University of Illinois at Urbana-Champaign
Department of Computer Science

First Examination - RUBRIC

CS 225 Data Structures and Software Principles
Spring 2011
7-9p, Tuesday, February 22

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

- **Labs this week are officially cancelled.** We are loading up the calendar with lab hours, so if you are still working on MP3 you will be able to find course staff to help you.
- 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.
- We will be grading your code by first reading your comments to see if your plan is good, and then reading the code to make sure it does exactly what the comments promise. In general, complete and accurate comments will be worth approximately 30% of the points on any coding problem.
- 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 statements, and assume the standard `iostream` library has been included:

```
int v;
int * w;
v = 10;
*w = v;
cout << *w << endl;
```

What is the result of executing these statements?

(a) 10 is sent to standard out.

(b) The memory address of `w` is sent to standard out.

(c) This code does not compile.

(d) This code results in a runtime error.

(e) None of these options is correct.

MC2 (2.5pts)

Consider the following statements, and assume the standard `iostream` and `string` libraries have been included:

```
string * b = new string("NULL");
cout << *b << endl;
delete b;
```

What is the result of executing these statements?

(a) `NULL` is sent to standard out.

(b) The memory address of `b` is sent to standard out.

(c) This code does not compile.

(d) This code results in a runtime error for dereferencing a `NULL` pointer.

(e) None of these options is correct.

MC3 (2.5pts)

What is the output of the following sequence of C++ statements? (The sphere class interface is included at end of the exam.)

```
sphere * a, * b;

a = new sphere(1.0);
b = a;
b->setRadius(2.0);
delete b;

a->setRadius(4.0);
sphere * c = new sphere(5.0);
b = new sphere(3.0);

cout << a->getRadius() << endl;
```

(a) 4.0

(b) 3.0

(c) An unpredictable runtime memory error.

(d) A compiler error on line `a->setRadius(4.0);`.

(e) None of these is the correct choice.

MC4 (2.5pts)

Consider this definition of the function `ugly`:

```
int & ugly(int x) {

   return x;
}
```

Which of the following statements is true?

(a) This code is ugly because it returns a value parameter by reference.

(b) This code is ugly because there is a type mismatch between the return value and the return type.

(c) This code is ugly because the parameter is not `const int x`.

(d) This code is ugly for more than one of these reasons.

(e) This code is not ugly.

MC5 (2.5pts)

Consider the following statements, and assume the standard `iostream` library has been included:

```
void funwon(int x) { x = 1;}
void funtoo(int * x) { *x = *x + 2; }
void funfor(int & x) { x = x + 4; }

int main() {

    int x = 0;

   funwon(x);
   funtoo(&x);
   funfor(x);

    cout << x << endl;

    return 0;
}
```

What is the result of executing these statements?

(a) 3 is sent to standard out.

(b) 5 is sent to standard out.

(c) 6 is sent to standard out.

(d) This code does not compile because of a type mismatch in `funfor`.

(e) None of these options is correct.

MC6 (2.5pts)

Using the `MyPair` class defined in lecture and the standard `string` class which of the following correctly declares a `MyPair` variable whose name is `boots` and whose parameterized type is a dynamic array of `string`s?

(a) `MyPair<string> * boots;`

(b) `MyPair<string *> boots;`

(c) `MyPair * boots = new string[size]`

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

(e) None of (a), (b), (c), are correct.

MC7 (2.5pts)

Suppose class `pictureRep` contains exactly one pure virtual function: the overloaded parentheses operator, `int operator()(int i, int j)`. Also suppose that class `hardBMP` is a public `pictureRep` that implements `operator()`.

Which of the following C++ statements will certainly result in a compiler error?

(a) `pictureRep * a = new hardBMP;`

(b) `hardBMP * a = new hardBMP;`

(c)
```
pictureRep * a = new hardBMP;
hardBMP * b;
b=a;
```

(d) Exactly two of these will result in a compiler error.

(e) None of these will result in a compiler error.

MC8 (2.5pts)

Which of the following concepts is **NOT** a fundamental characteristic of Object Oriented Programming?

(a) encapsulation

(b) inheritance

(c) memory management

(d) polymorphism

(e) All of these concepts are characteristics of Object Oriented Programming.

2. **[MP2ish – 20 points].** Consider the following partial class definition:

```
class Book{
private:
    string * text;
    int length;
public:
    Book(const char * fileName);  // places each word from a file into
                          // sequential locations in the text array.
                          // Sets length to be the number of words in the text.
    ~Book();
    // Member functions that allow access to the contents of the text array.
    // Declaration of the operator== function for this problem.
}

class Library
{
  private:
    Book ** collection;
    int collectionSize;

    // some helper functions

  public:
    // constructors and destructor
    Library(int num); // constructor for a Library with room for num books

    //  operator=   declaration

    // lots of other public member functions not relevant to this problem
};
```

A `Book` is implemented as a dynamically allocated array of `string`s in a variable called `text`. The `Book` constructor simply opens a given text file and places each word and punctuation mark into sequential positions in the `text` array. (White space is discarded upon reading the input file.)

A `Library` is implemented as a collection of books. The `collection` structure is a dynamically allocated array whose entries are `Book` pointers. The array `collection` has `collectionSize` elements. A pointer to a `Book` is added to the `collection` structure in the cell whose index corresponds to the location of the book in the `Library`.

You can assume the `string` class has been included and scoped, and that `Book`s always contain at least one word.

You may assume that all pointers are valid. That is, they are either `NULL` or they point to an object of the specified type. In particular, the `Library(int num)` constructor builds a dynamic array of length `num`, whose elements are all `NULL`, and sets `collectionSize` to `num`.

In this question you will help us implement some of the member functions for the `Library` and `Book` classes.

You will write your answers on the following pages. To grade the coding portions of 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. Comments will be worth up to 1/3 of the total points for any part of the problem. Adding comments to our code skeletons can get you partial credit, but it's not required.

(a) (12 points) In this part of the problem, you will write the code for an overloaded boolean comparison operator, `operator==()` so that two `Book`s can be compared. Two `Book`s are the same if their text arrays contain identical strings in each position. We have given you a skeleton for your code. If you need to add a couple lines, just do it neatly between our lines. If your approach is significantly different than ours, please write your response on the back of the previous page.

Solution:

```
// comments were worth 1 point
bool Book::operator==(const Book &  rhs) // 3 points
{

   bool  returnValue = true;  // 1 point


   if (length == rhs.length) // 2 points


      for(int i = 0;  i < length;  i++) // 1 point
      {

            if (text[i] != rhs.text[i]) // 2 points
               returnValue = false;  // 1 point
      }

   else
      returnValue = false; // 1 point


   return returnValue;
}
```

(b) (6 points) Write a private helper function called `copy(const Library & orig)` for the `Library` class, as it would appear in `library.cpp`. `copy` should make the current object's member variables contain exactly the same data as that of the parameter `orig`, but its memory should be completely independent.

Solution:

```
void Library::copy(const Library & orig) {
    collectionSize = orig.collectionSize;    // 1 point
    collection = new Book *[collectionSize];    // 2 points
    for (int i=0; i < collectionSize; i++) {
        if (orig.collection[i]!= NULL) { // 1 point
            collection[i] = new Book;
            *collection[i] = *(orig.collection[i]);    // 1 points
        }
        else collection[i] = NULL;    // 1 point
    }
}
```

(c) (2 points) List two member functions of the `Library` class that would employ the `copy` helper function. Give the member declaration for each as it would appear in `library.h`.

Solution:

Copy constructor: `Library(const Library & rhs);`

Assignment operator: `Library & operator=(const Library & rhs);`

3. **[MP3ish – 20 points].**

The following code is a partial definition of a doubly linked list implementation of the `List` class that you used for MP3. Note in particular that it does not contain sentinels, but it *does* have head and tail pointers.

```
template <typename Etype>
class List   {
public:

   // appendList
   //    - parameters : endList - a reference to a previously allocated
   //                     List object; this function changes endList
   //                     to an empty list.
   //    - appends endList after the end of the current list, and
   //      empties endList.
   void appendList(List<Etype> & endList);

   // a bunch of other List class functions

private:

   class ListNode
   {
   public:
      // ListNode constructor
      //    - initializes element to default Etype, and pointers to NULL
      ListNode();

      // ListNode constructor
      //    - parameters : value - the value to store in the element field
      //    - initializes node to hold value and NULL pointers
      ListNode(Etype const & value);

      // Maybe some other functions here.

      ListNode* next; // pointer to next node in list
      ListNode* prev; // pointer to prior node in list
      Etype element;   // holds element of node
   };

   ListNode* head;    // points to first node of list
   ListNode* tail;    // points to last node of list
   ListNode* current; // an extra pointer for your convenience
   int size;
};
```

(a) (10 points) Please implement the `appendList` function (as you did for MP3). This function simply appends the input list to the end of the current list, and leaves the input list (called `endList`) empty. You may assume that `endList` is not the same `List` as the current list. The restrictions from MP3 apply: you are not allowed to access the `element` field of any `ListNode`, nor can you create new `ListNode`s. You may, however, declare `ListNode` pointer variables if you think they would be useful to you. You may also use the private member `current`. We make no assumptions about its value after a call to `appendList`.

Be sure to make your code very neat and easy to read. Draw pictures where it will help us, and comment profusely.

Solution:

```
template <typename Etype>
void List<Etype>::appendList(List<Etype> & endList) {

// Sorry, no solution here.  You should have written it for MP3.  :)
// grading rubric:
// 1: handles endList empty case
// 1: handles *this empty case
// 2: appends in O(1) time
// 2: Clears endList where necessary
// 1: No memory leaks
// 1: No memory/logical errors (i.e. off by one)
// 2: Comments

}
```

(b) (5 points) Describe the running time of your implementation of `appendList` if the list contains n items.

Solution: The answer *should* be $O(1)$ since all operations are conditionals or assignment statements.

Rubric: Correct analysis of *your* appendList == 4 points; Brief justification == 1 point.

(c) (2 points) Explain why the `endList` parameter is passed by reference to `appendList`.

Solution: endList is passed by reference because 1) we wish to change its value (1 point) and 2) we don't want to incur the time it takes to do a copy (1 point).

(d) (3 points) Does the `List` class require a destructor? Briefly justify your answer.

SOLUTION: YES (1 point) ! We have implemented the `List` class as a sequence of dynamically allocated list nodes, each of which is created when data is inserted into the list. If a variable of type `List` goes out of scope, for example, the memory associated with the variable should be deallocated when the system calls the destructor associated with the class (2 points).

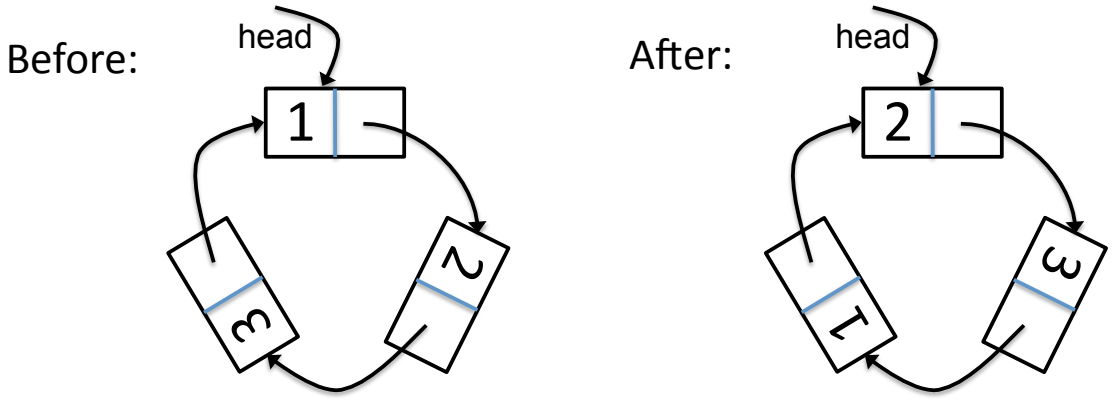
4. **[Links – 20 points].**

In each of the problem segments below, we have given you "before and after" models of linked lists. Your task is to transform the "before" into the "after" using simple pointer manipulations on the list nodes. For the first three parts you will use the `listNode` class below, and for the last two, the list nodes also contain a previous pointer called `listNode * prev`. Your solutions should follow these guidelines:

- You may declare `listNode` pointer variables to use in navigating the lists. When you are finished with them, just set them to `NULL`.
- You must never refer to the `data` member of the `listNode` class, except indirectly using the `listNode` constructor.
- You should only allocate memory if you are inserting new data into the structure.
- You don't need to try to write general solutions. Just write the statements that accomplish the "after" list, starting with the "before" list. (Almost every statement you write should be an assignment statement.)
- Any variables listed in the picture can be used in your solution.

```
struct listNode {
  int data;
  listNode * next;
  listNode(int e): data(e), next(NULL) {}
};
```

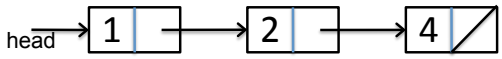
(a) (2 points)



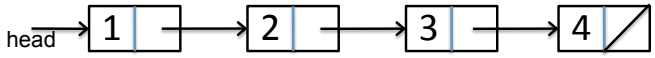
Solution: `head = head->next`

(b) (5 points)

Before:



After:

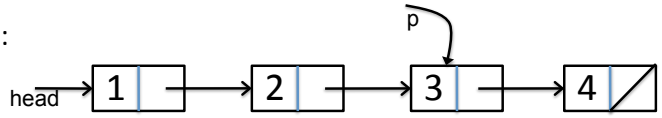


Solution:

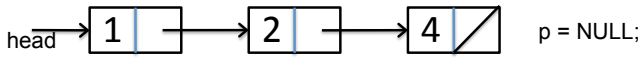
```
listNode *c = new ListNode(3);  // (1 point)
c->next=head->next->next;
head->next->next=c;  // (3 points)
c=NULL;  // (0.5 points)
```

(c) (5 points)

Before:



After:

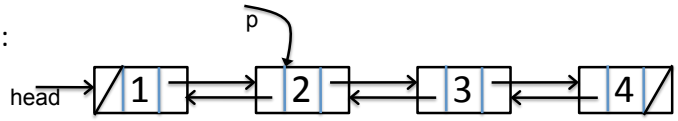


Solution:

```
head->next->next=p->next;  // (3.5 points)
delete p;  // (1 point)
p=NULL;  // (0.5 point)
```

(d) (3 points)

Before:



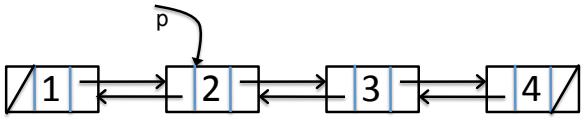
After:

Solution:

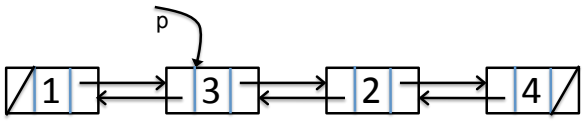
```
head -> next = p -> next;  // -1 point
head -> next -> prev = head; // -1 point
delete p; // -1 point
p = NULL; // -0.5 point
```

(e) (5 points)

Before:



After:



Solution:

```
ListNode * q = p -> next;
q -> prev = p -> prev;
p -> next = q -> next;
q -> next -> prev = p;
p -> prev -> next = q;
p -> prev = q;
q -> next = p;
// -1 point for each incorrect or missing pointer
p = q; // -1 point
q = NULL; // -0.5 point
// -0.5 point for using non-declared head pointer
```

5. **[Miscellaneous – 20 points].**

(a) (5 points) Complete the following (silly) function implementation so that no memory is leaked.

```
void weird() {
    int pearl = 7;
    int * diamond = &pearl;
    int * boa = new int[4];
    int * coffee = boa;
    int ** sea = new int * [3];
    sea[0] = &coffee[3];
    sea[1] = new int;
    sea[2] = sea[0];

    // add code to free each piece of dynamically
    // allocated memory exactly once

}
```

Solution:

```
delete sea[1];
delete [] sea;
delete [] boa;
```

- **full pts for deleting** $coffee$ **instead of** boa**;**
- **full pts for deleting** boa **before** sea**;**
- **-1 pts for deleting** $diamond$
- **-1 pts for deleting both** $coffee$ **and** boa
- **-2 pts for deleting** $sea[0]$ **and** $sea[2]$

(b) (2 points) Briefly describe the two instances when the destructor is called.

Solution:

- When the function ends or the object is out of scope
- When we explicitly use *delete* to destroy some pointers

Other instances will get point as long as it is valid.

(c) (6 points) For this problem, you will be a code critic. Please answer the questions below about the following `sphere` class member function. `setPictureGetRadius` is supposed to change the member variable `thePicture` and return the current value of member variable `theRadius`. The adapted `sphere` class appears at the end of the exam.

```
double  sphere::setPictureGetRadius(BMP & newPicture) const {

   thePicture = newPicture;
   return theRadius;

}
```

i. (2 points) Comment on the type specification in the parameter list. Is there a better way to specify that parameter? Why is it pass-by-reference?
Solution:

- Add a *const* before *newPicture* will be ideal in the sense that we don't accidently want to change *newPicture*.
- Pass-by-reference will be the best way since it doesn't make a copy of *newPicture*, which could be costly.

Since the first part is kind of overlapped with the following subproblem, any answer correctly points out the second item will get full points.

ii. (2 points) Comment on the `const` keyword in the function prototype.
Solution:
This *const* means that the function will not change any members of **sphere** class, which contradicts with our purpose. So it should be removed or moved before *newPicture*.

iii. (2 points) What does the assignment `thePicture = newPicture;` assume about the `BMP` class? Is this a reasonable assumption?
Solution:
By writing this assignment we assume that in the **BMP** class we have overrided the assignment operator "=". This is a reasonable assumption given that the assignment operator is part of Big three.

(d) (4 points) Scrutinize the following code. In this problem we are going to ask you to predict and to change its output. (Please assume that the standard `iostream` is included.)

```
class yell{
public:
    void display() {
        cout << "yell" << endl;
    }
};

class w00t:public yell {
public:
    void display() {
        cout << "w00t" << endl;
    }
};

int main() {

    yell * a = new w00t;
    a->display();

    return 0;
}
```

i. (2 points) What is output when the code above is compiled and run?
Solution:
"yell" (2 points), "w00t" (1 point), anything else (0 points).

ii. (2 points) How can you change the code so that the other expression is printed?
Solution:
"add virtual to yell::display" == 2 points
"add virtual to anything else" == 1 point

(e) (1 points) On a scale of 1 to 5, how much are you learning in the class? (1 is not much, 5 is a ton)

(f) (1 points) On a scale of 1 to 5, how is the pace of the course so far? (1 is too fast, 5 is too slow)

(g) (1 points) On a scale of 1 to 5, rate your general satisfaction with the course. (1 is profoundly dissatisfied, 5 is happy) If your response is not 4 or 5, please suggest a specific improvement we can make.

```
class sphere {
public:
   sphere();
   sphere(double r);

   double getDiameter() const;
   double getRadius() const;
   void setRadius(double r);
   void setPictureGetRadius(BMP & newPicture);

private:
   double theRadius;
   BMP thePicture;
};
```

scratch paper 1

scratch paper 2