University of Illinois at Urbana-Champaign
Department of Computer Science

First Examination

CS 225 Data Structures and Software Principles
Fall 2007
7p-9p, Thursday, October 4

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 20 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)

Consider the following C++ statements:

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

int main(){
   int *y;
   int x = 36;
   //code to be inserted here
   (*y)++;
   cout << x << endl;
   return 0;
}
```

Which of the following choices, if inserted at the line marked, will result in `37` being sent to standard ouput? **Answer: b**

(a) `y = x;`

(b) **y = &x;**

(c) `*y = x;`

(d) `*y = &x;`

(e) None of these will give the result we want.

MC2 (2.5pts)

We need to write our own destructor (instead of using the default one) when...**Answer:c**

(a) we have private members that are pointers.

(b) we have private members that are arrays.

(c) we have allocated dynamic memory in our constructors.

(d) two of these three require that we write our own destructor.

(e) all of the above require that we write our own destructor.

MC3 (2.5pts)

Consider the following C++ statements:

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

void myFun(int * x) {
int *y = new int;
*y = 16;
x=y;
delete y;
}

int main(){
   int i = 9;
   myFun(&i);
   cout << i << endl;
   return 0;
}
```

What is the result when this code is compiled and run?**Answer: c**

(a) Nothing. This code does not compile.

(b) 16 is sent to standard out

(c) 9 is sent to standard out

(d) The address of `i` is sent to standard out

(e) This compiles fine, but will generate a segmentation fault when executed (runtime error).

MC4 (2.5pts)

Consider this prototype for a template function:

```
template <class Item>
void foo(Item x);
```

What is the right way to call the `foo` function with an integer argument `i`?**Answer: a or b**

(a) `foo( i );`

(b) `foo<int>( i );`

(c) `foo<Item>( i );`

(d) `foo( <int> i );`

(e) `foo( <Item> i);`

MC5 (2.5pts)

Consider the following C++ statements:

```
class Ball {
   public:
       //constructors and other member functions
       ...
       void setSameRadius(Ball & orig) const;
   private:
      int radius;
}

void Ball::setSameRadius(Ball & orig) const {
    radius = orig.radius;
}
```

Consider the `Ball` class and the `setSameRadius` function shown above. This function is expected to set the radius of this Ball object to the same value as the radius of the orig Ball object.

This function: **Answer: d**

(a) compiles and runs without error but does not perform the desired task.

(b) compiles and runs without error and performs the desired task.

(c) encounters a run-time error.

(d) does not compile because of an error involving setSameRadius.

(e) compiles and runs without error, performs the desired task, but changes the value of `orig`.

MC6 (2.5pts)

Suppose we have implemented a queue as a singly linked list with a tail pointer modeled here:

Which of the following best describes the running time of the `enqueue` and `dequeue` operations if the rear of the queue must be at the head of the linked memory structure? (n is the number of elements in the queue.) **Answer: c**

(a) $O(1)$ for both `enqueue` and `dequeue`

(b) $O(n)$ for both `enqueue` and `dequeue`

(c) $O(1)$ for `enqueue` and $O(n)$ for `dequeue`

(d) $O(n)$ for `enqueue` and $O(1)$ for `dequeue`

(e) None of these is the correct choice.

MC7 (2.5pts)

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

(a) `myVar` is a dynamic array of integer pointers.

(b) `myVar` is a pointer to a dynamic array of integers.

(c) `myVar` is a dynamic array of dynamic arrays of integers.

(d) Two of these are valid descriptions (and one is not).

(e) All three are valid descriptions.

MC8 (2.5pts)

Consider the following partial class definitions:

```
class Sphere {
private:  double theRadius;
public:
     // Lots of member functions go here.
     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()`? **Answer: a**

(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.

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

```
class Slideshow
{
        private:
                Image** slides;
                int* durations;
                int slidecount;

               // perhaps some helper functions

        public:
               // default constructor and destructor

               Slideshow(const Slideshow & source);

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

The class is a Slideshow class, which stores information about a series of slides (of type `Image`), and their durations. The `slides` structure is a dynamically allocated array of `Image` pointers. `durations` is a dynamically allocated array of integers, representing how long each slide is to be displayed. Each array has `slidecount` elements and `slidecount` is greater than zero.

In this question you will write the copy constructor for the `Slideshow` class that you would include in the `slideshow.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` class has an appropriately defined "Big Three" (destructor, copy constructor, and assignment operator). You may not specify, nor may you assume the existence of, any particular helper functions in your solution.

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. You may use the template on the following page for your response, or you may write free-form on the back of this page or the next.

problem 2 continued...

Solution:

```
Slideshow::Slideshow(const Slideshow & origval) {

      slidecount = origval.slidecount;        //define  slidecount
      slides = new Image *[slidecount];    // allocate space for  slides
      durations = new int[slidecount];       // allocate space for durations
      for (int j = 0; j < slidecount; j++) {      // loop to fill arrays
         durations[j] = origval.durations[j];
         if (origval.slides[j] != NULL)
            slides[j] = new Image(*(origval.slides[j]));  //copy images
         else
            slides[j] = NULL;
      }
}
```

Grading scheme:

- **1 point for correct function signature**
- **1 point for copying slidecount**
- **1 point for correctly allocating slides**
- **1 point for correctly allocating durations**
- **2 points for correctly copying contents of durations**
- **8 points for correctly copying contents of slides:**
  - **5 points for correctly copying non-NULL contents of slides including making copies of the Image objects**
  - **3 points for properly handling NULL pointers in slides**
- **6 points for comments:**
  - **1 point for saying you are allocating new memory**
  - **1 point for commenting that you handling NULL pointers in slides**
  - **2 points for saying you are making a deep copy (i.e. copying objects rather than pointers)**
  - **2 points for writing comments indicating an understanding copy constructors**

3. **[Iterators – 20 points].**

Consider a `BackPack` class object and two iterators which we have declared in `main()` using the following statements:

```
BackPack<int> bp;
BackPack<int>::iterator it1;
BackPack<int>::iterator it2;
//  Some code which inserts an ODD number ( > 3) of integers into the BackPack
//  Some code you write here!!
```

Your task is to use the iterators and a single additional integer variable `sum3` to compute the sum of the middle three integers in the BackPack. That is, if the BackPack had values:

```
< 3 2 8 4 5 9 1 6 0 2 4 9 1 >
```

your code would compute `9 + 1 + 6 = 16`.

You may assume that the `BackPack` class has member functions `begin()` and `end()`, each of which returns an iterator to the appropriate element of the `BackPack`. That is, `begin()` returns an iterator indicating the first element of the `BackPack` and `end()` returns an iterator indicating the element *past* the last element of the `BackPack`. You may also assume that the operators `*`, `++`, `--`, `==`, and `!=` are overloaded for `BackPack` iterators. You may ONLY use variables `it1`, `it2`, and `sum3` in your code. Do not declare any additional variables, and do not assume that the `BackPack` class provides any other public member functions.

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. You may use the template on the following page for your response, or you may write free-form on the back of this page or the next.

problem 3 continued... **Solution:**

```
it1 = myVect.begin();
it2 = myVect.end();
it2--;  // it2 now points at the last element in the Vector

// N.B.: since the size of the Vector is odd, this loop will eventually terminate
while (it1 != it2) {
   it1++;
   it2--;
}

// at this point, it1 and it2 both point to the middle element in the Vector
sum3 = *it1;
it1--;
it2++;
sum3 += *it1 + *it2;
```

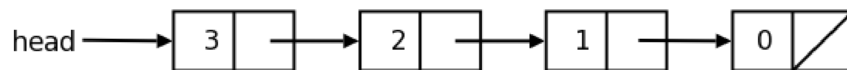
Grading scheme:

- **6 points for making the appropriate comments**
  - **2 points for comments about setting the start and end of the vector**
  - **2 points for comment about calculating the middle element**
  - **2 points for comment about calculating the sum**
- **1.5 points for setting the iterators to the beginning and end of the list.**
- **4 points for identifying the middle elements or the list**
- **1.5 points for properly accessing the value of iterator**
- **4 points for calculating the sum of three middle elements without using any other variables.**
- **2 points for the appropriate uses of the overloaded operators.**
- **1 point for no syntax errors in the code.**

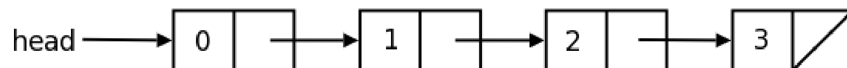
4. **[Linked List Reversal – 20 points].** Consider the following partial class definition:

```
template<class T>
class LinkedList {
private:
   class listNode{
    public:
       listNode():next(NULL){}
       T data;
       listNode * next;
    };
   listNode * head;
public:
   // constructors, destructor, and other member functions
   ...
    void reverse();
};
```

(a) (10 points) This class is a simple singly-linked list class. You are to write an iterative `reverse()` function that reverses a list by rearranging `next` pointers and changing the `head` pointer if necessary. You should not need to call any outside functions to write the code for `reverse()`. To illustrate the action reverse will perform, the following is a `LinkedList<int>` object before the call to `reverse()`:



The following is that same `LinkedList<int>` object after the call to `reverse()`:



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. You may use the template on the following page for your response, or you may write free-form on the back of this page or the next.

Solution

```
void LinkedList::reverse()
{
  listNode * cur = head;     // Traverse list with 2 pointers pointomg
  listNode * prev = NULL;    //   to the current and previous list nodes

  while (cur !=NULL)         // A node still needs it's pointer reversed
  {
    listNode * temp = cur->next; // Save a pointer to the next node

    cur->next = prev;            // Reverse the current node's ''next''
                                 //   to point to the previous node

    prev = cur;                  // Move the ''prev'' pointer to
                                 //   point to the current node

    cur = temp;                  // Move ''cur'' to the next node
                                 //  in the list
  }                    // End while loop
  head = prev;         // ''prev'' points to the end of the list
}                      //   Complete the reversal by setting ''head''
                       //     to point to the end of the list
```

Grading scheme:

- **3 points for good comments**
- **2 points for setting head pointer correctly**
- **5 points for working reversal code even if $\mathbf{O(n^2)}$**

(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.

The running time of reverse() on an n node list would be O(n). The code traverses the length of the list a single time thereby visiting n nodes and does O(1) work on every listNode it visits.

Grading scheme:

- **3 points for correct running time**
- **2 points for a good explanation**

(c) (5 points) If the LinkedList class were implemented as a doubly-linked list instead of a singly-linked list, could the running time of `reverse()` be improved? Why or why not?

No. The “previous” and “next” pointers of each node in doubly-linked list need to be swapped, so every listNode would still have to be visited.

Grading scheme:

- **3 points for correct answer (e.g. ”no” for an O(n) algorithm, ”yes” for an $O(n^2)$ algorithm)**
- **2 points for a good explanation**

5. **[Stack and Queue Interfaces – 20 points].**

   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 `Rev0` that takes one argument: a reference to a `Queue`. The function should reverse the order of any numbers appearing between a pair of consecutive `0`'s, where the pairs can't overlap (so we reverse the elements between the first `0` and the second one, leave alone the elements between the second and third, reverse the ones between the third and fourth, and so on). If there are an odd number of `0`'s in the given queue, the elements following the last `0` should also be reversed.

   For example, given the following queue,

```
front                rear
 1 0 5 3 0 3 1 0 2 6 5
```

   we get the following arrangement:

```
front                rear
 1 0 3 5 0 3 1 0 5 6 2
```

   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. You may use the template on the following page for your response, or you may write free-form on the back of this page or the next.

Solution:

```
void Rev0(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.

   temp2 = 0;  //  flag to indicate parity of # of 0s.

   while (!queue.isEmpty()) {     // process each queue value

      temp1 = queue.dequeue();  // grab front of queue
      if ((temp1==0) && (temp2==0)) {  // see a 0 and we weren't reversing
         temp2 = 1;                          // start reversing next character
         q.enqueue(temp1);          // put the 0 on queue
      }
      else if ((temp1 == 0) && (temp2 == 1)) { // see a 0 and we were reversing
         temp2 = 0;                         // flip reversal flag
         while (!s.isEmpty()) { // complete the reversal by
            q.enqueue(s.pop()); //putting stack values on queue
         }
         q.enqueue(temp1);  // put the 0 on the queue
      }
      else if ((temp1 != 0) && (temp2 == 0))  // not reversing, just enqueue
         q.enqueue(temp);
      else if ((temp1 != 0) && (temp2== 1))  // reversing, push to reverse
         s.push(temp);
   }
   while (!s.isEmpty()) {   // empty s into q if we ran out of elements
     q.enqueue(s.pop());    //    in queue while pushing to s
   }
   queue = q;  // relies on operator= in Queue class.
}
```

Grading scheme:

- **6 points for comments**
- **14 points for correct code:**
  - **1 pt - main while loop**
  - **1 pt - check if element==0**
  - **1 pt -enqueue nonzero elts until first 0**
  - **1 pt -enqueue first 0**
  - **1 pt -push elts between zeros**

- 1 pt -pop elts and enqueue them when second 0 is reached
- 1 pt -enqueue second 0 (after popped items)
- 1 pt -after big while loop terminated, pop any items left in s and enqueue them
- 1 pt -make sure final result is in the Queue "queue"

If using boolean flag version

- 1 pt - initialize flag
- 1 pt - change flag when start reversing
- 1 pt - change flag when stop reversing
- 2pts - if statements all correct (either using or nested)

If using nested while loops (5 pts):

- 3 pts - loops have correct guards, and handle edge cases correctly
- 2 pts - if statements all correct

Miscellaneous deductions:

- -1 pt for changing variable names, unless such change is indicated on the code segment we gave
- -1-2 pts for not storing the result of dequeue somewhere when using it to check for zeros
- -2 pts for improper use of push/pop/enqueue/dequeue
- -1 pt each if any dequeue is done where the queue involved could possibly be empty

(scratch paper, page 1)

(scratch paper, page 2)

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

      void push(char e);
      char pop();

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

   private:
      // we're not telling
};

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

      void enqueue(char e);
      char dequeue();

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

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