C++ Primer Plus, 5^{th} *Edition* by Stephen Prata
Chapter 16: The `string` Class and the Standard Template Library
Review Questions

1. Consider the following class declaration:

```
class RQ1
{
private:
    char * st;     // points to C-style string
public:
    RQ1() { st = new char [1]; strcpy(st,""); }
    RQ1(const char * s)
    {st = new char [strlen(s) + 1]; strcpy(st,s); }
    RQ1(const RQ1 & rq)
    {st = new char [strlen(rq.st) + 1]; strcpy(st,rq.st); }
    ~RQ1() {delete [] st};
    RQ & operator=(const RQ & rq);
    // more stuff
};
```

   Convert this to a declaration that uses a `string` object instead. What methods no longer need explicit definitions?

   This is our class rewritten to use a `string` *object.*

```
class RQ1
{
private:
    string st;
public:
    RQ1() { st = ""; }
    RQ1(const char * s)
    {st = string(s); }
    RQ1(const RQ1 & rq)
    {st = rq.st; }
    ~RQ1() {}
    RQ & operator=(const RQ & rq)
    {st = rq.st; return *this;}
    // more stuff
};
```

   Barring the inconsistency regarding accessing the private member of another `RQ1` *object directly, we have the following results:*

   (1) *We do not need an explicit copy constructor.*
   (2) *We do not need an explicit destructor*
   (3) *We do not need an explicit overloaded assignment operator*

   However, since we explicitly define a constructor, we are unable to use the default copy constructor, so to enable it we must define it explicitly.

2. Name at least two advantages `string` objects have over C-style strings in terms of ease-of-use.
   (1) *We can change the size of a* `string` *object.*
   (2) *We can use the assigment operator to assign string values.*
   (3) *We can use the addition operator for easy concatenation of two strings.*
   (4) *We can use relational operators (*`==, <, <=, >, >=`*) on string objects for easy comparisons.*
   (5) *The* `string` *class has member functions, such as* `size()` *which allows us to work with* `string` *objects in an intuitive way.*

3. Write a function that takes a reference to a `string` object as an argument and that converts the `string` object to all uppercase.

```
string & ToUpper(string & s)
{
   transform(s.begin(), s.end(), s.begin(), toupper);
   return s;
}
```

4. Which of the following are not examples of correct usage (conceptually or syntactically) of `auto_ptr`? (Assume that the needed header files have been included.)

```
auto_ptr<int> pia(new int[20]);
auto_ptr<string> (new string);
int rigue = 7;
auto_ptr<int> pr(&rigue);
auto_ptr dbl (new double);
```

   - *The first line of code is an incorrect usage because the destructor for* `auto_ptr<>` *uses the* `delete` *command in its destructor, not* `delete []`.
   - *The fourth line of code is incorrect because when the destructor to* `pr` *is invoked, it would apply the* `delete` *operator to non-heap memory.*
   - *The fifth line of code is incorrect because we must declare a specialization of* `auto_ptr` *since it is a template class.*

5. If you could make the mechanical equivalent of a stack that held golf clubs instead of numbers, why would it (conceptually) be a bad golf bag?

   It would be a bad golf bag because you would be unable to access any given golf club easily. For example, if you have nine golf clubs and needed to get the #3 club, firstly you would not know where in the stack the club was unless you remember the order you placed all of the clubs in the bag. Secondly, to access the #3 club, you would need to remove each club placed onto the stack after the #3 club to get to it.

6. Why would a `set` container be a poor choice for storing a hole-by-hole record of your golf scores?

   The best reason for why this would be a poor choice is that a `set` *object does not hold duplicate items. If you were to get the same score on two holes, the set would count them as equivalent (if the only thing stored is the score). Additionally, a* `set` *object automatically orders its elements by using the* `<` *operator. If the only thing stored was the numeric score per hole, you would be unable to tell which score corresponded to which hole.*

7. Because a pointer is an iterator, why didn't the STL designers simply use pointers instead of iterators?

   Pointers are not appropriate for all situations. In the case of an array, a pointer will work fine as it can be dereferenced, incremented, and random access is permissible. However, for some data structures, such as a linked list, a binary tree, and a heap, pointers themselves will not work the same way when incremented and may not allow random access. However, an iterator can be made so that operators such as incrementation work in the same way for a linked list as it does for an array. In short, pointers are only appropriate for a subset of data structures.

8. Why didn't the STL designers simply define a base iterator class, use inheritance to derive classes for the other iterator types, and express the algorithms in terms of those iterator classes?

   I suppose that to do so would be possible. However, because of the multitude of data structures, each of which requiring unique implementations of the iterator concept, the iterator member functions would have to be redefined for each iterator. In this case, inheritance would be in name only which defeats the purpose of inheritance. With this in mind, using inheritance seems to complicate the implementation of an iterator for a given class without any sort of benefit.

9. Give at least three examples of convenience advantages that a `vector` object has over an ordinary array.

   (1) *We can use STL functions with the* `vector<>` *class*
   (2) *We can resize a* `vector<>` *object easily*
   (3) *We can use iterators to initialize a* `vector<>` *object*
   (4) *The* `vector<>` *class has member functions for performing many operations.*

10. If Listing 16.7 were implemented with `list` instead of `vector`, what parts of the program would become invalid? Could the invalid part be fixed easily? If so, how?

    (1) *The STL sort requires a random access iterator, the* `list<>` *class only has bidirectional iterators, which do not allow for random access. However, this problem can be fixed by using the* `sort()` *member function of the* `list<>` *class.*
    (2) *The* `random_shuffle()` *function requires a random access iterator. To remedy this, the client program must provide a function that shuffles the values of a linked list. However, this is not necessarily an easy fix.*