Advanced Programming Exam — Some Answers

October 26, 2001

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**

   It must be a type that supports push_back and whose value type is `pair<int, string>` (thus any type `b<pair<int, string> >` where `b` is a model of the Back Insertion Sequence concept). (It must also be non-const.)

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

   No matter what type `x` is, this would be illegal, since map doesn't support push_back.

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**

   ```
   S::iterator new_end = remove(x.begin(), x.end(), 13);
   x.erase(new_end, x.end());
   ```

   Or, in one line,

   ```
   x.erase(remove(x.begin(), x.end(), 13), x.end());
   ```

   Besides being more compact, this conveniently avoids having to declare an iterator. (But it is a little harder to read.)

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**

Since the data member c of queue is protected, nonmember functions don't have access to it unless they are named as friends of the class.

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**

   2 pts for each correct "Yes/No" answer; 2 points for each correct explanation. Yes, I know that adds up to 16 pts.

   - `queue< int, list<int> >` **Yes.**
   - `queue< double, vector<double> >`

     No, because vector doesn't support pop_front, which is needed by queue's pop member.
   - `queue< vector<int> >` **Yes.**
   - `queue< int, set<int> >`

     No, because vector doesn't support back, push_back, front, or pop_front.
   - `queue< int, deque<double> >`

     Yes and no. Conceptually, the second template parameter of queue should be instantiated with a type whose value type is the same as the type instantiating the first template parameter. And in fact if you are compiling with a version of STL that does concept checking (e.g., GNU C++ 3.01), it will be flagged as a concept error. But otherwise, it is legal and the compiler lets it go by, since T is never actually used anywhere in class. (This is a weakness of the C++ type checking rules.) Thus, full credit for a Yes answer (I should have asked for explanations of these too!), and full credit for a No answer that points out that T is never used anywhere.

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**

   they are multi-pass, which means that they allow saving an iterator in a variable and using that variable later to return to a previously traversed element of the sequence.

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**

   Because this may allow other operations on the data structure to be more efficient. E.g., the list container supports constant time insertion and deletion, which could not be done if its iterators had to be random access. Partial credit (6 pts): "Because random access iterators would be inefficient for some structures, such as linked lists."

Why is this not as good an answer? Because random access iterators are *required* to be efficient (constant time) as part of their definitions.

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

   In order to implement it efficiently you would need `operator--` to be a constant time operation, but that would require the base iterators to be bidirectional, not just forward.

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**

template member functions (or any permutation of those words)

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;
```

Answer: **5 pts**

The simplest way to do this is to take advantage of the push_back operation, just as was being done in the Str constructor:

```
template <class In>
Vec(In b, In e) {
  create();  // this does data = avail = limit = 0;
  std::copy(b, e, std::back_inserter(*this));
}
```

This gets full credit (either with the create call or the assignments in the comment), because it works and only requires the iterators to be input iterators, which corresponds to the requirement for the corresponding Str constructor. The same is true if instead of the copy you called push_back repeatedly in a for loop:

```
template <class In>
Vec(In b, In e) {
  create();  // this does data = avail = limit = 0;
  for (In i = b; i != e; ++i)
   (*this).push_back(*i);
}
```

Here is another solution that is more efficient because it pre-allocates the storage rather than letting push_back do it incrementally.

```
template <class Ran>
Vec(Ran b, Ran e) {
  create(e - b, value_type());
  std::copy(b, e, data);
}
```

But the problem with this is that it requires the iterator type to be random access, because of the use of subtraction (`e - b`). Credit: 4 pts.

Similarly, solutions that do the storage allocation directly with new (ignoring the issues of allocation of uninitialized storage discussed in the textbook), e.g.,

```
template <class Ran>
Vec(Ran b, Ran e) {
  data = new value_type[e - b];
  limit = avail = data + (e - b);
  std::copy(b, e, data);
}
```

also need to compute the size of the array and therefore need random access iterators. Credit: 4 pts. [1]

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

Answer:

```
template <class In>
Str(In b, In e) : data(b, e) { }
```

Either of the following gets partial credit (3 pts) although neither really works:

[1] The corresponding constructor for the actual vector class is implemented with compile-time dispatching on the iterator category, so that the more efficient preallocation of storage can be used if the instantiated iterator type is random access (the size is computed as `e - b`) or bidirectional or forward (the distance from `b` to `e` is computed by `distance(b, e)`, which does it by stepping with `++`); otherwise, for input iterators the push_back solution is used.

```
template <class In>
Str(In b, In e) { data(b, e); }

template <class In>
Str(In b, In e) { Vec<char> data(b, e); }
```

If the constructor call is done in the function body rather than in the initialization list, it must be done as follows:

```
template <class In>
Str(In b, In e) { data = Vec<char>(b, e); }
```

This gets full credit (or 4 pts if the `<char>` is missing).

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**

Answer:

```
template <>
struct array_traits<double*> {
  typedef double value_type;
}
```

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**

Answer:

```
template <typename T>
struct array_traits<T*> {
  typedef T value_type;
}
```

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**

Since in_edges are not represented, clear_vertex cannot be implemented in constant time: the only way to find edges that have the given vertex as target is to iterate over all edges (thus $O(|E|)$) or iterate over all vertices and their out edges (thus $O(|V| + |E|)$). Hence the time for the above algorithm would be either $O(|V||E|)$ or $O(|V|^2 + |V||E|)$, instead of $O(|V| + |E|)$ as required. Also the function never checks to see if the set of vertices in G is the same as the set of vertices in S. (Mention of the latter is scored as a bonus of 4 points.)

11. Consider the following complete program:

```
// Another Depth First Visitor Example
... see the exam
```

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**

Answer:

```
template <typename Edge, typename Graph>
void back_edge(Edge e, const Graph& G) const {
  std::cout << "Cycle detected: Edge (" << source(e, G) << ", "
            << target(e, G) << ") is a back-edge." << std::endl;
  has_cycle = true;
}
```