Problem Set 3 Solutions

Problem 0

There are several steps involved in this problem that end up with a completely `const`-correct public interface for the `Map` class, and we'll consider each of them individually.

The first set of changes we should make to the `Map` is to mark all of the public member functions which don't modify state `const`. This results in the following interface:

```
/* Note: Still more changes to make.  Do not use this code as a reference! */
template <typename ValueType> class Map
{
public:
    Map(int sizeHint = 101);
    ~Map();

    int size() const;
    bool isEmpty() const;

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

    /* 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) const;
    ValueType& operator[](string key);

    void clear();

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

    Iterator iterator() const;

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

The `size`, `isEmpty`, and `containsKey` functions are all `const` because they simply query object properties without changing the `Map`. `get` is also `const` since accessing a key/value pair in the `Map` does not actually modify the underlying state, but `operator[]` should definitely *not* be marked `const` because it may update the container if the specified key does not exist.

The trickier functions to `const`-correct are `mapAll` and `iterator`. Unlike the STL iterators, CS106 iterators are read-only and can't modify the underlying container. Handing back an iterator to the `Map` contents therefore cannot change the `Map`'s contents, so we have marked `iterator const`. In addition, since `mapAll` passes its arguments to the callback function by value, there is no way for the callback function to modify the underlying container. It should therefore be marked `const`.

Now that the interface has its member functions `const`-ified, we should make a second pass over the `Map` and replace all instances of pass-by-value with pass-by-reference-to-`const`. In general, objects should *never* be passed by value and should always be passed either by pointer or reference with the appropriate `const`ness. This eliminates unnecessary copying and can make programs perform asymptotically better. The resulting class looks like this:

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

    int size() const;
    bool isEmpty() const;

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

    /* 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(const string& key) const;
    ValueType& operator[](const string& key);

    void clear();

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

    Iterator iterator() const;

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

Note that we did *not* mark the `ClientDataType&` parameter to `mapAll const`, since the `Map` client may actually want to modify that parameter.

This version of the `Map` is `const`-correct, since non-mutating interface operations are marked `const` and parameters to member functions are passed by reference-to-`const` where appropriate.

Problem 1

Because the `Map` constructor accepts a single parameter, C++ treats it as a conversion constructor and allows for implicit conversions between `int`s and `Map`s. For example, the following code is legal:

```
Map<int> myMap;
myMap = 137; // Whoops!
```

To fix this, we simply need to mark the constructor `explicit`.

Problem 2

The `parentModule` field of the `AltairModule` class should be defined as `OrionModule * const parentModule`, that is, as a `const` pointer. This allows the `AltairModule` class to modify its parent `OrionModule` as necessary, but prevents the `AltairModule` class from changing which module it's associated with.

Because the `parentModule` field is now marked `const`, we must initialize it in the initializer list. For example:

```
AltairModule::AltairModule(OrionModule* parent) : parentModule(parent)
{
   /* ... empty constructor ... */
}
```

Problem 3

A copy constructor cannot accept its parameter by value because this would create a circular dependency. C++ initializes all parameters to functions that are passed-by-value by calling the copy constructor. However, if the copy constructor took its parameter by value, it would then be called to initialize its own parameter, which would then initialize its parameter by value, calling the copy constructor, initializing the parameter by value, etc.

The assignment operator *can* accept its parameter by value because there is no circular dependency – the parameter is initialized with the copy constructor as usual.

Problem 4

Writing a `swap` function for the `Vector` is straightforward because we have the global `swap` function to assist us. Given these data members:

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

The `swap` function could be written as follows:

```
template <typename T> void Vector<T>::swap(Vector& other)
{
    std::swap(elems, other.elems);
    std::swap(logicalLength, other.logicalLength);
    std::swap(allocatedLength, other.allocatedLength);
}
```

This simply exchanges all of the data members of the two classes. Note that due to a C++ technicality the calls to the global function `swap` must be fully-qualified as `std::swap`, since otherwise C++ will think that we're trying to make a recursive call to `Vector<T>::swap` with the wrong number of arguments.

Problem 5

The copy-and-swap assignment operator works in three steps:

1. Create a temporary copy of the passed-in parameter using the class's copy constructor.
2. Use `swap` to move all of the data members from the current class into the copy and vice-versa. At this point our class now is a deep-copy of the parameter and the temporary object is responsible for all of the data members we used to have.
3. Let the temporary copy of the parameter go out of scope, triggering its destructor. This causes the old data members of the receiver object to be cleaned up.

Interestingly, this approach uses the implementation of the copy constructor to perform the deep-copy and the implementation of the destructor to perform the cleanup.

Problem 6

The self-assignment check is no longer necessary because we don't clean out the contents of the `Vector` before copying over the data members. However, it would still be a reasonable idea to insert the test before performing the full copy since if we detect self-assignment, we can avoid making an expensive deep copy of the class data.