Generic Programming and the Boost Graph Library

Jeremy Siek

Department of Electrical, Computer, and Energy Engineering
University of Colorado at Boulder

BoostCon 2010

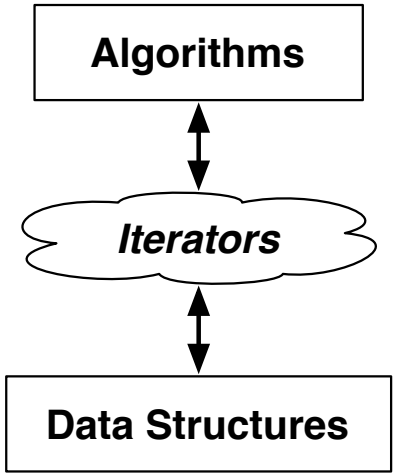
Outline

Please, stop me and ask questions at **any time**!

Generic Programming

- Generic programming is a *methodology* for developing software libraries that are highly **reusable** and **efficient**.
- Main idea: when implementing algorithms, make minimal assumptions about data representations of the inputs.
- e.g. the Standard Template Library

Alexander Stepanov



David Musser

The object-oriented approach

```
struct vector {
  void merge(const vector& a, const vector& b) { ... }
  void reverse() { ... }
  void sort() { ... }
  ...
};
struct list {
  void merge(const list& a, const list& b) { ... }
  void reverse() { ... }
  void sort() { ... }
  ...
};
```

- If we have M algorithms and N data structures, then we have $O(M \times N)$ code to write!

The generic programming approach

```
template<class InputIter1, class InputIter2, class OutputIter>
OutputIter merge(InputIter1 first1, InputIter1 last1,
                 InputIter2 first2, InputIter2 last2, OutputIter result);

template<class BidirectionalIter>
void reverse(BidirectionalIter first, BidirectionalIter last);

template<class RandomAccessIter>
void sort(RandomAccessIter first, RandomAccessIter last);
```

```
struct vector {
  struct iterator;
  iterator begin();
  iterator end();
};
```

```
struct list {
  struct iterator;
  iterator begin();
  iterator end();
};
```

- We write $O(M + N)$ code, leaving more time for skiing!

Efficiency

- Function templates are just as efficient as normal functions.
- The compiler stamps out specialized versions of the template.

```
template<class T>
const T& min(const T& x,
             const T& y)
 { return y < x ? y : x; }

min(1, 2);
min(1.0, 2.0);
```

$\Longrightarrow$

```
const int& min(const int& x,
               const int& y)
  { return y < x ? y : x; }

const float& min(const float& x,
                 const float& y)
  { return y < x ? y : x; }
```

- The cost: longer compile times.

Templates and Type Requirements

- Which types can be used with a given template?

```
1  int main() {
2    vector<int> v;
3    sort(v.begin(), v.end()); // ok
4
5    list<int> l;
6    sort(l.begin(), l.end()); // error!
7  }
```

```
stl_algo.h: In function 'void std::sort(_RandomAccessIterator,
                                        _RandomAccessIterator)
  [with _RandomAccessIterator = std::_List_iterator<int>]':
sort-error.cpp:6: instantiated from here
stl_algo.h:2570: error: no match for 'operator-' in '__last - __first'
```

Type Requirements and Concepts

- The requirements of a template are expressed in terms of "concepts".
- A *concept* is set of requirements.
- When a particular type satisfies the requirements of a concept, we say that the type *models* the concept. (Think "implements".)

```
template <class Iter>
void sort(Iter first, Iter last);
```

Requirements on types:

- Iter is a model of Random Access Iterator.
- Iter's value type is a model of Less Than Comparable.

Concepts

concept **Random Access Iterator**

A Random Access Iterator is an iterator that provides both increment and decrement (just like a Bidirectional Iterator), and that also can move forward and backward in arbitrary-sized steps in constant-time.

Refinement of Bidirectional Iterator and Less Than Comparable

Valid expressions:

Expression	Return type
i += n	X&
i + n	X&
i -= n	X&
i - n	X
i[n]	Convertible to value_type
i[n] = t	Convertible to value_type

Concepts

In general, a concept may include the following kinds of requirements:

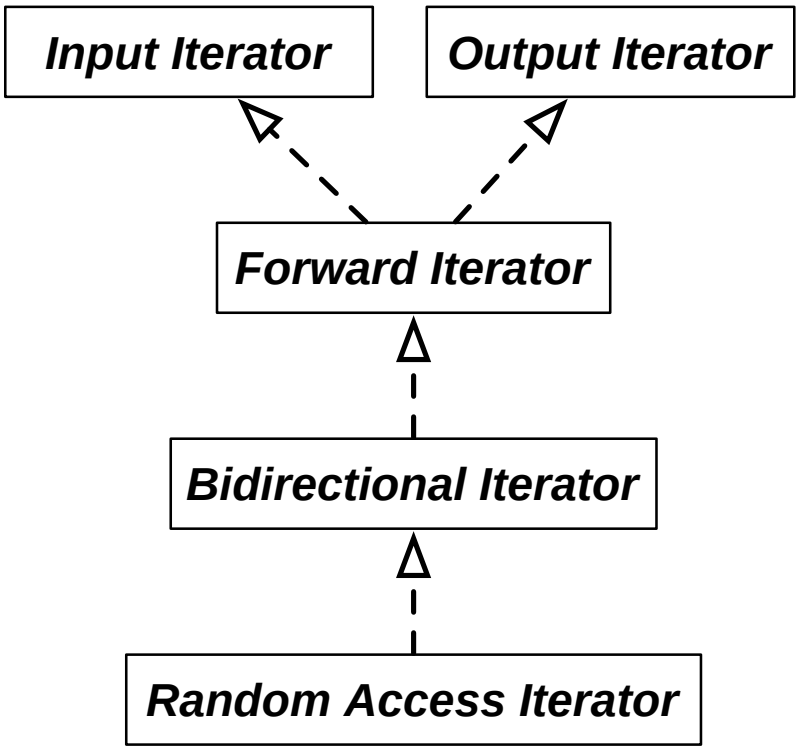
- valid expressions
- associated types, and requirements on those associated types
- refinements
- efficiency guarantees for operations
- semantic requirements for operations (pre and post-conditions, invariants, etc.)

Concepts and Models

- vector<int>::iterator is a model of Random Access Iterator.
- list<int>::iterator is not.

```
1  int main() {
2    vector<int> v;
3    sort(v.begin(), v.end()); // ok
4
5    list<int> l;
6    sort(l.begin(), l.end()); // error!
7  }
```

The Iterator Concept Hierarchy



- Where do concepts come from?
- Is it better to use few or many requirements in a template?
- When is it time to define a new concept?

Associated Types

```
template <class Iter>
void sort(Iter first, Iter last);
```

Requirements on types:

- Iter is a model of Random Access Iterator.
- Iter's value type is a model of Less Than Comparable.

An *associated type* is a helper type that is involved in the operations of a concept.

A model must specify particular types to play the roles of the associated types in the concept.

Example of Associated Types

concept **Input Iterator**

An Input Iterator is an iterator that may be dereferenced to refer to some object, and that may be incremented to obtain the next iterator in a sequence.

Associated types: value_type, difference_type

Valid expressions:

Expression	Return type
*i	Convertible to value_type
++i	X&

```
template<class T> struct list_iterator { // models Input Iterator
  typedef T value_type;
  typedef ptrdiff_t difference_type;
  T& operator*() const { return current->data; }
  list_iterator& operator++() { current = current->next; return *this; }
};
```

Accessing Associated Types in Templates

```
template<typename InputIterator1, typename InputIterator2>
void iter_swap(InputIterator1 a, InputIterator2 b)
{
  typedef typename InputIterator1::value_type T;
  T tmp = *a;
  *a = *b;
  *b = tmp;
}
```

- Problem: only class types may have nested typedefs. We'd like iter_swap to also work with built-in types like pointers.
- Solution: add a level of indirection using a traits class.

Accessing Associated Types in Templates

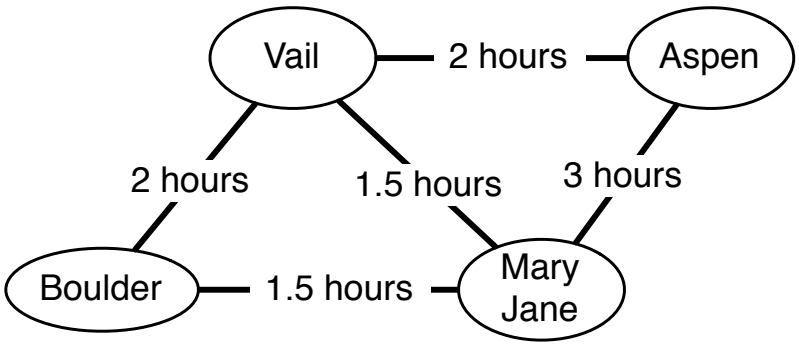
```
// "Default" version for class types
template<class Iter> struct iterator_traits {
  typedef typename Iter::value_type value_type;
  typedef typename Iter::difference_type difference_type;
};
// Partial specialization for pointer types.
template<class T> struct iterator_traits<T*> {
  typedef T value_type;
  typedef ptrdiff_t difference_type;
};

template<typename InputIterator1, typename InputIterator2>
void iter_swap(InputIterator1 a, InputIterator2 b) {
  typedef typename iterator_traits<InputIterator1>::value_type T;
  T tmp = *a; *a = *b; *b = tmp;
}
```

Generic Programming Recap

- Decouple algorithms from data structures using iterators, reducing the amount of code from $O(M \times N)$ to $O(M + N)$.
- Create *concepts* to describe and organize the type requirements for templates.
- Create traits classes to access the associated types of a concept.
- A type that satisfies the requirements for a concept is said to *model* the concept.
- Minimize the requirements on a template to maximize the potential for reuse.
- Any other questions at this point?

Graphs (aka. Networks)

- An *abstraction* is a simplified model of some problem that allows us to focus on the important parts and ignore the irrelevant parts.
- The graph abstraction is commonly used to solve problems in areas such as Internet packet routing, telephone network design, software build systems, molecular biology, and so on.
- A *graph* is a collection of vertices and edges, where each edge connects two vertices. In a *directed graph*, each edge points from a source vertex to a target vertex.



Graph Data Structures

- Edge list

  {(Boulder,Vail), (Boulder,Mary Jane), (Vail, Mary Jane), (Vail, Aspen), (Mary Jane, Aspen) }

- Adjacency list

  Boulder: {Vail, Mary Jane}
  Vail: {Boulder, Mary Jane, Aspen}
  Mary Jane: {Boulder, Vail, Aspen}
  Aspen: {Vail, Mary Jane}

- Adjacency matrix

	Boulder	Vail	Mary Jane	Aspen
Boulder		✓	✓	
Vail	✓		✓	✓
Mary Jane	✓	✓		✓
Aspen		✓	✓	

- Nodes with pointers, application specific structures, etc.

Graph Algorithms

- Problem: how can I find my way out of a maze?
- Solution: depth-first search, that is, mark where you've been and backtrack when you hit a dead end.
- Trivia: what's the difference between a maze and a labyrinth?

```
DFS(G,u)
  if u is the exit
    return Success
  mark[u] ← BLACK
  for each v ∈ Adjacent(u)
    if mark[v] = WHITE
      if DFS(G, v) = Success
        return Success
  return Failure
```

To create a generic implementation, what should the type requirements be?

Requirements for a generic Depth-First Search

- Need some way to refer to vertices.
- Need to access the vertices adjacent to a given vertex.
- Need to mark a vertex with color (black or white).
- Need a way to carry out custom actions during the search, such as checking for success and terminating.

What concepts should we create/use?
What requirements should they include?

The Adjacency Graph concept

concept **Adjacency Graph**

The Adjacency Graph concept defines the interface for accessing adjacent vertices within a graph.

Associated types: vertex_descriptor, adjacency_iterator (accessed through graph_traits). The adjacency_iterator must model Multi-Pass Input Iterator and its value_type must be vertex_descriptor.

Valid expressions:

Expression	Return type
adjacent_vertices(v,g)	pair<adjacency_iterator>

where g is a graph and v is a vertex_descriptor.

Requirements for a generic Depth-First Search

- Need some way to refer to vertices.
- Need to access the vertices adjacent to a given vertex.
- **Need to mark a vertex with color (black or white).**
- Need a way to carry out custom actions during the search, such as checking for success and terminating.

What concepts should we create/use?
What requirements should they include?

The Property Map concepts

concept **Readable Property Map**

Refinement of Copy Constructible

Associated types: key_type, value_type, and reference (accessed through property_traits). reference is convertible to value_type.

Valid expressions:

Expression	Return type
get(pmap, k)	reference

concept **Writable Property Map**

Refinement of Copy Constructible

Associated types: same as Readable Property Map

Valid expressions:

Expression	Return type
put(pmap, k, v)	void

Requirements for a generic Depth-First Search

- Need some way to refer to vertices.
- Need to access the vertices adjacent to a given vertex.
- Need to mark a vertex with color (black or white).
- **Need a way to carry out custom actions during the search, such as checking for success and terminating.**

What concepts should we create/use?
What requirements should they include?

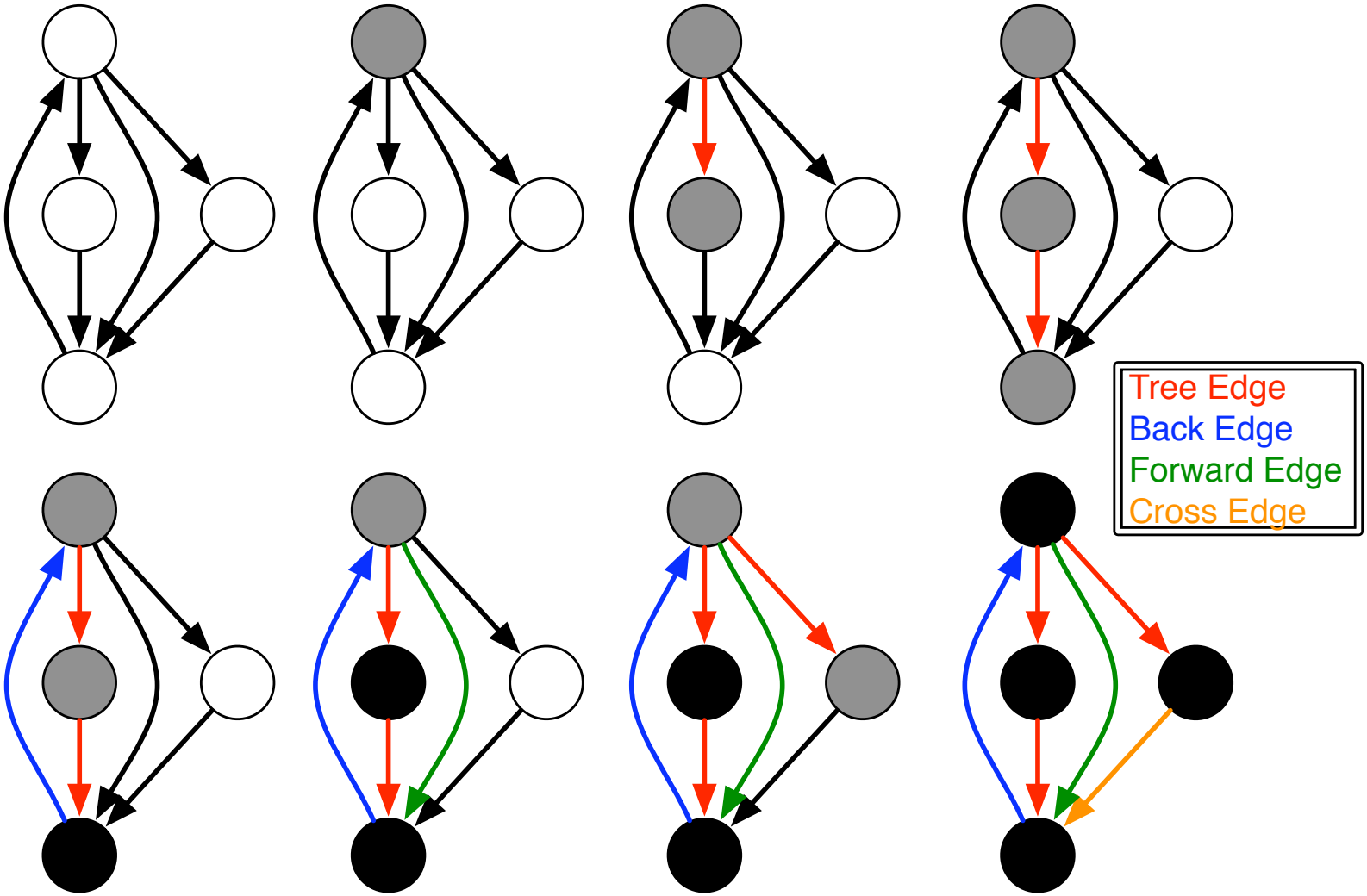
Depth-First Search, in Depth

We can extract more information during a depth-first search by using a more discerning color scheme:

- White for a vertex that is not yet visited.
- Gray for a vertex that has been visited but there are vertices reachable from it that are not yet visited.
- Black for a visited vertex for which all vertices reachable from it have also been visited.

For example, during the search, if we run into a vertex that is already gray, then we have detected a cycle.

Depth-First Search, in Depth



The Depth-First Search Visitor Concept

concept **DFS Visitor**

Valid expressions:

Expression	When
vis.initialize_vertex(v,g)	Before starting the search.
vis.discover_vertex(v,g)	First time the vertex is encountered.
vis.examine_edge(e,g)	After the source vertex is discovered but before the target is.
vis.tree_edge(e,g)	When the edge is added to the DFS-tree.
vis.back_edge(e,g)	When the target vertex is an ancestor of the source vertex in the DFS-tree.
vis.forward_or_cross_edge(e,g)	When the source and target are not descended from each other in the DFS-tree.

where g is a graph, v is a vertex_descriptor, and e is an edge_descriptor.

We need access to the out-edges...

concept **Incidence Graph**

The Incidence Graph concept defines the interface for accessing out-edges of a vertex within a graph.

Associated types: vertex_descriptor, edge_descriptor, out_edge_iterator (accessed through graph_traits). The out_edge_iterator must model Multi-Pass Input Iterator and its value_type must be edge_descriptor.

Valid expressions:

Expression	Return type
out_edges(v,g)	pair<out_edge_iterator>
source(e,g)	vertex_descriptor
target(e,g)	vertex_descriptor

where g is a graph, v is a vertex_descriptor, and e is an edge_descriptor.

Depth-First Search Function Template

```
template<class Graph, class Map, class Visitor>
void depth_first_visit(const Graph& G,
                       typename graph_traits<Graph>::vertex_descriptor u,
                       Map color,
                       Visitor vis);
```

Requirements on types:

- Graph is a model of Incidence Graph
- Map is a model of Readable and Writable Property Map. Map's key_type is the Graph's vertex_descriptor and the Map's value_type is bool.
- Visitor is a model of DFS Visitor. Visitor's vertex and edge types are the same as the Graph's.

Quiz Time

- How can you use this generic depth_first_visit to record a path that will find the way out of a maze?
- How can you use this generic depth_first_visit to detect a cycle in the dependencies of a makefile?

Depth-First Search Function Template

```
template<class Graph, class Map, class Visitor>
void depth_first_visit(const Graph& G,
                       typename graph_traits<Graph>::vertex_descriptor u,
                       Map color, Visitor vis)
{
  typedef typename graph_traits<Graph>::vertex_descriptor vertex;
  typedef typename graph_traits<Graph>::edge_descriptor edge;
  put(color, u, gray); vis.discover_vertex(u, G);
  for (edge e : out_edges(u, G)) { // new for-loop in C++0x!
    vertex v = target(e, G); vis.examine_edge(e, G);
    if (get(color, v) == white) {
      vis.tree_edge(e, G);
      depth_first_visit(G, v, color, vis);
    } else if (get(color, v) == gray) vis.back_edge(e, G);
    else vis.forward_or_cross_edge(e, G);
  }
  put(color, u, black);
}
```

Algorithms in the Boost Graph Library

- Breadth-First Search
- Shortests Paths: Dijkstra's, Bellman-Ford, Johnson's All-Pairs, Floyd-Warshall All-Pairs
- Minimum Spanning Tree: Kruskal's, Prim's
- Connected components, strongly connected components, biconnected components
- Max flow: Edmunds-Karp, Push relabel
- Sparse Matrix Ordering: Cuthill-McKee, King, Minimum Degree
- Topological Sort
- Transitive Closure
- Isomorphism
- ...

Graph Classes in the Boost Graph Library

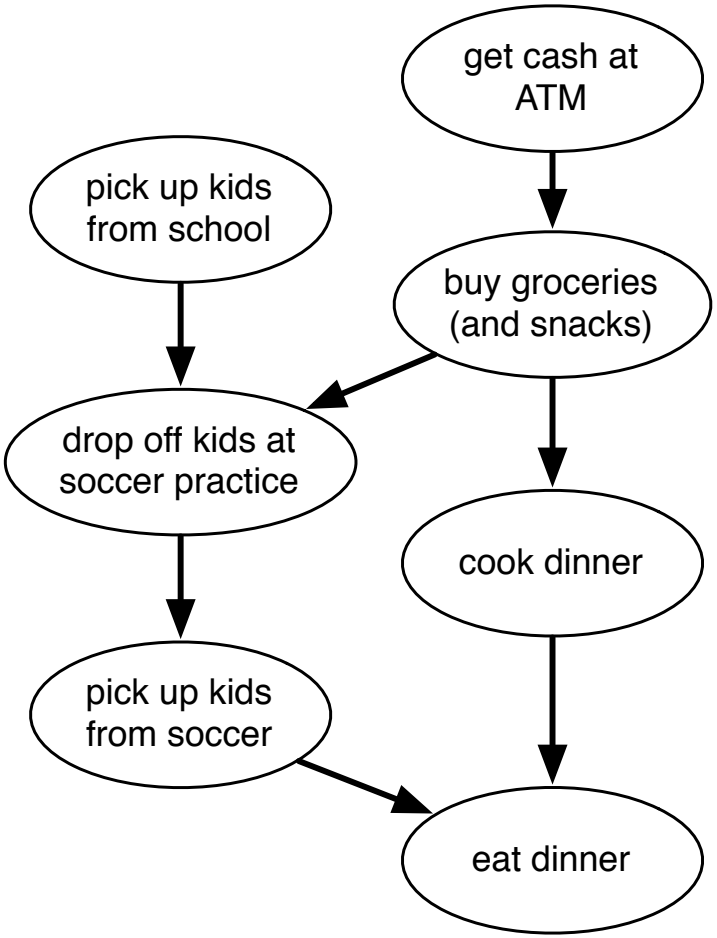
Graphs:

- adjacency_list: a swiss-army knife providing many variations on the traditional adjacency list.
- adjacency_matrix: the traditional adjacency matrix representation.

Graph Adaptors:

- reverse_graph
- filtered_graph
- edge_list
- Vector as Graph
- Matrix as Graph
- Leda Graph
- Stanford GraphBase

Scheduling your day with topological sort



A *topological ordering* is a total ordering on vertices, call it $<$, such that if $u \longrightarrow v$ then $u < v$.

Topological Sort

```
template <typename VertexListGraph, typename OutputIterator,
          typename P, typename T, typename R>
void topological_sort(VertexListGraph& g, OutputIterator result,
    const bgl_named_params<P, T, R>& params = /*all defaults*/);
```

Parameters

IN: VertexListGraph& g
A directed acylic graph (DAG). The graph type must be a model of Vertex List Graph. If the graph is not a DAG then a not_a_dag exception will be thrown and the user should discard the contents of result range.

OUT: OutputIterator result
The vertex descriptors of the graph will be output to the result output iterator in **reverse** topological order. The iterator type must model Output Iterator.

Topological Sort (cont'd)

Named Parameters

UTIL/OUT: color_map(ColorMap color)
This is used to keep track of progress through the graph. The type ColorMap must be a model of Read/Write Property Map and its key type must be the graph's vertex descriptor type and the value type of the color map must model Color Value.
Default: an iterator_property_map created from a std::vector of default_color_type of size num_vertices(g) and using the i_map for the index map.

IN: vertex_index_map(VertexIndexMap i_map)
This maps each vertex to an integer in the range [0, n) where n is the number of vertices in the graph. This parameter is only necessary when the default color property map is used. The type VertexIndexMap must be a model of Readable Property Map. The value type of the map must be an integer type. The vertex descriptor type of the graph needs to be usable as the key type of the map.
Default: get(vertex_index, g)

Putting topological sort to work

```
#include <deque>
#include <iostream>
#include <boost/graph/topological_sort.hpp>
#include <boost/graph/adjacency_list.hpp>
using namespace boost;
using namespace std;

int main() {
  // Create labels for each of the tasks
  // Create the graph
  // Perform the topological sort and output the results
}
```

Create labels for each of the tasks

```
const char *tasks[] = {
  "pick up kids from school",
  "buy groceries (and snacks)",
  "get cash at ATM",
  "drop off kids at soccer practice",
  "cook dinner",
  "pick up kids from soccer",
  "eat dinner"
};
const int num_tasks = sizeof(tasks) / sizeof(char *);
```

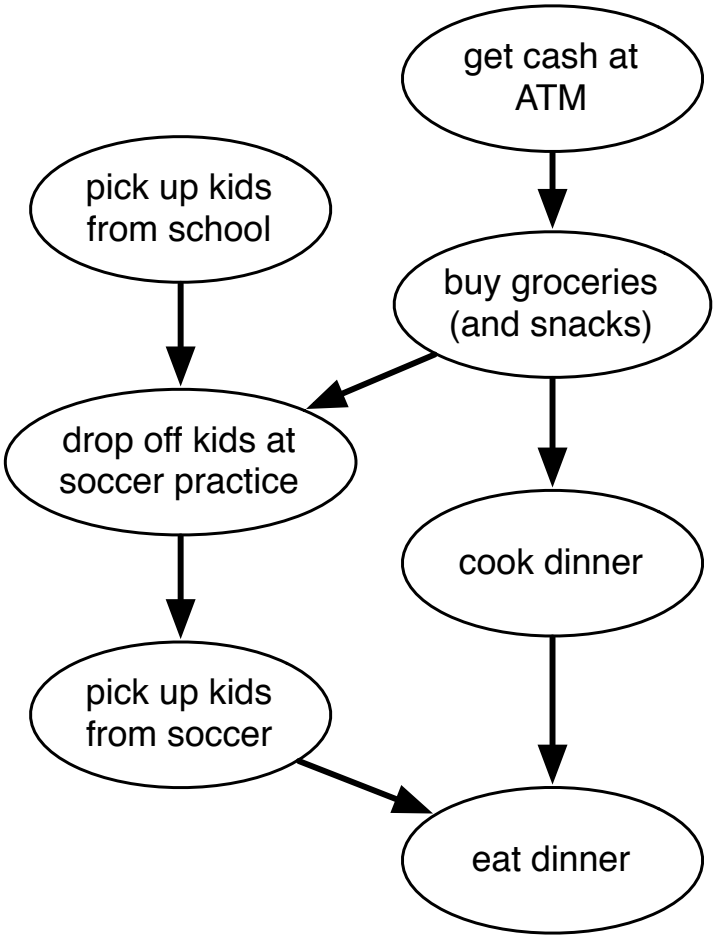
Create the graph

```
// Using the adjacency_list template with default parameters
adjacency_list<> g(num_tasks);

// Add edges between vertices
// With this variant of adjacency_list, vertex_descriptor == int
add_edge(0, 3, g);
add_edge(1, 3, g); add_edge(1, 4, g);
add_edge(2, 1, g);
add_edge(3, 5, g);
add_edge(4, 6, g);
add_edge(5, 6, g);
```

Perform the topological sort and output the results

```
// The front insertion reverses the reversed ordering
// produced by topological_sort.
deque<int> topo_order;
topological_sort(g, front_inserter(topo_order));

for (int v : topo_order)
  cout << tasks[v] << endl;
```

The topological ordering of your schedule



1. get cash at ATM
2. buy groceries (and snacks)
3. cook dinner
4. pick up kids from school
5. drop off kids at soccer practice
6. pick up kids from soccer
7. eat dinner

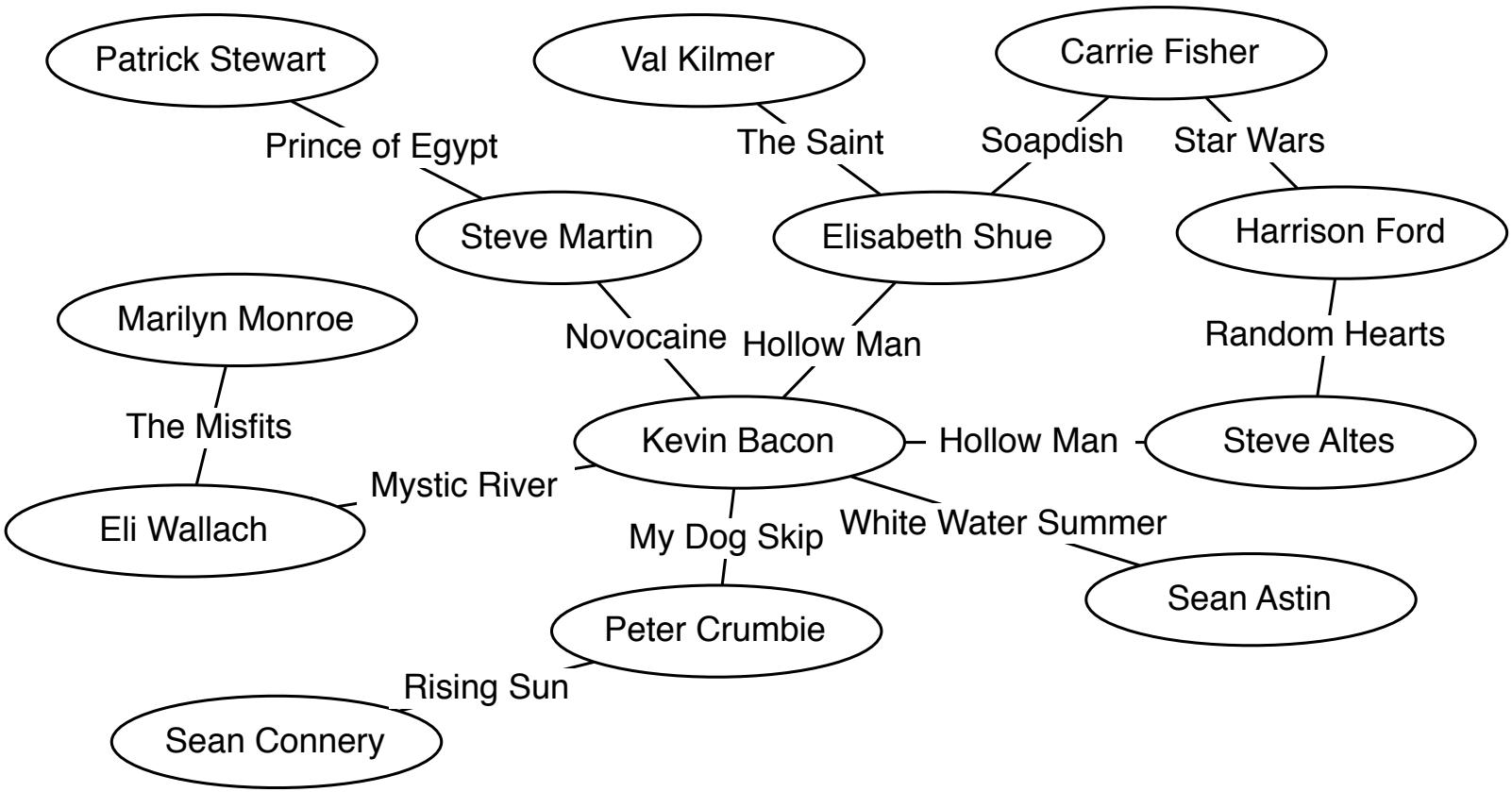
Challenge question

- How can you implement topological_sort using the generic depth_first_search?
- Hint: remember, topological_sort computes the reverse topological ordering.

Topological Sort Implementation using DFS

```
template <typename OutputIterator>
struct topo_sort_visitor : public dfs_visitor<> {
  topo_sort_visitor(OutputIterator iter) : out_iter(iter) { }

  template <typename Vertex, typename Graph>
  void finish_vertex(const Vertex& u, Graph&) { *out_iter++ = u; }

  OutputIterator out_iter;
};

template <typename VertexListGraph, typename OutputIterator>
void topological_sort(VertexListGraph& g, OutputIterator result) {
  typedef topo_sort_visitor<OutputIterator> TopoVisitor;
  depth_first_search(g, visitor(TopoVisitor(result)));
}
```

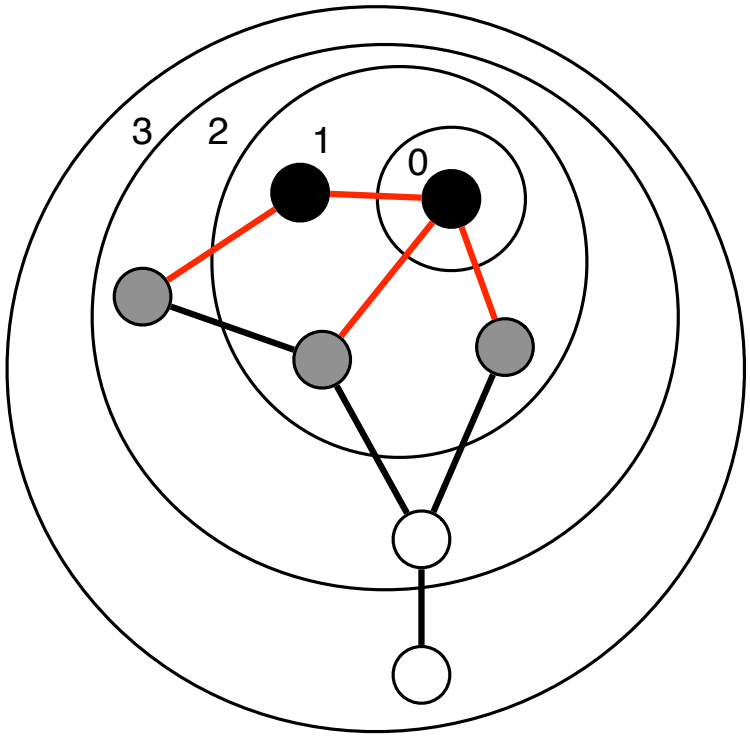
Six degrees of Kevin Bacon



An actor's *Bacon number* is the shortest path to Kevin Bacon through a trail of actors who appear together in the same movie.

Breadth-First Search

A breadth-first search visits all of the vertices in a graph reachable from the starting vertex, with vertices closer to the starting vertex visited before vertices that are farther away. The BFS-tree gives the shortest path from the source to every reachable vertex.



Attaching data to vertices and edges

```
#include <string>
#include <boost/graph/adjacency_list.hpp>
using namespace std; using namespace boost;

struct actor {
  string name;
};
struct movie {
  string name;
};
typedef adjacency_list<vecS, vecS, undirectedS, actor, movie> graph;
typedef graph_traits<graph>::vertex_descriptor vertex;
typedef graph_traits<graph>::edge_descriptor edge;
```

Enable serialization to and from an archive

```
#include <string>
#include <boost/graph/adjacency_list.hpp>
using namespace std; using namespace boost;

struct actor {
  string name;
  template<class Archive>
  void serialize(Archive & ar, const unsigned int version)
    { ar & name; }
};
struct movie {
  string name;
  template<class Archive>
  void serialize(Archive & ar, const unsigned int version)
    { ar & name; }
};
```

The Bacon number program

```
#include <iostream>
#include <fstream>
#include <boost/archive/text_iarchive.hpp>

int main() {
  ifstream ifs("./kevin-bacon.dat");
  archive::text_iarchive ia(ifs);
  graph g;
  ia >> g;
  vector<int> bnum(num_vertices(g));
  // Find Kevin Bacon and perform breadth-first search
  for (vertex v : vertices(g))
    cout<<"bacon_number(" << g[v].name <<")= " << bnum[v] <<"\n";
}
```

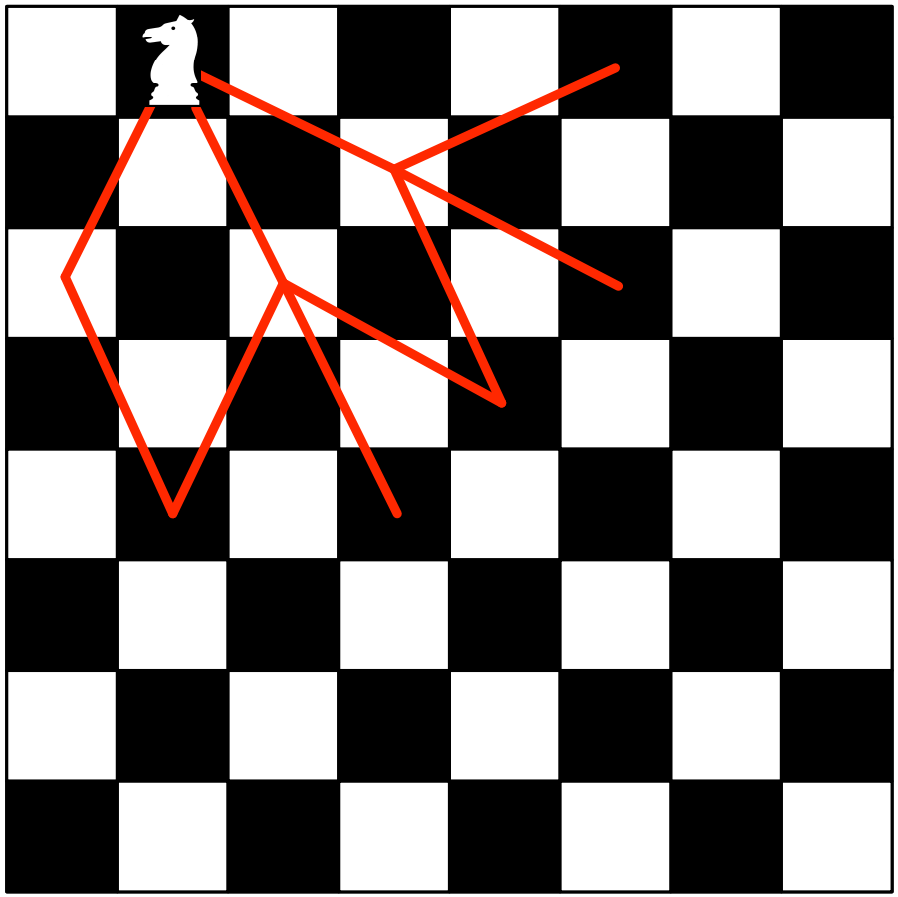
Find Kevin Bacon and perform breadth-first search

```
struct bacon_number_recorder : public default_bfs_visitor {
  bacon_number_recorder(vector<int>& distances) : d(distances) { }
  void tree_edge(edge e, const graph& g) const {
    vertex u = source(e, g), v = target(e, g);
    d[v] = d[u] + 1;
  }
  vector<int>& d;
};
// Find Kevin Bacon, the source
vertex src;
for (v : vertices(g))
  if (g[v].name == "Kevin Bacon") { src = v; break; }
// Perform breadth-first search
breadth_first_search(g, src, visitor(bacon_number_recorder(bnum)));
```

Quiz time

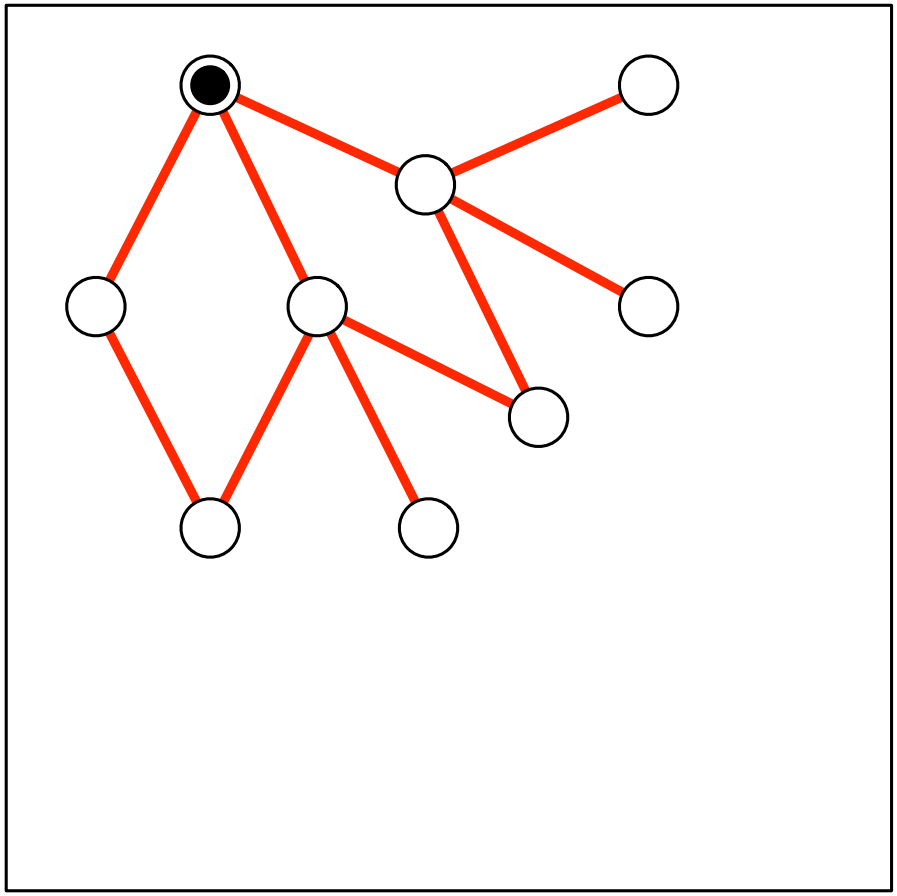
- Suppose you want to report the shortest path that links an actor to Kevin Bacon. How can you use breadth_first_search to compute this?

A Knight's Tour and Implicit Graphs



Can the Knight make a tour around the chess board, touching each square exactly once?

A Knight's Tour



Find a Hamiltonian path through the Knight's Graph.

Hamiltonian Path via Backtracking Search

```
template<typename Graph, typename TimePropertyMap>
bool backtracking_search
      (Graph& g, typename graph_traits<Graph>::vertex_descriptor u,
       TimePropertyMap time_stamp, int n) {
  put(time_stamp, u, n);
  if (n == num_vertices(g) - 1) return true;
  bool result = false;
  for (v : adjacent_vertices(u, g))
    if (get(time_stamp, v) == -1) {
      result = backtracking_search(g, v, time_stamp, n + 1);
      if (result == true) break;
    }
  if (result == false)
    put(time_stamp, u, -1);
  return result;
}
```

Brainstorm

- How do we create a Knight's graph that we can use with the backtracking_search function template?

Adjacency Iterator for Knight's Graph

```
typedef pair<int,int> point;
point knight_jumps[8] = { point(2, -1), point(1, -2), point(-1, -2),
    point(-2, -1), point(-2, 1), point(-1, 2), point(1, 2), point(2, 1) };

struct knight_adjacency_iterator {
  knight_adjacency_iterator(int ii, point p, const knights_graph & g)
    : m_pos(p), m_g(&g), m_i(ii) { valid_position(); }
  point operator*() const { return m_pos + knight_jumps[m_i]; }
  void operator++() { ++m_i; valid_position(); }
  bool operator==(const knight_adjacency_iterator & x) const
    { return m_i == x.m_i; }
protected:
  point m_pos;
  const knights_graph* m_g;
  int m_i;
};
```

An Implicit Knight's Graph

```
struct knights_graph {
  typedef point vertex_descriptor;
  typedef pair<vertex_descriptor> edge_descriptor;
  typedef knight_adjacency_iterator adjacency_iterator;
  knights_graph(int n): m_board_size(n) { }
  int m_board_size;
};
int num_vertices(const knights_graph& g)
  { return g.m_board_size * g.m_board_size; }

const pair<knights_graph::adjacency_iterator>
adjacent_vertices(knights_graph::vertex_descriptor v,
                  const knights_graph & g) {
  typedef knights_graph::adjacency_iterator Iter;
  return make_pair(Iter(0, v, g), Iter(8, v, g));
}
```

A property map for recording the time stamps

```
struct board_map {
  typedef int value_type;
  typedef point key_type;
  typedef read_write_property_map_tag category;
  board_map(int n) : m_board(new int[n]), m_size(n)
    { fill(m_board, m_board + n * n, -1); }
  int *m_board;
  int m_size;
};
int get(const board_map& ba, point p) {
  return ba.m_board[p.first * ba.m_size + p.second];
}
void put(const board_map& ba, point p, int v) {
  ba.m_board[p.first * ba.m_size + p.second] = v;
}
```

Searching for a Knight's Tour

```
const int N = 8;
knights_graph g(N);
board_map chessboard(N);

bool ret = backtracking_search(g, point(0, 0), chessboard, 0);

for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j)
    cout << get(chessboard, point(i, j)) << "\t";
  cout << endl;
}
```

A Knight's Tour

0 23 8 61 38 25 44 59
9 62 1 24 43 60 37 26
22 7 10 39 20 41 58 45
63 2 21 42 57 46 27 36
6 11 4 19 40 35 56 47
3 16 13 30 49 28 53 34
12 5 18 15 32 51 48 55
17 14 31 50 29 54 33 52

Conclusion

- Hope you enjoyed this whirlwind tour of the Boost Graph Library!
- Generic programming can be applied to many more domains than sequences (as in the STL). Let's do it and create more reusable libraries!
- Generic libraries have a steep learning curve, but don't give up! Once you've mastered one, it becomes a powerful tool.
- Interested in adding algorithms or data structures to the BGL? Contribute to Boost!