

```
#include <CGAL/Homogeneous.h>
#include <CGAL/Point_2.h>
#include <CGAL/leda_integer.h>
#include <CGAL/Gmpz.h>

using CGAL::Homogeneous;
using CGAL::Point_2;

Point_2< Homogeneous <long> > pi1;
Point_2< Homogeneous <leda_integer> > pi2;
Point_2< Homogeneous <CGAL::Gmpz> > pi3;
Point_2< Homogeneous <double> > pi4;
```

The order in which the include files appear is important in CGAL. The files `Cartesian.h` and `Homogeneous.h` must be included before any other CGAL include files. If they are both included, the order in which this is done does not matter. But if you include `Point_2.h` before `Cartesian.h`, the preprocessor should give an error message.

There is another point to note, which is not specific to CGAL, but is a peculiarity of C++ syntax of nested templates. Note that we use a lot of spaces in the declarations above. Most of them are not necessary, except the one between the two `>` brackets. Without a space, the lexical analyser would interpret `>>` as a right shift token instead of two closing brackets, which results in compilation errors.

As you can see there were good reasons to hide the complete names of the types in the header file `tutorial.h`. We advise you to use typedefs to get shorter names. For instance:

[1]Users familiar with projective geometry may disagree here.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/Line_2.h>

typedef CGAL::Cartesian<double> Rep_class;
typedef CGAL::Point_2<Rep_class> Point_2;
typedef CGAL::Line_2<Rep_class> Line_2;
```

This kind of definitions can also be found in the header file `tutorial.h`.

3.3 Example

Example file: examples/Getting_started/exact_orientation.C
Example file: examples/Getting_started/exact_orientation_gmpz.C

We return to the example of section 2.2.1. There we encountered a round-off error which led to a wrong decision. Now we will use exact arithmetic to solve this problem.

```
#include <CGAL/Homogeneous.h>
#include <CGAL/Point_2.h>
#include <CGAL/predicates_on_points_2.h>
#include <iostream>

using std::cout;

typedef CGAL::Homogeneous<long> Rep_class;
typedef CGAL::Point_2<Rep_class> Point;

main()
{
    Point p1(0, 0), p2(3, 17, 10), p3(9, 51, 10);
    switch (CGAL::orientation(p1,p2,p3)) {
      case CGAL::LEFTTURN:
        cout << "Left turn.";
        break;
      case CGAL::RIGHTTURN:
        cout << "Right turn.";
        break;
      case CGAL::COLLINEAR:
        cout << "The three points lie on a line.";
        break;
    }
    cout << "\n";
}
```

We used the built-in type *long* and not an arbitrary precision integer as number type. This ensures that the code compiles on all systems, without the need for additional libraries besides CGAL. In this case, where we know that we only compute with small integers, there is no problem. In real code this would usually not be the case.

3.4 Trade-offs between number types

We are aware that, despite the fact that typedefs can be used to alleviate the biggest problems, parameterisation by number type makes the CGAL code a little harder to read. On the other hand, parameterisation by a number type makes the library easier to use. For example, plug in *leda_real* and you don't have robustness problems any longer.

Moreover, there are many geometric algorithms and there are many fields in which they can be used. If there would have been a single number type that would suit everybody's needs perfectly, we would have chosen it. But alas, there are tradeoffs, both from the implementation side and from the user side.

- For some applications it is very important not to loose any precision during the computation. Others may not care so much about that, but may be more interested in the speed with which the results are computed (perhaps their input data is based on measurements with a large error). In the latter case, built-in floating point types are a good candidate. In the first case, some high precision or exact number type libraries may be a better choice.

- Implementing an algorithm is easier when exact arithmetic can be assumed. Otherwise it is often necessary to very carefully analyse an algorithm to see what major consequences a minor round-off error may have. The algorithm may have to be adapted to be more robust under such circumstances. This may also mean that the efficiency gained by using a number type with faster operations is lost through the more complicated algorithm.

  An exactness problem may be inherent to the input of the problem. That is, a small perturbation of the input would lead to a (radically) different output. In this case, the algorithm is allowed to give the output belonging to the disturbed output when plugging in an inexact number type. For example, if a point lies (almost) on the boundary of a polygon, asking whether it lies inside or outside may give the wrong answer. This type of behaviour should always be expected by the user when inexact number types are used.

  On the other hand, the exactness problem may be caused by the chosen implementation. This case should be carefully documented. We can elaborate on the previous example. Suppose we use the following method to decide if a point lies inside a polygon: cast a ray from the point in some arbitrary direction and count the number of times that an edge of the polygon is crossed. If this number is odd, the point lies inside, otherwise outside. Now, if the ray passes (almost) through a polygon edge endpoint, inexact computation may miss it (or count it twice), leading to a wrong result, even though the point lies well inside or outside the polygon.

- Another aspect is the availability of operations on number types. For example, if the length of a vector is computed, a square root operation is needed. For a number type like *int* this is not available. Even the finite precision *double* is not closed under the square root operation. Although there are number types (e.g. LEDA's real) that can compute exactly with of square roots, exact arithmetic may become infeasible when more operations (like sine or natural logarithm) are needed.

Which of the above considerations are important depends on the particular algorithm, the particular number type, and the particular application. The good thing about our solution is that it makes it very easy to switch between number types, so testing what number type is best suited is little work. Normally, all one has to do is change one or two typedefs placed in a strategical header file and recompile.

Chapter 4

Stepping through

4.1 Iterators

In Section 2.3 we discussed code to compute the centre of mass of a number of point masses. We wrote the code in a somewhat peculiar way, to make it easier to generalise. In this section we will explain what we mean by that. We will introduce the concept of iterators. Readers familiar with iterators and the C++ standard template library won't find anything new here, and can skip to the next section.

A set of objects can be stored in different ways; in an array, a list, a tree or other containers. Often, it is not very important to an algorithm how the objects are stored. But if we implement the algorithm in a routine it seems that we have to make a choice how to pass the objects to the routine. If we choose to pass them as a list and the caller has them stored in an array, the caller first has to create a list and copy the objects into it. Of course, it is possible to implement the algorithm for various containers. However, besides being tedious and error prone, this approach works only when the type of containers needed is known beforehand and the number of different types is not too large.

The iterator concept helps in those cases. Instead of passing a container to a routine, we pass iterators. Now, what is an iterator? An iterator is some kind of pointer to an object in a container. When we say *some kind of pointer* we mean that an iterator must satisfy a number of requirements. Any object that satisfies these requirements is an iterator. In this sense, an iterator is a concept rather than a language element. It must be possible to go to the next element (advance the iterator) and to get to the object to which the iterator points (dereferencing). Furthermore, the syntax used to advance and dereference must be the same as with normal pointers. So, if *it* is an iterator, we can advance it by $++it$ or $it++$ and dereference it by $*it$.

Now, to be usable in this framework, every container must have associated iterators that iterate over the elements of the container. For arrays the iterators are pointers. For other containers, it should be possible to get iterators to the first and last element by means of member functions *begin()* and *end()*. More precisely, the *end()* function gives an iterator that points one position beyond the last element. Because of the precise syntactic and semantic rules that an iterator must obey, it is now possible to write just one implementation that works for all iterators.

4.1.1 Example: centre of mass revisited

Example file: examples/Getting_started/templ_centre_of_mass.C

We repeat here the example of computing the centre of mass of a set of point masses. There are no new CGAL calls, but this time we use three different data structures to store the points: an array, a vector, and a list. The array is a built-in construct of C++, the vector (a dynamic array) and the list are data types of the Standard Template Library [Musser & al. 96]. STL is part of the C++ standard and free implementations of it are available, see [STL]. Modern versions of compilers should be able to handle it. You will need to have STL installed in order to be able to compile the example.

We start with including header files and making some typedefs. The include files `vector` and `list` are part of STL.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/Vector_2.h>
#include <iostream>
#include <vector>
#include <list>

typedef CGAL::Cartesian<double> Rep_class;
typedef CGAL::Point_2<Rep_class> Point_2;
typedef CGAL::Vector_2<Rep_class> Vector_2;
```

The definition of the point mass structure is almost the same as in the original program. We added a constructor without parameters, which is convenient if we deal with vectors and lists.

```
struct Point_mass {
    Point_2 pos;
    double mass;
    Point_mass(const Point_2 & p, double m): pos(p), mass(m) {}
    Point_mass() {}
};
```

In the main routine below, three containers are declared: points1 (an array), points2 (a vector) and points3 (a list). The vector and the list both take a template parameter that indicates what type of values are stored in the container. The argument *Point_mass* is written after the class name, in brackets < >. The array is initialised in a standard way. Both the vector and the list are initialised with the elements of the array. This is the first use of iterators: a pointer to the first element as begin iterator and a pointer past the last argument as the end iterator. Remember that in C and C++, a pointer is allowed to point one element past an array.

The three calls to the routine *centre_of_mass* take a begin and end iterator as arguments. We already saw in Section 2.3 how to supply the array iterators. Both the vector and the list class supply their begin and end iterator by functions *begin()* and *end()*. This is in accordance to the requirements of STL.

```
void write(const Point_2 &centre)
{
    std::cout << "The centre of mass is: ("
          << centre.x() <<", "<< centre.y() <<")\n";
}

main()
{
    const int N = 4;
    Point_mass points1[N] = {
            Point_mass(Point_2(3,4), 1),
            Point_mass(Point_2(-3,5), 1),
            Point_mass(Point_2(2.1,0), 10),
            Point_mass(Point_2(7,-12), 1)
            };
    write(centre_of_mass(points1, points1+N));

    std::vector<Point_mass> points2(points1, points1+N);
    write(centre_of_mass(points2.begin(), points2.end()));

    std::list<Point_mass> points3(points1, points1+N);
    write(centre_of_mass(points3.begin(), points3.end()));
}
```

Now let's direct our attention to the implementation of *centre_of_mass()*. (This function is described last, but should actually be defined before the main function in the complete program text.) Although a user of CGAL does not need to know the following, it may still be interesting to see it. Since the iterator types of the vector and list may be different from each other and from 'pointer to *Point_mass*', how is it possible that we have one function that works for arguments of different types ?

We see that the *centre_of_mass()* routine below is almost the same as the original one in Section 2.3:

```
template <class Iterator>
Point_2 centre_of_mass(Iterator cur, Iterator beyond)
{
    Vector_2 sumv(0.0, 0.0);
    double sumw = 0.0;
    for ( ; cur != beyond; ++cur) {
        sumv = sumv + ((*cur).pos - CGAL::ORIGIN) * (*cur).mass;
        sumw += (*cur).mass;
    }
    return CGAL::ORIGIN + sumv/sumw;
}
```

The line

template <class Iterator>

says that the following routine is templated by a class, which is given the name *Iterator*. If a function is templated, it represents a whole family of functions; one function for every type that we fill in for the template parameter. Wherever the parameter name is encountered in the definition of the routine, it is replaced by that specific type. This process of filling in a type to get one particular function of the family is called instantiation.

In the argument list we see twice *Iterator* instead of *Point_mass **. The two iterators *cur* and *beyond* define a so-called range *[cur,beyond)*, which means that applying a finite number of times the operator '++' to *cur* makes that it is equal to *beyond*. The range refers to the points starting with *cur* up to but not including *beyond*. The iterator *beyond* is said to point 'past the end' of the range. If *cur* is equal to *beyond*, the range *[cur, beyond)* is empty.

Note that in standard C++ you can use the arrow notation *cur−>mass* rather than *(∗cur).mass*. This notation is not supported by older compilers and hence not implemented in older versions of STL. There is more to say about iterators than we did here. For instance, there are different classes of iterators (input iterators, forward iterators, random access iterators, ...) which have different requirements. For more information about iterators and sequence containers, see the companion document 'The Use of STL and STL Extensions in CGAL'.

4.2 Circulators

Example file: examples/Getting_started/circulate.C

For inherently circular structures, such as polygons, CGAL provides so-called circulators. Circulators are similar to iterators, but there is no past-the-end value, because of the circularity. A container providing circulators has no *end()* method, only a *begin()* method. For a circulator *cir*, the range *[cir, cir)* denotes the sequence of all elements in the data structure. By contrast, for iterators this range would be empty. For a circulator *cir*, *cir==NULL* tests whether the data structure is empty or not.

The following example shows a typical use of a circulator. The program reads a polygon from a file, circulates over the edges, and sums their lengths to compute the perimeter.

```
#include "tutorial.h"
#include <CGAL/Polygon_2.h>
#include <fstream>
```

```
4
5   typedef Polygon::Edge_const_circulator Edge_circulator;
6
7   main()
8   {
9     Polygon polyg;
10
11    std::ifstream from("polygon.dat");              // input file stream
12    CGAL::set_ascii_mode(from);                 // file contains ascii
13    from >> polyg;                            // read polygon
14
15    Edge_circulator start = polyg.edges_circulator();
16    double perimeter=0;
17
18    if (start != 0) { // polygon not empty
19      Edge_circulator cur = start;
20      do {
21        Segment edge = *cur;
22        double l2 = edge.squared_length();
23        perimeter += sqrt(l2);
24        ++cur;
25      } while (cur != start);
26    }
27
28    std::cout << "Perimeter of the polygon is "
29              << perimeter << std::endl;
30  }
```

First the file `tutorial.h` is included, which contains the definitions for a number of geometric objects in Cartesian coordinates, represented in `double` number type. Also the *Polygon_2* type is defined there. Line 5 shows a type definition. The circulator type to step through the edges of a polygon, *Polygon_2::Edge_const_circulator* is named *Edge_circulator* for readability. In lines 11 through 13, a polygon is read from a file.

A circulator *start* is declared in line 15. The polygon method *edges_circulator()* returns a circulator that can be used to successively address the edges. We first test if there are edges at all, i.e. whether the range *[start,start)* is non-empty (line 18). If so, we circulate over the edges in a do-loop. The running circulator *cur* is initialised with the starting value in line 19 , incremented in line 24, and tested in line 25. The loop runs as long as the current iterator position does not reach the starting position. In the body of the loop the circulator is dereferenced, which yields an edge of value type *Segment_2*.

Note that a segment has no method for its length, only for the square of the length. This is to avoid square root computations, since most number types are not closed under the square root operation. It is often possible to work with squared lenghts, but because we want to add length to calculate the perimeter, we take the square root explicitly. Since we are working with numbers of type *double*, which are not closed under the square root operation, we get only approximate results. Should you wish to compute the length exactly, one of the possible solutions is to use the LEDA type `real`, which *is* closed under the square root operation.

4.2.1 Example: centre of mass revisited again

Example file: examples/Getting_started/polygon_centre.C

In the previous section we computed the centre of mass of a number of point masses, by stepping through the container of the point masses with an iterator. In this section we compute the centre of mass of a (filled) polygon, by stepping through the container of the vertices with a circulator.

The formula for the center of mass of a (filled) polygon is similar to, but slightly different from, the formula for the centroid of point masses. For a polygon in the plane, with at least three vertices $v_1, \ldots, v_n, v_{n+1} = v_1$, the center of mass is:

$$\frac{\sum_{i=1}^{n} a_i(\vec{v}_i + \vec{v}_{i+1})}{3 * \sum_{i=1}^{n} a_i}, \tag{4.1}$$

with $a_i = x_i y_{i+1} - x_{i+1} y_i$. To code this formula we could step through the vertices with an iterator, but we see that we have to address the starting vertex v_1 a second time as v_{n+1}. For such applications a circulator is particularly useful:

```
 5  typedef Polygon_2::Vertex_circulator Vertex_circulator;
 6
 7  using CGAL::ORIGIN;
 8
 9  Point_2 centroid (Polygon_2 polyg)
10  {
11    // check if the polygon has at least three vertices
12    assert (polyg.size() >= 3);
13
14    Vertex_circulator start = polyg.vertices_circulator();
15    Vertex_circulator cur = start;
16    Vertex_circulator next = cur;
17    ++next;
18
19    Vector_2 centre(0,0);
20    double a=0.0, atot=0.0;
21    do {
22      a = ((*cur).x()) * ((*next).y()) - ((*next).x()) * ((*cur).y());
23      centre = centre + a * ((*cur - ORIGIN) + (*next - ORIGIN));
24      atot = atot + a;
25      cur = next;
26      ++next;
27    } while (cur != start);
28    atot = 3*atot;
29    centre = centre/atot;
30    return ORIGIN + centre;
31  }
```

First the type *Vertex_circulator* is defined as the circulator type for vertices of a polygon, *Polygon_2::Vertex_circulator* (line 5). Before the computation starts, it is checked in line 10 whether the polygon has at least three vertices (if not, the program terminates). Then the circulator variables *start*, *cur*, and *next* are declared. Within the loop, the numerator (lines 20 and 21), and the denominator of formula (4.1) (line 22) are computed. The loop continues as long as *cur* is not equal to the starting value *start*. In the last iteration of the loop, *next* is equal to *start*, something that could not happen if it were an iterator.

Note again the conversion between points and vectors in lines 21 and 28, as explained in Section 2.1.1. Also note that the number type used to represent coordinates has been defined as *double* in the file `tutorial.h`, so that the type of variable *a* has also been chosen to be *double*. In general, the number type must be the field type associated with the representation class, see the CGAL Reference Manual.

For more information about circulators, see the companion document The Use of STL and STL Extensions in CGAL.

Chapter 5

Intersections and Boolean operations

5.1 Introduction

In many applications such as animation, computer aided design, and ray tracing, it is necessary to test whether two objects intersect, and to actually compute the intersection. Bounding boxes can be used to decide whether two objects do not intersect. If the bounding boxes do not intersect, then the objects do not either, otherwise further checking may be necessary.

5.2 Bounding boxes

Example file: examples/Getting_started/boundingbox.C

The primitive objects in the kernel that have a position and have limited extent (point, segment, triangle and tetrahedron, iso-rectangle, and circle, but for example not vector and line), have a member function *bbox()*, which returns a bounding box of type *Bbox_2*. This is illustrated for triangles in the program below, see also Figure 5.1. Lines 8 and 9 declare two triangles, in lines 12 and 13 their bounding boxes are taken.

```
 1  #include "tutorial.h"
 2  #include <CGAL/Point_2.h>
 3  #include <CGAL/Triangle_2.h>
 4  #include <CGAL/Bbox_2.h>
 5
 6  main()
 7  {
 8    Triangle t1(Point(-5.2,1.7), Point(-7.1,-6.3), Point(-0.9,-2.3));
 9    Triangle t2(Point(-2.8,-4.5), Point(4.5,-1.1), Point(2.4,-7.6));
10    Triangle t3(Point(5.5,8.8), Point(-7.7,8.3), Point(1.3,2.9));
11
12    Bbox bb1 = t1.bbox();
13    Bbox bb2 = t2.bbox();
14    Bbox bb12, bb3;
15
16    std::cout << "Bounding box 1: " << bb1
17              << "\n and bounding box 2: "<< bb2 << std::endl;
18    if ( !CGAL::do_overlap(bb1, bb2) )
19       std::cout << "do not ";
20    std::cout << "overlap." << std::endl;
21
```

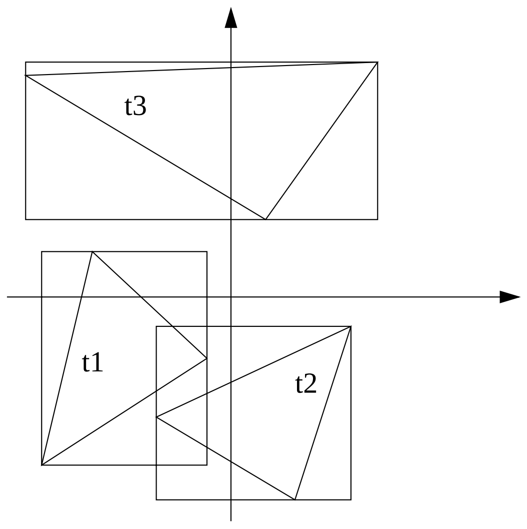


Figure 5.1: Bounding boxes.

```
22     bb12 = bb1 + bb2;
23     bb3 = t3.bbox();
24
25     if (bb12.ymax()<bb3.ymin()) {
26        std::cout << "Triangle 3:\n" << t3 << std::endl;
27        std::cout << "lies above triangle 1:\n" << t1 << std::endl;
28        std::cout << "and triangle 2:\n" << t2 << std::endl;
29     }
30  }
```

There is a global function *do_overlap()*, operating on two bounding boxes, which determines whether they overlap, see line 17. The member functions *xmin()*, *xmax()*, *ymin()*, and *ymin()* return the minimum and maximum coordinate values, always as a *double*, independent of the representation of the object from which the bounding box is taken. In the file `tutorial.h` the triangles are defined with Cartesian coordinates and number type *double*. However, also for other number types (for instance *leda_rational*), or for homogeneous representations of the object, the bounding box is represented with number type *double*. This means that the *xmin* coordinate of the bounding box is not necessarily the smallest x-coordinate of the object, but of course the bounding box is guaranteed to contain the object completely.

A bounding box object need not come from an object, it can be constructed independently. For example, in line 14 an uninitialised bounding box *b12* is created, which is set in line 21 to the bounding box of the two boxes *b1* and *b2* with the + operator, which 'adds' two boxes.

5.3 Intersection

Example file: examples/Getting_started/inters_check.C
Example file: examples/Getting_started/inters_comp.C

If two bounding boxes of objects intersect, it may be necessary to test whether the objects themselves actually intersect. Checking for intersection is often easier than actually computing the intersection result. Therefore, CGAL provides the function *do_intersect(obj1, obj2)*, which merely tests for intersection and returns a *bool*. The parameters *obj1* and *obj2* can have various types. For two-dimensional kernel objects for example, they can be *Point_2*, *Line_2*, *Ray_2*, *Segment_2*, *Triangle_2*, and *Iso_rectangle*. Internally, function *do_intersect()* uses the bounding box overlap test.

The following little program shows the testing for intersection of two triangles. Note that the file `intersections.h` must be included.

```
#include "tutorial.h"
#include <CGAL/Point_2.h>
#include <CGAL/Triangle_2.h>
#include <CGAL/intersections.h>

main()
{
  Triangle t1(Point(-5.2,1.7), Point(-7.1,-6.3), Point(-0.9,-2.3));
  Triangle t2(Point(-4.8,-4.5), Point(4.5,-1.1), Point(2.4,-7.6));

  std::cout << "Triangle 1:\n" << t1 <<
                "\nand triangle 2:\n" << t2 << std::endl;
  if ( ! CGAL::do_intersect(t1,t2))
    std::cout << "do not intersect" << std::endl;
  else
    std::cout << "do intersect" << std::endl;
}
```

In order to actually compute the intersection of two objects, use the function *intersection(obj1, obj2)*. If *obj1* and *obj2* are both triangles, then depending on their coordinates, the intersection result can be empty, a point, a segment, a triangle, or a polygon. Because the result type is not known in advance, the return type of the function is *Object*. An object of this class can represent an object of arbitrary type. In order to identify the true type of such an object, the function *assign()* should be used. *assign(spec_obj, generic_object)*, returns *true* and assigns *generic_obj* to *spec_obj* if they have the same type, and returns *false* otherwise.

A typical use is to test all possible types such a generic object can have in a particular situation, and to handle these cases appropriately. This is illustrated in the following piece of code. Two triangles are first tested for intersection (line 15). If they do intersect, the actual intersection is computed (line 19), which returns a generic object. All possible types in this situation (point, segment, triangle, and polygon) are subsequently tested (lines 25-35).

Note that if the result is a polygon, it is returned as a vector of points, rather than as a *Polygon_2*. This is because the polygon is a templated data structure in the CGAL basic library, and the *do_intersect()* function of the kernel is kept independent from the basic library. To get the result as a polygon, use the boolean operations functions from the basic library, see the next section.

```
 7  main()
 8  {
 9    Triangle_2 t1(Point_2(2,6), Point_2(-6,5), Point_2(-2,-7));
10    Triangle_2 t2(Point_2(6,0), Point_2(-6,0), Point_2(2,-5));
11
12    cout << "The intersection of triangle 1:\n" << t1;
13    cout << "\nand triangle 2:\n" << t2 << "\n is ";
14
15    if ( ! CGAL::do_intersect(t1,t2) ) {
16      cout << "empty" << endl;
17    }
18    else {
19      CGAL::Object result = CGAL::intersection(t1,t2);
20      Point_2 point;
21      Segment_2 segment;
22      Triangle_2 triangle;
23      vector<Point_2> polypoint;                   // not a Polygon_2 !
24
25        if (CGAL::assign(point, result)) {
26          cout << "point." << endl;
27        } else
28        if (CGAL::assign(segment, result)) {
29          cout << "segment." << endl;
30        } else
31        if (CGAL::assign(triangle, result)) {
32          cout << "triangle." << endl;
33        } else
34        if (CGAL::assign(polypoint, result)) {
35          cout << "a polygon." << endl;
36        } else
37        cout << "unknown!" << endl;
38      }
39    }
```

Alternatively, the intersection function can be called first, and the function *result.is_empty()* can be used to test if the intersection is empty.

5.4 Boolean operations

Example file: examples/Getting_started/polygon_match.C

For a number of combinations of objects, the intersection has only a single result, see the previous section. However, the result of the intersection of two polygons is unknown in advance. Therefore, the result is not a single *Object*, but a list of generic objects. To take the intersection of two polygons, they must be simple, and oriented counterclockwise. A third parameter, an iterator, must be specified to indicate where the result should be put. A typical use is illustrated in the next piece of code, which computes the area of overlap of two polygons. The function only takes the intersection, and passes the result to another funciton. The iterator for the result is here the *back_inserter* of a list, see line 67, which makes that the intersection objects are added at the end of the list.

```
63    double area_overlap(Polygon_2 &polyg1, Polygon_2 &polyg2)
64    {
65      list<CGAL::Object> result;
66
67      CGAL::intersection (polyg1, polyg2, back_inserter(result) );
68      return sum_area( result.begin(), result.end() );
69    }
```

The result of the intersection is now a list of generic objects. The start and past the end iterator values are passed to the function *sum_area()*, which iterates over the corresponding range, and inspects the type of the generic object, just as in the previous section. (Note that the result type can now be a real *Polygon_2*.)

```
48    template< class ForwardIterator >
49    double sum_area( ForwardIterator start, ForwardIterator beyond )
50    {
51      Point_2 point;
52      Segment_2 segment;
53      Polygon_2 polygon;
54      double area=0;
55
56    for( ForwardIterator it= start; it != beyond; it++) {
57      if( CGAL::assign( polygon, *it) ) {       // it’s a polygon
58        area = area + polygon.area();
59      }
60    }
61    return area;
62  }
```

Similarly, the union and the difference of two polygons can be computed with the functions *union()* and *difference()*. In both cases, the result can be a polygon with zero or more holes in it (at most one for a difference). If this is the case, the list first contains the outer polygon, which has a counterclockwise orientation, and then the inner polygons that represent holes, which have a clockwise orientation.

As an example, the next function computes the area of symmetric difference of two polygons *polyg1* and *polyg2*, that is, the area of *polyg1*−*polyg2* plus the area of *polyg2*−*polyg1*. In lines 76 and 80 the differences are computed, and the result is passed on to *sum_area()* again.

```
71  double area_difference (Polygon_2 &polyg1, Polygon_2 &polyg2)
72  {
73    list<CGAL::Object> result;
74    double area1, area2;
75
76    CGAL::difference( polyg1, polyg2, back_inserter(result) );
77    area1 = sum_area( result.begin(), result.end() );
78
79    result.erase( result.begin(), result.end() );
80    CGAL::difference( polyg2, polyg1, back_inserter(result) );
81    area2 = sum_area( result.begin(), result.end() );
82
83    return area1+area2;
84  }
```

Note that the list containing the result must be emptied before using it for the second computation (line 79).

If a polygon has a clockwise orientation, its member function *area()* returns a negative value. Therefore, in function *sum_area()* the area of a hole is automatically subtracted from the area of the outer polygon, should the result contain holes.

5.5 Example: matching polygons

Example file: examples/Getting_started/polygon_match.C

The above functions can be used to perform the approximate matching of polygons. In general, matching an object B onto an object A means finding a transformation T such that $T(B)$ and A are as similar as possible. For polygons, a possible measure of similarity is the area of overlap $area(A \cap T(B))$, which should be maximized. Alternatively, a possible measure of dissimilarity is the area of symmetric difference $area(A - T(B)) + area(T(B) - A)$, which should be minimized. Naturally, the transformation that gives the maximal overlap also gives the minimal symmetric difference.

Here we restrict the type of transformations to translations only. The translation that moves the centroid of B to the centroid of A can be used as an approximation of the optimal translation. The program below computes this approximate transformation, and the resulting areas of overlap and symmetric difference.

The program first reads two polygons from a file (lines 95-97), tests for emptiness with the member function *is_empty()*, for simplicity with *is_simple()*, and the orientation with *orientation()*. When necessary, the orientation is reversed to make them counterclockwise (which is a precondition for the boolean operations), with the member function *reverse()*.

```
void main(int argc, char *argv[])
{
  Polygon_2 polygA;
  Polygon_2 polygB;

  ifstream from("polygon_match.dat");
  CGAL::set_ascii_mode(from);
  from >> polygA >> polygB;

  if (polygA.is_empty() || polygB.is_empty()) {
    cout << "Polygons must be non-empty, exiting program." << endl;
    exit (1);
  }

  // check simplicity
  if ( !polygA.is_simple() || !polygB.is_simple()) {
    cout << "Polygons must be simple, exiting program." << endl;
    exit (1);
  }

  // check counterclockwise orientation
  if (polygA.orientation() == CGAL::CLOCKWISE) {
    cout << "Polygon A is entered clockwise.";
    polygA.reverse_orientation();
    cout << " Its orientation has been reversed." << endl;
  }
  else
    cout << "Polygon A is entered counterclockwise." << endl;

  // check counterclockwise orientation
  if (polygB.orientation() == CGAL::CLOCKWISE) {
    cout << "Polygon B is entered clockwise.";
    polygB.reverse_orientation();
    cout << " Its orientation has been reversed." << endl;
  }
  else
    cout << "Polygon B is entered counterclockwise." << endl;

  cout << "Area of intersection: " << area_overlap(polygA, polygB) << endl;
  cout << "Area of symmetric difference: "
       << area_difference(polygA, polygB) << endl;

  // transform B to put centroid B over centroid A
  Point_2 centA = centroid(polygA);
  Point_2 centB = centroid(polygB);
  Vector_2 vec = centA - centB;
  Transformation_2 transl(CGAL::TRANSLATION, vec);

  polygB = transform (transl, polygB);
  cout << "Polygon B translated over " << vec << endl;

  cout << "New area of intersection: "
       << area_overlap(polygA, polygB) << endl;
  cout << "New area of symmetric difference: "
144          << area_difference(polygA, polygB) << endl;
145
146     // check convexity
147     if ( polygA.is_convex() && polygB.is_convex() )
148       cout << "Polygons A and B are both convex." << endl;
149     else
150       cout << "Polygons A and B are not both convex." << endl;
151   }
```

Computing the area of overlap and symmetric difference have been done in the previous section, and computing the centroid of a polygon has been done in Section 4.2.1. In order to transform polygon *B* such that its centroid is moved to the centroid of *A*, we first subtract the two centroids to obtain the translation vector, see line 135. The next line instantiates an object of type *Transformation_2*, which implements an affine transformation. The constructor *transl(TRANSLATION, vec)* tells that the transformation object *transl* is a translation over vector *vec*. Other special transformation can be constructed similarly; to define an arbitrary affine transformation, all the transformation matrix coefficients can be given explicitly. The transformation object has a function operator *Transformation_2()* for CGAL kernel primitives such as *Point_2*, so that a transformation *t* can be applied to a point *p* as *t(p)*. However, there is no function operator *()* defined for a polygon, which is not a kernel type. Instead, there is a global function *transform()* which returns the image of the input polygon under the transformation, see line 138. The centroids are computed, and the polygons are translated over the difference vector. Before and after the translation, the areas of overlap and symmetric difference are computed and printed on standard output.

At the end of the program, the convexity of the polygons is tested (line 145), using the member function *is_convex()*. Surprisingly, for convex polygons, the resulting area of overlap is at least 9/25 of the maximum [Berg & al. 97], and the area of symmetric difference is at most 11/4 of the minimum [Alt & al. 96]!

Chapter 6

Triangulations

6.1 Introduction

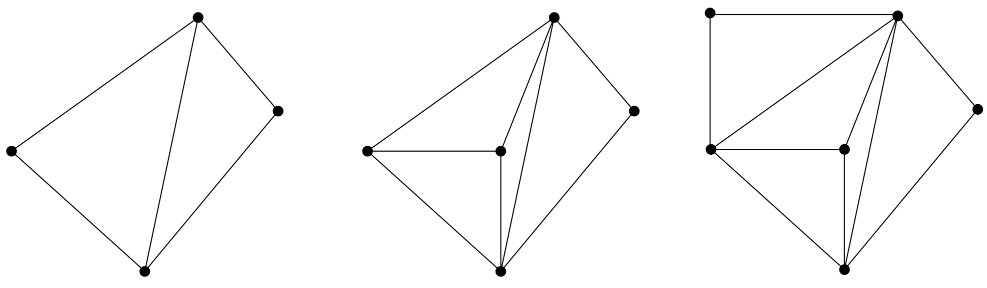
Apart from elementary geometric objects such as points and lines from the kernel, CGAL offers a number of geometric datastructures, for example polygons (introduced in the previous chapter) and triangulations. Triangulations of point sets are used in many areas, for example numerical analysis (finite elements), computer aided design (meshes), and geographic information systems (triangulated irregular networks). In this chapter we will consider triangulations in the plane.

6.2 Construction

Example file: examples/Getting_started/triangulation1.C

Given a triangulation *tr*, a point *pt* can be inserted with *tr.insert(pt)*. If point *pt* coincides with an already existing vertex, the triangulation is not changed. If it lies on an edge, the two adjacent faces are split into two new faces. If it lies inside a face, the face is split into three new faces. And if it lies outside the current triangulation, *pt* is connected to the visible points on the convex hull in order to form new faces. This is illustrated in Figure 6.1. The order of insertion is important, because edges are not removed.

A whole range of points can be inserted with *tr.insert(first,last)*, where *first* and *last* are input iterators. This is illustrated in the program below. First an empty triangulation is created in line 25. Then *points1*, an array of points with length *numPoints1* is inserted. The iterator *points1* refers to the first, and *points1+numPoints* points past the last element of the array. The corresponding range *[points1, points1+numPoints)* is inserted into



Triangulation of 4 points | Extra interior point | Extra exterior point

Figure 6.1: Triangulations.

the triangulation with *tr.insert(points1, points1+numPoints)*, see line 25. This results in the left triangulation of Figure 6.1. Line 26 inserts a single point, which happens to be an internal point, resulting in the middle triangulation of Figure 6.1. Line 27 inserts an exterior point, giving the right triangulation of Figure 6.1. Finally, an STL vector *points4* of points is inserted. An STL vector is a sequence container that linearly stores objects of a single type, see Chapter 4. The first element of *point4* is pointed to by the iterator *point4.begin()* and the past-the-end iterator is *point4.end()*.

```
1   #include "tutorial.h"
2   #include <CGAL/Point_2.h>
3   #include <CGAL/Triangulation_2.h>
4   #include <vector>
5
6
7   main()
8   {
9     const int numPoints1 = 4;
10
11    static Point points1[numPoints1] = {
12      Point(0.4, 1),
13      Point(1, 0.3),
14      Point(0.0, -0.9),
15      Point(-1, 0)
16      };
17
18    Point point2(0.0, 0.0);
19    Point point3(-1,1);
20
21    std::vector<Point> points4(3);
22    points4[0] = Point(1, 0.9);
23    points4[1] = Point(1.4, -0.3);
24    points4[2] = Point(0.6, 0);
25
26    Triangulation tr;                           // create an empty triangulation
27
28    tr.insert(points1, points1+numPoints1);       // insert array of Point-s
29    tr.insert(point2);                            // insert interior Point
30    tr.insert(point3);                            // insert exterior Point
31    tr.insert(points4.begin(),points4.end());     // insert vector of Point-s
32
33    std::cout << tr;
34
35
36  }
```

6.3 Access

Vertices

A triangulation is stored as a collection of vertices and faces. A vertex is an object type that is defined locally within the triangulation class: *Triangulation_2::Vertex*. Vertices are created automatically when a point is inserted. The point associated with a vertex *v* can be accessed with the method *v.point()*, which returns a *Point_2*. Each triangulation has a special vertex 'at infinity', see Figure 6.2.

All the finite vertices can be accessed through a vertex iterator, defined within the triangulation class: *Triangulation_2::Vertex_iterator*. The value type of a vertex iterator is *Vertex*, i.e. dereferencing a vertex

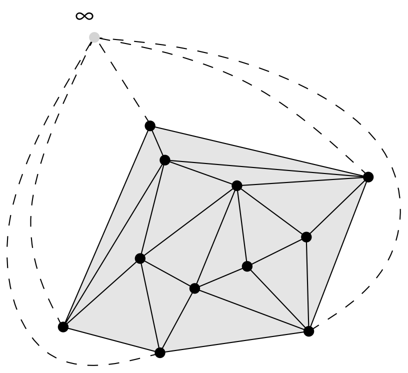


Figure 6.2: Vertex at infinity.

iterator yields a vertex. The method *vertices_begin()* gives an iterator referring to the first vertex in the range, *vertices_end()* gives the past-the-end iterator of the range. As an example, the following piece of code prints all the points of all finite vertices in a triangulation *tr*.

```
// short hand type definitions
typedef Triangulation_2::Vertex Vertex;
typedef Triangulation_2::Vertex_iterator Vertex_iterator;

Vertex v;                                         // pointer to vertex
Vertex_iterator it = tr.vertices_begin(),        // begin iterator
                     beyond = tr.vertices_end(); // past-the-end iterator

while(it != beyond) {
  v = *it;                                        // access vertex
  ++it;                                           // advance the iterator
  cout << v.point() << endl;                      // print vertex point
}
```

Faces

A face is an object type that is defined locally within the triangulation class: *Triangulation_2::Face*. Faces are created automatically when a point is inserted. A face f has three vertices: *f.vertex(0)*, *f.vertex(1)*, and *f.vertex(2)*. These methods return a *Vertex_handle*, a sort of pointer to the vertex. Dereferencing the handle yields the vertex itself. Conversely, if v is a handle of a vertex of f, then *f.index(v)* returns the vertex index of v in f.

The vertices with indices 0, 1, 2 are ordered counterclockwise. In order to facilitate taking the next vertex in counterclockwise or clockwise order, faces have the member functions *ccw(i)* which returns $i+1$ modulo 3, and *cw(i)*, which returns $i+2$ modulo 3.

Each face has three neighbors: *f.neighbor(0)*, *f.neighbor(1)*, and *f.neighbor(2)*. These methods return a *Face_handle*, a sort of pointer to the face. Conversely, if *nb* is a handle of a neighboring face of f, then *f.index(nb)* returns the index of *nb* in f, either 0, 1, or 2. The neighbor with index i is always opposite the vertex with index i, see Figure 6.3.

Note that *each* face has three neighbors. An interior triangle with an edge on the convex hull has a neighboring face that has an infinite vertex, and lies outside the convex hull. To test if a handle v refers to an infinite vertex, the triangulation has the method *is_infinite(v)*. Similarly, *is_infinite(f)* tests if a face handle f refers to a face with an infinite vertex.

Analogous to the vertex iterator, there is a face iterator to address all finite faces: *Triangulation_2::Face_iterator*. As an example, the following piece of code looks at all faces, inspect all three neighbors, and prints the number of infinite neighbors.

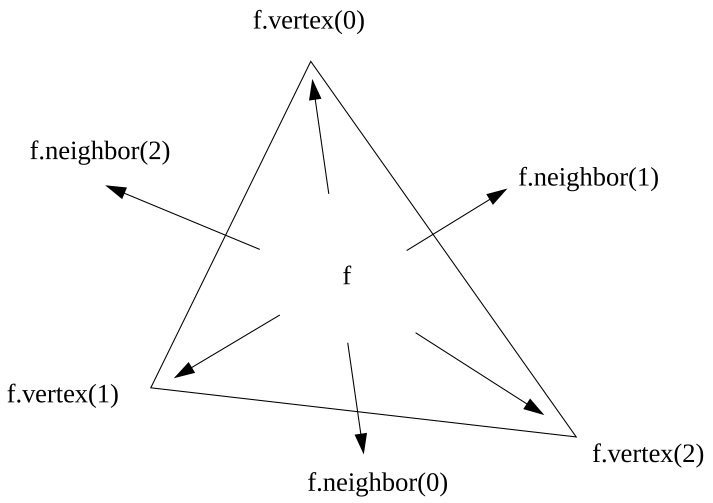


Figure 6.3: Relation between indices, vertices, and neighbors of face *f*.

```
// short hand type definitions
typedef Triangulation_2::Face_iterator Face_iterator;
typedef Triangulation_2::Face Face;
typedef Face::Face_handle Face_handle;

Face_iterator it = tr.faces_begin(),
                  beyond = tr.faces_end();
Face face;
Face_handle neighbor;

while (it != beyond) {
    face = *it;                                 // get face
    ++it;                                       // advance the iterator
    int count=0;                                // initialize counter
    for (int i=0; i<3; ++i) {                   // for index 0,1,2
       neighbor = face.neighbor(i);             // get neighbor
       if (tr.is_infinite(neighbor)) {          // test its infinity
          ++count;
       }
    }
    cout << tr.triangle(face) << endl
          << " has " << count << " infinite neighbor(s)" << endl;
}
```

Edges

Note that edges are not explicitly stored in the triangulation. However, an edge is identified by a pair of a face handle and an index, where the index denotes the edge between the face and the neighbor of that index, see Figure 6.4. The triangulation provides an iterator for edges, The value type of the iterator is *Edge*, i.e. dereferencing an edge iterator yields an object of type *Edge*. In fact an edge is an STL pair of a face handle and an integer: *pair*<*Face_handle*, *int*>. Given an object *edge* of type *Edge*, the face handle is accessed with *edge.first*, and the index value with *edge.second*.

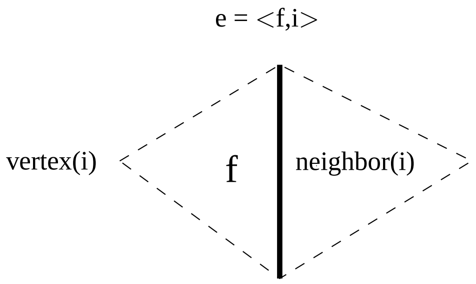


Figure 6.4: Relation between edge *e*, face handle *f*, and index *i*.

6.4 Putting it all together

Example file: examples/Getting_started/triangulation2.C

In the following example program, we construct a triangulation, and test for all edges with vertices *edgev1* and *edgev2* if the opposite vertices lie within the smallest circle passing through the edge vertices. The triangulation is constructed with random points, generated by the random point generator declared in line 18. Points are successively taken from the generator in line 23, and inserted into the triangulation.

The edges of the resulting triangulation are accessed through the begin and past-the-end edge iterators declared in lines 26 and 27. In the while loop the iterator is dereferenced, giving an *edge*. An edge is a pair of a face handle and an index. The handle (*face*) is taken as the first element from the pair (line 33), the second element is the index (*nbIndex*) of the neighbor (*neighbor*) that is the other adjacent face (line 35). This index is also the index of the opposite vertex (line 41), so the edge vertices have indices *face->cw(nbIndex)* and *face->ccw(nbIndex)*. The neighbor index of *neighbor* referring to *face* is *neighbor->index(face)*, so the second opposite vertex is the vertex with the same index (line 42), see also Figure 6.5.

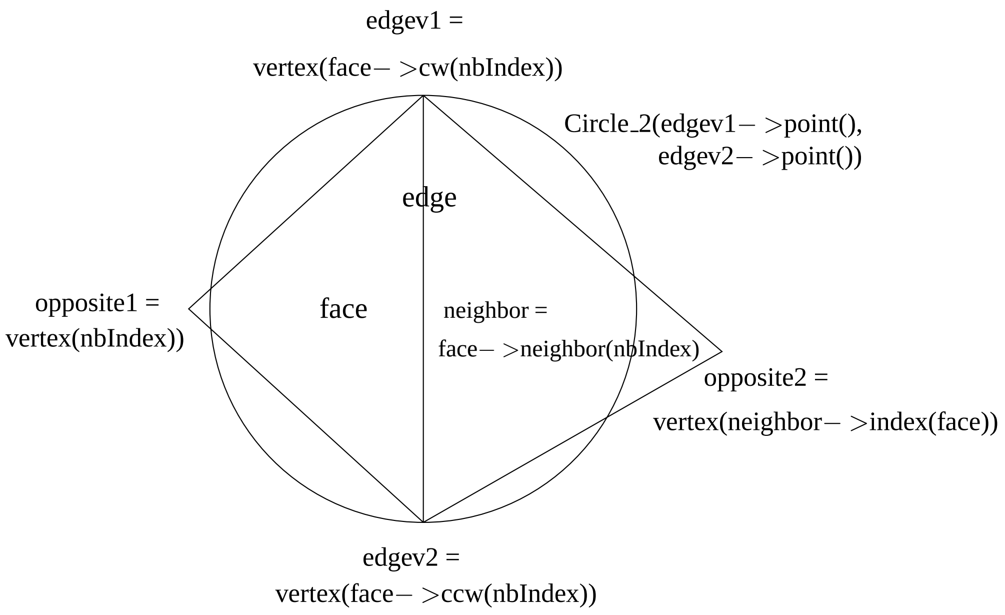


Figure 6.5: Relation between faces, neighbor, vertices, and indices in the example program.

```
#include "tutorial.h"
#include <CGAL/Point_2.h>
#include <CGAL/Circle_2.h>
#include <CGAL/Triangulation_2.h>
#include <CGAL/point_generators_2.h>
#include <iostream>

typedef Triangulation::Edge_iterator  Edge_iterator;
typedef Triangulation::Face  Face;
typedef Triangulation::Edge  Edge;
typedef Triangulation::Vertex  Vertex;
typedef Face::Face_handle Face_handle;
typedef Vertex::Vertex_handle Vertex_handle;

main()
{
    const int numPoints = 50;
    CGAL::Random_points_in_square_2<Point> g(100.0);  // random points generator
    Triangulation tr;                                // empty triangulation

    // construct a triangulation
    for (int i=0; i<numPoints; ++i) {
        tr.insert( *g++ );                             // take next point from generator
    }

    Edge_iterator it = tr.edges_begin(),               // initialize with begin value
                      beyond = tr.edges_end();         // past the end value

    while (it != beyond) {
       Edge edge = *it;                                // take ‘edge’
       ++it;                                           // advance iterator

       Face_handle face = edge.first;                  // take the face
       int nbIndex = edge.second;
       Face_handle neighbor = face->neighbor(nbIndex); // take neighbor

       Vertex_handle edgev1 = face->vertex(face->cw(nbIndex));  // edges vertices
       Vertex_handle edgev2 = face->vertex(face->ccw(nbIndex));

       // two opposite vertices of adjacent face
       Vertex_handle opposite1 = face->vertex(nbIndex);
       Vertex_handle opposite2 = neighbor->vertex(neighbor->index(face));

       // smallest circle through edge vertices
       Circle circle(edgev1->point(), edgev2->point());

       if ( ! tr.is_infinite(opposite1) ) {            // opposite1 infinite?
          if (circle.has_on_bounded_side(opposite1->point()) ) {
             // opposite vertex 1 lies inside circle, continue with next edge
             continue;
          }
       }
       if ( ! tr.is_infinite(opposite2)) {             // opposite2 infinite?
          if (circle.has_on_bounded_side(opposite2->point()) ) {
                // opposite vertex 2 inside circle, continue with next edge
                continue;
            }
        }

        // opposite vertex 1 and 2 not inside circle, output edge as segment
        std::cout << Segment(edgev1->point(), edgev2->point())
                  << std::endl;
    }
}
```

6.5 Delaunay triangulation

The successive insertion can result in very elongated triangles. To avoid this we want the minimum angle over all the triangles to be not too small. A triangulation that *maximizes* the minimum angle is called a Delaunay triangulation. It has the property that the circumscribing circle of all faces is empty, i.e. none of the vertices lies strictly inside that circle. If no four points lie on an empty circumscribing circle, the Delaunay triangulation is unique, see [de Berg & al. 97].

The *Delaunay_triangulation_2* inherits from *Triangulation_2*, and redefines the insert and remove functions. When a new point is inserted, the face containing that point is not just split, but the triangulation is modified so as to satisfy the max-min angle property, see Figure 6.6. As long as no four points lie on an empty circle, the order of insertion does not matter.

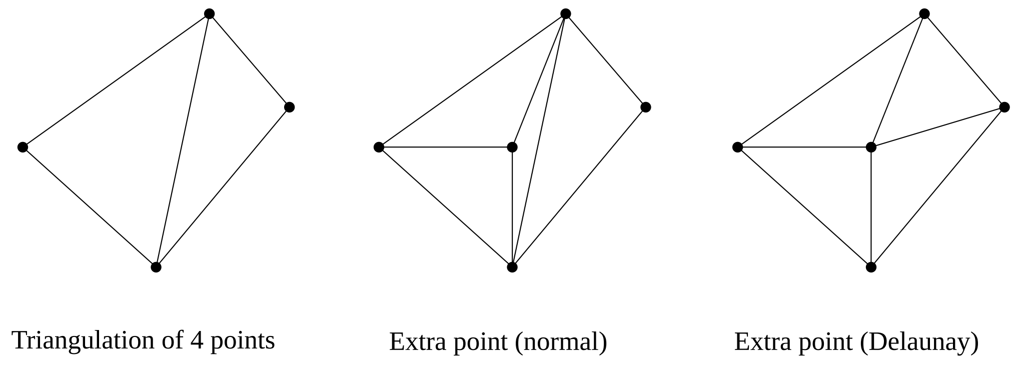


Figure 6.6: Difference between insertion into a normal and a Delaunay triangulation.

6.6 Using your own point type

Rather than using the 2D CGAL point type, you can use your own point type, provided that it fulfills certain requirements. The geometric datastructures and algorithms in the basic library are in fact parameterized by a ‘traits class’, which defines an interface to the geometric primitives. For example, the triangulation needs a traits class that defines an interface to points. There is a predefined triangulation traits class to the 2D CGAL point, so nothing special had to be done in the examples above. However, to use your own point type, you must provide your own triangulation traits class that fulfills certain requirements. Chapter 8 explains how this works for convex hull algorithms; for the specific requirements for triangulations, see the CGAL Reference Manual.

Chapter 7

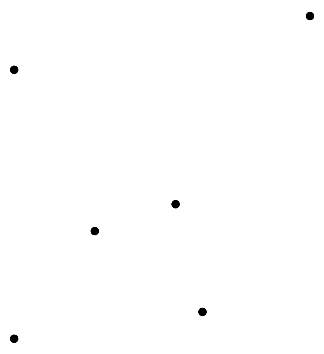
Convex Hulls

The convex hull is one of the most familiar concepts in computational geometry. Nevertheless, at the risk of being boring, we'll give a definition of a convex object and of a convex hull.

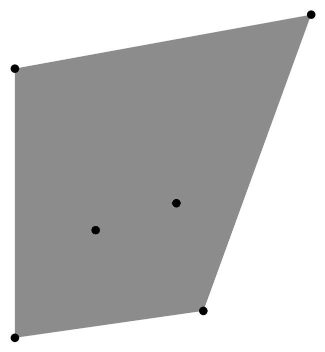
Definition 7.1 *An object O is convex if for any two points p1 and p2 that are part of O, all points on the line segment between p1 and p2 are also part of O.*

Definition 7.2 *The convex hull of a set of objects is the smallest convex object that contains all objects of the set.*

Those definitions hold in arbitrary dimensions and for arbitrary objects. In this chapter we have a look at a special case, the convex hull of a set of points in the plane. Below we see an example of the convex hull of six points.



a set of points



the convex hull

This chapter describes the basic way to use the convex hull routines. A more advanced use is possible, giving more flexibility in the input. This is described in Chapter 8.

7.1 An example program

Example file: examples/Getting_started/convex_hull.C

The following program shows how we can compute the convex hull of those six points in CGAL. The program has more than thirty lines, but most of them have to do with defining the points and writing them to standard output. Only two are directly related to computing the convex hull. In the second line we include the header file where the convex hull algorithms are declared. In the second line of the main procedure we call the convex hull function. This function takes a set of points as input and computes the convex polygon that is the convex hull of those points.

The convex hull function, in its simplest form, takes three parameters. The first two are input iterators that are the begin and end iterator of the sequence of input points. The third parameter is an output iterator. The resulting convex hull polygon is represented by its sequence of vertices in counterclockwise order, not as an object of type *Polygon_2*. The output iterator tells the function where it should write those vertices to. After each write, the function increments the output iterator.

```
#include "tutorial.h"
#include <CGAL/convex_hull_2.h>
#include <list.h>
#include <iostream.h>

typedef list<Point_2> pointlist;

const int IN_COUNT = 6;

static Point_2 in[IN_COUNT] = {
    Point_2(0, 0),
    Point_2(3, 4),
    Point_2(0, 10),
    Point_2(11, 12),
    Point_2(7, 1),
    Point_2(6, 5),
};

template <class iterator>
void write(iterator cur, iterator beyond)
{
    for (; cur != beyond; ++cur)
        cout << *cur << '\n';
}

void main()
{
    pointlist out;
    CGAL::convex_hull_points_2(in, in+IN_COUNT, back_inserter(out));
    write(out.begin(), out.end());
}
```

7.1.1 Output in a static array

The convex hull function also has a return value. This is the value of the output iterator that was passed as third parameter just before the function returns, after all increments are done. We did not need this value in the previous program. There we started with an empty list and the back inserter of the list took care that all elements were appended to the list. So the whole list was the result.

Now we will do it differently. Instead of writing to a list, we will write the result to an array. Now we must take care: we must use an array big enough to hold the result. In the worst case all input points lie on the convex hull. So, we'll take an array of the same size as the input array. The return value of the function now gives us a pointer to the first position that was not filled.

We can substitute the main function of the previous function by the one below.

```
void main()
{
    Point_2 out[IN_COUNT], *beyond;
    beyond = CGAL::convex_hull_points_2(in, in+IN_COUNT, out);
    write(out, beyond);
}
```

Chapter 8

Traits classes in CGAL

In this chapter we revisit the convex hull algorithm and go into some details that were skipped before. We deal with an extra parameter that can be passed to the convex hull algorithm. This parameter, a so called traits class, makes it possible to use the convex hull algorithm in a much more flexible manner. In particular, we can let it operate on other types of points than just two dimensional CGAL points.

The traits class is a class that brings together all data types and operations that are needed for the computation of the convex hull. That is, for all its geometrical computations, the convex hull algorithm relies only on the types and operations as declared in the traits class. If we pass no traits class, a default traits class is taken which defines the point type to be a two dimensional CGAL point and specifies some kernel routines that should be used for various operations. In this chapter we shall see how we can compute the convex hull of a different point type. To this end we will have to define our own traits class.

Traits classes are a technique that are widely applied in CGAL. Usually, the algorithms come with a default traits class implementation that is used automatically if plain CGAL data types are used. There may be some more predefined traits class implementations, which can be used as an alternative. Finally, the documentation of a class (or function) gives the precise requirements that every traits class implementation must obey.

Writing your own traits classes is an advanced topic. Still, because it is a powerful mechanism which is widely used in CGAL, we want to explain something about it. You can skip this chapter on first reading or just skim it over. Just remember that most CGAL algorithms can be used in a flexible manner. See [Myers 95] for more detailed information about traits classes.

8.1 Our problem

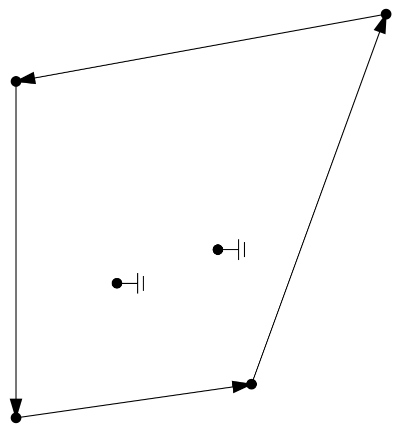
Suppose, we want to store the convex hull of a set of points. A possible way to do this is to add a field to the points which is a pointer to a point. If the point does not lie on the convex hull, the pointer is 0 (NULL). Otherwise, it points to the next point (counterclockwise) on the convex hull. The figure illustrates the this. So, we can have a struct like the following[1]

```
struct Special_point {
    Special_point() {}
    Special_point(int x, int y) : pt(x, y), next_on_hull(0) {}
    CGAL::Point_2<Rep_class> pt;
    Special_point *next_on_hull;
};
```

The input to our problem will be an array of those points.

Now the question is, how can we use our convex hull algorithm to compute the convex hull of those special points. And also, how can we use the result of the computation to fill in the pointers.

[1]Another possibility is to inherit from *CGAL::Point_2<Rep_class>*. You can rewrite this example in such a way as an exercise.



connected convex hull points

8.2 Our own traits class

The answer lies in using the traits class parameter. The convex hull can use any type as point type, as long as accompanying operations are supplied as well. In the reference manual a precise list of the requirements is given. It is a good idea to have a look at this list while you are reading this.

The first thing that the traits class should define is the point type. The first idea would be to use *Special_point* here. This would solve our first goal, computing the convex hull of the special points, but not the second –fill in the pointers in the point. A better idea is to use pointers to the *Special_point*s.[2] Then the result will also be a list of pointers to the original points, which we can use to fill in the internal pointers.

Now let's turn our attention to the operations that our traits class must support. The traits-class-requirements section gives a list of traits class operations and types. The documentation of the convex hull tells us which of those are really used.[3] Our complete traits class declaration looks as follows.

```
struct Special_point_traits {
    typedef Special_point * Point_2;
    typedef Special_less_xy Less_xy;
    typedef Special_less_yx Less_yx;
    typedef Special_right_of_line Right_of_line;
    typedef Special_leftturn Leftturn;
    Less_xy get_less_xy_object() const
      { return Less_xy(); }
    Less_yx get_less_yx_object() const
      { return Less_yx(); }
    Right_of_line get_right_of_line_object(
                      const Point_2& p, const Point_2& q) const
      { return Right_of_line( p, q); }
    Leftturn get_leftturn_object() const
      { return Leftturn(); }
    Special_point_traits() {}
};
```

Of course, we are not ready yet. The traits class uses a lot of typedefs. Except the first one, they are defined in terms of unknown types. As we will see below, those types are function objects, that is, types with an *operator()*

[2]If the special points were stored in a different container than an array, we could have used the iterator type of that container instead of pointers.

[3]The full list serves also for other convex hull algorithm implementations. That is why it contains superfluous elements for our purpose.

defined. Every function object typedef is accompanied by a function that constructs such an object. That are the four *get_* member functions.

8.3 Implementation of the traits class

All the functionality in the interface must be implemented by means of function objects. Function objects are widely used in STL, but we'll explain something about them here. A function object is a type with an operator() defined. For instance, we will define a class *Special_less_xy* below. This class has an operator() which takes two pointers to *Special_point* as parameters and returns a bool. This is a requirement that is defined in the traits class requirements for convex hulls. In order to implement this operator, we can use again a function from the CGAL kernel.

```
22  struct Special_less_xy {
23      bool operator()(Special_point *p, Special_point *q) const
24          { return CGAL::lexicographically_xy_smaller(p->pt, q->pt); }
25  };
```

The following function object is implemented in the same way. The function object *Special_right_of_line* deserves more attention. This function object has an operator() that takes one *Special_point* pointer as argument and returns if this point lies to the right of a line. This line is specified at construction time of the objectj by two point parameters.

The implementation makes use of a function object, *CGAL::r_Right_of_line* in line 38, which is a predicate object in the CGAL kernel (this type is parametrised by the representation class).[4] So, here we do not only define a function object, we also see how to use it. In the constructor of *Special_right_of_line* in line 33 and 34 we call the constructor of *CGAL::r_Right_of_line*. In the operator() function of *Special_right_of_line* in line 35 and 36 we call the operator() of *CGAL::r_Right_of_line* (*rol(r->pt)*). All those different meanings of parentheses may be confusing at first sight, but there is positive evidence that people can get used to it.

```
32  struct Special_right_of_line {
33      Special_right_of_line(Special_point *p, Special_point *q)
34                      : rol(p->pt, q->pt) {}
35      bool operator()(Special_point *r) const
36                      { return rol(r->pt);}
37  private:
38      CGAL::r_Right_of_line<Rep_class> rol;
39  };
```

Finally, the last function object defines an operator() function that decides if three points define a left turn or something else. The CGAL kernel has a predicate that computes just this. All we need to do is call this function with the CGAL points stored inside the *Special_point* to which the pointers refer.

```
41  struct Special_leftturn {
42      bool operator()(Special_point *p, Special_point *q, Special_point *r)const
43          { return CGAL::leftturn(p->pt, q->pt, r->pt); }
44  };
```

8.4 The complete program

Example file: examples/Getting_started/advanced_hull.C

[4] Function objects in the kernel are new, undocumented and still subject to change. So, the precise way of doing this is not stable, but the technique of using a CGAL object to implement your own object will remain.

By now we have seen the most important parts of the program. Below we give the full listing. There are still a lot of details that we did not give in the preceding discussion. Those details do not bring anything new.

We compute with longs, a choice only meant for tu-toy-rial programs (where general availability and ease of compilation is more important than correctness). In line 73 we choose to store the pointers to the points in an STL vector (this is a rather arbitrary choice). The linking of the convex hull points in a chain was not described. This is done in lines 77 to 97. It is not very interesting and documented in comments only.

The main program initialises the vector of pointers first (line 101–105). Then comes the call of the convex hull routine. The fourth argument is the interesting one. It is an object of class *Special_point_traits*, which is created by calling the default constructor. After that the internal pointers of the points are set and the points on the convex hull are printed to standard output.

```
 1  #include <CGAL/Homogeneous.h>
 2  #include <CGAL/convex_hull_2.h>
 3  #include <CGAL/Point_2.h>
 4  #include <CGAL/predicates_on_points_2.h>
 5  #include <CGAL/predicate_objects_on_points_2.h>
 6  #include <vector.h>
 7  #include <iostream.h>
 8
 9  typedef CGAL::Homogeneous<long> Rep_class;
10
11  typedef CGAL::Point_2<Rep_class> Point_2;
12
13  const int IN_COUNT = 6;
14
15  struct Special_point {
16      Special_point() {}
17      Special_point(int x, int y) : pt(x, y), next_on_hull(0) {}
18      CGAL::Point_2<Rep_class> pt;
19      Special_point *next_on_hull;
20  };
21
22  struct Special_less_xy {
23      bool operator()(Special_point *p, Special_point *q) const
24          { return CGAL::lexicographically_xy_smaller(p->pt, q->pt); }
25  };
26
27  struct Special_less_yx {
28      bool operator()(Special_point *p, Special_point *q) const
29          { return CGAL::lexicographically_yx_smaller(p->pt, q->pt); }
30  };
31
32  struct Special_right_of_line {
33      Special_right_of_line(Special_point *p, Special_point *q)
34                      :rol(p->pt, q->pt) {}
35      bool operator()(Special_point *r) const
36                      { return rol(r->pt);}
37  private:
38      CGAL::r_Right_of_line<Rep_class> rol;
39  };
40
41  struct Special_leftturn {
42      bool operator()(Special_point *p, Special_point *q, Special_point *r)const
43          { return CGAL::leftturn(p->pt, q->pt, r->pt); }
};

struct Special_point_traits {
    typedef Special_point * Point_2;
    typedef Special_less_xy Less_xy;
    typedef Special_less_yx Less_yx;
    typedef Special_right_of_line Right_of_line;
    typedef Special_leftturn Leftturn;
    Less_xy get_less_xy_object() const
      { return Less_xy(); }
    Less_yx get_less_yx_object() const
      { return Less_yx(); }
    Right_of_line get_right_of_line_object(
                      const Point_2& p, const Point_2& q) const
      { return Right_of_line( p, q); }
    Leftturn get_leftturn_object() const
      { return Leftturn(); }
    Special_point_traits() {}
};

static Special_point in[IN_COUNT] = {
    Special_point(0, 0),
    Special_point(3, 4),
    Special_point(0, 10),
    Special_point(11, 12),
    Special_point(7, 1),
    Special_point(6, 5),
};

typedef vector<Special_point *> Pointer_collection;

// Link the points of the convex hull together, given a vector of pointers
// to the points on the convex hull.
Special_point *
link(Pointer_collection &c)
{
    Pointer_collection::iterator cur, prev;
    cur = c.begin();
// return NULL if there are no points.
    if (cur == c.end())
        return 0;
// prev and cur iterate over all CH points. prev lags one behind.
// Every time we set the next pointer in prev to cur.
    prev = cur;
    ++ cur;
    while (cur != c.end()) {
        (*prev)->next_on_hull = *cur;
        prev = cur;
        ++cur;
    }
// Close the chain.
    (*prev)->next_on_hull = c.front();
    return *prev;
}
```

```

void main()
{
// Initialise a vector with pointers to the input points.
    Pointer_collection pointers(IN_COUNT), out;
    for (int i=0; i<IN_COUNT; i++)
        pointers[i] = in+i;

// Compute the convex hull of the pointers.
    CGAL::convex_hull_points_2(
                pointers.begin(),
                pointers.end(),
                back_inserter(out),
                Special_point_traits());

// Link the points of the convex hull together.
    Special_point *first, *cur;
    cur = first = link(out);

// Print all points of the convex hull.
    if (first != 0)
        do {
            cout << cur->pt << '\n';
            cur = cur->next_on_hull;
        } while (cur != first);
}
```

Appendix A

A Short Introduction to C++

This chapter gives a short introduction to some of the features that C++ added to plain C. Of course, you cannot expect to get a complete overview of a language as complex as C++ in a few pages. Consider this introduction as a basic cookbook that tells you how to warm up some precooked dishes. If you know your way in a supermarket, this is enough to keep you alive, not to keep you satisfied. If you want to cook your own dish, you'll have to consult a book. See for example [Stroustrup 97], [Lippman 98], [Musser & al. 96], and [Myers 95].

If you are familiar with C++, you will want to skip this chapter. This is what you'll miss. The first section explains the class concept. The next section explains various things: overloading, reference parameters, new and delete, iostreams and templates. The last section deals with the list and vector data types of the C++ library.

A.1 The Use of C++ Classes

The most important feature that is added in C++ is the *class*. A class can be seen as a user defined type. As such the class resembles a struct. An important difference is that a class, apart from being a repository for data, may also contain a number of operations on those data. Those operations are called the *member functions*. We also use the term *methods* as an equivalent.

A class can hide some data and operations from the users of the class, thus making it possible to separate implementation and interface. Another aspect of classes is inheritance. In this case a class (called a derived class) is built with the help of one or more other classes (base classes). That is, the data and operations of the base classes are also available in the derived class. Everywhere where a base class is expected as parameter, a derived class may be provided instead. Inheritance is not much used in CGAL.

A.1.1 Example of a class

To make it somewhat clearer how classes are described and how they can be used, we give a description of the interface of a class here.

```
class int_stack {
  public:
                int_stack() ;
                int_stack(int_list il) ;
    bool        is_empty() const ;
    int         top() const ;
    void        push(int i) ;
    void        pop() ;
};
```

The class is named *int_stack* and models the well known stack of integers. On the first two lines there are two member functions with the same name as the class. Those special functions are called *constructors* and play a part in the declaration of variables of this class. The next four lines contain normal member functions. The type *bool* is a boolean type with values *true* and *false*. It is a built-in type like *char* or *int*, although that is not yet supported by all compilers. The type *int_list* is supposed to be defined elsewhere.

Constructors give the possibility to initialise an object during declaration. In this case there are two constructors. This gives the following possibilities to declare a variable of type *int_stack*, where we assume that *ilist* is a variable of type *int_list*.

```
int_stack s1;
int_stack s2(ilist);
```

In the first declaration the first constructor is used. There are no parameters; the stack will start empty. In the second declaration a list is given as parameter. This could be a list of initial values of the stack. Note that when no parameters are required, no parentheses should be used. Those two ways of declaring can be compared to:

```
int i1;
int i2 = 42;
```

Except for the two explicitly mentioned constructors, a third one can be used:

```
int_stack s3(s2);
```

This constructor initialises *s3* as a copy of *s2*. In C++ such a copy constructor exists for each class, that is why it is not mentioned explicitly in the class descriptions.

After the declaration of an object, the member functions for that object can be called. A member function is called for an object by placing the function name behind the object name, separated by a dot. For instance:

```
if (!s2.is_empty())
    s1.push( s2.top() );
```

Here we check if the second stack contains an element and, if so, put the top element also on stack *s1*.

Now let's look more closely to the declarations of the member functions. The methods *is_empty* and *top* take no parameters. They return whether the stack is empty and the value of the top element respectively. The keyword *const* after the declarations means that the object (s2 in the example) does not change because of this call.[1]

The method *push* puts an integer on the top of the stack. The method *pop* removes the top value from the stack. Note that those declarations are not followed by *const*, because in this case the stack is altered.

Objects of a class can be assigned to variables of that class. Because this is always possible, this is not mentioned for every individual class.

```
s1 = s2;
s2 = int_stack(ilist);
```

In the second line a constructor is used in the same way as a constant of a standard type (compare: *i2 = 42;*). In fact, a temporary object of type *int_stack* is created and assigned to *s2*. The syntax for this use of a constructor differs from the syntax used for declaration; in the former case the arguments follow the class name, in the latter they follow the variable name.

What apparently is missing in the class definition, is a description of how the data are stored. The implementor of the class must decide how to do this, but for the user of the class this is not interesting. C++ has a mechanism to hide those details from the user of a class. Users of a class may only access a class via its public interface. The class may have some extra, private members, which are only accessible to himself. In general, we do not document those private members. So, in the *int_stack* case, we do not document how we store the data. Of course, if you want to define your own classes, you have to know about this. But this is beyond the scope of this introduction.

[1]Actually, the object may change internally, but this is invisible when using only the public interface to the class. This allows for caching things for more complicated objects.

A.2 Various aspects of C++

A.2.1 Overloading

In C, a program can not have two functions with the same name. C++ permits this, if the parameter lists of the functions differ. This is called overloading. Except functions, also operators may be defined. This is a special case of overloading. Suppose that, for some strange reason, we want to write a routine that multiplies every value of an *int_stack* with a constant. We could redefine the operator *= for this purpose, as it looks like the whole stack is multiplied with an integer. This operator can now be used in the following way:

```
s1 *= 3;
```

The precedence of operators cannot be changed, neither can new operators be defined.

A.2.2 Reference parameters

In C, all arguments to functions are passed by value. If we want that a function changes an object, we should pass the address of the object. Although this is still possible in C++, there also is another mechanism: reference parameters. A parameter of a function is passed by reference if its name is preceded by an ampersand.

```
void remove_one(int_stack &param)
{
    if (!param.empty())
        param.pop();
}

void foo()
{
    int_stack s;
    s.push(3);
    remove_one(s);
}
```

In this example, the function *remove_one* changes the stack *s*, so after the call in *foo*, the stack is empty.

We already saw the usage of the keyword *const* in the definition of classes. This keyword is also quite often used when passing parameters, especially if they are passed by reference but we don't want the routine to change the argument. When a parameter is passed by const reference, the routine is only allowed to call methods that were marked const on those parameters. Otherwise an error will be reported at compilation time.

```
void remove_one(const int_stack &param)
{
    if (!param.empty())     // ok.
        param.pop();        // not allowed.
}
```

We rewrote the preceding example so that we pass the parameter by const reference. The first call in this fragment is still valid, as the method *empty* is marked const. The method *pop* is not, so the second call is not allowed in this context.

Call by reference is used a lot, because it is much cheaper to pass a parameter by reference than by value, if we want to pass a more complicated structure. Passing by value would mean copying the whole structure. For instance, if we consider the second constructor of the *int_stack* example we would normally have declared the second constructor as follows:

```
int_stack(const int_list &il) ;
```

A.2.3 New and delete

The use of malloc and free in C++ is not advisable. Instead there are the language constructs new and delete. There are two ways of using those constructs, depending on whether a single object is created or an array of objects. Here follows an example of the dynamic creation of several int_stacks:

```
    int_stack *is_pt;
    is_pt = new int_stack;
    delete is_pt;
    is_pt = new int_stack[11];
    delete [] is_pt;
```

Which can be compared with the C code:

```
    int_stack *is_pt;
    is_pt = (int_stack *) malloc(sizeof(int_stack));
    free(is_pt);
    is_pt = (int_stack *) malloc(11*sizeof(int_stack));
    free(is_pt);
```

As can be seen, the syntax of *new* is somewhat simpler than the syntax of malloc. More important though is that *new* always calls a constructor for every object that is created, so that every object is initialised in a valid state.

The two different forms of *delete* sometimes cause confusion. Remember: when square brackets are used with new, square brackets should be used with delete.[2] So, even when an object is created as

```
is_pt = new int_stack[1];
```

it should be deleted with

```
delete [] is_pt;
```

A.2.4 Namespaces

Namespaces are a mechanism for grouping declarations together in one scope. The user of CGAL will encounter two namespaces frequently: *std* and *CGAL*. The first holds all the functions, constants, variables and so on of the standard C++ library. Everything supplied by the CGAL library is in namespace *CGAL*.

Names that are in a namespace can be accessed in two ways. One is to qualify the name by the namespace that it belongs to. For instance, the name *cout* in namespace *std* may be referred to as *std::cout*. If a name is used often in a file, the repetition of the qualification may become tedious. In that case, a *using declaration* may be helpful. After declaring *using std::cout;*, the name *cout* may be used without qualification.

```
#include <iostream>  // this header declares cout in namespace std

std::cout; // ok, explicit qualification
cout;      // error. cout is in namespace std.
using std::cout;
cout;      // now ok.
```

A.2.5 C++ style IO

Example file: examples/Getting_started/basic_io.C
Example file: examples/Getting_started/file_io.C

[2]Except for freeing the space used by the object, delete also calls destructors for all objects. Destructors play a role when objects cease to exist. Because this happens transparently to the user, they are not treated here.

C++ defines a new model for doing input and output. A problem with C style IO is that it is not very well extendible. The new C++ model is based on streams. Characters stream out of input stream into your program, or they stream out of your program to output streams. A stream can be a file, standard input, standard output or other things.

In the example below we see how it is possible to read and write integers and floats. We read from standard input, which is always represented by the stream *cin*. We write to standard output, represented by the stream *cout*. There is a third stream, *cerr*, which represents the standard error output.

Values are read from a stream into variables. We separate the input stream and the variables with right shift operators. The direction of the arrows can be memorized by thinking of the data flowing from the stream to the variables. The number of variables is arbitrary. In the example below there are two: an integer and a float.

To write data to an output stream, the left shift operator should be put between the output stream and the data. Again, one or more data values can be written in a single line. In the example four types are written: a float, an integer, character strings and characters (end of line characters, in this case).

```
#include <iostream>

main()
{
    int i;
    float f;
    std::cin >> i >> f;
    std::cout << "The float read was: " << f << '\n';
    std::cout << "The int read was: " << i << '\n';
}
```

We can compile and run this example.

```
% CC basic_io.C -o basic_io
% basic_io
42 3.1842
```

The float read was: 3.1842
The int read was: 42

User defined types are usually read and written in the same way as built-in types (the writer of the class should provide the necessary routines). In the next example we suppose that in header file `Segment.h` a type *Segment* is declared, together with appropriate routines for input and output. In this case we read and write from file. In order to do this we need to include the header file `fstream.h`. A file from which we want to read should be declared as an ifstream. We can initialise an ifstream object with the name of the file. Likewise, a file to which we want to write must be declared as an ofstream and can be initialised with the file name.

In line 10 we check if the input stream is ok. If some error occurs during reading, an error flag is set in the stream. So, if the file segin contains no valid segment as first item, we skip the writing to segout.

```
 1   #include <fstream.h>
 2   #include <Segment.h>
 3
 4   main()
 5   {
 6       Segment seg;
 7       ifstream fin("segin");
 8       ofstream fout("segout");
 9       fin >> seg;
10       if (fin.good())
11           fout << seg;
12   }
```

Formatting.

A.2.6 Templates

In section A.1.1 we showed the interface for a stack of integers. Now, what would the interface for a stack of floats look like? A good guess is:

```
class float_stack {
                float_stack() ;
    bool        is_empty() const ;
    float       top() const ;
    void        push(float i) ;
    void        pop() ;
};
```

The two interfaces look very much alike. We had to invent a different name for the class (two classes with the same name are not allowed), the *top* member returns a float instead of an int and the *push* member expects a float as parameter. If we wanted to have stacks for more types, this same pattern would arise again. All those classes would have the same interface (and implementation), after substitution of the int type for another. It would be nice if we could write the code once.

C++ has a mechanism to deal with this: templates. A templated class is a class that can be parametrized with different types. Those types can be user defined (classes) or built-in (int, char, double ...). This is how we would write the interface declaration of the templated stack class. We write the name of the type parameter (T) between < > brackets.

```
class stack<T> {
                stack() ;
    bool        is_empty() const ;
    T           top() const ;
    void        push(T i) ;
    void        pop() ;
};
```

A templated class is not a complete type. You still have to fill in the dots, that is, choose a type for the type parameter(s). Here is how you could use such a templated stack class. The only place where you have to do something new is when you declare an object of a templated class. Here we show how to make and manipulate a stack of customers, where customer is supposed to be a user defined type.

```
class Customer { /* ... */ };

main()
{
    Customer c;
    stack<Customer> cstack;
    cstack.push(c);
    c = cstack.top();
    while (!cstack.is_empty())
        cstack.pop();
}
```

A.3 Lists and Vectors

Apart from a language, the C++ standard also mandates some libraries. Among these are libraries for some standard containers and algorithms on them. These libraries are known as the Standard Template Library (or STL). Although future compilers can be expected to ship with these libraries, currently this is not always so. Luckily, there are free implementations available[3] that work on most compilers. Here we describe two very common containers that are part of STL: lists and vectors.

[3]See URL `ftp://butler.hpl.hp.com/stl/` and URL `http://www.sgi.com/Technology/STL/`.

A.3.1 STL vectors

Example file: examples/Getting_started/vectorex1.C

The vectors of STL are much like the built-in array types. There are two major differences.

- The number of elements is part of the vector.
- A vector can be resized. Elements can be added and deleted.

First we compare the declaration of a vector to the declaration of an array. We declare a container of 10 elements of type T, where T can be a built-in type (like int) or a user defined type (some struct or class).

```
T t_array[10];
vector<T> t_vec(10);
```

There are two variants for declaring a vector. We can omit the number of elements, in which case we get a vector with zero elements. We can also add an argument of type T. Then all elements of the vector will be initialised with this value instead of with some default value.

```
vector<float> fvec1(3, 9.781);
vector<float> fvec2;
fvec2 = fvec1;
```

In the example above we also see that we can assign a vector to another vector. This means copying all elements. In this case, *fvec2* will also contain three elements with value 9.781 after the assignment. Note that assignment is not possible with built-in arrays.

Now we come to a more elaborate example. This example illustrates three new aspects of vectors.

- There is the method *push_back*, which appends an element to the vector.
- The method *size* returns the number of elements of the vector.
- Access to individual elements of a vector is done in the same way as access to elements of an array.

```
#include <vector>
#include <iostream>
using std::vector;

main()
{
    vector<float> fvec;
    fvec.push_back(3.14159265358979323846);
    fvec.push_back(2.7182818284590452354);
    vector<int> ivec(fvec.size());
    int i;
    for (i=0; i<ivec.size(); i++)
        ivec[i] = (int) fvec[i];
    std::cout << ivec[0] <<' '<< ivec[1] <<'\n';
}
```

A.3.2 STL lists

Example file: examples/Getting_started/listmanip.C

An STL list is a doubly connected list of elements. STL lists can contain elements. We can go forward and backward in a list and insert and erase elements anywhere in a list.

The declaration of lists resembles the declaration of vectors. Here are three declarations of list, one containing 7 *int* values, an empty list of a user defined type *T* and one containing 9 NULL pointers to type *T*.

```
#include <list>
using std::list;

struct T {/* anything here */};

list<int> ilist(7);
list<T> tlist;
list<T*> tplist(9, 0);
```

Now we come to the access aspect of lists. How do we refer to a particular position in the list? We need some kind of pointer when we move in the list or when we want to indicate what element must be erased or where we want to insert an element. The STL has a special concept for this sort of thing: iterators. Iterators are a generalisation of pointers. That is, a lot of operations can be performed on iterators that are syntactically and semantically similar to operations on pointers. Every data structure of STL has an associated iterator type. In the following example we will highlight the most common usage of both lists and iterators.

First, let's define a function that removes all the occurrences of the value 2 from a list of integers.

```
#include <list>
using std::list;
void remove_2(list<int> &l)
{
    typename list<int>::iterator x, cur(l.begin());
    while (cur != l.end()) {
        if (*cur == 2) {
            x = cur;
            ++cur;
            l.erase(x);
        } else {
            ++cur;
        }
    }
}
```

The list is passed by reference to the function. This implies that the changes that occur in this routine affect the list that was passed as argument, not just a copy of it. In the function we first declare two iterators, *x* and *cur*. The syntax for getting the type of the iterator that is associated with the list of ints may seem strange at first sight:

```
list<int>::iterator
```

The double colon indicates that the type *iterator* is defined inside the (parametrised) class *list<int>*. The variable *cur* is initialised at declaration with the begin iterator of the list. The member function *begin* returns an iterator which points to the first element of the list. Also, there is a member function *end()* which returns an iterator that points just beyond the end of the list. Here, the iterator *cur* will pass over the list, from the first position until it has passed the last element. We step forward by doing ++*cur*. At every step, we check if the value to which the iterator points (**cur*) is equal to 2. If this is the case, we save the iterator in a temporary variable, advance the iterator and remove the retained element from the list. This is done by means of the method *erase*.

By now, it should be clear why we called iterators a generalisation of pointers. They will advance to the next element by applying the ++ operator (in the same way the −− operator can be applied to move backward) and the operator $*$ gives access to the value to which the iterator points, just like we can manipulate with a pointer in an array. This syntax is shared by all iterators.

The next routine inserts a new element containing the value 7 after each element in the list.

```
void insert_7(list<int> &l)
```

```
{
    list<int>::iterator cur(l.begin());
    while (cur != l.end())
        l.insert(++cur, 7);
}
```

The method *insert* inserts a new list element before the the element to which its first argument is pointing. Here, the iterator is first moved to the next element (because the prefix increment operator is used) before the new element is inserted. Note that this routine also inserts a new element after the last list element, because it inserts an element before the end iterator, which points one position beyond the last element.

Then we show how to count all occurrences of the value 7.

```
int count_7(const list<int> & l)
{
    list<int>::const_iterator cur;
    int n = 0;
    for (cur = l.begin(); cur != l.end(); ++cur) {
        if (*cur == 7)
            ++n;
    }
    return n;
}
```

Here we see a different iterator: the const_iterator. In fact, every STL container has *two* associated iterator types, the normal iterator and the const iterator. The difference is that it is not possible to change values by means of a const iterator. So, if *iter* is a const iterator, it is invalid to write something like

```
*iter = 5;
```

In this case the begin method of the list returns a *const_iterator* because the list is passed as a const reference to the routine.

```
int main()
{
    list<int> l;
    for (int i=-1; i<4; i++)
        l.push_back(i);
    // l contains -1 0 1 2 3
    remove_2(l);
    //  l contains -1 0 1 3
    insert_7(l);
    //  l contains -1 7 0 7 1 7 3 7
    l.pop_front();
    //  l contains 7 0 7 1 7 3 7
    return count_7(l);
}
```

The main routine calls the routines that were described above. The only method that are new are *push_back* and *pop_front*. Just like for a vector, *push_back* inserts a value at the end of the list. The method *pop_front* removes the first element of a list.

Lists have a number of methods which we did not yet mention. *clear()* removes all elements of the list. This routine, although mandated by the C++ standard, is not yet available in older implementations. *size()* gives the number of elements in the list.

A.3.3 Vectors and iterators revisited

Example file: examples/Getting_started/copyex.C

The vector interface that we described in section A.3.1 was not the whole story. Vectors can also be accessed by means of iterators, in the same as lists can. In particular, a vector has *begin* and *end* methods that return iterators to the first and past the last element of the vector.

We introduce the notion of vector iterators here because they are useful in the CGAL library. Whenever a collection of objects is expected as input to a function, this collection should normally be supplied by means of a begin and an end iterator. We illustrate this in the following example. Here we use the function copy, which is part of STL. It copies a collection of objects from one place to another. The first two parameters are two iterators that indicate the range of values that must be copied. We see from the three calls of copy that we can use all kinds of iterators here: list iterators, vector iterators and pointers in an array. The last parameter indicates where the values must be copied to. We won't explain the details here. Let it suffice to say that with the syntax below the values are appended to the back of a list or vector respectively.

```
#include <list>
#include <vector>
#include <algobase.h>
#include <iterator.h>

main()
{
    double d_array[2] = { 1.3, 1.2};
    std::vector<double> d_vec(3, 0.5);
    d_vec[2] = 1.4;
    std::list<double> d_list;
    copy(d_vec.begin(), d_vec.end(), back_inserter(d_list));
// d_list: 0.5, 0.5, 1.4 ;
    copy(d_array, d_array+2, back_inserter(d_list));
// d_list: 0.5, 0.5, 1.4, 1.3, 1.2 ;
    d_vec.clear();
// d_vec:  ;
    copy(d_list.begin(), d_list.end(), back_inserter(d_vec));
// d_vec: 0.5, 0.5, 1.4, 1.3, 1.2 ;
}
```

Bibliography

[Alt & al. 96] H. Alt, U. Fuchs, G. Rote, G. Weber. Matching Convex Shapes with Respect to the Symmetric Difference. In *Proceedings of the 4th Annual European Symposium on Algorithms, Lecture Notes in Computer Science, Vol. 1148*, pages 320-333. Springer, 1996.

[de Berg & al. 97] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. Computational Geometry, Algorithms and Applications. Springer, 1997, ISBN 3-540-61270-X.

[Berg & al. 97] M. de Berg, O. Devillers, M. van Kreveld, O. Schwarzkopf, and M. Teillaud. Computing the maximum overlap of two convex polygons under translations. In *Proceedings 7th Annual International Symposium on Algorithms and Computation*, pages 126–135, 1996.

[Fabri & al. 98] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the Design of CGAL, the Computational Geometry Algorithms Library. Research Report MPI-I-1-98, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1998 To appear in Trends in Software.

[Fabri & al. 96] A. Fabri, G.J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL Kernel: A Basis for Geometric Computation. In *Proceedings of the 1st ACM Workshop on Applied Computational Geometry*, Lecture Notes in Computer Science, Vol. 1148. Springer, 1996.

[Granlund 96] T. Granlund. The Gnu Multiple Precision Arithmetics Library 2.0.2, 1996. `http://www.nada.kth.se/~tege/gmp/`.

[Lippman 98] S. B. Lippman, J. Lajoie. C++ primer, 3d ed. Addison Wesley Longman, 1998.

[Mehlhorn & al. 98] K. Mehlhorn, S. Näher, M. Seel and C. Uhrig. The LEDA user manual (version 3.6). `http://www.mpi-sb.mpg.de/LEDA/www/MANUAL/MANUAL.html`

[Musser & al. 96] D. R. Musser and A. Saini. STL tutorial & reference guide: C++ programming with the standard template library. Addison-Wesley, 1996.

[Myers 95] Nathan C. Myers. Traits: a New and Useful Template Technique. C++ Report, 1995.

[Overmars 96] M. H. Overmars. Designing the Computational Geometry Algorithms Library CGAL. In *Proc. of the 1st ACM Workshop on Applied Computational Geometry*, Lecture Notes in Computer Science, Vol. 1148. Springer, 1996.

[Schirra 96] S. Schirra. Designing a Computational Geometry Algorithms Library. In *Lecture Notes for Advanced School on Algorithmic Foundations of Geographic Information Systems, CISM, Udine, September 16-20, 1996.*

[STL] Standard Template Library. URL `ftp://butler.hpl.hp.com/stl/`. URL `http://www.sgi.com/Technology/STL/`.

[Stroustrup 97] B. Stroustrup. The C++ Programming Language, 3d edition. Addison-Wesley, 1997.