Generic Programming

Michael Hemmer

Tel Aviv University, Israel

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

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
Concept and Model (brief introduction)

```cpp
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
Example: `normalize_rational (I)`

- Code for `leda::rational`

```cpp
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`

```cpp
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: `Rational_traits (I)`

- A template version using a traits class:

```cpp
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: `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 (II)

```cpp
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
};
```

Introduction to Generic Programming (in CGAL) 27

## Outline

1. **Generic Programming**
   - Introduction and Motivation
   - Concepts and Models
   - Traits Classes
   - Dispatching

2. **Generic Programming in CGAL**
   - Coding Conventions
   - Traits classes in CGAL
   - CGAL Kernels
   - Function Objects

3. **STL (Based on Jak Kirman's tutorial)**

4. **Appendix**
   - Literature

Introduction to Generic Programming (in CGAL) 30

### Traits class

**Original definition:**

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

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

- may also provide unified interface

```cpp
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

Introduction to Generic Programming (in CGAL) 31

### 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`:

```cpp
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:

```cpp
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

1. Generic Programming
   - Introduction and Motivation
   - Concepts and Models
   - Traits Classes
   - Dispatching

2. Generic Programming in CGAL
   - Coding Conventions
   - Traits classes in CGAL
   - CGAL Kernels
   - Function Objects

3. STL (Based on Jak Kirman’s tutorial)

4. Appendix
   - Literature

Coding Conventions in CGAL

- ThisIsAConceptName
- This_is_a_class_name
- this_is_a_function_name
- THIS_IS_AN_ENUM_NAME
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.

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
      - usually only one template argument
    - Disadvantage:
      - this overload of the term ‘traits’ usually causes confusions
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.

---

**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.

```cpp
#include <algorithm>

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

---

### 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.

---

**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.

```cpp
#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.

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

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:

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

This may cause an overflow though!

Typedefs

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

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

versus

```cpp
typedef vector<int> int_vector;
back_insert_iterator<vector<int>> 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).
CGAL
The Computational Geometry Algorithms Library

Efi Fogel
Tel Aviv University, Israel

Algorithmic Robotics and Motion Planning
March 14th, 2011

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

Bounding Volumes Polyhedral Surfaces Boolean Operations

Triangulations Voronoi Diagrams Mesh Generation

Subdivision Simplification Parametrisation Streamlines Ridge Detection Neighbor Search Kinetic Data Structures

Envelopes Arrangements Intersection Detection Minkowski Sums PCA Polytope Distance QP Solver
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 F.O.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

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

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
...

**CGAL Bibliography**

  On the design of CGAL a computational geometry algorithms library.

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

- M. H. Overmars.
  Designing the computational geometry algorithms library CGAL.

- The CGAL Project.