# Generic Programming

**Michael Hemmer**

Tel Aviv University, Israel

Introduction to Generic Programming (in CGAL)
March 14th, 2011

## Outline





## Generic Programming Paradigm

**Definition (Generic Programming)**

A discipline that consists of the gradual lifting of concrete algorithms abstracting over details, while retaining the algorithm semantics and efficiency.

[MS88]

Translation:

- You do not want to write the same algorithm again and again !

⇒ You even want to make it independent from the used types.

See also: http://en.wikipedia.org/wiki/Generic_programming



## Motivation - Generic Programming

Reasons to write generic code

- Laziness
  - you don't want write code again and again
  - code is maintained at one place
- Flexibility
  - others can use your code with their types
  - behavior of algorithm can be adjusted later

# Outline





# Trivial Example: swap

```
template <typename T> void swap(T& a, T& b)
{
  T tmp(a);
  a = b;
  b = tmp;
}
```

- announce that you write a template
- name the template arguments
- write template as if arguments where known types



# Concept and Model (brief introduction)

```
template <typename T> void swap(T& a, T& b)
{
  T tmp(a);   a = b;    b = tmp;
}
```

This code implies certain requirements on T

- T must have a copy constructor that copies
- T must have an assignment operator that assigns

Or in formal words:

- T must be a Model of the Concept CopyConstructible
- T must be a Model of the Concept Assignable



# Concept

A concept is a set of certain requirements.

Four principle categories:

- Valid Expressions
  - compiling C++ expressions involving the template argument
- Associated Types
  - types that are related to the model as they participate in one or more of the valid expressions
  - e.g. the return type of some required member function
- Run-time Characteristics
  - semantic guarantees, e.g., pre/post-conditions
- Complexity Guarantees
  - maximum limits on execution time or required resources

## Outline





## Example: normalize_rational (I)

- Code for leda::rational

```
void normalize_rational(leda::rational& x) {
  leda::integer num   = x.numerator();
  leda::integer denom = x.denominator();
  leda::integer g     = leda::gcd(num,denom);
  x = leda::rational( num/g , denom/g );
}
```

- Code for CORE::BigRat

```
void normalize_rational(CORE::BigRat& x) {
    CORE::BigInt num   = CORE::numerator(x);
    CORE::BigInt denom = CORE::denominator(x);
    CORE::BigInt g     = CORE::gcd(num,denom);
    x = CORE::BigRat( num/g , denom/g );
}
```



## Example: normalize_rational (II)

Algorithm is actually clear:

- Take numerator and denominator.
- Compute the greatest common divisor (gcd).
- Divide both by the gcd and construct the new rational.

Problems for possible template code

- Types are external and have different interface.
- How to get the type for integer ? (associated type)

Solution: Unify interface by using a traits class that you control!



## Example: Rational_traits (I)

- a template version using a traits class:

```
template<class Rational>
void normalize_rational(Rational& x) {
  typedef Rational_traits<Rational> Traits;
  typedef typename Traits::Integer Integer;

  Integer num   = Traits::numerator(x);
  Integer denom = Traits::denominator(x);
  Integer g     = gcd(num,denom);
  x = Traits::make_rational( num/g , denom/g );
}
```

- use traits to get the integer type
- use traits to provide a unified interface

# Example: Rational_traits (II)

```
template <typename T> struct Rational_traits{}; // default

// full specialization for leda::rational
template<> struct Rational_traits<leda::rational> {
  typedef leda::integer  Integer;

  static Integer numerator(const Rational& x){
    return x.numerator();
  }
  // similar for denominator, make rational
};

// full specialization for CORE::BigRat
template<> struct Rational_traits<CORE::BigRat> {
  typedef CORE::BigRat  Integer;

  static Integer numerator(const Rational& x){
    return CORE::numerator(x);
  }
  // similar for denominator, make_rational
};
```



# Traits class

Original definition:

- A traits class is a class template that associates information to compile-time entity, e.g. associated type or constants

```
typedef typename Traits::Integer Integer;
```

- may also provide unified interface

```
Integer num   = Traits::numerator(x);
```

Sometimes the traits class is already used as the template argument:

- Advantage:
  provides all types and functionality the algorithm needs
  ⇒ usually only one template argument
- Disadvantage:
  this overload of the term *'traits'* may causes confusion



# Outline





# Dispatching using Tags (I)

- Goal:
  - Provide a function `simplify` that is valid for any type.
  - If the argument is a rational it should normalize it.
  - Otherwise it should do nothing.
- Problem:
  - `normalize_rational` can only be instantiated for rationals.
  - This decision must be taken at compile time.

## Dispatching using Tags (II)

First a small modification to `Rational_traits`:

```
struct Tag_false{}; // just two different classes
struct Tag_true{};

template <typename T> struct Rational_traits{ // default
  typedef Tag_false Is_rational;
};

template<> struct Rational_traits<leda::rational> {
  typedef Tag_true Is_rational;

  ... // the remaining definitions
};

template<> struct Rational_traits<CORE::BigRat> {
  typedef Tag_true Is_rational;

  ... // the remaining definitions
};
```



## Dispatching using Tags (III)

Simplify some number:

```
template <class T> inline
void simplify_(T& x, Tag_false){}

template <class T> inline
void simplify_(T& x, Tag_true){normalize_rational(x);}

template <class T> inline
void simplify(T& x){
    typedef Rational_traits<T> Traits;
    typedef typename Traits::Is_rational Is_rational;
    simplify_(x, Is_rational());
}
```



## Outline





## Coding Conventions in CGAL

- `ThisIsAConceptName`
- `This_is_a_class_name`
- `this_is_a_function_name`
- `THIS_IS_AN_ENUM_NAME`

# Outline





# Use of traits classes in CGAL

- used in traditional way on lower levels
  e.g. traits classes related to number types
  - `CGAL::Algebraic_structure_traits<T>`
  - `CGAL::Real_embeddable_traits<T>`
  - `CGAL::Fraction_traits<T>`
- higher level name the template argument *'traits'*
  - Advantage:
    provides all types and functionality the algorithm needs
    $\Rightarrow$ usually only one template argument
  - Disadvantage:
    this overload of the term *'traits'* usually causes confusions



# Outline





# Generic Programming Dictionary

**Concept** A set of requirements that a class must fulfill.

**Model** A class that fulfills the requirements of a concept.

**Traits** Models that describe behaviors.

**Refinement** An extension of the requirements of another concept.

**Generalization** A reduction of the requirements of another concept.

## Some Generic Programming Libraries

STL — The C++ Standard Template Library.

BOOST — A large set of portable and high quality C++ libraries that work well with, and are in the same spirit as, the C++ STL.

LEDA — The Library of efficient data types and algorithms.

CGAL — The computational geometry algorithms and data structures library.



## STL Components

Container — A class template, an instance of which stores collection of objects.

Iterator — Generalization of pointers; an object that points to another object.

Algorithm

Function Object (Functor) — A computer programming construct invoked as though it were an ordinary function.

Adaptor — A type that transforms the interface of other types.

Allocator — An objects for allocating space.



## The sort() algorithm

- The sort() routine is generic.
  - It works on many different types of containers.
  - It deals with Iterators instead of the containers themselves.
- Takes two iterators to specify the semi-open range source
  - Sort is an overloaded name. Another version takes a functor that compares 2 values for ordering.
- Incidentally, this is much faster than qsort; presumably due to the fact that comparisons are done inline.

```
#include <algorithm>

std::sort(first, beyond);
std::sort(first, beyond, std::greater<int>());
```



## Sorting: Using Iterator Adaptors

- Iterators can iterate over streams, either to read elements or to write them.
- std::cin must be "adapted" to have an iterator interface.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>

using namespace std;

int main(int argc, char* argv[])
{
  vector<int> v;
  istream_iterator<int> start(cin), end;
  back_insert_iterator<vector<int> > dest(v);

  copy(start, end, dest);
  sort(v.begin(), v.end());
  copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
  return 0;
}
```

# Adaptors

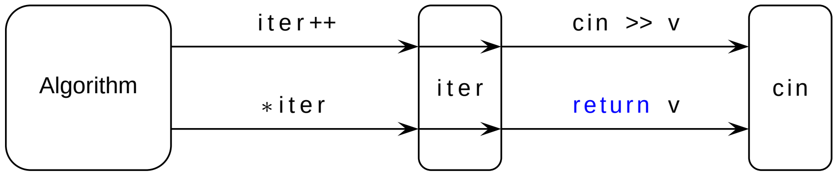
- Adaptors transform the interface of other types.
- This is very much how electrical adaptors work.



# The istream_iterator Adaptor

- A template type (of course!); parameterized by the type of object to be read from the stream.
- Initialized with a stream.
- Dereferencing the iterator reads an element from the stream.
- Incrementing the iterator has no effect.
- An istream iterator created with the default constructor has the past-the-end value.
  - as does an iterator whose stream has reached the end of file.

```
iter = istream_iterator<int>(cin);
```






# Typedefs

- Shorten the length of type definitions.
- Eliminate the problem introduced by overloading the '<' '>' operators.

```
back_insert_iterator<vector<int> > dest(v);
```

versus

```
typedef vector<int> int_vector;
back_insert_iterator<int_vector> dest(v);
```

- C++0x improves the specification of the parser:
  - multiple right angle brackets are interpreted as closing the template argument list (where it is reasonable).



# The copy() Algorithm

- Accepts three iterators as arguments.
  - The first two specify the source range.
  - The third specifies the destination.

Copy from standard input (from the current position in the input stream to the end of the stream) into a vector:

```
typedef istream_iterator<int> int_istream_iterator;
copy(int_istream_iterator(cin), int_istream_iterator(), v.begin());
```

This may cause an overflow though!

# CGAL

## The Computational Geometry Algorithms Library


**Efi Fogel**

Tel Aviv University, Israel


Algorithmic Robotics and Motion Planning
March 14th, 2011

# Outline

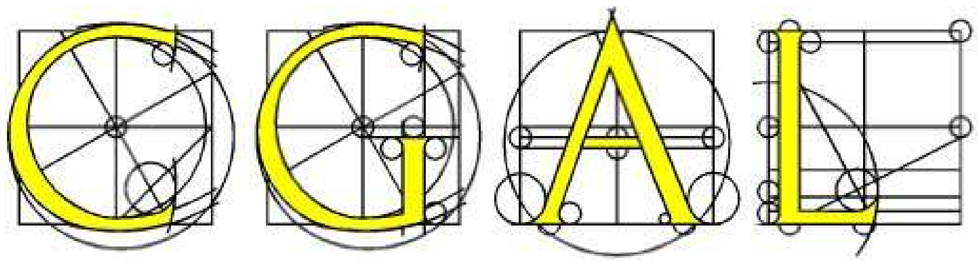




# CGAL: Mission

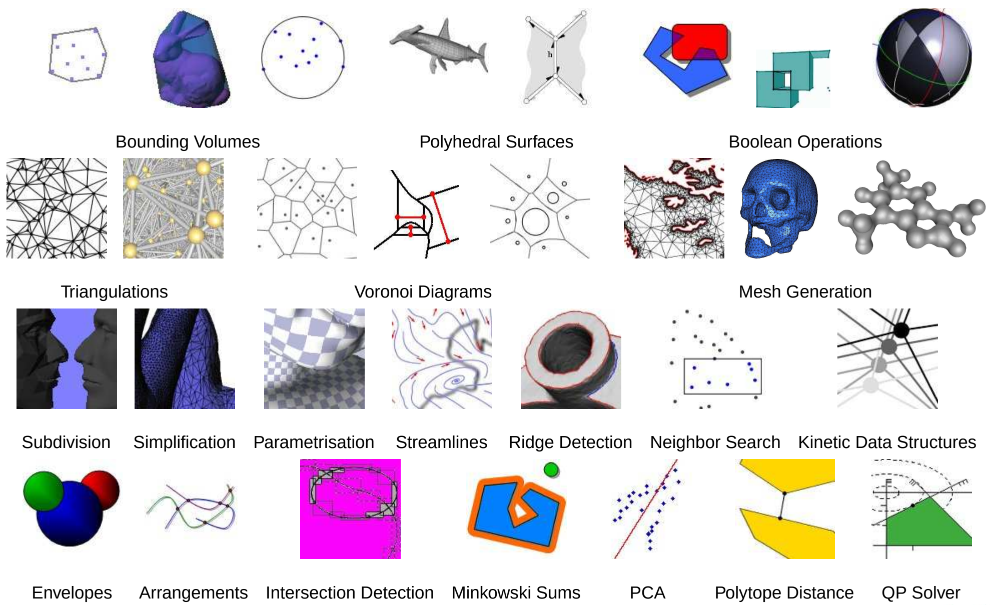
“Make the large body of geometric algorithms developed in the field of computational geometry available for industrial applications”

CGAL Project Proposal, 1996

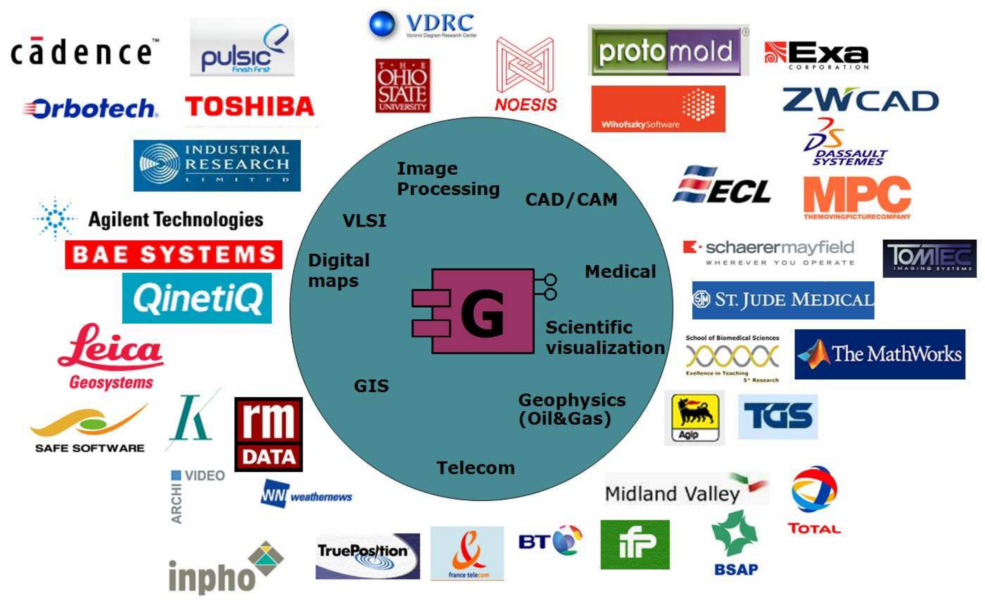




# Some of CGAL Content



# Some CGAL Commercial Users






# CGAL Facts

- Written in C++
- Follows the *generic programming* paradigm
- Development started in 1995
- Active European sites:

1. INRIA Sophia Antipolis
2. MPII Saarbrücken
3. Tel Aviv University
4. ETH Zürich (Plageo)
5. University of Crete and FO.R.T.H.
6. INRIA Nancy
7. Université Claude Bernard de Lyon
8. ENS Paris
9. University of Eindhoven
10. University of California, San Francisco
11. University of Athens



# CGAL History

| Year | Version Released | Other Milestones |
|---|---|---|
| 1996 | | CGAL founded |
| 1998 | July 1.1 | |
| 1999 | | Work continued after end of European support |
| 2001 | Aug 2.3 | Editorial Board established |
| 2002 | May 2.4 | |
| 2003 | Nov 3.0 | GEOMETRY FACTORY founded |
| 2004 | Dec 3.1 | |
| 2006 | May 3.2 | |
| 2007 | Jun 3.3 | |
| 2009 | Jan 3.4, Oct 3.5 | |
| 2010 | Mar 3.6, Oct 3.7 | CGAL participated in Google Summer of Code |
| 2011 | | CGAL applies to participate in GSoC |



# CGAL in Numbers

| | |
|---|---|
| 900,000 | lines of C++ code |
| 10,000 | downloads per year not including Linux distributions |
| 3,500 | pages manual |
| 3,000 | subscribers to cgal-announce list |
| 1,000 | subscribers to cgal-discuss list |
| 120 | packages |
| 60 | commercial users |
| 25 | active developers |
| 6 | months release cycle |
| 7 | Google's page rank for cgal.org.com |
| 2 | licenses: Open Source and commercial |

# CGAL Properties

- Reliability
  - Explicit degeneracy handling.
  - Exact Geometric Computation (EGC) adherence.
- Flexibility
  - Open library.
  - Depends on other libraries (e.g., BOOST, GMP, MPFR, QT, & CORE)
  - Modular structure. Separation between to geometry and topology.
  - Adaptable to user code.
  - Extensible, e.g., data structures can be extended.
- Ease of Use
  - Didactic and exhaustive Manuals.
  - Follows standard concepts (e.g., C++ and STL).
  - Smooth learning-curve.
- Efficiency
  - Follows the generic-programming paradigm.
  - Polymorphism is resolved at compile time.



# CGAL Structure

**Basic Library**

Algorithms and Data Structures
e.g., Triangulations, Surfaces, and Arrangements

**Kernel**

Elementary geometric objects
Elementary geometric computations on them

**Support Library**

Configurations, Assertions,...

Visualization
Files
I/O
Number Types
Generators
...



# Outline

1. CGAL
   - Introduction
   - Literature



# CGAL Bibliography

A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr.
On the design of CGAL a computational geometry algorithms library.
*Software — Practice and Experience*, 30(11):1167–1202, 2000. Special Issue on Discrete Algorithm Engineering.

A. Fabri and S. Pion.
A generic lazy evaluation scheme for exact geometric computations.
In $2^{nd}$ *Library-Centric Software Design Workshop*, 2006.

M. H. Overmars.
Designing the computational geometry algorithms library CGAL.
In *Proceedings of ACM Workshop on Applied Computational Geometry, Towards Geometric Engineering*, volume 1148, pages 53–58, London, UK, 1996. Springer.

The CGAL Project.
CGAL *User and Reference Manual*.
CGAL Editorial Board, 3.7 edition, 2010.
`http://www.cgal.org/Manual/3.7/doc_html/cgal_manual/contents.html`.