Generic programming and CGAL

Sylvain Pion

INRIA Sophia Antipolis

November 30, 2004

Overview

- Basic C++ constructs
- Templates functions and classes
- STL and generic programming
- Generic programming and CGAL

Basic C++ constructs

Classes

```
struct A {
    int i;
};
```

We call the type A a class. If we create a variable of type A, we call this variable an object of class A, which is also known as an instantiation of class A. Each object of type A has a member variable i which can be accessed with the dot notation:

```
int main() {
    A a;
    a.i = 5;
}
```

Access control

```
class A {
public:
    int i;
protected:
    void f();
private:
    A() {}
};
```

- private : only accessible within the class member functions.
- public : accessible from everywhere.
- protected : accessible from derived classes.

struct versus class.

Member functions

```
class A {
    int i; // private
public:
    int  get_i();
    void set_i( int n);
};
```

Scope operator :: to refer to members, from outside the class.

Hidden this parameter of type A*.

```
int A::get_i() {
    return i;
}
void A::set_i( int n) {
    i = n;
}
```

Inline

Efficiency considerations :

```
class A {
    int i;
public:
    int  get_i()        { return i; }   // both inline
    void set_i( int n) { i = n; }
};

inline
void f(double d)
{
    ...
}
```

Const correctness

```
class A {
    int i;
public:
    int  get_i() const { return i; }   // inline and const
    void set_i( int n) { i = n; }      // inline
};


int main() {
    A a;
    a.set_i(5);
    int j = a.get_i();

    const A a2;          // uninitialized constant
    a2.set_i(5);         // error
}
```

Special member functions

Constructors, Assignment, and Destructor

```
class A {
    A();                    // default constructor
    A( const A& a );        // copy constructor
    A( int n);              // user defined
    ~A();                   // destructor
};

int main() {
    A a1;           // default constructor
    A a2 = a1;      // copy constructor (not assignment operator)
    A a3(a1);       // copy constructor
    A a4(1);        // user defined constructor
} // destructor calls for a4, a3, a2, a1 at end of block
```

Special member functions

```
class A {
    int i; // private
public:
    A()
        : i(0) {}       // default constructor
    A(const A& a)
        : i(a.i) {}     // copy constructor, equal to default
    A(int n)
        : i(n) {}       // user defined
    ~A() {}             // destructor, equal to default
};
```

Typical uses : ressource management (e.g. memory).

Derivation

B inherits the members of A.

```
class B : public A {
    int j;
};

int main() {
    B b;
    A a = b;
}
```

Objects of class B can be assigned to objects of class A.

There are also protected and private derivations, multiple inheritance, and virtual inheritance.

Virtual member functions

```
struct Shape {
    virtual void draw() = 0;
};
```

We derive different concrete classes from Shape and implement the member function draw for each of them.

```
struct Circle : public Shape {
    void draw();
};
struct Square : public Shape {
    void draw();
};
```

Dynamic polymorphism and object-oriented programming.

Static class members

```
struct A {
    static int i;
    int j;
    static void init();
};

void A::init() {
    i = 5; // fine
    // j = 6; // is not allowed
}

int main() {
    A::init();
    assert( A::i == 5);
}
```

Template functions and classes

Template functions

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

int main() {
    int i = 5;
    int j = 7;
    swap( i, j);   // Instantiation : uses "int" for T.
}
```

The actual types for a class template are explicitly provided by the programmer. An example is a generic list class for arbitrary item types.

Template classes

```
template <class T>
struct list {
    void push_back( const T& t); // append t to list.
};

int main() {
    list<int> ls;  // uses "int" for T.
    ls.push_back(5);
}
```

Pattern matching for matching functions

The C++ compiler uses pattern matching to derive automatically the template argument types for functions template. Consider for example a function template that works only for lists:

```
template <class T>
void foo(list<T>& ls);

std::list<int>   l;   ... foo(l);
std::vector<int> v;   ... foo(v); // does not match
```

Default arguments

```
template <class T, class Container = list<T> >
struct stack { ... };

stack<int> s;                          // easy
stack<double, my_container> sd;        // customizable
```

Non-type template parameters

```
template <int dim>
struct Point {
    double coordinates[dim]; // coordinate array
    // ...
};

int main() {
    Point<3> // a point in 3d space
}
```

Also template template parameters...

Lazy instantiation

A member function of a class template is not instantiated when the class is instantiated. Only when the function is called.

```
template <class T>
class list {
    ...
    void sort();
};

template <class T>
void list<T>::sort()
{
  ...
}

list<int> l; // instantiates the class
l.sort();    // instantiates the function
```

Member templates

```
template <class T1, class T2>
struct pair {
    T1 first;
    T2 second;
    pair() {}  // don't forget if there are also others ctors
    template <class U1, class U2>     // template constructor
    pair( const pair<U1,U2>& p);
};
```

Syntax used when not inside the class body :

```
template <class T1, class T2>
template <class U1, class U2>
pair<T1,T2>::pair( const pair<U1,U2>& p)
    : first( p.first), second( p.second) {}
```

Specializations and partial specializations

```
template <>
struct vector<bool> {
    // specialized implementation
};
```

The compiler matches vector<bool> automatically with this specialization. The empty template declaration was previously superfluous, but is now mandatory.

Now suppose, vector has a second template argument for a memory allocator (which it does in the STL, but hidden by a default setting). The resulting partial specialization is still a template.

```
template <class Allocator = std::allocator<bool> >
struct vector<bool, Allocator> {
    // partially specialized implementation
};
```

For template functions : partial overloading

- Try them all (same name, number of arguments, and scope),
- see which are possible, (pattern matching)
- and pick the "most specific",
- ambiguity otherwise.

Local types and keyword typename

```
template <class T>
struct list {
    typedef T value_type;
};

int main() {
    list<int> ls;
    list<int>::value_type i; // is of type int
}

template <class Container>
struct X {
    typedef Container::value_type value_type; // not correct
    typedef typename Container::value_type value_type; // OK
};
```

Namespaces

```
namespace CGAL {
    int max( int a, int b);
    class A;
    void foo( const A& a);
} // ends the namespace CGAL
```

We could use the above declarations in the following way:

```
int main() {
    int i = CGAL::max( 3, 4);
    A a; // assumes that we have also seen the full definition
    CGAL::foo( a);
}
```

Koenig lookup

Useful for operators :

```
int main() {
    A a;
    foo( a); // Koenig lookup (ADL)
}
```

Importing a whole namespace with the using directive :

```
using namespace std;
using std::vector;
```

STL and generic programming

The Standard Template Library

It's a foundation library. It provides :

- basic data types, such as list, vector, set and map, and
- basic algorithms, such as find and sort.

It is using the generic programming paradigm. Here is one definition [Jazaeri98]:

Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction.

Generic programming

Key ideas include:

- Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.

- Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.

- When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.

- Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient.

Concept and Model

Consider our first example of a function template, swap:

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

Instantiation with an actual type : it needs to provide assignment operator and a copy constructor.

```
class A {
  int i;
  A(const A& a) : i(a.i) {}
public:
  A() : i(0) {}
};

A a, b;        swap(a, b);    // error
int c, d;      swap(c, d);    // OK
```

Requirements

syntactic versus semantic requirements (checked or not).

Sets of requirements documented as concepts.

The concept for the swap function parameter is called Assignable.

An actual type is a model for a concept if it fulfills its requirements.
int is a model of Assignable.

Common basic concepts :

Concept	Syntactic requirements
Assignable	copy constructor and assignment operator
Default Constructible	default constructor
Equality Comparable	equality and inequality operator
LessThan Comparable	order comparison with operators $<$, $<=$, $>=$ and $>$

Analogy with dynamic polymorphism :
model (derived class), concept (base class).

Generic Algorithms Based on Iterators

Iterators are an abstraction of pointers.

Two purposes :

- refer to an item, and
- traverse a sequence (e.g. stored in a Container)

Iterator categories

Five categories depending on the accessing capabilities.

Concept	Refinement of	Syntactic requirements
TrivialIterator/Handle	Assignable, EqualityComparable	*, ->
InputIterator	TrivialIterator	++ ...
OutputIterator	Assignable	*, ++ ...
ForwardIterator	InputIterator, OutputIterator, DefaultConstructible	...
BidirectionalIterator	ForwardIterator	– ...
RandomAccessIterator	BidirectionalIterator, LessThanComparable	+, +=, -, [], ...

Pointers and `std::vector<T>::iterator` are RandomAccessIterator.
`std::list<T>::iterator` is a BidirectionalIterator.

Complexity requirements as well.

Sequences as iterator ranges

Range [first,beyond) of two iterators (half-open interval, beyond = past-the-end)

A container class is supposed to provide

- a member type called iterator, model of Iterator,
- two member functions : begin() and end().

```
template <class T>
class list {
    void push_back( const T& t); // append t to list.
    typedef ... iterator;
    iterator begin();
    iterator end();
};
```

STL algorithms are independent of container classes, but use iterators instead.

Exemple : the contains function

Returns true iff the value is contained in the values of the range [first,beyond).

```
template <class InputIterator, class T>
bool contains( InputIterator first, InputIterator beyond,
               const T& value)
{
    while ((first != beyond) && (*first != value))
        ++first;
    return (first != beyond);
}
```

Using it :

```
int a[100]; // ... initialize elements of a.
bool in_third_quarter = contains( a+50, a+75, 42);

std::list<int> l; // ... initialize elements of l.
bool found = contains(l.begin(), l.end(), 42);
```

The `std::copy` function

Copies a sequence to an output iterator.

```
template <class InputIterator, class OutputIterator>
OutputIterator copy( InputIterator first,
                     InputIterator beyond,
                     OutputIterator result)
{
    while (first != beyond)
        *result++ = *first++;
    return result;
}
```

Using with stream iterators :

```
copy( istream_iterator<int>( cin ),
      istream_iterator<int>( ),
      ostream_iterator<int>( cout, "\n" ));
```

Function Objects

Definition : An instance of a class with the operator() member function (looks like a function call).

Generator, UnaryFunction, BinaryFunction, Predicate, BinaryPredicate.

Goal : parameterize the behavior of generic algorithms.

Similar to function pointers, but can store a state.

Example:

```
template <class T>
struct equals {
    bool operator()( const T& a, const T& b) const
    { return a == b; }
};
```

Example

```
template <class InputIterator, class T, class Eq>
bool contains( InputIterator first, InputIterator beyond,
               const T& value, Eq eq )
{
    while ((first != beyond) && ( ! eq( *first, value )))
        ++first;
    return (first != beyond);
}
```

Using it with equal<int>.

```
int a[100];
// ... initialize elements of a.
bool found = contains( a, a+100, 42, equals<int>());
```

More STL

- std::iterator_traits
- Adaptable functors and binders.
- ...

Generic programming and CGAL

Generic programming and CGAL

CGAL uses the STL.

CGAL uses STL concepts : iterators, function objects...

Specificities of CGAL : circulators, geometric traits...

Circulators

Iterating over circular structures (e.g. edges incident to a vertex).

Variant of Iterators, with different past-the-end condition.

Used by the Halfedge Data Structure, and the Triangulation Data Structure.

Kernel interface

CGAL geometric algorithms are parameterized by a geometric traits.

```
template < class TriangulationTraits_3 ,
           class TriangulationDataStructure_3 = ... >
class Triangulation_3;
```

It provides a (attempted minimal) set of types and functions for the geometry :

Nested Type	Requirements
Point_3	Assignable, DefaultConstructible
Segment_3	Assignable, DefaultConstructible
Orientation_3	Function object taking 4 Point_3, returning enum...
..._3	...

The Kernel concept is a superset for many different geometric traits concepts. There are many different possible (and useful) models of Kernel.

The end

```
std::istream_iterator<char> question(std::cin);

while ( question != std::istream_iterator<char>() )
{
    std::cout << answer(question);
    ++question;
}

goto lunch;
```