**CISS350: Data Structures and Advanced Algorithms**
**Assignment 1**

Name: ______________________________________

Q1. Complete the following class that implements a class similar to the C++ STL vector class. The purpose is to understand the inner workings for the iterator and constant iterator classes so that

1. You can implement iterator classes in your own container classes.
2. You can work quickly with C++ STL classes (since most uses of C++ STL classes involve the use of builtin iterator classes).

Therefore you need not complete all the methods for the vector class. Remember that skeleton code is incomplete and might have errors. The purpose of skeleton code is only to give you a rough overview of the intention of the code.

Refer to the relevant iterator sections in chapter on containers. You might also want to look for the `IntPointer` class in the CISS245 notes.

```
// File: vector.h

template < typename T >
class vector
{
public:
    class iterator;
    class const_iterator;

    class iterator
    {
    public:
        iterator(T * const p)
        : p_(p)
        {}

        const iterator & operator++()
        {}

        const iterator operator++(int)
        {}

        const T & operator*() const
        {}

        T & operator*()
```

```
        {}

        // This is used by const_iterator(const iterator &)
        T * p() const
        {
            return p_;
        }
    private:
        T * p_;
    };

    class const_iterator
    {
    public:
        const_iterator(T * p)
        {}

        const_iterator(const iterator & p)
            : p_(p.p())
        {}

        const_iterator & operator++()
        {}

        void operator++(int)
        {}

        const T & operator*() const
        {}

    private:
        T * p_;
    };

    vector(int size, int val)
        : capacity_(2 * size), size(size_), (p_(new T[size])
    {
        for (int i = 0; i < size; ++i)
        {
            p[i] = val;
        }
    }

    ~vector()
    {}

    // TODO: copy constructor
    // TODO: assignment operator

    iterator begin()
    {
```

```
        // TODO: return iterator object that points to p_[0];
    }

    const_iterator begin() const
    {
        // TODO: return const_iterator object that points to p_[0];
    }

    iterator end()
    {
        // TODO: return iterator object that points to p_[size_];
    }

    const_iterator end() const
    {
        // TODO: return const_iterator object that points to p_[size_];
    }

    const T & operator[](int i) const
    {
        return p_[i];
    }

    T & operator[](int i)
    {
        return p_[i];
    }

private:
    int capacity_;
    int size_;
    T * p_;
};
```

When completed, you should be able to do the following:

```
vector< int > v(10, 0); // v is 10 0s in array of capacity 20
typename vector< int >::iterator p = v.begin();
typename vector< int >::const_iterator q = v.begin();

std::cout << v[0] << '\n'; // prints 0
std::cout << (*p) << '\n'; // prints 0
std::cout << (*q) << '\n'; // prints 0

v[0] = 42;
std::cout << v[0] << '\n'; // prints 42
std::cout << (*p) << '\n'; // prints 42
std::cout << (*q) << '\n'; // prints 42

*p = -1;
```

```
std::cout << v[0] << '\n'; // prints -1
std::cout << (*p) << '\n'; // prints -1
std::cout << (*q) << '\n'; // prints -1

// *q = 0; // ERROR during compilation

v[1] = 1;
++p;
++q;
std::cout << v[1] << '\n'; // prints 1
std::cout << (*p) << '\n'; // prints 1
std::cout << (*q) << '\n'; // prints 1

v[10] = -9999; // Note: this is outside the valid index range of 0..9
p = v.end();
q = v.end();
std::cout << v[10] << '\n'; // prints -9999
std::cout << (*p) << '\n';  // prints -9999
std::cout << (*q) << '\n';  // prints -9999

v[9] = -9998;
--p;
--q;
std::cout << v[9] << '\n'; // prints -9998
std::cout << (*p) << '\n'; // prints -9998
std::cout << (*q) << '\n'; // prints -9998

*p = -9997;
std::cout << v[9] << '\n'; // prints -9997
std::cout << (*p) << '\n'; // prints -9997
std::cout << (*q) << '\n'; // prints -9997
```

Other features you can implement and test for your own understanding:

```
p = v.begin();
std::cout << (*(p++)) << '\n';
std::cout << (*(++p)) << '\n';
p += 3; // NOTE: operator+= is not available for all \cpp\ STL classes
*p = 42;
std::cout << v[3] << '\n';
std::cout << *p << '\n';
p -= 2;
*p = 43;
std::cout << v[1] << '\n';
std::cout << (*p) << '\n';
```

You should be able to include the following to your `main.cpp`:

```
template < typename T >
std::ostream & operator<<(std::ostream & cout, const vector< T > & v)
{
    std::string delim = "";
    cout << '{';
    for (typename vector< T >::const_iterator p = v.begin();
         p != v.end();
         ++p)
    {
        cout << delim << (*p);
        delim = ", ";
    }
    cout << '}';
    return cout;
}
```

Note that in the above I'm using the `const_iterator` because the `v` is a constant reference. You cannot use `iterator`. If your object `v` is from class `C< T >` (which is not necessary from the `vector` class) has a `const_iterator`, and methods `begin()` and `end()`, you can do this:

```
template < typename T >
std::ostream & operator<<(std::ostream & cout, const C< T > & v)
{
    std::string delim = "";
    cout << '{';
    for (typename C< T >::const_iterator p = v.begin();
         p != v.end();
         ++p)
    {
        cout << delim << (*p);
        delim = ", ";
    }
    cout << '}';
    return cout;
}
```

Your submission should include code for your vector class (`vector.h`) and a `main.cpp` that includes the above tests. (Remember: Template code should be complete in a header file. There should be no `vector.cpp`.)

For further practice on use of your own iterators, you can implement other vector methods/functions (example: the `std::vector` class has an `erase` method).

Note: If you want to write a complete vector class, you can google for g++ implementation details. There are some optimizations that I did not mention in the CISS245 assignment the integer dynamic array class assignment. Also, once you have done several iterator classes for different containers, you'll see that it's beneficial to have some iterator base classes.

Q2. You are given the following

```
// File: SLNode.h

#ifndef SLNODE_H
#define SLNODE_H

template < typename T >
class SLNode
{
public:
    SLNode(const T & key, SLNode * next=NULL)
        : key_(key), next_(next)
    {}
private:
    T key_;
    SLNode * next_;
};
#endif
```

```
// File: SLList.h

#ifndef SLLIST_H
#define SLLIST_H


class IndexError          // An IndexError object is thrown if
{};                       // an invalid index value is used in
                          // a method/operator in the SLList
                          // class.

class ValueError          // A ValueError object is thrown if
{};                       // an invalid value is used in a
                          // method/operator in the SLList class.
                          // If the value is an index value, then
                          // the IndexError class is used instead.

template < typename T >
class SLList
{
public:
    SLList()
        : phead_(NULL)
    {}
private:
    SLNode < T > * phead_;
};

#endif
```

Implement the above classes so that the following test code works. The behavior of the methods is clear from the description below.

```
// File: test.cpp

#include <iostream>
#include "SLNode.h"
#include "SLList.h"

int main()
{
    SLNode< int > node(5);
    std::cout << node << std::endl; // Prints the class of node, the
                                    // address of the node, the key_ and
                                    // the next_ pointer in this format:
                                    // <SLNode 0xbff9b4fc key:5, next:0>
                                    // where 0xbff9b4fc is the address of
                                    // this node, 5 is the value of key_,
                                    // 0 is the address in node.next_.
                                    // Different nodes will of course have
                                    // different values printed.
                                    // The format must however follow the
                                    // above.

    // The above format for printing is suitable for debugging your class.
    // Once your class is working you can turn off debug-printing and just
    // print the key_ in the node. This can be done as follows:
    SLNode< int >::debug_printing(false);
    std::cout << node << std::endl; // Prints 5 (and newline).
    SLNode< int >::debug_printing(true);// Turn on debug printing again.

    // The get_key(), set_key() and get_next(), set_next() of the node
    // does the obvious.
    std::cout << node.get_key() << std::endl;   // Prints 5
    std::cout << node.get_next() << std::endl;  // Prints 0
    node.set_key(6);                            // node.key_ is 6
    SLNode< int > node2(7);
    node.set_next(&node2);                      // node.next_ is &node2

    SLList< int > list;
    std::cout << list << std::endl;
    // Prints newline and then
    // <SLList 0x9788038 phead:0
    // >

    list.insert_head(5);
    std::cout << list << std::endl;
    // Prints newline and then
    // <SLList 0x9788038 phead:0x9788028
    //     <SLNode 0x9788028 key:5, next:0>
```

```
// >

list.insert_head(6);
std::cout << list << std::endl;
// Prints newline and then
// <SLList 0x9788038 phead:0x9788018            <-- CORRECTION
//     <SLNode 0x9788018 key:6, next:0x9788028> <-- CORRECTION
//     <SLNode 0x9788028 key:5, next:0>         <-- CORRECTION
// >

// The above format for printing is suitable for debugging.
// Once your SLList class is working, you can turn debug-printing
// off.
SLList< int >::debug_printing(false);
SLNode< int >::debug_printing(false);

list.insert_head(3);
std::cout << list << std::endl; // Prints [3, 6, 5]
list.insert_tail(3);
std::cout << list << std::endl; // Prints [3, 6, 5, 3]
list.insert_tail(2);
std::cout << list << std::endl; // Prints [3, 6, 5, 3, 2]

int x;

x = list.remove_head();
std::cout << list << std::endl; // Prints [6, 5, 3, 2]
std::cout << x << std::endl;    // Prints 3

x = list.remove_tail();
std::cout << list << std::endl; // Prints [6, 5, 3]
std::cout << x << std::endl;    // Prints 2

SLNode< int > * p = list.find(5);// p points to the first node
                                // with key_ = 5.
x = list.remove(p);
std::cout << list << std::endl; // Prints [6, 3]
std::cout << x << std::endl;    // Prints 5

try
{
    SLNode< int > * p;
    list.remove(p);             // A ValueError exception object is
                                // thrown if the pointer argument is
                                // not valid.
}
catch (ValueError & e)
{
    std::cout << "ValueError caught" << std::endl;
}
```

```
list.clear();                    // list becomes empty.
std::cout << list << std::endl; // Prints []

list.insert_tail(5);
list.insert_tail(6);
list.insert_tail(7);
list.insert_tail(8);
list.remove(list.find(5));
std::cout << list << std::endl; // Prints [6, 7, 8]
list.remove(list.find(8));
std::cout << list << std::endl; // Prints [6, 7]
list.clear();

list.insert_tail(5);
list.insert_tail(6);
list.insert_tail(7);
list.insert_tail(6);
list.remove(6);                  // Remove first node whose key_ is 6.
std::cout << list << std::endl; // Prints [5, 7, 6]
try
{
    list.remove(100);            // A ValueError exception object is
                                 // thrown if the key_ value to be
                                 // removed is not found.
}
catch (ValueError & e)
{
    std::cout << "ValueError caught" << std::endl;
}
list.clear();

list.insert_tail(5);
list.insert_tail(6);
list.insert_tail(7);
list.insert_tail(6);

SLList< int > newlist(list);
std::cout << newlist << std::endl; // Prints [5, 6, 7, 6]
list.insert_tail(8);
newlist = list;
std::cout << newlist << std::endl; // Prints [5, 6, 7, 6, 8]

list.clear();
list.insert_tail(5);
list.insert_tail(6);
list.insert_tail(7);
list.insert_tail(6);
std::cout << list[0] << std::endl; // Prints 5, i.e., list[0] returns
                                   // the key_ in the first node of
                                   // list. Note that this should be
                                   // returned as a reference so that
```

```
                                     // it's possible to do this:
                                     // list[0] = 42;
    std::cout << list[1] << std::endl; // Prints 6
    std::cout << list[2] << std::endl; // Prints 7
    std::cout << list[3] << std::endl; // Prints 6
    try
    {
        list[4];                       // This will cause an IndexError
                                       // object to be thrown since list
                                       // has only 4 nodes.
    }
    catch (IndexError & e)
    {
      std::cout << "IndexError caught" << std::endl;
    }
    try
    {
        list[-1];                      // This will also cause an IndexError
                                       // to be thrown since the index is < 0.
    }
    catch (IndexError & e)
    {
        std::cout << "IndexError caught" << std::endl;
    }
    list.clear();

    list.insert_tail(5);
    list.insert_tail(6);
    list.insert_tail(7);
    list.insert_tail(6);
    std::cout << list.size() << std::endl;     // Prints 4.
    std::cout << list.is_empty() << std::endl; // Prints 0

    list.clear();
    std::cout << list.is_empty() << std::endl; // Prints 1

    // You can also compare lists. In other words if list1 and list2
    // are two SLList objects, then list1 == list2 is true exactly
    // when the values in list1 and list2 are the same and appear in
    // order, starting from the head.
    // Of course operator!= is just the "opposite" of operator==.

    // If list is an SLList object and p points to a node in list,
    // then calling list.insert_before(p, 5) will insert 5 (as a node)
    // in front of the node that p points to.
    // list.insert_after(p, 5) will insert 5 (as a node) behind the
    // node that p points to.

    // If list is an SLList object, then list.front() returns a reference
    // to the key_ member of the head node of list. This is similar to
    // list[0].
```

```
    // If list is an SLList object, then list.back() returns a reference
    // to the key member of the tail node of list.

    return 0;
}
```

Note that a linked list is not meant to act completely like an array/vector. So a linked list class (singly-linked or doubly-linked) usually wouldn't have `operator[]`. This is included here only for practice. So for instance in the C++ STL library, `std::list`, you won't find `operator[]`.

Q3. The `DLNode` and `DLList` class is analogous to the classes in Q2 except that these are doubly-linked. Furthermore you must implement the `DLList` using sentinel nodes:

```
class DLList
{
private:
    DLNode head_sentinel_;
    DLNode tail_sentinel_;
};
```

Note that the `head_sentinel_.prev_` is `NULL` while `tail_sentinel_.next_` is `NULL`.

The corresponding node class of course looks like this:

```
template < typename T >
class DLNode
{
public:
    DLNode(const T & key, DLNode * prev=NULL, DLNode * next=NULL)
        : key_(key), next_(next), prev_(prev)
    {}

private:
    T key_;
    DLNode * prev_;
    DLNode * next_;
};
```

Implement the above classes with methods completely analogous to those in Q2.

Recall: Although a `DLList` object has the same behavior as a `SLList` object, recall that the runtime is very different. For instance inserting at the tail of a `SLList` has a runtime of $O(n)$ where $n$ is the size of the list whereas for a `DLList` object, the runtime is $O(1)$.

Q4. Implement a list-based stack. As mentioned in class, a singly linked list can be used where the top of the stack corresponds to the head (not the tail!) The following is what you can do:

```
Stack< int > stack;
stack.push(5);           // Top of stack is 5
stack.push(6);           // Top of stack is 6 (5 is below 6)
stack.push(4);           // Top of stack is 4 (6 is below 4)
stack.pop();             // Top of stack is 6 (5 is below 6)
int x = stack.top();     // x has value 6
stack.top() = 7;         // Top of stack is 7 (5 is below 7)
int size = stack.size(); // size has value 2
bool b = stack.is_empty();// b has value false
stack.clear();           // stack.size is 0
                         // Calling stack.pop() here will
                         // result in the throwing of an
                         // UnderflowError object
```

Here's the skeleton:

```
// Stack.h
class UnderflowError     // An UnderflowError object is thrown if
{};                      // you execute pop() on an empty stack.

template < typename T >
class Stack
{
public:

private:
    SLList< T > list_;
};
```

(I'm using composition instead of inheritance.)

Once the singly linked list is done, the stack class is extremely easy. So you must finish the singly linked list class first.

Q5. Implement a list-based double-ended queue, or deque. As mentioned in class, a doubly linked list should be used. Treat the front of a deque as the head of the underlying list and the back of the deque as the tail of the underlying list. The following is what you can do:

```
Deque< int > deque;
deque.push_front(5);
deque.push_back(6);
int x = deque.front();      // x = 5
deque.front() = 4;          // front of deque is now 4
int y = deque.back();       // y = 6
deque.back() = 7;           // back of deque is 7
deque.pop_front();          // deque has now 1 value, i.e., 7
deque.pop_back();           // deque is now empty
int size = deque.size();    // size is zero
bool b = deque.is_empty();  // b is true
deque.push_front(1);
deque.push_front(1);
deque.push_front(1);
deque.clear();              // deque is now empty
```

Here's the skeleton:

```
// Deque.h
class UnderflowError        // An UnderflowError object is thrown
{};                         // if you execute any operation to
                            // remove a value from a deque that is
                            // empty.

template < typename T >
class Deque
{
public:

private:
    DLList< T > list_;
};
```

You can print a Deque object. The output is similar to previous questions – just make sure you print the front first.

Once the doubly linked list is done, the Deque class is extremely easy. So you must finish the doubly linked list class first.

Q6. Now implement a Queue class. Note that a queue is just a deque except that you can only add to the back and remove from the front. Therefore the Queue class can just inherit from the Deque class. Here's the skeleton:

```
// File: Queue.h
template < typename T >
class Queue
{
public:
    void enqueue(const T & x)
    {
        deque_.push_back(x);
    }
// etc.
private:
    Deque< T > deque_;
};
```

(I'm using composition instead of inheritance.)

You can do the following

```
Queue< int > queue;
queue.enqueue(5);           // 5 is at the front
queue.enqueue(6);           // 5 is at the front and 6 is at the back
int x = queue.front();      // x = 5
queue.front() = 4;          // front of queue is now 4
int y = queue.back();       // y = 6
queue.dequeue();            // queue has 1 value
int size = queue.size();    // size is 1
queue.clear();              // queue is now empty
bool b = queue.is_empty(); // b is true           <-- CORRECTION
queue.enqueue(1);
queue.enqueue(1);
queue.enqueue(1);
queue.clear();              // queue is now empty
```

You can print a Queue object. The output is similar to previous questions – just make sure you print the front first.

The Queue class is very easy once the Deque class is done. So first make sure your Deque class is correct.