Problem Set 3

Due May 21, 11:59PM

This problem set should give you a chance to test out the language features we've been exploring this week. I strongly suggest reading over the handouts before starting this problem set, since there are a few C++ subtleties that we may not have had a chance to cover in lecture. As usual, email your solution (preferably as plain-text or as a PDF) to htiek@cs.stanford.edu. Please include your name and SUNetID at the top of all of your files so that the graders have an easier time organizing your files.

Problem 0

The first two questions concern the following (slightly modified) version of the CS106 `Map` class:

```
template <typename ValueType> class Map
{
public:
    Map(int sizeHint = 101);
    ~Map();

    int size();
    bool isEmpty();

    void put(string key, ValueType value);
    void remove(string key);
    bool containsKey(string key);

    /* get causes an Error if the key does not exist.  operator[] (the
     * function which is called when you use the map["key"] syntax) creates
     * an element with the specified key if the key does not already exist.
     */
    ValueType get(string key);
    ValueType& operator[](string key);

    void clear();

    void mapAll(void (fn)(string key, ValueType val));
    template <typename ClientDataType>
       void mapAll(void (fn)(string key, ValueType val, ClientDataType& data),
                   ClientDataType &data);

    Iterator iterator();

private:
    /* ... Implementation specific ... */
};
```

As mentioned in lecture, the CS106 ADTs are not `const`-correct, and consequently do not use `const` in places where it would likely be appropriate. Modify the `Map` interface above such that the `Map` is `const`-correct. In particular, member functions that do not modify the state of the object should be marked `const`, and parameters to functions should be passed by reference-to-`const` where appropriate.

Problem 1

The `Map` interface, as provided, has a problem with its constructor. What is it? Write a short program that illustrates this problem, then suggest a change to the `Map` interface to resolve this problem.

Problem 2

NASA is currently working on Project Constellation, which aims to resume the lunar landings and ultimately to land astronauts on Mars. The spacecraft under development consists of two parts – an orbital module called Orion and a landing vehicle called Altair. During a lunar mission, the Orion vehicle will orbit the Moon while the Altair vehicle descends to the surface. The Orion vehicle is designed such that it does not necessarily have to have an Altair landing module and consequently can be used for low-Earth orbital missions in addition to lunar journeys.

You have been hired to develop the systems software for the spacecraft. Because software correctness and safety are critically important, you want to design the system such that the compiler will alert you to as many potential software problems as possible.

Suppose that we have two classes, one called `OrionModule` and one called `AltairModule`. Since every Altair landing vehicle is associated with a single `OrionModule`, you want to define the `AltairModule` class such that it stores a pointer to its `OrionModule`. The `AltairModule` class should be allowed to modify the `OrionModule` it points to (since it needs to be able to dock/undock and possibly to borrow CPU power for critical landing maneuvers), but it should under no circumstance be allowed to change which `OrionModule` it's associated with.

Here is a skeleton implementation of the `AltairModule` class:

```
class AltairModule
{
public:
    /* Constructor accepts an OrionModule representing the Orion spacecraft
     * this Altair is associated with, then sets up parentModule to point to
     * that OrionModule.
     */
    explicit AltairModule(OrionModule* owner);

    /* ... */
private:
    OrionModule* parentModule;
};
```

Given the above description about what the `AltairModule` should be able to do with its owner `OrionModule`, appropriately insert `const` into the definition of the `parentModule` member variable. Then, implement the constructor `AltairModule` such that the `parentModule` variable is initialized to point to the `owner` parameter.

Problem 3

It is illegal to have a copy constructor which accepts its argument by value. Why is this? However, it is *not* illegal to have an assignment operator which accepts its argument by value. Why isn't this a problem?

Problem 4

The next few problems deal with a another approach for writing assignment operators using a technique known as *copy-and-swap*.

Suppose that we have a `Vector<T>` class that has the following data members:

```
T* elems;             // Actual array of elements; allocated with new[].
int logicalLength;    // Logical size of the Vector
int allocatedLength; // Size of the array
```

You can assume that `Vector` has a correctly-defined copy constructor and destructor.

Using the `swap` function exported by `<algorithm>`, write a member function `Vector::swap` which accepts as input a reference to another `Vector`, then exchanges the contents of the source `Vector` and the `Vector` parameter. For example, if we had a `Vector` called `one` containing {1, 2, 3, 4} and a `Vector` `two` containing {5, 6, 7, 8}, then after a call to `one.swap(two)`, `one` would contain {5, 6, 7, 8} and `two` would contain {1, 2, 3, 4}. Your function should not make any deep-copies of the data stored in the `Vector`; instead, just use the global function `swap` to exchange the values of every data member with its corresponding member in the other `Vector`.

Problem 5

Consider the following implementation of a `Vector` assignment operator:

```
template <typename T>
Vector<T>& Vector<T>::operator= (const Vector& other)
{
    Vector<T> temp(other);

    /* Uses Vector<T>::swap rather than the global swap function to make
     * the swap, so no data is copied during this step.
     */
    swap(temp);
    return *this;
}
```

Explain why this implementation of `operator =` correctly sets the receiver object to be a deep-copy of the `other` object. What `Vector` member function actually performs the deep copy? What `Vector` member function actually does the cleanup?

Problem 6

When writing an assignment operator using the pattern covered in Handout #23, we had to explicitly check for self-assignment in the body of the assignment operator. Explain why this is no longer necessary using the copy-and-swap approach, but why it still might be a good idea to insert the self-assignment check anyway.

Problem 7

How long did this problem set take you? How hard was it? Did the questions help you get a better understanding of the material? Any suggestions for future problem sets?