Code Number:____________________

Advanced Programming Exam

October 23, 2001

Write your answers on the exam pages. Draw a **Code Number** from the **brown** envelope (circulating) and enter it in the space provided above. Do **not** write your name on any of the exam pages. (I plan to grade all of the exams before matching up code numbers to names.)

This exam consists of eleven questions, worth a total of 100 points. No notes, printed material, or computers may be used. The exam must be completed during the regular class period, 10:00-11:20 am (80 minutes). I suggest that you read over the entire exam before beginning to write your answers.

1. Suppose that `m` has type `map<int, string>`, and we encounter a call to `copy(m.begin(), m.end(), back_inserter(x))`.
   What can we say about the type of `x`? **5 pts**

   What if the call were `copy(x.begin(), x.end(), back_inserter(m))` instead? **5 pts**

2. Suppose `x` is an object of some type `S` that is a model of the Sequence concept. Suppose also the value type of `S` is `int`. Write one or two lines of C++ code that remove all of the elements in `x` equal to 13, using the generic function `remove` and the `erase` member function of `S`. (If you need to declare an iterator, try to do so in a way that makes the code as generic as possible.) **10 pts**

3. Consider following code that implements the STL queue adaptor (same code as appeared in the Practice Exam):

```
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;
}
```

The == and < operators are defined outside of queue's class definition, rather than as member functions. However, they have been granted friendship privileges by the queue class. Why are the friendship privileges necessary? **6 pts**

---

4. This question is also about the `queue` adaptor code given in the previous question. For each of the following instantiations, indicate whether it would be legal ("Yes") or not ("No"). For those instantiations for which the answer is "No," explain why it is not legal. **12 pts**

   - `queue< int, list<int> >`
   - `queue< double, vector<double> >`
   - `queue< vector<int> >`
   - `queue< int, set<int> >`
   - `queue< int, deque<double> >`

5. Forward Iterators are a refinement of Input Iterators and Output Iterators: they support the Input Iterator and Output Iterator operations and also provide additional functionality. In particular, **8 pts**

6. Why would you want to define a data structure that doesn't support random access iterators (i.e., that only provides, say, forward iterators or bidirectional iterators)? Give an example. **8 pts**

7. Why doesn't STL define a template class `reverse_forward_iterator`? **8 pts**

8. In the `Str` class in Chapter 12 of *Accelerated C++*, one of the member functions defined is the constructor shown here:

```
class Str {
public:
  // ...
  template <class In>
  Str(In b, In e) {
    std::copy(b, e, std::back_inserter(data));
  }
private:
  Vec<char> data;
};
```

This constructor is an example of the C++ feature of class definitions known as **2 pts**

In their definition of the `Vec` class in Chapter 11 of *Accelerated C++*, the authors did not have a constructor that was this kind of member function. Add such a constructor to `Vec`. Recall that the private data members of `Vec` are

```
iterator data;
iterator avail;
iterator limit;
```

5 pts

Now rewrite the `Str` constructor given at the beginning of this question to use the new `Vec` constructor instead of calling `copy`. **5 pts**

9. A common way a class definition can define *associated types* is by means of `typedef`s nested inside the class definition, e.g.,

```
class my_array {
public:
 typedef double value_type;  // the type for elements in the array
 double& operator[](int i) { return m_data[i]; }
private:
 double* m_data;
};
```

If a function receives an object of type `my_array` through a parameter of that type, it can obviously access the object's value type with an expression such as `my_array::value_type`. However, if it receives an array type object through a template parameter `Array`, one that might be an object of either `my_array` type or of some other class-defined array type or of a builtin array type like `double*` or `int*`, it would not necessarily be legal to access the value type by writing `Array::value_type`. The solution to this problem is a *traits class*, which is a class template whose sole purpose is to provide a mapping from a type to other types, functions, or constants. The mapping is accomplished by creating different versions of the traits class to handle specific type parameters.

The default (fully templated) case will assume the array is a class with a nested typedef such as `my_array`:

```
template <typename Array>
struct array_traits {
 typedef typename Array::value_type value_type;
};
```

We can then create a specialization of the `array_traits` template to handle the case when the `Array` template argument is a built-in type like `double*`. First, create a *full specialization* that handles just the case `double*`. **5 pts**

Next, create a *partial specialization* that handles the general case `T*` where `T` is any type. (With such a partial specialization defined, the preceding full specialization would be unnecessary.) **5 pts**

10. For Homework 2, Joe McGraph proposed to implement a topological sort acceptance testing function using the following method:

```
To test whether a linear sequence S is a topological ordering
of graph G:
   reverse the sequence S
   while S is not empty {
     remove a vertex x from the front of S
     if there are any edges in G with x as source
       return false
     else {
       clear all edges in G that have x as target
       remove vertex x from G
     }
   }
   return true
```

It appears it would fairly easy to implement this method using BGL, since BGL provides a `clear_vertex` function that removes all edges (in-edges or out-edges) from a given vertex, and a `remove_vertex` function that removes an already cleared vertex from the graph. However, the resulting function would not meet all of the requirements stated in Homework 2 for the acceptance testing function. What is the problem with it? (Hint: keep in mind that in-edges are not represented in BGL's representation of a directed graph.) **8 pts**

11. Consider the following complete program:

```
// Another 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 cycle_detector
  : public dfs_visitor<> {
public:
  cycle_detector(bool& cycle) : has_cycle(cycle) { }
  template <typename Edge, typename Graph>
  void back_edge(Edge e, const Graph& G) const {
    has_cycle = true;
  }
private:
  bool& has_cycle;
};

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);
  add_edge(7, 2, g);
  bool b = false;
  cycle_detector vis(b);
  depth_first_search(g, visitor(vis));
  if (b)
    std::cout << "Cycle detected." << std::endl;
  else
    std::cout << "No cycle was detected." << std::endl;
  return 0;
}
```

This code shows how BGL's `depth_first_search` algorithm can be parameterized with a visitor that detects cycles, simply by recording that a back edge has been traversed. However, the program gives no indication at all about where in the graph the cycle occurs. Add code to the visitor so that it writes, on `std::cout`, the back edge(s) detected: write the new code below and draw an arrow on the preceding page to show where it should be inserted in the visitor class. (Note: Your code does *not* have to write out all of the edges in a detected cycle.) **8 pts**