C++ Software Tools for Cutting and Packing Workshop

Ophir Setter
Tel Aviv University

May 26, 2008

Generic Algorithms

- Generic algorithms are written over unknown types that are then somehow instantiated later by the compiler
- A generic algorithm has two parts:
  - The actual instructions that describe the steps of the algorithm
  - A set of requirements that specify which properties its argument types must satisfy — aka Concept

A Basic Example: `swap()`

- Generic programming in C++ = templates (TAVNIOT)
- When the function call is compiled, it is instantiated with a data type
- This data type must have an assignment operator (copy constructor)
- This defines the concept of our algorithm.
- In this example, the int data type is a model of our concept commonly called assignable (copy constructable)

```
template <typename T>
void swap(T& a, T& b)                   // int x, y;
{                                       // swap(x, y);
    T tmp = a; a = b; b = tmp; // swap(const int, const int) is an error
}
```

A Simple Example: Version 1 – Standard C++

A program that reads integers, sorts them, and prints them out
cons: flexibility, lack of compile-time check, a lot of code

```
int cmp(const void * a, const void * b) {
  int aa = *(int *)a; int bb = *(int *)b;
  return (aa < bb) ? -1 : (aa > bb) ? 1 : 0;
}

int main(int argc, int * argv[]) {
  int array[1000]; int n = 0;
  while (std::cin >> array[n++]);
  n--;                    // it got incremented once too many times
  qsort(array, n, sizeof(int), cmp);
  for (int i = 0; i < n; ++i)
    std::cout << array[i] << std::endl;
  return 0;
}
```

STL – Standard Template Library

- Software library partially included in the C++ standard library
- Uses the generic programming paradigm through the use of C++ templates
- Provides containers, iterators, algorithms and functors
- Containers – represent objects that contain other objects. STL includes (but not only):

  `vector` – a random-access dynamic container

  `list` – a doubly linked list

  `set` – no 2 elements are the same

  `map` – associates objects of one type (Key) with objects of another type (Data)

A Simple Example: Version 2 – Containers, Iterators, Algorithms

Using STL `vector` container

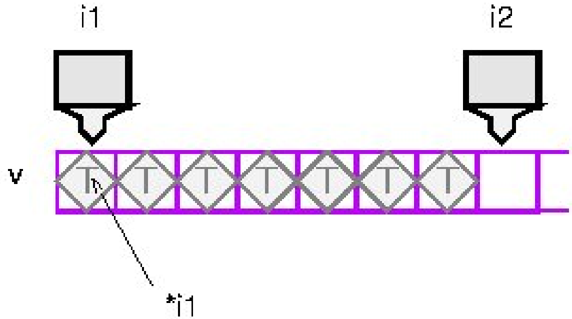
```
#include <algorithm>
#include <vector>
#include <iostream>
int main(int argc, int * argv[]){
  int input; std::vector<int> v;// create an empty vector of integers
  while (std::cin >> input)      // while not end of file
    v.push_back(input);          // append to vector
  std::sort(v.begin(), v.end()); // sort using < operator
  int n = v.size();
  for (int i = 0; i < n; ++i)
    std::cout << v[i] << std::endl;
  return 0;
}
```

Iterators

- Provide a way of specifying a position in a container (like pointers)
- Can be dereferenced with * operator
- Two iterators can be compared
- Refined iterator concepts – Some can be incremented/decremented/indexed (++/−−/[ ] operators)
- There is a special iterator value called "past-the-end"
- `c.begin()` and `c.end()` return the first and "past-the-end" iterators of the container c

Iterators – cont.

```
vector<int> v;
vector<int>::iterator i2 = v.end();
for (vector<int>::iterator i1 = v.begin(); i1 != i2; ++i1)
{
   ...
}
```



Iterator Types

- There are several different ways of generalizing pointers. Each is a refined concept of the Trivial Iterator concept:
  - Input Iterator — `*it++;`
  - Output Iterator — `*it++ = t;`
  - Forward Iterator — `it++;`
  - Bidirectional Iterator — `it++; it--;`
  - Random Access Iterator — `it++; it--; it[n]; it + n;`

Output Iterators and Iterators Adaptors

- We wish to treat streams as iterators both to read elements or to write them
- `std::cin`, `std::cout` and `std::vector` must be "adapted" to have an iterator interface

```
using namespace std;
vector<int> v;
istream_iterator<int> start(cin), end;        // iterator for reading ints
back_insert_iterator<vector<int> > dest(v);// iterator for adding elements

copy(start, end, dest);  // copy(start, end, back_inserter(v));
sort(v.begin(), v.end());
// output using iterators
copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
```

Typedefs

- Shorten the length of type definitions
- Removes definition repetition
- Eliminate the extra space needed due to overloading the < > operators
- Important ingredient in writing generic algorithms

```
map<const string, int>::iterator cur = months.find("june");
```

versus

```
typedef map<const string, int> Map_monsth_days;
Map_monsth_days::iterator cur = months.find("june");
```

Boost Library

- Is a free portable C++ source of a collection of libraries

  smart pointers – automatic deletion of pointers at the appropriate time

  regex – support of regular expressions

  filesystem – directory and file iteration

  graph – generic graph components and algorithms

  more
- Written in STL style
- Used by many programmers across a broad spectrum of applications
- Parts will become part of a future C++ Standard soon

Example: shared_ptr

Automatic deletion of allocated variables

```
#include <boost/shared_ptr.hpp>

typedef boost::shared_ptr<Foo> Foo_ptr;

std::vector<Foo_ptr> foo_vector;

Foo_ptr foo_ptr (new Foo (2));
foo_vector.push_back (foo_ptr);

foo_ptr.reset (new Foo (1));
foo_vector.push_back (foo_ptr);

foo_ptr.reset (new Foo (3));
foo_vector.push_back (foo_ptr);
```

The Boost Graph Library (BGL)

- Is a header-only library (not need to be built to use)
  - GraphViz input parser is the only exception
- Is generic in three ways (like the STL):
  - Algorithm/Data-Structure Interoperability

    Single template functions operate on many different classes of containers
  - Extension through Function Objects

    Algorithms and containers are extensible and adaptable
  - Element Type Parameterization

    Its containers are parameterized on the element type

The interface of the BGL graph-algorithms

- Is abstract — hides the details of the particular graph data-structure of the BGL graph-algorithms
- Defined by iterators for data-structure traversal:
  - Traversal of all vertices in the graph
  - Traversal of all edges in the graph
  - Traversal along the adjacency structure of the graph (from a vertex to each of its neighbors)
- Allows template functions (`breadth_first_search()`) to work on a large variety of graph data-structures
  - Without copying/placing the data inside adaptor objects
  - Custom-made graph structures can be used as-is
    - e.g., CGAL arrangements are custom-made graphs

BGL Graph Representation

- "Built-in" graph classes include:
  - adjacency_list — each vertex holds an edge list
  - adjacency_matrix — each element a_{ij} is a boolean flag that says whether there is an edge from i to j
  - compressed_sparse_row_graph — high-performance, non-mutable graph
- vertex_descriptor and edge_descriptor to represent vertex and edge objects in BGL algorithms

```
// use vectors to hold lists
typedef adjacency_list <vecS, vecS, undirectedS> graph_t;
typedef graph_traits <graph_t>::vertex_descriptor Vertex;
typedef graph_traits <graph_t>::edge_descriptor Edge;
```

Extension through Visitors

- Are extensible through *Visitors* — a function object with multiple methods
- User-defined operations are inserted into “event points”
  - particular event points and corresponding visitor methods depend on the particular algorithm

```
template <class TimeMap, class TimeT, class Tag>
time_stamper<TimeMap, TimeT, Tag>;

vertex_descriptor dis_time[N];
// Using stamp_times Object Generator for convenience.
stamp_times(dis_time, initial_time, on_discover_vertex());
```

Named Parameters

- C++ only supports positional parameters (in function call parameters are determined by position)
- Used to overcome a long and exhausting list of parameter some (or all) have defaults
- Don't have to remember the order of the parameters – only their names
- Periods are used instead of commas

```
bellman_ford_shortest_paths(g, int(N), weight_map(weight).
distance_map(&distance[0]).predecessor_map(&parent[0]));
```

Wrapping It All Up Example

```
typedef adjacency_list <vecS, vecS, undirectedS> graph_t;
enum { r, s, t, u, v, w, x, y, N };
typedef std::pair <int, int> Edge;
Edge edge_array[] = { Edge(r, s), Edge(r, v), Edge(s, w), Edge(w, r),
    Edge(w, t), Edge(w, x), Edge(x, t), Edge(t, u), Edge(x, y),
    Edge(u, y)};
typedef graph_traits<graph_t>::vertices_size_type v_size_t;
graph_t g(edge_array, edge_array + n_edges, v_size_t(N));

std::vector<int> p(boost::num_vertices(g));
boost::graph_traits<graph_t>::vertices_size_type d[N];
std::fill_n(d, size_t(N), 0);

boost::breadth_first_search (g, s, boost::visitor(boost::make_bfs_visitor
  (std::make_pair(boost::record_distances(d, boost::on_tree_edge()),
                  boost::record_predecessors(&p[0], boost::on_tree_edge())
     ))));
```

CGAL – the Computational Geometry Algorithm Library

- The goal of the CGAL Open Source Project is to provide *easy access to efficient and reliable geometric algorithms*
- Developed in C++ and follows the Generic Programming paradigm
- Primary design goals: Correctness, Flexibility, Efficiency and Ease of Use
- Some numbers:
  - 600,000 lines of code
  - 3,500 manual pages
  - 1,000 subscribers for cgal-discuss list

Architecture

- Geometric Kernel
  - Constant-size geometric objects (e.g., points, lines, planes, etc.)
  - Predicates and constructors of these objects
- Basic Library — Data structure and algorithms (e.g., Triangulations, Polyhedrons, Arrangements, etc.)
- Support Library
  - Number types
  - Geometric-object generators
  - Input/Output
  - Visualization
  - More none geometric types (e.g., Circulators, etc.)

Geometric Kernels

- Consists of:
  - Constant-size non-modifiable geometric primitive objects (e.g., point, vector, direction, line, ray, segment, etc.)
  - Operations on these objects
- Predefined kernels:
  - `Exact_predicates_inexact_constructions_kernel`
  - `Exact_predicates_exact_constructions_kernel`
  - `Exact_predicates_exact_constructions_kernel_with_sqrt`

Basic Library

- Basic geometric data structures and algorithms
- Generic data structures are parameterized with *Traits* classes
  - Separates algorithms and data structures from the geometric kernel
- Generic algorithms are parameterized with iterator ranges
  - Decouples the algorithm from the data structure

2D Polygons

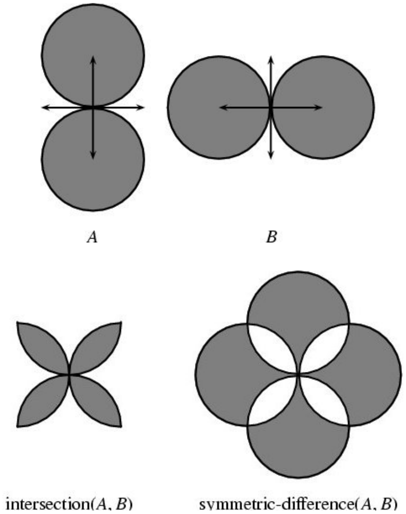
- A *polygon* is a closed chain of edges
- A *simple polygon* is a polygon whose edges don't intersect (except neighboring edges)
- CGAL support algorithms for 2D polygons (`Polygon_2` class):
  - Find the leftmost, rightmost, topmost and bottommost vertex.
  - Compute the (signed) area.
  - Check if a polygon is simple.
  - Check if a polygon is convex.
  - Find the orientation (clockwise or counterclockwise)
  - Check if a point lies inside a polygon.

2D Polygons – Example

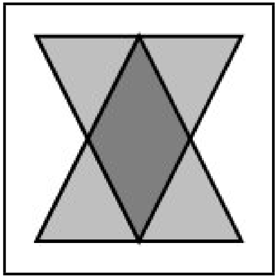
```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Polygon_2_algorithms.h>
#include <iostream>

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef K::Point_2 Point;

int main() {
  Point points[] = { Point(0,0), Point(5.1,0), Point(1,1), Point(0.5,6)};
    // check if the point is inside the polygon.
    if (CGAL::bounded_side_2(points, points+4, K()) ==
    CGAL::ON_BOUNDED_SIDE) {
      std::cout << "The point is inside the polygon." << std::endl;
    }
  return 0;
}
```

Boolean Set-Operations

- Used to perform regularized boolean-set operations on polygons (and general polygons) in 2D
- Includes intersection predicates, and point containment predicates
- Operations include:
  - Intersection
  - Join
  - Difference
  - Symmetric Difference
  - Complement



Boolean Set-Operations – Example



```
#include <CGAL/Exact_predicates_exact_constructions_
kernel.h>
#include <CGAL/Boolean_set_operations_2.h>

typedef CGAL::Exact_predicates_exact_constructions_
kernel K;
typedef K::Point_2                             Point_2;
typedef CGAL::Polygon_2<K>                     Polygon_2;
typedef CGAL::Polygon_with_holes_2<K>          Polygon_with_holes_2;
typedef std::list<Polygon_with_holes_2>        Pwh_list_2;

void main () {
  Point points_p[] = { Point(-1,1), Point(0,-1), Point(1,1)};
  Polygon_2 P(points_p, points_p + 3);
  Point points_q[] = { Point(-1,-1), Point(1,-1), Point(0,1)};
  Polygon_2 Q(points_q, points_q + 3);
  Pwh_list_2 R;  // intersection is a list of polygons with holes
  CGAL::intersection (P, Q, std::back_inserter(R));
}
```

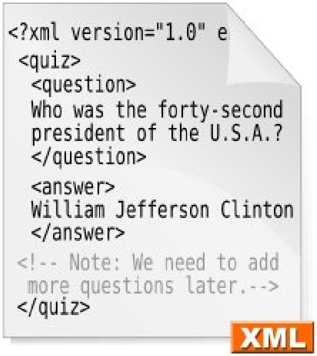
2D Minkowski Sums

Example using decomposition method:

```
typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef CGAL::Polygon_2<K>                                Polygon_2;
typedef CGAL::Polygon_with_holes_2<K>                     Polygon_with_holes_2;

void main () {
  std::ifstream    in_file ("polygons.dat");
  if (false == in_file.is_open()) {
    std::cerr << "Failed to open the input file." << std::endl;
    return;
  }
  Polygon_2   P, Q;
  in_file >> P >> Q;

  // Compute the Minkowski sum using the decomposition approach.
  CGAL::Small_side_angle_bisector_decomposition_2<K>  ssab_decomp;
  Polygon_with_holes_2  sum = minkowski_sum_2 (P, Q, ssab_decomp);
}
```

XML – eXtensible Markup Language

- *XML* is a general-purpose specification for creating custom markup languages
- Text based
- Very convenient for hierarchical (tree-like) data model
- Can represent common computer science data structures: records, lists and trees



- XML is heavily used as a format for document storage and processing, both online and off-line
- XML-based formats can be found in in: OpenOffice, RSS, SOAP protocol, XHTML, Microsoft Office, etc.

eXaMpLe – wikipedia-based

```
<recipe name="bread" prep_time="5 mins" cook_time="3 hours">
   <title>Basic bread</title>
   <ingredient amount="3" unit="cups">Flour</ingredient>
   <ingredient amount="0.25" unit="ounce">Yeast</ingredient>
   <ingredient amount="1.5" unit="cups" state="warm">Water</ingredient>
   <ingredient amount="1" unit="teaspoon">Salt</ingredient>
   <instructions>
     <step>Mix all ingredients together.</step>
     <step>Knead thoroughly.</step>
     <step>Cover with a cloth, and leave for one hour in warm room.</step>
     <step>Knead again.</step>
     <step>Place in a bread baking tin.</step>
     <step>Cover with a cloth, and leave for one hour in warm room.</step>
     <step>Bake in the oven at 350(degrees)F for 30 minutes.</step>
   </instructions>
 </recipe>
```

TOC