CS 104 (Spring 2014) — Midterm
03/06/2014

Good Luck

Your Name, USC username, and Student ID:

This exam has 9 pages and 5 questions. If yours does not, please contact us immediately. Please read and reread each question carefully before trying to solve it.

In your solutions, you should try hard to write mostly correct C++. Minor syntax errors will (at most) lose small numbers of points, but you should also demonstrate a mastery of most C++ syntax.

You don't need to duplicate the code we're giving you — just give the piece of code to add to what we are giving you in response to questions.

Question	Points possible	Points
1	10	
2	15	
3	30	
4	20	
5	25	
Total	100	

(1) [3+4+3=10 points]

(a) Consider the declaration of the following `Map<K,V>` class member function:

```
template <class K, class V>
V const & Map<K,V>::get(const K& key) const
{ ... }
```

Explain what each of the 3 `const` keywords (ordered from left to right) will cause the compiler to check/enforce.

(b) For each of the descriptions, indicate what ADT is appropriate to store the following information and show what types would be used for the template arguments (e.g., `map<string, int>` or `list<double>`). If you think that there are multiple equally good options, feel free to justify the choice you make.

- Data structure allowing you to find a book's title from its ISBN identifier (13 characters, mostly digits, but could contain the letter 'X').

- Data structure storing the type of obstacle at each square of a 40×40 level of an arcade game.

- Waitlist of students who want to enroll in CS104, but couldn't get in yet.

- Data structure to store the content of each line of code of a possibly long C++ program.

(c) Suppose that you have a class which requires a deep copy. Which functions must you define? (Mark all that apply.)

[ ] Copy constructor
[ ] Destructor
[ ] Assignment operator

(2) [7+8=15 points]
In this problem, we will work on a singly linked list of integers. The items are of the following type:

```
struct IntItem {
  int value;
  IntItem *next;
  IntItem (int v, IntItem *n) { value = v; next = n; }
};
```

(a) Write a *recursive* function `printLL` to print all items in the linked list in the order in which they appear in the list, on one line. For full credit, you should print an endline after the last number, but you will get most of the credit if you do not print it. Your function *must* be recursive, and it must fit the following definition:

```
void printLL (IntItem* head)
{  // your code goes here









}
```

(b) Assume that `printLL` works as asked, and prints all items in the linked list in order (even if yours doesn't) on one line. What will be printed by the following program?

```
#include <iostream>

// IntItem definition from above inserted here

// your printLL function is here as well.

IntItem* mystery1 (int start, int end) {
  if(start < end) return new IntItem(start, mystery1(start+1,end));
  else return NULL;
}

IntItem* mystery2 (IntItem* head, IntItem* prev) {
  if(head != NULL) return mystery2(head->next, new IntItem(head->value, prev));
  else return prev;
}

int main() {
  IntItem* p1 = mystery1(0,5);
  printLL(p1);
  IntItem* p2 = mystery2(p1, NULL);
  printLL(p2);
  return 0;
}
```

Output:

(3) [3+17+10=30 points]

In this problem, you will be implementing a data structure for a "forgetful brain". The way a forgetful brain works is as follows: it has a fixed and limited capacity for facts (which for our purposes are just strings). Initially, your brain is empty. As you learn more facts, they are added to the brain. When the brain is full, any newly added fact displaces one that was previously there, meaning that you forget the previous fact. Which fact gets displaced? The one that was used least recently. There are two ways in which your brain can use a fact: (1) Learning a new fact is using it; that is, you remember the things you learned recently. (2) You can deliberately recall a fact.

As an example, suppose that the brain has a capacity of 3 facts, and you learn A, B, C in order. Then, you recall A, and then you learn D. At this point, B is the least recently used fact, so you forget B in order to learn D. When you learn E, you next forget C, and if you next learn F, you forget A. If instead, you recalled A again before learning F, then you would next forget D. The `Brain` class thus looks as follows:

```
class Brain
{
  public:
    Brain (int capacity);
      // create a new Brain with the given fixed capacity.
    void remember (const string & fact);
      /* access the fact, i.e., mark it as freshly remembered.
         We will never ask you to remember a fact that you haven't learned. */
    void learn (const string & fact);
      /* add the given fact to the brain, and mark it as freshly remembered.
         If the brain is full, throw out the least recently used fact
         to make room for the newly added fact. */
};
```

In order to implement this `Brain` class, you should use the following modified version of a `List<T>` class, which a friend has already written for you, and which you cannot modify. We guarantee that you will never be asked to learn the same fact twice, so you don't need to worry about duplicates in your brain or list.

Your friend's modified list class is called `LimitedList<T>`; it's basically a `List<T>` which will never resize, even if you reached the capacity. Instead, it will throw an exception if you exceed its capacity. Here is your friend's header file.

```
template <class T>
class LimitedList {
  public:
    LimitedList (int capacity);
     /* creates a list fixed to this capacity. It can still grow and
        shrink with insert/remove, but if an insert would make the
        size exceed the capacity, it will throw an exception. */

    void set (int i, const T & item); // exactly the same as standard set
    const T & get (int i) const; // exactly the same as standard get
    void insert (int i, const T & item);
      /* almost the same as standard insert, except if the list is
         full, it will throw an exception rather than resizing.
         In the case of the exception being thrown, it will not alter
         the list. */
    void remove (int i); // exactly the same as standard remove

  protected:
    int find (const T & item) const;
      /* returns the first location at which item is stored in the
         list. Returns -1 if the item isn't in the list. */
    int size () const;
      // runs in time O(1) and returns the number of items currently stored in the list

  private:
    // the actual variables used to store stuff
};
```

(a) How should you use `LimitedList` to build `Brain`? Inheritance (if so, what type), composition, or other? Why?

(b) Give an implementation of the `Brain` class, by adding your code in the following piece of code.

```
class Brain
// relevant code here if you want
{
  public:
    Brain (int capacity)
     {









      }

    void remember (const string & fact) {









      }

    void learn (const string & fact) {









      }

    private:   // any data or methods you would like to add




};
```

(c) In big-$O/\Omega/\Theta$ notation, analyze the worst-case running time of the `learn` function under your implementation. Do this analysis for two different implementations that your friend may have made of `LimitedList<T>`:

- Implemented as a fixed-size array internally, doing insertions and removals the way you learned in class. Show your work and explain how you arrived at the result.

- Implemented as a doubly-linked list internally with both a `head` and `tail` pointer, and finding the i^{th} element by searching from the right or left end of the linked list, whichever is closer. (Doing insertions and removals the way you learned.) Show your work and explain how you arrived at the result.

(4) [20 points]
Here is a piece of code. Tell us what it outputs. (You will get partial credit for partially correct answers.)

```
class Question {
public:
  Question(int v) : val(v) { }
  virtual ~Question() { cout << "d1" << endl; }
  virtual string studentResponse() = 0;
  int getValue() { return val; }

private:
  int val;
};

class NonTrivialQuestion : public Question {
public:
  NonTrivialQuestion() : Question(10) { }
  NonTrivialQuestion(int v) : Question(v) { }
  ~NonTrivialQuestion() { cout << "d2" << endl; }
  string studentResponse() { return "I got this!"; }
  int getValue() { return 15 + Question::getValue(); }
};

class DifficultQuestion : public NonTrivialQuestion {
public:
  DifficultQuestion() : NonTrivialQuestion() { }
  ~DifficultQuestion() { cout << "d3" << endl; }
  string studentResponse()
      { return "When are office hours?"; }
};

int main()
{
  Question* p[2];
  p[0] = new NonTrivialQuestion(15);
  p[1] = new DifficultQuestion;
  for(int i=0; i < 2; i++){
    cout << p[i]->getValue() << endl;
    cout << p[i]->studentResponse() << endl;
  }
  NonTrivialQuestion* q[2];
  q[0] = new NonTrivialQuestion(15);
  q[1] = new DifficultQuestion;
  for(int i=0; i < 2; i++){
    cout << q[i]->getValue() << endl;
    cout << q[i]->studentResponse() << endl;
  }
  delete p[1];
  return 0;
}
```

Output:

(5) [25 points]

Imagine that you are writing a simple game for young children to learn to recognize and match shapes. For our purposes, shapes are restricted to squares and circles. Shapes arrive over time and fall on a stack of items. See Figure 1.

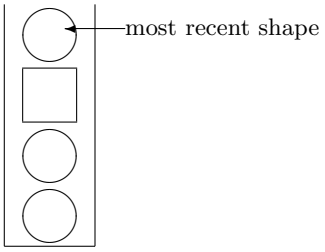

Figure 1: An example of a game state.

At each point, the child may press one of two buttons, 'L' and 'R'. 'L' should be the button to press if there is a square on top, and 'R' if there is a circle on top. More precisely, the rules are as follows:

- If the child presses the correct button ('L' when the top shape is a square, 'R' when it is a circle), the child gets a point, and the top shape disappears.
- If the child presses the wrong button ('L' when the top shape is a circle, 'R' when it is a square), the child loses a point (the points could become negative), and no shape disappears.
- If the child presses a button when the stack is empty, nothing happens.
- The game starts at 0 points.
- Items appear at a regular pace, and their appearance has nothing to do with what buttons are pressed.
- The game may end even when there are still items left.

You will be given a sequence consisting of the letters 'L', 'R', 'S', 'C' (representing the pressing of the two buttons, and the arrival of the two shapes, respectively), and are to determine from it the final score of the child. We promise that the sequence will contain only those four letters, and nothing else.

As an example, for the sequence "SCLLR", the final score will be -1, and there will be a square left in the end. The two presses of the 'L' button are wrong, so each incurs a penalty of 1 point and changes nothing. The press of 'R' is correct, so it earns a point and makes the circle disappear, but the square remains.

Hint: the use of a stack is strongly recommended. So much so that we already enter `#include<stack>` for you. That gives you a templated `stack<T>` class, which provides the functions `void push(T)`, `void pop()`, `T top()`, and `bool isEmpty()`. You can use this without defining your own stack.

Insert your code in the following:

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

int main ()
{
   string s;   // the sequence of characters describing the game's events
   int score = 0;
   cin >> s;

   // your code goes here.









   cout << "The score is " << score << endl;
   return 0;
}
```