# CSCI 104 Practice Midterm Exam —

**Your Name, USC username, Class time, and Student ID:**

| Question | Points possible | Points |
|---|---|---|
| 1 | 16 | |
| 2 | 12 | |
| 3 | 10 | |
| 4 | 10 | |
| 5 | 16 | |
| 6 | 16 | |
| 7 | 20 | |
| Total | 100 | |

**Do not open the exam until instructed to do so!**

**Turn off all cell phones!**

This packet has 9 pages (including this cover page) and 7 questions. If yours does not, please contact us immediately.

This exam is closed book. You are allowed one $8.5 \times 11$ inch note sheet (front and back). You will have 110 minutes to work on this exam.

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. And of course, you should make sure to not have any memory leaks.

Good Luck

**(1)** [4+4+4+4=16 points]

(a) Consider the declaration of the following class member function:

```
const MyClass2& map::get(const MyClass1& key) const
{ ... }
```

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

(b) Identify and explain any errors you see in the following code.

```
#include<iostream>
class BaseClass {
  public:
    virtual ~BaseClass() {};
};
class SubClass : public BaseClass {};
int main() {
  SubClass *s = new BaseClass;
  delete s;
  return 0;
}
```

(c) If you feel that you need to implement a destructor for your class, what other functions do you most likely need to implement as well?

(d) Consider the following function signature. Explain the rationale behind the choice of return value.

```
ostream& operator<< (ostream &o, const LinkedList &ll);
```

**(2)** [3+3+3+3=12 points]
Carefully consider the following applications, and indicate which ADT is most appropriate for each (map, set, list, stack, or queue). Make sure to explain your reasoning. More detail is better: instead of just "List" you could state "a list of student records".

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

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

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

(d) A data structure that allows you to input an academic semester and find all students who earned an A in CS104 that semester.

**(3)** [10 points]
Imagine that you are developing an Office Suite, and you anticipate that you will need classes for **Document**, **Text Document**, **Spreadsheet**, **Read-Only Text Document**, and **Author**. What relationships exist between these classes? Specifically, describe (in words and/or diagrams) all instances of public or private inheritance, as well as has-a relationships.

**(4)** [10 points]
Assume that `printLL` prints all items in a linked list in order on one line. What will be printed by the following program?

```
// printLL function is defined here.

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

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

IntItem* mystery2 (IntItem* head, IntItem* prev) {
  if(head != nullptr) 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, nullptr);
  printLL(p2);
  return 0;
}
```

**(5)** [16 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:

**(6)** [16 points]
Consider the following code:

```
class IntArray {
  public:
    IntArray(const IntArray &other);
    //other class functions are here, which you don't need to worry about
  private:
    int *myarray; //data
    int used; //number of elements in myarray
    int alloc; //number of allocated indices.  Unused indices have garbage values.
};
class ArrayOfArrays {
  public:
    ArrayOfArrays(const ArrayOfArrays &other);
    //other class functions are here, which you don't need to worry about
  private:
    IntArray **myarray; //an array of IntArray pointers.
    int used; //number of arrays in myarray
    int alloc; //number of allocated indices.  Unused indices have garbage values.
};
```

Using the front and back of this page, implement deep copy constructors for both `IntArray` and `ArrayOfArrays`.

**(7)** [3+17=20 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`, 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`; it's basically a `List` 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.

```
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 string & item); // exactly the same as standard set
    const string & get (int i) const; // exactly the same as standard get
    void insert (int i, const string & 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 string & 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;
      // 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




};
```