THE HONG KONGUNIVERSITY OF SCIENCE & TECHNOLOGY

Department of Computer Science & Engineering

COMP 152: Object-Oriented Programming and Data Structures

Spring 2011

Final Examination

Instructor: Albert Chung (L1) / Long Quan (L2) / Gary Chan (L3)

Date: Tuesday, 24 May 2011
Time: 4:30pm – 7:30pm
Venue: G017 / LG1031 / LG1

- **This is a closed-book examination. However, you are allowed to bring with you ONE piece of A4-sized paper with notes written or typed on both sides for use in the examination.**
- **Your answers will be graded according to correctness, precision, clarity and efficiency.**
- **During the examination, you should put aside your calculators, mobile phones, PDAs and all other electronic devices. In particular, all mobile phones should be turned off completely.**
- **This booklet has 24 single-sided pages. Please check that all pages are properly printed before you start working.**
- **You may use the reverse side of the pages for your rough work or continuation of work.**

Student name: ___________________________ English name (if any):_______________________

Student ID: ________________ Email:_______________________ Lecture & lab:____________

I have not violated the Academic Honor Code in this examination (your signature): ____________________

Question	Your score	Maximum score	Question	Your score	Maximum score
1		15	4		25
2		17	5		20
3		23			
Total					100

1. Insertion Sort with Recursion *(15 points)*

Insertion sort can be implemented with recursion stated as follows. In order to sort the array with *n* elements *A[0..n-1]*, we first (recursively) sort *A[0..n-2]* (in increasing order) and then insert *A[n-1]* into the sorted array *A[0..n-2]*.

Write a recursive function `ISRecur(int A[], int n)` that implements the recursive insertion sort as stated above. The argument A[] is the array to be sorted and *n* is the number of elements to be sorted.

For example, given `data[]={5,3,1,4,7}`, we can sort it by calling `ISRecur (data,5)` to yield the result `data[]={1,3,4,5,7}`.

2. Validity Checking for n-Queen Problem Using STL vector *(17 points)*

In chess, a queen can attack horizontally, vertically, or diagonally. The n-queen problem is to determine the positions of *n* queens on an $n \times n$ chessboard so that no any two queens can attack each other; in other words, any two queens must never share the same row, column and diagonal. The column and row are indexed from 0 to $n-1$. The solution of n-queen problem may not be unique. The following figures show two valid solutions and an invalid one for 5-queen problem.

			Q1	
Q2				
		Q3		
				Q4
	Q5			

Valid Solution A

Q1				
			Q2	
	Q3			
				Q4
		Q5		

Valid Solution B

~~Q1~~			~~Q2~~	
		~~Q3~~		
				Q4
	Q5			

Invalid Solution C
(invalid positions are crossed)

You are given that each column is placed a queen. We use a vector ***r*** to indicate the row index of the queens, i.e., r[i] is the *row* index of the queen for column *i*, where 0 <= i <= n-1. For example, Solution A above is indicated *by* `r[]={1,4,2,0,3}`, while Solution C is indicated by `r[]={0,4,2,0,3}`.

In this problem, you are to write some functions to check whether the queen positions given by the vector r[] is valid or not, i.e., whether r[] is a solution of n-queen problem. The following is the header statements of the file:

```
/* ---------------------------------------- main.cpp ---------------------------------------------- */

#include <iostream>

#include <vector>

using namespace std;


typedef vector<int> Vect;    //row indexes of queen positions r[]

//...
```

a) *(6 points)* Implement the function

```
bool ValidRow(const Vect& r)
```

which, given the queen position vector *r*, returns `true` if no two queens share the same *row* (hence attacking each other horizontally), and `false` otherwise. For example, the function returns `true` for Solution A, and `false` for Solution C because Q1 and Q2 share the same row (and hence attack each other horizontally).

b) *(7 points)* Implement the function

```
bool ValidDiag(const Vect& r)
```

which, given the queen position vector r, returns `true` if no two queens attack diagonally, and `false` otherwise. For example, the function returns `true` for Solution A, and `false` for Solution C because Q1 and Q3 attack each other diagonally.

c) *(4 points)* Implement the function

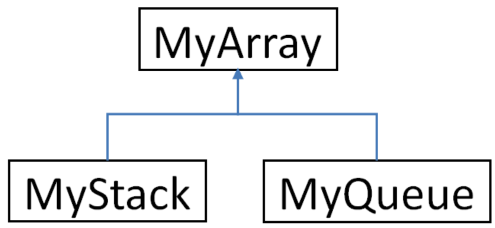
```
bool isValid(const Vect& r)
```

which, given the queen position vector r, returns `true` if r is a valid solution of the n-queen problem, and `false` otherwise. For example, the function returns `true` for Solution A, and `false` for Solution C.

3. Implementing Stacks and Queues Using Inheritance and Polymorphism

(23 points)

An abstract base class *MyArray* is to be inherited by the classe *MyStack* and *MyQueue*, which are simply the data structures of *stack* and *queue,* respectively. The figure below illustrates the inheritance relationship between the three classes, *MyArray*, *MyStack* and *MyQueue.*



The *MyArray* class is defined below. *MyArray* has an array `data` which *always* holds just enough of all the elements, i.e., the `size` of the data is equal to the number of valid elements. In this problem, all the elements are of positive values (i.e., >=0). Note that in the `push` operation an element is always added to the end of the array.

```
class MyArray
{
public:
      MyArray();   // construct MyArray: default is 0 element
      ~MyArray();
      void push(int elem); // add an element elem at the end of the array
      virtual int pop()=0;
protected:
      int* data;  // array holding just enough all the elements
      int size;   // number of elements in the array
};
```

(a) *(7 points)* Complete the implementation of the abstract base class `MyArray`.

(b) *(6 points)* `MyStack` is a concrete class inherited from `MyArray` and has no other member functions of its own. Complete the class implementation below. Recall that the `pop` operation of a stack returns and removes the most recent element that has been added to it, and it returns -1 upon error.

(c) *(6 points)* `MyQueue` is a concrete class inherited from `MyArray` and has no other member functions of its own. Complete the class implementation below. Note that the `pop` operation of a queue is the dequeue operation which returns and removes the elements in the first-in-first-out manner; it returns -1 upon error.

(d) *(4 points)* Implement a polymorphic function
`printAndClear(MyArray* stkQ)`
which pops (and hence removes) element by element in the *stkQ* and prints them.

For example, given that `data={1,2,3,4,5}` with elements pushed in such sequence, the function outputs `1 2 3 4 5` if *MyQueue* object is used and prints `5 4 3 2 1` if *MyStack* object is used.

4. List Implementation Using Template and Overloading *(25 points)*

In this problem, you are going to implement a list class template using the STL deque. The definition of the class is as follows:

```
#include <iostream>
#include <deque>
using namespace std;

template<typename T>
class List{
public:
  // some public functions to be implemented here

private:
  deque<T> list;
};
```

(a) *(6 points)* You are to implement `List` to support concatenation of an element with the list using the operator `+` (so you are overloading the operator `+`). For example, the following codes concatenate`list1` with the element 1 by putting1 at the *end* of the list, and return the concatenated result (note that `list1` does not get changed):

```
List<int> list1, list2;
list2 = list1 + 1;     // concatenate list1 with 1 and assign the
// result to list2 (list1 is not changed)
list2 = list1 + 2 + 3; // put 2 and then 3 at the end of list1
// and then assign the result to list2 (list1 is not changed)
```

Write down the function prototype to be added into the `List` class and its implementation below.

(b) *(6 points)* You are now to implement `List` to concatenate an element at the *head* of the list, using the same operator +. For example, the following codes concatenate the element 3.4 at the *beginning* of the `list1` and return the result (Note that `list1` is not changed):

```
List<double> list1, list2;

list2 = 1.2 + list1;      // add 1.2 at the beginning of list1
                 // and assign it to list2(list1 is not changed)
list2 = 5.6 + (3.4 + list1); // concatenate 3.4 and then 5.6
           // at the head of list1, and then assign the
           // concatenated result to list2 (list1 is not changed)
```

Write down what needs to be added to `List` and its implementation to achieve the above.

(c) *(6 points)* You are to implement the addition of lists of the same type using the operator +. For example, the following codes add three lists together:

```
List <int> list1, list2, list3, list4;
// …

// concatenate list1 with list2 and assign the result to list3
// list3 is [list1 list2]
list3 = list1 + list2;  // no change in list1 and list2

// concatenate list1, list2 and list3 to assign to list4
// list4 is [list1 list2 list3]
list4 = list1 + list2 + list3;    // no change in list1 to list3
```

Write down what needs to be added to `List` and the implementation to achieve the above.

(d) *(7 points)* You are to implement the deletion of the leading element using the prefix operator `--`. For example, the following codes delete the element at the *beginning* of the list (`list1` is hence changed):

```
List<int> list1;
//…
--list1;  // deleting the first element of list1
----list1;  // deleting the first two elements of list1
```

Write down what needs to be added to `List` and the implementation to achieve the above. If it is an empty list at the call, simply exit with code -1.

5. Binary tree *(20 points)*

Consider an arbitrary binary tree with *root* pointing to the root of the tree. The binary tree has the following definitions:

```
class Node{
public:
Node(int value = 0, int l = 0):
        left(NULL), right(NULL), data(value), level(l) {}

 Node* left; // pointer to the left subtree, NULL if no left subtree
 Node* right; // pointer to the right subtree, NULL if no right subtree

 int data;   // data of the node
 int level;  // level of the node in the tree
};

// binary tree class
class BT{
public:
   BT(): root(NULL){}
   BT(const BT& src){  // copy constructor
        copyNodes(src.root, root);
   }
   ~BT() {deleteNodes(root);}

   // return the number of internal nodes in the binary tree
   int internalNodes(){ return CountINodes( root );}

   // assign all the node level of the binary tree
   void assignLevel (){ assignNodeLevel(root, 0); }

   // …some other member functions not relevant to this question
private:
   Node* root;  // root of the binary tree
   // helper function for copy constructor
   void copyNodes(const Node const*, Node* &);


   void deleteNodes( Node* );    // helper function for destructor


   //helper function to count the internal nodes
   int CountINodes( const Node const*);


   //helper function for level assignment
   void assignNodeLevel( Node*, int );
};
```

(a) *(5 points)* Implement the helper function `deleteNodes(Node* node)` which deletes all the nodes in the binary tree rooted at *node*. You may get partial credit if you just describe your algorithm without implementing it.

(b) *(5 points)* Implement the helper function
`void copyNodes(const Node const* src, Node* &dest)`
which copies all the nodes from a binary tree rooted at *`src`* to form another binary tree rooted at *`dest`*. You may get partial credit if you just describe your algorithm without implementing it.

(c) *(5 points)* Recall that an internal node in a tree is a node that has at least one child. Implement the helper function

```
int CountINodes( Node* nptr) ;
```

which returns the total number of internal nodes of a tree rooted at *nptr*. You may get partial credit if you just describe your algorithm without implementing it.

(d) *(5 points)* Each node in a binary tree has a certain level which is the depth of the node from the root, i.e., the root is at level 0, its children are at level 1, and so on. Implement the helper function

```
void assignNodeLevel( Node* nptr, int level )
```

which assigns appropriately the levels of all the descendants of a binary tree rooted at `nptr` that has node level `level`. You may get partial credit if you just describe your algorithm without implementing it.

Appendix – STL Reference

C++ vector

Constructors

Syntax:

```
explicit vector ( const Allocator& = Allocator() );
explicit vector ( size_type n, const T& value= T(), const Allocator& =
Allocator() );
template <class InputIterator>
        vector ( InputIterator first, InputIterator last, const Allocator& =
Allocator() );
vector ( const vector<T,Allocator>& x );
```

Default constructor: constructs an empty vector, with no content and a size of zero.

Repetitive sequence constructor: Initializes the vector with its content set to a repetition, *n* times, of copies of *value*.

Iteration constructor: Iterates between *first* and *last*, setting a copy of each of the sequence of elements as the content of the container.

Copy constructor: The vector is initialized to have the same contents (copies) and properties as vector x.

begin

Syntax:

```
     iterator begin();
const_iterator begin() const;
```

Returns an iterator referring to the first element in the vector container.

end

Syntax:

```
     iterator end();
const_iterator end() const;
```

Returns an iterator referring to the past-the-end element in the vector container.

size

Syntax:

```
size_type size() const;
```

Returns the number of elements in the vector container.

empty

Syntax:

```
bool empty () const;
```

Returns whether the vector container is empty, i.e. whether its size is 0.

---

operator[]

Syntax:

```
      reference operator[] ( size_type n );
const_reference operator[] ( size_type n ) const;
```

Returns a reference to the element at position n in the vector container.

---

push_back

Syntax:

```
void push_back ( const T& x );
```

Adds a new element at the end of the vector, after its current last element. The content of this new element is initialized to a copy of x.

---

pop_back

Syntax:

```
void pop_back ( );
```

Removes the last element in the vector, effectively reducing the vector size by one and invalidating all iterators and references to it.

---

clear

Syntax:

```
void clear ( );
```

All the elements of the vector are dropped: their destructors are called, and then they are removed from the vector container, leaving the container with a size of 0.

C++ deque

Constructor

Syntax:

```
explicit deque ( const Allocator& = Allocator() );
explicit deque ( size_type n, const T& value= T(), const Allocator& =
Allocator() );
template <class InputIterator>
       deque ( InputIterator first, InputIterator last, const Allocator& =
Allocator() );
deque ( const deque<T,Allocator>& x );
```

Default constructor: constructs an empty deque container, with no content and a size of zero.
Repetitive sequence constructor: Initializes the container with its content set to a repetition, n times, of copies of value.
Iteration constructor: Iterates between first and last, setting a copy of each of the sequence of elements as the content of the container.
Copy constructor: The deque container is initialized to have the same contents (copies) and properties as deque container x.

begin

Syntax:

```
    iterator begin ();
const_iterator begin () const;
```

Returns an iterator referring to the first element in the container.

end

Syntax:

```
    iterator end ();
const_iterator end () const;
```

Returns an iterator referring to the past-the-end element in the deque container.

size

Syntax:

```
size_type size() const;
```

Returns the number of elements in the deque container.

empty

Syntax:

```
bool empty ( ) const;
```

Returns whether the deque container is empty, i.e. whether its size is 0.

operator[]

Syntax:

```
     reference operator[] ( size_type n );
const_reference operator[] ( size_type n ) const;
```

Returns a reference to the element at position n in the deque container.

push_back

Syntax:

```
void push_back ( const T& x );
```

Adds a new element at the end of the deque container, after its current last element. The content of this new element is initialized to a copy of x.

push_front

Syntax:

```
void push_front ( const T& x );
```

Inserts a new element at the beginning of the deque container, right before its current first element. The content of this new element is initialized to a copy of x.

pop_back

Syntax:

```
void pop_back ( );
```

Removes the last element in the deque container, effectively reducing the container size by one.

pop_front

Syntax:

```
void pop_front ( );
```

Removes the first element in the deque container, effectively reducing the deque size by one.