Adaptors in Generic Programming
Applied Geometric Computation

Efi Fogel
Tel Aviv Univ.

May 5, 2005

Type of Adaptors

An *adaptor* is anything that transforms one interface into another

- Iterator Adaptors
- Container Adaptors
- Function-Object Adaptors

Iterator Adaptors

The STL defines several iterator adaptors:

- `front_insert_iterator`
- `back_insert_iterator`
- `reverse_iterator`

`reveres_iterator<Iterator>`

- Enables backward traversal of a range
- Applying `operator++` is equivalent to applying `operator--` to an object of type `Iterator`
- `&*(reverse_iterator(i)) == &*(i-1)`
- Has same reference type and pointer type as `Iterator`
- Is mutable if and only if `Iterator` is mutable

Container Adaptors

- Provide a limited subset of container operations
  - Are not containers
- Are implemented in terms of underlying containers

  `stack` — a "last in first out" (LIFO) data structure

  `queue` — a "first in first out" (FIFO) data structure

  `priority_queue` — largest out

queue<T, Sequence = dequeue<T> >

- Only front and back (2) elements can be accessed
  - Has no iterators
  - Elements are added to the back
  - Elements may be removed from the front
- Makes clear that queue operations are performed (and no other)

`front()`, `back()`, `push_front()`, `push_back()` `pop_front()`, `pop_back()`

Function Objects

- The simplest kind of function object is an ordinary function pointer

```
bool is_even(int x) { return (x & 1) == 0; }
```

- A class that has an operator () is a function object

```
template <class Number>
struct even {
  bool operator()(Number x) const { return (x & 1) == 0; }
}
```

Algorithm `find_if`

```
template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred)
{
  while (first != last && !pred(*first)) ++first;
  return first;
}
```

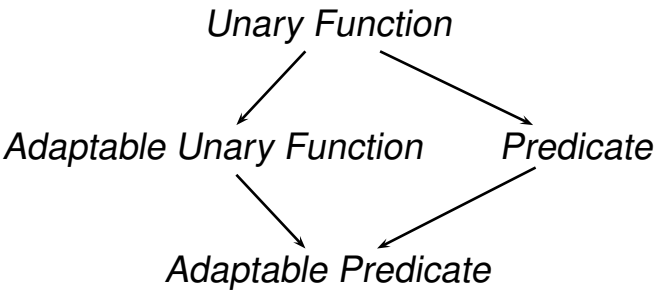
Searches for an element in the given range that satisfies the condition `pred`.

```
find_if(f, l, is_even) == find_if(f, l, even<int>())
```

Concept Hierarchy

A function object that takes a single arguments and returns a `true` or `false` result is called a *Predicate*.

The concept *Adaptable Unary Function* requires the provision of the types of its argument and result.



Adaptable Predicate

```
template <class Arg, class Result> struct unary_function {
  typedef Arg argument_type;
  typedef Result result_type;
}
```

A model of *Unary Function* is made a model of *Adaptable Unary Function* by inheriting from `unary_function`.

```
template <class Number>
struct even : public unary_function<Number, bool> {
  bool operator()(Number x) const { return (x & 1) == 0; }
}
```

`even` is a model of *Adaptable Predicate*, cause its `operator()` returns `bool`.

Function-Object Adaptors

- Enables function composition
- Enables composition of complicated operations out of simple ones
- Are easy to define (the problem reduces to picking a good name...)

Example

We want to find the first odd number in a range

It is possible to find the first even number in a range:
`find_if(f, l, even<int>())`

`find_if` searches for an element `e` of type `int` in the range `f,l` that satisfies the condition `even<int>()(e)`

Code should be reused, and not rewritten $\Rightarrow$ ~~odd<T>~~

Unary Negate

```
template <class AdaptablePredicate> class unary_negate {
private:
  AdaptablePredicate pred;

public:
  typedef typename AdaptablePredicate::argument_type argument_type;
  typedef typename AdaptablePredicate::result_type result_type;

  unary_negate(const AdaptablePredicate & x) : pred(x) {}
  bool operator()(const argument_type & x) const {
    return !pred(x);
  }
}
```

The template parameter `AdaptablePredicate` must be a model of *Adaptable Predicate*.

Putting it Together

- The expression `unary_negate<F>(f)` is cumbersome!
- Instead, define a tiny helper "higher-order" function `not1`

```
template <class AdaptablePredicate>
inline unary_negate<AdaptablePredicate>
not1(const AdaptablePredicate & pred) {
  return unary_negate<AdaptablePredicate>(pred);
}
```

  - `not1` is a function that operates on functions
- Putting it together

```
find_if(first, last, not1(even<int>()))
```

Boost

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

  concept check — generic-programming tools
  functional — enhancements of function-object adaptors
  graph — generic graph components and algorithms
  more
- Used by many programmers across a broad spectrum of applications
- Will become part of a future C++ Standard soon
- The Boost web site provides all necessary information

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 Planar Maps are custom-made graphs

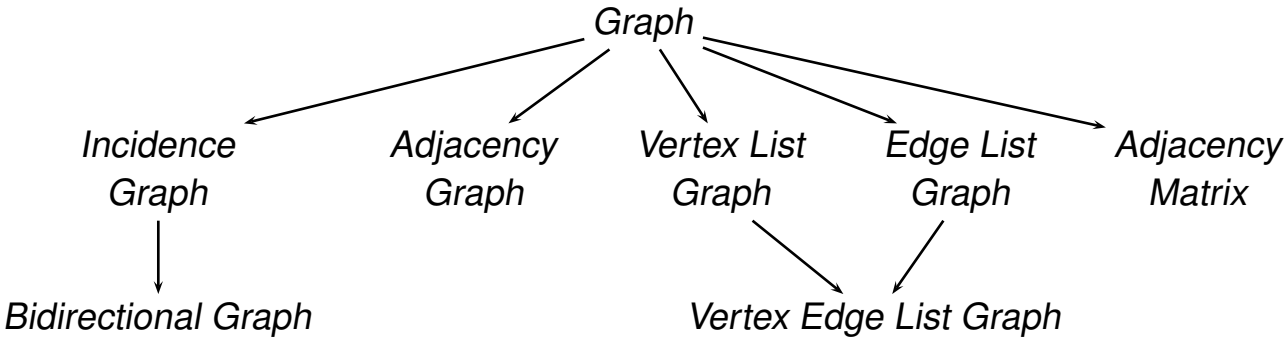
Extension through Visitors

- 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

Vertex and Edge Property Multi-Parameterization

- Multiple properties can be associated with both the vertices and the edges
- Properties are identified by a tag
- A *property map* is used to obtain the property value attached to a particular vertex or edge

Graph Concepts Hierarchy



Factoring the graph interface into many concepts $\Rightarrow$ Algorithm interfaces require only the minimum interface of a graph, thereby increasing the reusability of the algorithm.

Converting Planar Maps to BGL Graphs

- External Function Adaptation
  - BGL graph interface consists solely of free functions $\Rightarrow$ All free functions required by the interface must be overloaded
- Iterator Adaptation
  - Transform CGAL *Circulator* interface into *Iterator* interface
  - Transform *Iterator* interface to another

Graph Concept Type-Requirements

1. `vertex_descriptor` — a unique vertex in a graph
   Must be *Default Constructible*, *Assignable*, and *Equality Comparable*
2. `edge_descriptor` — a unique edge in a graph
   Must be *Default Constructible*, *Assignable*, and *Equality Comparable*
3. `directed_category` — Directed or undirected?
   `directed_tag` or `undirected_tag`
4. `edge_parallel_category` — Allow parallel edges?
   `allow_parallel_edge_tag` or `disallow_parallel_edge_tag`
5. `traversal_category` — the ways in which a graph vertices and edges can be visited

Vertex-List-Graph Concept Type-Requirements

1. `traversal_category` — must be convertible to `vertex_list_graph_tag`
2. `vertex_iterator` — provides access to all of the vertices in a graph
   - Value type of vertex iterator is vertex descriptor
3. `vertices_size_type` — represents the number of vertices

Vertex-List-Graph Concept
Function-Requirements

1. `pair<vertex_iterator, vertex_iterator> vertices(g)` — returns an iterator-range providing access to all the vertices in the graph `g`
2. `vertices_size_type num_vertices(g)` — returns the number of vertices in the graph `g`

Incidence-Graph Concept Type-Requirements

1. `traversal_category` — must be convertible to `incidence_graph_tag`
2. `out_edge_iterator` — provides access to the out-edges of a vertex
   - Value type of an out-edge iterator is edge descriptor
   - Must be *Multi Pass Input Iterator*
3. `degree_size_type` — represents the number out-edges or incident edges of a vertex

Incidence-Graph Concept
Function-Requirements

1. `vertex_descriptor source(e,g)` — returns the source vertex of edge `e`
2. `vertex_descriptor target(e,g)` — returns the target vertex of edge `e`
3. `pair<out_edge_iterator, out_edge_iterator> out_edges(u,g)` — returns an iterator-range providing access to the incident edges of vertex `u` in graph `g`
4. `degree_size_type out_degree(u, g)` — returns the number of incident edges of vertex `u` in graph `g`

Dijkstra Shortest Path — Synopsis

This algorithm solves the single-source shortest-paths problem on a weighted, directed or undirected graph for the case where all edge weights are nonnegative.

```
template <typename Graph, typename DijkstraVisitor,
          typename PredecessorMap, typename DistanceMap,
          typename WeightMap, typename VertexIndexMap,
          typename CompareFunction, typename CombineFunction,
          typename DistInf, typename DistZero>
void
dijkstra_shortest_paths(const Graph & g,
                        typename graph_traits<Graph>::vertex_descriptor s,
                        PredecessorMap predecessor, DistanceMap distance,
                        WeightMap weight, VertexIndexMap index_map,
                        CompareFunction compare, CombineFunction combine,
                        DistInf inf, DistZero zero, DijkstraVisitor vis)
```

Dijkstra Shortest Path — Parameters

1. `const Graph & g` — the graph object on which the algorithm will be applied. The type Graph must be a model of *Vertex List Graph* and *Incidence Graph*
2. `vertex_descriptor s` — the source vertex. All distances will be calculated from this vertex, and the shortest paths tree will be rooted at this vertex

Planar-Map Graph External Adaptor

```
#ifndef GRAPH_TRAITS_H
#define GRAPH_TRAITS_H

#include <CGAL/circulator.h>

namespace boost {
  // Type requirements:
  template <typename Dcel, typename Traits>
  struct graph_traits<CGAL::Planar_map_2<Dcel, Traits> > {
  };

  // IncidenceGraph graph function requirements:

  // VertexListGraph graph function requirements:
}

#endif
```

IncidenceGraph Function Requirements

```
//! Return the incident halfedge range:
inline std::pair<out_edge_iterator, out_edge_iterator>
out_edges(vertex_descriptor u, const graph & g) {}

//! Return the vertex degree:
inline
degree_size_type out_degree(vertex_descriptor u, const graph & g) {}

//! Return the edge source:
inline vertex_descriptor source(edge_descriptor e, const graph & g) {}

//! Return the edge target:
inline vertex_descriptor target(edge_descriptor e, const graph & g) {}
```

VertexListGraph graph function requirements

```
//! Return the planar map vertex range:
inline std::pair<vertex_iterator, vertex_iterator>
vertices(const graph & g) {}

//! Return the number of vertices:
inline vertices_size_type num_vertices(const graph & g) {}
```

Planar-Map Graph Traits — Overall

```
template <typename Dcel, typename Traits>
struct graph_traits<CGAL::Planar_map_2<Dcel, Traits> > {
public:
  // Requirements for graph concept:
  typedef directed_tag                          directed_category;
  typedef allow_parallel_edge_tag               edge_parallel_category;
  // typedef vertex_list_graph_tag                traversal_category;
  typedef incidence_graph_tag                   traversal_category;

  // IncidenceGraph graph type requirements:
  // out_edge_iterator, degree_size_type,
  // edge_descriptor, vertex_descriptor

  // VertexListGraph and EdgeListGraph graph type requirements:
  // vertex_iterator, vertices_size_type
  // edge_iterator, edges_size_type
};
```

Planar-Map Graph Traits — IncidenceGraph

```
template <typename Dcel, typename Traits>
struct graph_traits<CGAL::Planar_map_2<Dcel, Traits> > {
public:
  // Requirements for graph concept:

  // IncidenceGraph graph type requirements:
  typedef CGAL::Planar_map_2<Dcel, Traits>              Planar_map;
  typedef typename Planar_map::Halfedge_around_vertex_const_circulator
    Halfedge_around_vertex_const_circulator;

  typedef CGAL::Counting_iterator<Halfedge_around_vertex_const_circulator>
    Halfedge_around_vertex_const_iterator;
  typedef Halfedge_around_vertex_const_iterator         out_edge_iterator;
  typedef typename Planar_map::Size                     degree_size_type;
  typedef typename Planar_map::Halfedge                 edge_descriptor;
  typedef typename Planar_map::Vertex_const_handle      vertex_descriptor;
};
```

Planar-Map Graph Traits — ListGraph

```
template <typename Dcel, typename Traits>
struct graph_traits<CGAL::Planar_map_2<Dcel, Traits> > {
public:
  // Requirements for graph concept:

  // IncidenceGraph graph type requirements:

  // VertexListGraph graph type requirements:
  typedef typename Planar_map::Vertex_const_iterator   Base_vertex_iter;
  typedef Vertex_iterator_adaptor<Base_vertex_iter>    vertex_iterator;
  typedef typename Planar_map::Size                    vertices_size_type;

  // EdgeListGraph graph type requirements:
  typedef typename Planar_map::Halfedge_const_iterator edge_iterator;
  typedef typename Planar_map::Size                    edges_size_type;
};
```

Vertex Iterator Adaptor — Overall

```
template <typename Iterator>
class Vertex_iterator_adaptor {
public:
  typedef typename Iterator::iterator_category        iterator_category;
  typedef Iterator                                    value_type;
  typedef typename Iterator::difference_type          difference_type;
  typedef Iterator*                                   pointer;
  typedef Iterator&                                   reference;

private:
  Iterator m_it;

public:
  // Constructors:

  // Operators:
};
```

Vertex Iterator Adaptor — Constructors

```
template <typename Iterator>
class Vertex_iterator_adaptor {
private:
  Iterator m_it;

public:
  // Default constructor:
  Vertex_iterator_adaptor () : m_it() {}

  // Constructor:
  Vertex_iterator_adaptor (Iterator it) : m_it(it) {}

  // Operators:
};
```

Vertex Iterator Adaptor — Operators

```
template <typename Iterator>
class Vertex_iterator_adaptor {
public:
  // operator*:
  const Iterator & operator*() const { return m_it; }
  Iterator & operator*()  return m_it;

  // Equality operators:
  bool operator==(const Vertex_iterator_adaptor & it) const
  { return m_it == it.m_it; }

  bool operator!=(const Vertex_iterator_adaptor & it) const
  { return m_it != it.m_it; }

  // More operators:
};
```

Vertex Iterator Adaptor — Increment

```
template <typename Iterator>
class Vertex_iterator_adaptor {
public:
  // operator++:
  Vertex_iterator_adaptor & operator++()
  {
    ++m_it;
    return *this;
  }

  Vertex_iterator_adaptor operator++(int)
  {
    Vertex_iterator_adaptor temp = *this;
    ++m_it;
    return temp;
  }
};
```

Dijkstra Shortest Path

```
Planar_map pm;
graph_traits::vertex_descriptor vd = pm.vertices_begin();

// Property maps:
Vertex_vertex_map pred_map;
boost::associative_property_map<Vertex_vertex_map> pred_pmap(pred_map);

Vertex_int_map distance_map;
boost::associative_property_map<Vertex_int_map> distance_pmap(distance_map);

Vertex_int_map index_map;
boost::associative_property_map<Vertex_int_map> index_pmap(index_map);

Edge_map weight_map;
boost::associative_property_map<Edge_map> weight_pmap(weight_map);
```

Dijkstra Shortest Path

```
boost::dijkstra_shortest_paths(pm, vd, pred_pmap, distance_pmap,
                               weight_pmap, index_pmap,
                               std::less<int>(),
                               boost::closed_plus<int>(),
                               std::numeric_limits<int>::max(), 0,
                               boost::default_dijkstra_visitor());
```

TOC

Incidence-Graph Concept Function-Requirements ❖

Dijkstra Shortest Path — Synopsis ❖

Dijkstra Shortest Path — Parameters ❖

Planar-Map Graph External Adaptor ❖

IncidenceGraph Function Requirements ❖

VertexListGraph graph function requirements ❖

Planar-Map Graph Traits — Overall ❖

Planar-Map Graph Traits — IncidenceGraph ❖

Planar-Map Graph Traits — ListGraph ❖

Vertex Iterator Adaptor — Overall ❖

Vertex Iterator Adaptor — Constructors ❖

Vertex Iterator Adaptor — Operators ❖

Vertex Iterator Adaptor — Increment ❖

Dijkstra Shortest Path ❖

Dijkstra Shortest Path ❖