Advanced Programming Practice Exam — Some Answers

CSCI-6090

October 18, 2001

1. The word frequency program discussed in this course illustrated several properties of the associative container classes `map` and `multimap`. Here is an excerpt from that program:

```
typedef map<string, int>
  frequency_map;  // Type of table that holds words and frequencies
typedef istream_iterator<string>
  string_input;   // Type of iterator for traversing an input stream

frequency_map fm;
for (string_input j(cin); j != string_input(); ++j)
  ++fm[normalize(*j)]; // Increment frequency of normalized string
```

(The function `normalize` translates characters to lower case and eliminates punctuation characters in its string argument.) It may appear that this code is in error, in that the line

```
++fm[normalize(*j)]; // Increment frequency of normalized string
```

should have a test to check whether `fm` is already defined at a key:

```
if (fm.find(normalize(*j)) == fm.end()) // If normalized string is not
  fm[normalize(*j)] = 1;              // present, initialize its count
else
  ++fm[normalize(*j)];  // Increment frequency of normalized string
```

Explain why this check and initialization step aren't necessary, based on a property of the `map` `operator[]`.

When fm[k] appears in an expression but fm is undefined at key k, operator[] inserts an entry in the map for k:

```
pair<key_type, T>(k, T())
```

In case T is a built-in type T() is defined as 0 converted to type T.

2. Consider following code that implements the STL `queue` adaptor:

```
template <class T, class Sequence = deque<T> >
class queue {
  friend bool operator==(const queue&, const queue&);
  friend bool operator<(const queue&, const queue&);
public:
  typedef typename Sequence::value_type      value_type;
  typedef typename Sequence::size_type       size_type;
  typedef          Sequence                  container_type;
  typedef typename Sequence::reference       reference;
  typedef typename Sequence::const_reference const_reference;
protected:
  Sequence c;
public:
  queue() : c() {}
  explicit queue(const Sequence& c0) : c(c0) {}

  bool empty() const { return c.empty(); }
  size_type size() const { return c.size(); }
  reference front() { return c.front(); }
  const_reference front() const { return c.front(); }
  reference back() { return c.back(); }
  const_reference back() const { return c.back(); }
  void push(const value_type& x) { c.push_back(x); }
  void pop() { c.pop_front(); }
};
template <class T, class Sequence>
bool
operator==(const queue<T, Sequence>& x, const queue<T, Sequence>& y)
{
  return x.c == y.c;
}
template <class T, class Sequence>
bool
operator<(const queue<T, Sequence>& x, const queue<T, Sequence>& y)
{
  return x.c < y.c;
}
```

Explain why there are two versions of the `front` member function. In particular, write declarations and a call of `front` for which the compiler would select the first version, and explain why, and then do the same for the second version.

"stacktest.cpp" 3 ≡

```
#include <iostream>
#include <queue>
using namespace std;
int foo(const queue<int>& q)
{
   return q.front();  // calls const front member
}
int main()
{
  queue<int> q;
  q.push(2);
  int j = q.front();   // calls non-const front member
  q.front() = 4;       // calls non-const front member
  int k = foo(q);    // foo calls const front member
  cout << j << ", " << k << endl; // outputs 2, 4
}
```

In the calls of front in main, the first front member is called and it returns a non-const reference since q is not declared as a constant. (Therefore this reference can be used on the left hand side of an assignment statement, as in the second call.) It would be rather pointless to declare a queue as a constant in the main program—you couldn't push anything on it if you did—but when an object is passed to a function through a const reference parameter, it is considered to be a constant inside the function. Thus in the front call in foo, the const front member is called. (Therefore even if foo returned a reference, it would have to be a const reference—i.e., the return type would have to be const int&—and foo(q) could not be used on the left hand side of an assignment.)

3. This question is also about the `queue` adaptor code given in the previous question. In many textbook versions of a queue data type, the `pop` operation returns the front element of the queue. However, in the STL `queue`, `pop` returns void. Explain why this design decision makes sense (i.e., why `value_type` or `value_type&` isn't returned).

   Returning value_type would require a copy, which could be expensive if value_type objects are large. Returning value_type& would mean the caller would receive a "dangling reference," that is, it would point to deallocated storage. The front function returns a reference but it can remain valid for a while—until the element is popped off with pop. If the queue client needs to hold onto the value after the pop, it can copy it, but it pays the price of the copy only in that case.

4. A "concept," as the term has been used in this course, is a collection of abstrac-

tions that all obey a given set of requirements. By "abstraction" we usually mean a(n abstract data) type or an algorithm abstraction. In discussing container concepts and ordering and equivalence concepts we have used graphical depictions of the concept hierarchies, with the most generic concepts (the ones containing the most abstractions) at the top of the diagram. Edges in these graphs represent concept refinement, and more refined concepts (those having more requirements and containing fewer abstractions) appear lower than the concepts they are refined from. Also represented in the diagrams are some of the models of the concept, which are types that have all of the properties specified by the concept; these are distinguished from concepts by putting their names in italics. Draw such a diagram for all of the iterator concepts that are used in STL, including for each concept at least one STL model of it (label it with the name of a type defined by an STL component and use underlining instead of italics). (Use a separate sheet of paper.)

5. What three special operations must a Sequence have in order to be a Back Insertion Sequence?

   push_back, **pop_back**, and **back**.

   All three operations have the same complexity guarantee—what is it?

   Amortized constant time

6. The elements in a(n) _____**Unique Sorted Associative**_____ Container are always arranged in strictly ascending order by key, based on the < ordering relation on the key type or on a ordering relation on keys as defined by a function object used in constructing the container.

   Two models of this concept in STL are _____**set**_____and _____**map**_____.

7. Why would you want to write an algorithm to use other than random access iterators?

   To make the algorithm usable with data structures, like lists or streams, that don't provide random access iterators.

8. BGL defines several new iterators (not defined in STL) for traversing different graphs or parts of graphs Name three such iterators.

   vertex_iterator, **edge_iterator**, **adjacency_iterator**, **out_edge_iterator**, **in_edge_iterator**.

9. Consider the `binary_search` function below.

```
template<class Ran, class X>
bool binary_search(Ran begin, Ran end, const X& x)
{
  while (begin < end) {
    // find the midpoint of the range
    Ran mid = begin + (end - begin) / 2;

    // see which part of the range contains 'x';
    // keep looking only in that part
    if (x < *mid)
      end = mid;
    else if (*mid < x)
      begin = mid + 1;
    // if we got here, then '*mid == x' so we're done
    else return true;
  }
  return false;
}
```

Why didn't we write `(begin + end)/2` instead of the more complicated `begin + (end - begin)/2`?

Addition of two iterators is not defined (for good reason: it makes no semantic sense). (Subtraction of two iterators is defined and produces an integral value, and addition of an iterator and an integer is defined, so the second expression is defined.)

10. In Chapter 12 of *Accelerated C++*, the authors define a class named `Str`, a simplified version of the standard `string` class.

```
class Str {
// ...
public:
// ...
private:
Vec<char> data;
};
```

Give `Str` an operation that will let us implicitly use a `Str` object as a condition. The test should fail if the `Str` is empty, and should succeed otherwise. You may use any operations of the `Vec` template class defined in Chapter 11 of *Accelerated C++* (or any operations of the standard `vector` template class, since `Vec` is a simplified version of it).

Insert following line in the public members of Str:

```
  operator bool() { return !data.empty(); }
```

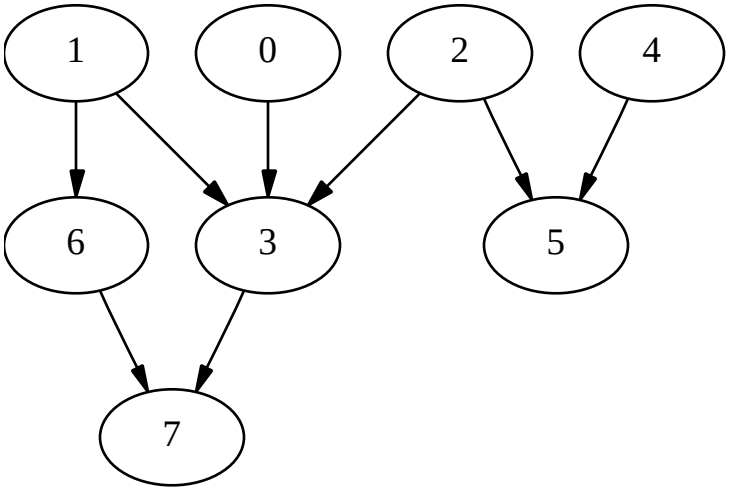
11. Consider the following complete program:

```
// Depth First Visitor Example
#include <iostream>
#include <fstream>
#include <boost/graph/depth_first_search.hpp>
#include <boost/graph/adjacency_list.hpp>
using namespace boost;

class mystery_visitor
  : public dfs_visitor<> {
public:
  mystery_visitor() { }
  template <typename Vertex, typename Graph>
  void discover_vertex(Vertex u, const Graph&) const {
    std::cout << "(" << u << " ";
  }
  template <typename Vertex, typename Graph>
  void finish_vertex(Vertex u, const Graph&) const {
    std::cout << u << ") ";
  }
};
int main()
{
  adjacency_list<listS, vecS, directedS> g;
  add_edge(0, 3, g);
  add_edge(1, 3, g);
  add_edge(2, 3, g);
  add_edge(3, 7, g);
  add_edge(4, 5, g);
  add_edge(2, 5, g);
  add_edge(1, 6, g);
  add_edge(6, 7, g);

  mystery_visitor vis;
  depth_first_search(g, visitor(vis));
  std::cout << std::endl;
  return 0;
}
```

Draw a picture of the graph g constructed by this program.



What would this program output on `std::cout`?

(0 (3 (7 7) 3) 0) (1 (6 6) 1) (2 (5 5) 2) (4 4)

12. Describe the difference between an algorithm and an acceptance testing function for that algorithm, in terms of their inputs and outputs.

    If the algorithm A has input x and output y, the corresponding acceptance testing function F has inputs x and y and outputs a boolean value: true if y is an acceptable output when A is applied to x, false otherwise.

13. Consider the following code:

```
template <typename VertexListGraph, typename OutputIterator>
void toposort(VertexListGraph& g, OutputIterator result)
{
  typedef typename graph_traits<VertexListGraph>::vertex_descriptor
          vertex_t;
  typedef typename graph_traits<VertexListGraph>::degree_size_type
          degree_size_t;
  vector<degree_size_t> indegree;
  // ...
}
```

    The problem with this code is that it is not as generic as it could be; the way it is written limits its use to an unnecessarily small subset of the possible graph representations.

First, write a declaration of a graph `G1` with an `adjacency_list` representation such that `toposort` could be called with `G1` as its first argument.

```
adjacency_list<listS, vecS, directedS> G1
```

The essential part of this declaration is `vecS` specifying the vertex list type as having a vector representation. This results in the `vertex_descriptor` type being an integral type, and therefore usable as the domain of a vector.

Next, write a declaration of a graph `G2` with an `adjacency_list` representation such that `toposort` could *not* be called with `G2` as its first argument (because it wouldn't compile).

```
adjacency_list<listS, listS, directedS> G1
```

In this case `listS` specifies the vertex list type has a list representation (as would `setS` or `hash_setS`). This results in the `vertex_descriptor` type being an nonintegral type (`void*` to be precise), and therefore it is not usable as the domain of a vector.

Finally, write a new declaration of `indegree` in `toposort` that would allow `toposort` to be applied to graphs with any representation belonging to the VertexListGraph concept.

```
map<vertex_t, degree_size_t> indegree;
```

Also acceptable:

```
hash_map<vertex_t, degree_size_t> indegree;
```

although this doesn't work without some tweaking (see the posted solution to Homework 2).