Department of Computer Science

First Examination

CS 225 Data Structures and Software Principles
Spring 2008
7p-9p, Tuesday, February 19

Name:
NetID:
Lab Section (Day/Time):

- This is a **closed book** and **closed notes** exam. No electronic aids are allowed, either.
- You should have 5 problems total on 17 pages. The last two sheets are scratch paper; you may detach them while taking the exam, but must turn them in with the exam when you leave.
- Unless otherwise stated in a problem, assume the best possible design of a particular implementation is being used.
- Unless the problem specifically says otherwise, (1) assume the code compiles, and thus any compiler error is an exam typo (though hopefully there are not any typos), and (2) assume you are NOT allowed to write any helper methods to help solve the problem, nor are you allowed to use additional arrays, lists, or other collection data structures unless we have said you can.
- Please put your name at the top of each page.

Problem	Points	Score	Grader
1	20		
2	20		
3	20		
4	20		
5	20		
Total	100		

1. **[Pointers, Parameters, and Miscellany – 20 points].**

MC1 (2.5pts)

Which of the following is a correct way to declare an instance of a `stack` whose parameterized type is an integer?

(a) stack s⟨int⟩;

(b) **stack⟨int⟩ s;**

(c) stack s(int);

(d) more than one of (a), (b), and (c) are correct.

(e) none of the above is correct.

MC2 (2.5pts)

Consider the following C++ statements:

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

void increment1(int  x) { x++; }
void increment2(int* x) { x++; }
void increment3(int& x) { x++; }

int main() {
  int x = 1;
  increment1(x);
  increment2(&x);
  increment3(x);
  cout << x << endl;
  return 0;
}
```

What is the printed out when this code is compiled and run?

(a) 1

(b) **2**

(c) 3

(d) 4

(e) none of the above

MC3 (2.5pts)

Consider the following lines of C++:

```
int* x, y;
int z = 10;
x = &z;
y = x;
*x = 15;
cout <<  ‘‘z is ’’  <<  z  << endl;
```

What is their result?

(a) Print `z is 10` to standard out.

(b) Print `z is 15` to standard out.

(c) **The code will not compile due to a type error.**

(d) The code will compile but it will segfault.

(e) None of these options describes the behavior of this code.

MC4 (2.5pts)

Given an array-based implementation of a list of n elements, what is the worst case running time for inserting an element to an arbitrary position in the list?

(a) O(1)

(b) O($\log n$)

(c) **O(n)**

(d) O(n^2)

(e) None of the above

MC5 (2.5 pts)

Consider the following partial function definition:

```
sphere bounce(sphere & s) {
     sphere t(s);

     // play with s and t (no copy constructors happen here)

     return t;
}
```

How many times is the copy constructor employed when the `bounce` function is executed?

(a) 0

(b) **1**

(c) **2**

(d) 3

(e) There is no way to tell how many times the copy constructor is called.

MC6 (2.5pts)

Suppose `myVar` is declared as follows: `int ** myVar;`. Which of the following could describe the variable `myVar` once it has been initialized?

I. `myVar` is a dynamic array of integer pointers.
II. `myVar` is a pointer to a dynamic array of integers.
III. `myVar` is a dynamic array of dynamic arrays of integers.

(a) None are a valid description of `myVar`.

(b) Only I. is a valid description of `myVar`.

(c) I. and III. are valid descriptions of `myVar`.

(d) I. and II. are valid descriptions of `myVar`.

(e) **All three are valid descriptions of `myVar`.**

MC7 (2.5pts)

Consider the following partial class definitions:

```
class Sphere {
private:  double theRadius;
public:
     // Lots of member functions go here.
     virtual double getArea() const;  //    ***computes surface area***
     virtual void displayArea() const;
          //  displayArea() includes the statement "cout << getArea() << endl;"
};

class Ball: public Sphere {
private:  string theName;
public:
     // Lots of member functions.  Note:  Ball class inherits displayArea().
     double getArea() const;  //     ***computes cross sectional area***
};
```

Now suppose you have the following in your `main()` function:

```
Sphere mySphere;
Ball myBall;
myBall.displayArea();
```

Which of the following describes the behavior of this code in `main()`?

(a) `myBall`'s surface area is displayed.

(b) **`myBall`'s cross sectional area is displayed.**

(c) The call to `getArea()` within `displayArea()` is ambiguous, so there is a compile error.

(d) A bus error occurs at run time.

(e) None of these options describes the behavior of this code.

MC8 (2.5pts)

Which of the following characterize C++ as an object oriented programming language?

(a) Pointers, memory management, and inheritance.

(b) Polymorphism, memory management, and encapsulation.

(c) **Polymorphism, inheritance, and encapsulation.**

(d) Pointers, inheritance, and overloading operators.

(e) Overloading operators, inheritance and encapsulation.

2. **[The Big Three – 20 points].** Consider the following partial class definition:

```
class CollegeCourse
{
  private:
    Image** studentPhotos;
    int* studentIDs;
    std::string* professor;
    int maxClassSize;
    int courseID;

    // some helper functions

  public:
    // constructors and destructor

    CollegeCourse(const CollegeCourse & source);

    // lots of public member functions
};
```

The `studentPhotos` structure is a dynamically allocated array of `Image` pointers. The `studentIDs` structure is a dynamically allocated array of ints.

Both `studentIDs` and `studentPhotos` arrays have `maxClassSize` elements. You can assume that `maxClassSize` is greater than zero.

`professor` is a dynamically allocated string object.

In this question you will write the copy constructor for the `CollegeCourse` class that you would include in the `collegecourse.cpp` file.

You may assume that all pointers are valid. That is, they are either `NULL` or they point to an object of the specified type. Furthermore, you may assume that the `Image` and `string` classes have an appropriately defined "Big Three" (destructor, copy constructor, and assignment operator).

You may write your answer on the following page. To grade this problem, we will first read your comments to make sure you intend to do the right thing, and then we'll check your code to make sure it does what your comments say it should. As a result, be sure your comments are coherent, useful, and reflective of your approach to the problem.

Solution:

```
CollegeCourse(const CollegeCourse & source) {
  maxClassSize = source.maxClassSize;
  courseID = source.courseID;

  // allocate new memory for studentPhotos and studentIDs
  studentPhotos = new Image*[maxClassSize];
  studentIDs = new int[maxClassSize];

  // make a deep copy of professor using string copy constructor
  if(source.professor != NULL) {  // not required but is better to check
    professor = new string(*source.professor);
  }
  else {
    professor = NULL;
  }

  for(int i = 0; i < maxClassSize; ++i) {
    // check for NULL pointer to avoid segfault
    if(source.studentPhotos[i] != NULL) {
      // make a deep copy of studentPhotos using Image copy constructor
      studentPhotos[i] = new Image(*source.studentPhotos[i]);
    }
    // source Image pointer is NULL, so set the copy's pointer to NULL
    else {
      studentPhotos[i] = NULL;
    }
    // make a deep copy of studentIDs
    studentIDs[i] = source.studentIDs[i];
  }
}
```

```
Grading Rubric (20pts)
1pt  - copy maxClassSize
1pt  - copy courseID
2pts - dynamically allocate new memory for studentPhotos
1pt  - dynamically allocate new memory for studentIDs
2pts - deep copy of professor
2pts - check for NULL before dereferencing source.professor
2pts - deep copy of studentPhotots
1pt  - set studentPhotos[i] to NULL if source.studentPhotos[i] is NULL
2pts - deep copy of studentIDs
2pts - comment about memory allocation
1pt  - comments about NULL handling
```

3pts - comments about deep copying studentIDs, studentPhotos, and professor

Penalties:
-1 pt for a syntax error
-1 pt setting a local CollegeCourse instance rather than the instance variables
-1 pt redeclaring instance variables as local variables
-1 pt for self assignment checking
-1 pt for returning anything from constructor
-2 pts for deallocating invalid pointers in a copy constructor

3. **[Short answer – 20 points].**

(a) (5 points) One problem with an array-based implementation of a stack is the resizing we must do when the stack fills. A scheme for overcoming this drawback employs periodic resizing of the array by some constant amount c. That is, when the array of size k fills, we create a new array of size $k + c$ and copy the elements from the original array into the new one, maintaining the original order so that the stack is not corrupted. Give a complete analysis of this scheme over a sequence of n `push` operations on the stack. Assume that the stack is empty to start. Please report your answer as an average cost per `push`.

Solution: over n pushes there are n/c copying events, and that in the ith copy event, $i \cdot c$ data items are copied from the old full array to the new one. The following sum reflects the total number of copies:

$$\sum_{i=0}^{\frac{n}{c}} i \cdot c = c \sum_{i=0}^{\frac{n}{c}} i = \frac{c}{2} \cdot \frac{n}{c}(\frac{n}{c} + 1) = O(n^2)$$

Over n pushes the average cost per push is thus $O(n)$.

Rubric: 3 points for analysis, 2 points for correct answer. Two of three analysis points were awarded for any answer in which the process was decomposed appropriately into the number of copy events and the size of each, even if the values used were incorrect. Forgetting to report the average was a 1 point penalty.

(b) (5 points) Suppose a queue is implemented as a singly-linked list with head and tail pointers, and the back of the queue is at the head of the list (that is, `enQueues` occur at the head of the list). What is the worst case running time of the best implementation of a `deQueue()` operation?

Solution: in order to complete the deQueue operation and end with a tail pointer at the last element in the list, we must first traverse the list to find the node BEFORE the one we wish to delete. This is a linear time operation in the number of nodes in the list, or $O(n)$.

Rubric: 3 points for analysis, 2 points for correct answer. Any answer that assumed the tail pointer could be moved back without traversing the list received a maximum of 2 points.

(c) (5 points) Suppose a queue is implemented as a singly-linked list with head and tail pointers, and the back of the queue is at the head of the list (as in the previous part of the question). We want to implement a `clearQueue()` function that removes all the data from the queue. What is the worst case running time of the best implementation of a `clearQueue()` operation?

Solution: we can simply traverse the list, deleting elements as we go, for a total running time of $O(n)$, where n is the number of elements in the list.

Rubric: 3 points for analysis, 2 for correct answer. Note that an incorrect answer to part 2 could result in a correct answer for this problem. Such responses received only the 2 points for the correct answer, and no analysis points. Answers that called dequeue repeatedly received 3 points if the analysis was correct.

(d) (5 points) Consider the following template function definition:

```
template <class T, class U>
T addEm(T a, U b) {
      return a + b;
}
```

Describe what happens in each of the following calls to `addEm`.

i. `addEm<int,int>(3,4);`
Solution: returns integer 7.

ii. `addEm<double,int>(3.2,4);`
Solution: returns double 7.2.

iii. `addEm<int,double>(3,4.5);`
Solution: returns integer 7.

iv. `addEm<string,int>("hi",4);`
Solution: This is a compiler error unless the + sign is overloaded to add integers to strings.

v. `addEm<string,string>("bye","hi");`
Solution: returns string "byehi".

Rubric: one point each part.

4. **[Lists: Orders of Magnitude – 20 points].**

You are given the following `ListNode` class:

```
class ListNode {
public:
   int element;
   ListNode* next;
};
```

and a singly-linked list made up of such nodes, with a head pointer but no tail pointer. We will further guarantee that all the integers stored in the nodes are nonnegative. You should write a function that reorders the list so that it is sorted by order of magnitude. That is, all the 1-digit numbers would appear first, followed by all the 2-digit numbers, and so on. Within a given order of magnitude, the list elements should appear in their original order.

As an example, the input list
`35->2->44->420->56789->65->287->8->48->NULL`
should end up looking like
`2->8->35->44->65->48->420->287->56789->NULL` .

Your function should also print out how many elements of each order of magnitude are in the list. The function should be named `OrderMagnitude`, return nothing, and have one parameter, a reference to a `ListNode` pointer, which will point to the head node of a singly-linked list. You may not create or delete any `ListNode`s in writing this function, but you can declare as many additional `ListNode` pointers as you need. (Hint: You may find it useful to know that the largest number an int can hold is 2147483647.)

You may write your answer on the following page. To grade this problem, we will first read your comments to make sure you intend to do the right thing, and then we'll check your code to make sure it does what your comments say it should. As a result, be sure your comments are coherent, useful, and reflective of your approach to the problem.

Finally, your score on this problem will be based on both the accuracy and the efficiency of your approach.

Solution:

```
void OrderMagnitude(ListNode*& list_head) {
  if (list_head == NULL)
    return;

  // declarations and initializations
  ListNode* temp = list_head;  // == head, by assumption
  ListNode *newHeads[10], *newTails[10], *tail;
  int counts[10], order;
  for (int i=0; i<10; i++) {
    newHeads[i] = NULL;
    newTails[i] = NULL;
    counts[i] = 0;
  }

  // determine order of the element -- we know temp->element >= 0
  while (temp != NULL) {
    if ( temp->element < 10)       // order 10^0
      order = 0;
    elseif ( temp->element < 100)  // order 10^1
      order = 1;
    elseif ( temp->element < 1000)
      order = 2;
    elseif ( temp->element < 10000)       // if the math library
      order = 3;                          // is included, this can
    elseif ( temp->element < 100000)      // be reduced to
      order = 4;                          // order = log10(temp->element);
    elseif ( temp->element < 1000000)     // which implicitly converts
      order = 5;                          // the double given by log10
    elseif ( temp->element < 10000000)    // to an int for us
      order = 6;
    elseif ( temp->element < 100000000)
      order = 7;
    elseif ( temp->element < 1000000000)
      order = 8;
    else  // order 10^9
      order = 9;

    // put it in the correct sublist
    counts[order]++;
    if (newHeads[order] == NULL)  // first element of this magnitude
      newHeads[order] = newTails[order] = temp;
    else {   // add it to the sublist, and update the tail
      newTails[order]->next = temp;
      newTails[order] = temp;
```

```
    }
    temp = temp->next;
  }    // now all the nodes are in their correct sublists

  // link up all the sublists
  int i = 0;
  while (counts[i] == 0)
    i++; // increment until the first sublist with nodes is found
  list_head = newHeads[i];  // set head for complete list
  tail = newTails[i];       // set tail of new list so far

  for (int j=i+1; j<10; j++) {
    if (newHeads[j] != NULL) {   // if anything is in this sublist
      tail->next = newHeads[j];  // link it up
      tail = newTails[j];        // move tail to new tail
    }
  }
  tail->next = NULL;  // set end of completed list

  // print out orders
  cout << "The numbers in the list are distributed as follows:" << endl;
  for (int i=0; i<10; i++) {
    cout << i+1 << "-digit numbers in the list:  " << counts[i] << endl;
  }
}  // end function
```

Grading scheme:

- 5 points for comments
- 11 points for correct code:
  - 2 pts - declare and initialize appropriate variables
  - 1 pt - determine the order of each element
  - 1 pt - keep track of number of each order
  - 2 pts - move element into the correct sublist
  - 3 pts - correctly link the sublists
  - 2 pts - print out the number of elements of each order
- 4 points for efficiency:
  - +4 pts if $O(n)$ with one pass through the list
  - +2 pts if other $O(n)$ solution (usually 10 passes through the list)
  - +1 pt for anthing else that works (e.g., $O(n^2)$ running time)
- Miscellaneous deductions:
  - -12 pts if the whole list was sorted instead of following the problem spec
  - -2 pts if elements in an order of magnitude appear in a different order in the final list

- -1 pt for misuse of a library function
- -1/2 pt each for syntax errors (up to -4 pts)
- -2 pts if any `ListNode`s were created or deleted
- -2 pts if helper functions were declared or used

5. **[Queues and Stacks – 20 points].**

(a) Imagine that you are given a standard `Stack` class and `Queue` class, both of which are designed to contain integer data (interface provided on following pages).

Your task is to write a function called `scramble` that takes one argument: a reference to a `Queue`. The function should reverse the order of SOME of the elements in the queue, and maintain the order of others according to the following pattern:

The first element stays on the front of the queue.
Then the next two elements are reversed.
Then the next three elements are placed on the queue in their original order.
Then the next four elements are reversed.
Then the next five elements are place on the queue in their original order.
etc.

For example, given the following queue,

```
front                                                 rear
 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16
```

we get the following result:

```
front                                                 rear
 00 02 01 03 04 05 09 08 07 06 10 11 12 13 14 16 15
```

Any "leftover" numbers should be handled as if their block was complete. See for example the way 15 and 16 were treated in our example above.

We are putting some constraints on your implementation. Specifically, you must write the function using only the variable declarations we provide. We are allowing you four local variables: two `int`s, one `Stack`, and one `Queue`.

You may write your answer on the following page. To grade this problem, we will first read your comments to make sure you intend to do the right thing, and then we'll check your code to make sure it does what your comments say it should. As a result, be sure your comments are coherent, useful, and reflective of your approach to the problem.

problem 5 continued...

Solution:

```
void scramble(Queue & queue) {
   int temp1, temp2;  //  These names are generic to avoid
   Stack s;           //  giving you too many clues about
   Queue q;           //  how we think you should use them.

   temp1=0; //keeps track of current block size

   while(!queue.isEmpty())
   {
      temp1++;  //increment block size

      if(temp1%2) //odd block size
         //move next block from queue to q
         for(temp2=0;temp2<temp1;temp2++)
            if(!queue.isEmpty()) q.enqueue(queue.dequeue());
      else //even block size
      {
         //move next block from queue to s
         for(temp2=0;temp2<temp1;temp2++)
            if(!queue.isEmpty()) s.push(queue.dequeue());
         //move s to q, reversing elements
         while(!s.isEmpty()) q.enqueue(s.pop());
      }

   //move back into the original queue
   while(!q.isEmpty()) queue.enqueue(q.dequeue());
}

Grading Rubric (15pts)
up to 5 pts - comments
up to 5 pts - concept/logic
up to 5 pts - c++

Penalties:
-1 pt for each small problem, i.e.
-1 pt for pop or dequeue on an empty stack or queue
-1 pt for defining extra variables
-1 pt for invalid return
```

(b) (5 points) Analyze the running time of the code you wrote in part (a). You should state the worst-case time complexity of your code when operating on a list with n nodes and briefly explain why it would take that much time.

Solution:
Each push, pop, enqueue, and dequeue takes O(1) time. Elements in odd blocks each force a dequeue from queue, an enqueue to q, a dequeue from q, and an enqueue to queue. Elements in even blocks each force a dequeue from queue, a push to p, a pop from p, an enqueue to q, a dequeue from q, and an enqueue to queue. Thus for each element originally in queue a constant number of operations are performed so scramble runs in O(n) time.

Grading Rubric (5 pts)
3 pts for correct answer
2 pts for analysis

```
class Stack {  // partial class definition
   public:
      Stack();

      void push(int e);
      int pop();

      bool isEmpty(); // returns true if the stack is empty

   private:
      // we're not telling
};

class Queue {  // partial class definition
   public:
      Queue();

      void enqueue(int e);
      int dequeue();

      bool isEmpty(); // returns true if the queue is empty

   private:
      // we're not telling
};
```