STAT 598W Homework 3

Due by March 11, 2016

In this homework, you are required to complete some part of section 1, all of section 2 and section 3. You are also required to complete at least one of section 4, 5 and 6. It is strongly recommended that you try to complete all sections, since they are all very important.

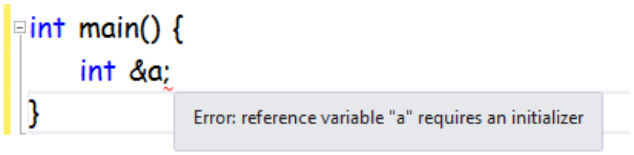
For section 2 to 6, use a header file ".h" to write the declaration of each class. And implement all member functions in a corresponding source file ".cpp". For example, you will have a "complex.h" and "complex.cpp" for declaration of the complex number class and implementation of member functions of complex number class.

Submit a pdf file for section one. For section 2 to 6, submit all your ".h" and ".cpp" files for all classes, and in addition, a ".cpp" file that includes your main function, which tests all your classes.

1 Basic Exercises

In this section, write codes to illustrate that each statement is true. You can either achieve this by showing the error messages from compiler, or print some messages to the screen.

For example, for the first statement of "Reference", it is enough to show the following error.



You do not need to spend too much time to get such a nice screen shoot to illustrate the result. The point is to have a try for each statement.

1. Reference.
   - A reference must be initialized when defined.
   - A reference can be used as return value, in which case we can do things like $f() + +$.
   - We can use const reference as input parameter for a function, in which situation we cannot modify the value of the original variable through this reference inside the function.
2. Private.
   - By default, a member of a class is private.
   - You can access private member of ANOTHER object of the SAME class.

- Two functions with same name, parameters and constancy, one being private and one being public, are not function overloading.

3. Constructor
   - If no constructor is defined, compiler will generate a default constructor, who has no parameter and does nothing.
   - If a constructor is defined, no default constructor will be generated.
   - After an object is defined, constructor cannot be called explicitly.

4. Copy constructor.
   - Copy constructor must have the form X(X& x) or X(const X& x), but cannot be X(X x).
   - If a copy constructor is defined, no default copy constructor is generated.
   - If a copy constructor is defined, no default constructor is generated.
   - If a constructor is defined but no copy constructor is defined, then no default constructor is generated but a default copy constructor is defined.
   - When pass an object to function by reference, the copy constructor is not called, but when pass by value, the copy constructor is called.
   - When return an object by reference, the copy constructor is not called, but when return by value, the copy constructor is called.
   - When define an object by "A a1 = a2", the copy constructor is called. After the object is defined, "a1 = a2" does not call copy constructor.

5. Static.
   - Static members must be initialized in THE SAME FILE that the class is defined.
   - Static members cannot be initialized when declared.
   - Static member functions cannot have access to non-static member variables and functions.
   - You can get access to static members by "ClassName::StaticMemberName".
   - sizeof() will not count the size of static members.

6. Order of constructor and destructor: Slides page 19, 24 and 32.

7. Const methods.
   - Inside const methods, cannot modify value of non-constant members of the class.
   - Inside const methods, cannot call non-constant member functions.
   - Inside const methods, can modify other non-constant variables outside the class.
   - Constant objects cannot call non-constant methods.
   - Two member functions, one const method and one not const method, is function overloading. In such case, const objects call const methods and non-constant objects call non-const methods.
   - Two functions that are only different from return types are not overloading.

8. Parameter list.

- Member objects, const members and references must be initialized in parameter list, or when they are declared.
- The order that constructor of each member is called is based on the order that each member is declared, but is irrelevant with the order that they appear in the parameter list.
- If a member does not appear in the parameter list, it is initialized with default constructor.
- If one initialize member objects, const members and references when they are declared as well as in the parameter list, then the version of parameter list is used.

2 Complex Number Class

Complete the following complex number class.

```
class complex {
        double re;
        double im;
public:
        complex();
        complex(double);
        complex(double, double);
        //Why don't we write a copy constructor and destructor?

        complex operator+(const complex &);
        //Also need to include -, *, /, =.

        friend ostream& operator<<(ostream&, const complex&);
        //Also need to include >> in istream.

        friend double re(const complex &);
        //Also need to implement im, obs, arg, conj, sqrt, pow as friend functions.
        //Need to use appropriate type of parameters and return value.
};
```

3 Vector Class

Complete the following vector class.

```
class myVector {
        double *p;
        int nElements;
public:
        myVector(int);
        //Constructor with input of nElements.
        myVector(myVector&);
        //Copy constructor that performs deep copy.
        ~myVector();
        //Destructor. You need to delete what you new in constructor.

        friend myVector operator+(const myVector&, const myVector&);
```

```
        friend myVector operator*(double, const myVector&);
        friend myVector operator*(const myVector&, double);
        friend double operator*(const myVector&, const myVector&);
        //vector addition, scalar multiplication and inner product.
        //Need to consider that you cannot add and inner product
        //two vectors with different length.

        myVector& operator= (const myVector&);
        //Overloading assignment operator.
        double& operator[] (int);
        //Get the ith element of the vector.
        //i should start from 0. Need to consider the case that i >= nElements.

        friend double norm(const myVector&);
        //Euclidean norm.

        friend ostream& operator<< (ostream&, const myVector&);
        friend istream& operator>> (istream&, const myVector&);
        //Overloading of "<<" and ">>".

        //Also need to overload ++/--, which plus/minus each elements by one.
        //Need to consider both prefix and postfix.
};
```

4 Stack

Complete the following stack class.

```
template<class T>
struct stack_node {
        T* next;
        T val;
        stack_node(T* _next) : next(_next) {}
};
template<class T>
class stack {
        typedef stack_node<T> node;
        typedef stack_node<T>* iterator;
        typedef stack_node<T>& reference;
        //Some type define.

        iterator first;
        iterator last;
        size_t length;
        //Three variables we keep.
public:
        stack();
        //Constructor with no parameter.
        stack(T);
        stack(node);
        //Constructor with one node as parameter.
        stack(stack &);
        //Copy constructor.
```

```
        ~stack();
        //Destructor.

        size_t size();
        bool empty();
        //Return the size of the list  and whether it is empty.

        T& top();
        void push(const T&);
        void pop();
        //Get the top element, push new element to the top
        //and remove the top element.
};
```

5 Queue

Complete the following queue class.

```
template<class T>
struct queue_node {
        T* next;
        T val;
        queue_node(T* _next) : next(_next) {}
};
template<class T>
class queue {
        typedef queue_node<T> node;
        typedef queue_node<T>* iterator;
        typedef queue_node<T>& reference;
        //Some type define.

        iterator first;
        iterator last;
        size_t length;
        //Three variables we keep.
public:
        queue();
        //Constructor with no parameter.
        queue(T);
        queue(node);
        //Constructor with one node as parameter.
        queue(stack &);
        //Copy constructor.
        ~queue();
        //Destructor.

        size_t size();
        bool empty();
        //Return the size of the list  and whether it is empty.

        T& front();
        T& back();
        //Get the next and last element.
```

```
        void push(const T&);
        void pop();
        //Push new element to the queue.
        //Remove the next element.
};
```

6 Single Linked List

Complete the following single linked list class.

```
template<class T>
struct linkedlist_node {
        T* next;
        T val;
        linkedlist_node(T* _next) : next(_next) {}
};
template<class T>
class linkedlist {
        typedef linkedlist_node<T> node;
        typedef linkedlist_node<T>* iterator;
        typedef linkedlist_node<T>& reference;
        //Some type define.

        iterator first;
        iterator last;
        size_t length;
        //Three variables we keep.
public:
        linkedlist();
        //Constructor with no parameter.
        linkedlist(T);
        linkedlist(node);
        //Constructor with one node as parameter.
        linkedlist(linkedlist &);
        //Copy constructor.
        ~linkedlist();
        //Destructor.
        linkedlist& operator=(const linkedlist&);
        //Overloading of assignment operator.

        iterator begin();
        iterator end();
        //Return the iterator (pointer) to the first and last element.

        size_t size();
        bool empty();
        //Return the size of the list  and whether it is empty.

        reference front();
        reference back();
        //Return the first and last elements.
```

```
        void push_back(const T&);
        void pop_back(const T&);
        void push_front(const T&);
        void pop_front(const T&);
        void insert(iterator, const T&);
        iterator erase(iterator);
        //Add or delete an element at the end or at the front.
        //Insert an element after the iterator.
        //Erase an element at the iterator.

        void remove(const T&);
        void sort();
        void reverse();
        //Remove certain element in the list.
        //Sort the list.
        //Reverse the list.
};
```