University of Illinois at Urbana-Champaign
Department of Computer Science

First Examination RUBRIC

CS 225 Data Structures and Software Principles
Fall 2012
Monday, October 1, 7-9p

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

- This is a **closed book** and **closed notes** exam. No electronic aids are allowed.
- You should have 5 problems total on 20 pages. The last sheet is scratch paper; you may detach it while taking the exam, but must turn it in with the exam when you leave.
- The points assigned to each problem are a *rough* estimate of the time it should take you to solve the problem. Use your time wisely.
- 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.
- Please put your name at the top of each page.

Problem	Points	Score	Grader
1	25		
2	20		
3	15		
4	35		
5	5		
Total	100		

1. **[Pointers, Parameters, and Miscellany – 25 points].**

MC1 (2.5pts)

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

```
int * s;
int t = 37;
*s = t;
cout << *s << endl;
```

What is the result of executing these statements?

(a) 37 is sent to standard out.

(b) The memory address of `s` 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 class definitions:

```
class Sport{
public:
        virtual int loser();
private:
        int score;
};
class Volleyball: public Sport {
public:
        int winner();
};
```

Where could the assignment `score = 20;` appear for the private variable `score`?

(a) Both `winner()` and `loser()` can make the assignment.

(b) `winner()` can make the assignment, but `loser()` cannot.

(c) `loser()` can make the assignment, but `winner()` cannot.

(d) Neither `winner()` nor `loser()` can make the assignment.

(e) The answer to this question cannot be determined from the given code.

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 insidious runtime memory error.

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

(e) A compiler error on line `b = new sphere(3.0);`

MC4 (2.5pts)

Consider this definition of the function `horrible`:

```
int * horrible(int & y) {
   int x = y;
   y = 15;
   return &x;
}
```

Which of the following statements is true?

(a) This code is horrible because it returns the memory address of a local variable.

(b) This code is horrible because there is at least one type mismatch.

(c) This code is horrible because the parameter is not `const int & y`.

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

(e) This code is not horrible at all, despite its name.

MC5 (2.5pts)

```
void fac2(int x) { x = 2*x; }
void fac3(int * x) { *x = 3 * (*x); }
void fac5(int & x) { x = 5*x; }

int main() {
    int z = 1;
    fac2(z);
    fac3(&z);
    fac5(z);
    cout << z << endl;
    return 0;  }
```

What is the result of compiling and executing this code (assume `iostream` is included)?

(a) 6 is sent to standard out.

(b) 10 is sent to standard out.

(c) 15 is sent to standard out.

(d) 30 is sent to standard out.

(e) None of these options is correct.

MC6 (2.5pts)

```
class Bear {
  public:    Bear() { cout << "Growl" << endl; }
            ~Bear() { cout << "Stomp stomp stomp" << endl; }
};
int main()  {
    Bear beary;
    cout << "Run!" << endl;
    return 0;  }
```

What is the result of compiling and executing this code (assume `iostream` is included)?

(a) Run!

(b) Growl
Run!

(c) Growl
Run!
Stomp stomp stomp

(d) Run!
Stomp stomp stomp

(e) Garbage is printed to the terminal.

MC7 (2.5pts)

Consider the following class definition:

```
class Pumpkin {
  public:
    Pumpkin(const Pumpkin & other);
    ~Pumpkin();
    // more public member functions
  private:
    double radius;
    // more private member variables
};
```

Which of the following functions must also be implemented for the Pumpkin class for it to function correctly?

(a) no parameter constructor

(b) operator=

(c) operator()

(d) setRadius

(e) operator delete

MC8 (2.5pts)

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

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

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

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

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

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

MC9 (2.5pts)

Suppose class `modPNG` contains exactly one pure virtual function whose name is `print`. Also suppose that class `flipImage` is a public `modPNG` that implements `print`.

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

(a) `modPNG a;`

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

(c)
```
modPNG * a;
flipImage * b;
a=b;
```

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

(e) All three of (a) thru (c) will result in a compiler error.

MC10 (2.5pts)

Consider this slight modification of a snippet of code from a recent lecture (and assume all STL classes are available):

```
struct animal {
   string name;
   string food;
   animal(string n="blob", string f="you"): name(n), food(f) { }
};

int main() {
   animal g("giraffe","leaves"), b("bear");
   list<animal *> zoo;

   zoo.push_back(&g); zoo.push_back(&b); //STL list insertAtEnd

   for(list<animal *>::iterator it = zoo.begin(); it != zoo.end(); it++)
      cout << (*it)->name << "  " << (*it)->food << endl;
}
```

Which of the following describes the result when this function is compiled and run?

(a) There is a compiler error because there is no constructor matching the one called for variable `b`.

(b) There is a compiler error because of a type mismatch in the parameterized type for the list class.

(c) There is a runtime error because the iterator is dereferenced twice.

(d) The code sends "giraffe leaves" to standard out.

(e) The code sends "giraffe leaves" and "bear you" to standard out.

2. **[MP2ish – 20 points].**

A TARDIS (Time and Relative Dimension in Space) is a time traveling spaceship used by the Time Lords of Gallifrey. It transports a set of people through time and space to their destination. Unfortunately, because the TARDIS moves through time, it can't rely on nature to keep its occupants at the correct age. It would be a shame to arrive in London 5120 with all of the occupants having aged three thousand years. To avoid that, a TARDIS keeps track of the age of each of its occupants.

A `TARDIS` is implemented as a dynamically allocated array of pointers to `string`s, representing the names of occupants, and a dynamically allocated array of `double`s, representing the ages of occupants (in Earth years).

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

class TARDIS{
private:
    string ** occupants;
    double * ages;
    int maxOccupants;
    int populationSize;
    // Lots of other member functions

public:
    TARDIS(int capacity); // creates a new empty TARDIS object:
                          // sets maxOccupants equal to capacity,
                          // allocates arrays of size capacity for occupants/ages,
                          // and sets all uninitialized pointers to NULL

    // Declaration of the operator+= function for this problem. (You will write this)
    void letIn(const string & name, double age); //adds a person to the TARDIS

    // Big Three declarations
    // Lots of other member functions
};
```

If there are fewer than `maxOccupants` people inside the TARDIS, some of the occupants' names may be NULL, in which case their corresponding ages are garbage. You may assume that all pointers are valid. That is, they are either `NULL` or they point to an object of the specified type. Also, please ensure that the inhabitants of the TARDIS are placed in the first `populationSize` positions in each of the arrays.

In this question you will implement some of the member functions for the `TARDIS` class. **Your comments will be graded**, and will be worth up to 1/3 of the total points for any part of the problem. They should be coherent, useful, and reflective of your approach to the problem.

(a) (4 points) As a TARDIS travels through space and time, it keeps its occupants ages up-to-date using the overloaded compound addition operator, `operator+=`. This operator works by taking a single double parameter, the amount by which to increment each occupant's age.

For example, the following code updates the occupants' ages during a 0.025 year trip:

```
TARDIS theDoctorsTardis(4); // setup
theDoctorsTardis.letIn("Amy Pond", 19.2);
theDoctorsTardis.letIn("River Song", 40.6);
theDoctorsTardis += 0.025; // you will be implementing operator+=
```

After this code executes, "Amy Pond" should have an age of 19.225, while "River Song" will have an age of 40.625.

Write the `operator+=` function for the `TARDIS` class, as it would appear in `tardis.cpp`, so that the above code will compile and work as specified.

Solution:

```
void TARDIS::operator+=(double amount)
{
     for (int i = 0; i < populationSize; i++)
          ages[i] += amount;
}
```

Rubric:
1 point for signature, 3 points for body.
I won't take off points if they look through "maxOccupants" instead of "populationSize" (still should work)
-1 points for bugs/compiler errors

(b) (6 points) Write a private helper function called `copy(const TARDIS & orig)` for the `TARDIS` class, as it would appear in `tardis.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 TARDIS::copy(const TARDIS & orig)
{
     maxOccupants = orig.maxOccupants;
     populationSize = orig.populationSize;
     ages = new double[maxOccupants];
     occupants = new string*[maxOccupants];

     for (int i = 0; i < populationSize; i++)
     {
          ages[i] = orig.ages[i];
          occupants[i] = new string(*orig.occupants[i]);
     }
     for (int i = populationSize; i < maxOccupants; i++)
     {
          occupants[i] = NULL;
     }
}
```

Rubric:
1 point for correctly copying the integers (maxOccupants and populationSize)
2 points for realizing you have to allocate new memory
1 point for copying ages correctly from orig.ages
2 points for copying occupants correctly from orig.occupants
-1 points for bugs/compiler errors

(c) (2 points) List two member functions of the `TARDIS` class that would employ the `copy` helper function.

Solution: assignment operator and copy constructor (1 point each)

(d) (6 points) The `letIn` function puts a person into the TARDIS object. It takes a string name and a double age, and puts those into the first available slot in the occupants array. Since a TARDIS is bigger on the inside than it is on the outside, new occupants should always be able to enter a TARDIS. Therefore, if the array is out of space, it should be resized so that there is room for the new occupant. Your code should do only as much copying of data as is necessary, and no more.

```
void TARDIS::letIn(const string & name, double age)
{
     if (populationSize == maxOccupants) {
          maxOccupants *= 2; // or +1
          double *newAges = new double[maxOccupants];
          string **newOccupants = new string*[maxOccupants];
          for (int i = 0; i < populationSize; i++)
          {
               newAges[i] = ages[i];
               newOccupants[i] = occupants[i];
          }
          for (int i = populationSize; i < maxOccupants; i++)
          {
               newOccupants[i] = NULL;
          }
          delete [] ages;
          ages = newAges;
          delete [] occupants;
          occupants = newOccupants;
     }
     ages[populationSize] = age;
     occupants[populationSize] = new string(name);
     populationSize++;
}
```

4 points for resizing array:

- 1 point for doubling maxOccupants
- 1 point for allocating new memory
- 1 point for copying over values correctly
- 1 point for properly deleting memory (no memory leaks)

2 points for adding occupant

- 1 point for adding age to ages array and name to occupant array
- 1 point for incrementing populationSize

-1 points for bugs/compiler errors

(e) (2 points) As mentioned above, if the TARDIS is out of space and a new occupant is added, the arrays must be resized so that there is room for the new occupant. For one new occupant, how long does this resizing take (big-O) in terms of $n =$ `maxOccupants`? Solution: $O(n)$, points: all or nothing.

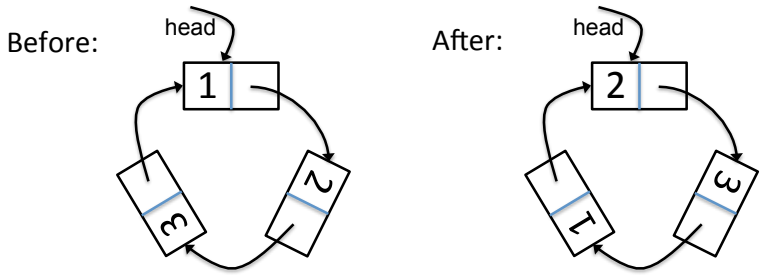
3. **[MP3ish – 15 points].**

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. Refer to the elements of the list nodes using the `listNode` class below. 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.
- You may write loops to simplify your solutions, but your answers don't need to be general... they just need to work on the given lists. (Don't worry about even/odd length, or empty lists, for example.)
- Any pointer variables named in the picture can be used in your solution.
- Only part (d) uses the `prev` pointer. Just ignore it for parts (a)-(c).

```
struct listNode {
  string data;
  listNode * next;
  listNode * prev;  //NULL except in part (d)
  listNode(string e): data(e), next(NULL), prev(NULL) {}
};
```

(a) (2 points)

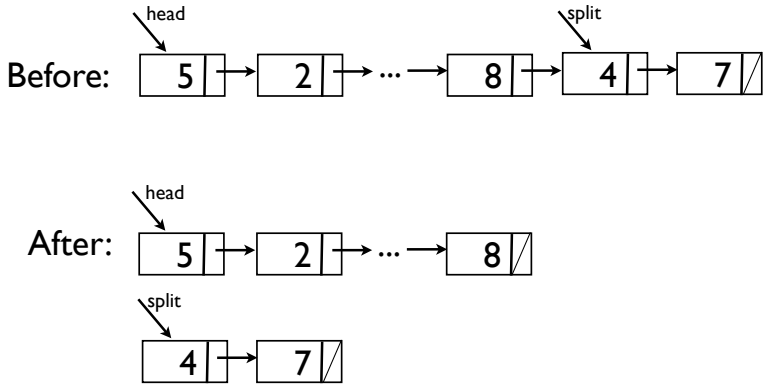


Solution:

```
head = head->next;
```

Rubric:
2 points - The exact solution given, or something similar that gets to the correct outcome
1 point - Using prev to get to the "correct" outcome
0 points - Does not reach correct outcome

(b) (3 points)

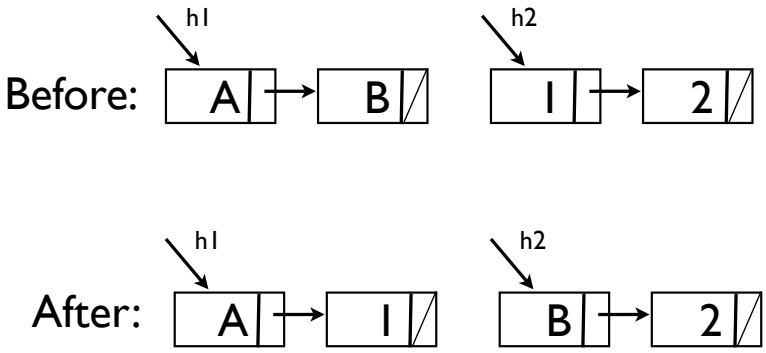


Solution:

```
listNode* temp = head;

while (temp->next != split)
  temp = temp->next;

temp->next = NULL;

temp = NULL;
```

Rubric:
3 points - The exact solution given, or something similar that gets to the correct outcome
2 points - Off-by-one type error that gets close to the solution and shows good understanding but with a small error
1 point - Using prev to get to the "correct" outcome
0 points - Not even close

(c) (5 points)

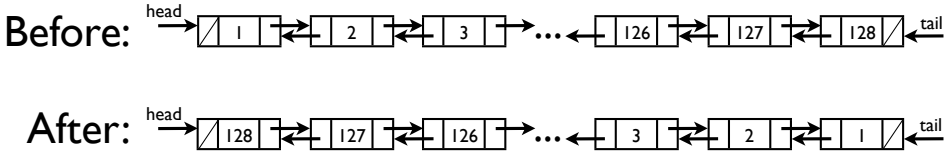


Solution:

```
listNode* temp = h1->next;
h1->next = h2;
temp->next = h2->next;
h2->next = NULL;
h2 = temp;

temp = NULL;
```

Rubric:
5 points - The exact solution given, or something similar that gets to the correct outcome
4 points - Very small error such as forgetting to unlink the lists
2 points - Getting one of the two lists correct, but not the other
1 point - Using prev to get the "correct" outcome
0 points - Not even close

(d) (5 points)



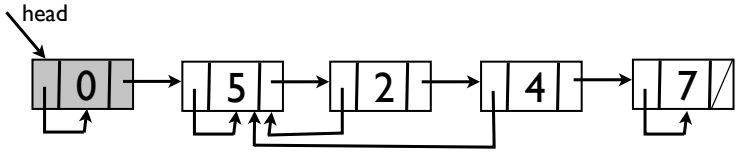
Solution:

```
listNode* current = head;
while (current != tail) {
  listNode* temp = current->next;
  current->next = current->prev;
  current->prev = temp;

  current = temp;
}

tail = head;
head = current;

current = NULL;
```

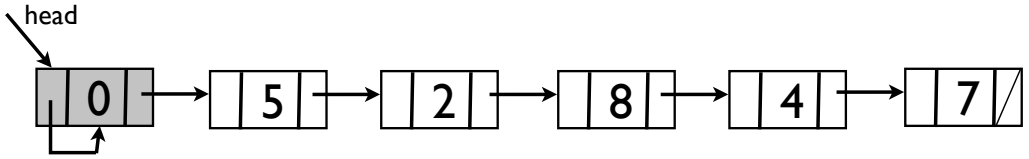
Rubric:
5 points - The exact solution given, or something similar that gets to the correct outcome
4 points - Very small error such as forgetting to switch head and tail after correctly reversing
2 points - Switching the next links properly, but forgetting to switch the prev links (or some similar sort of error).
1 point - Demonstrated an understanding of how to reverse the list, but had some sort of logic/pointer error
0 points - Not even close

4. **[MaxList – 35 points].**

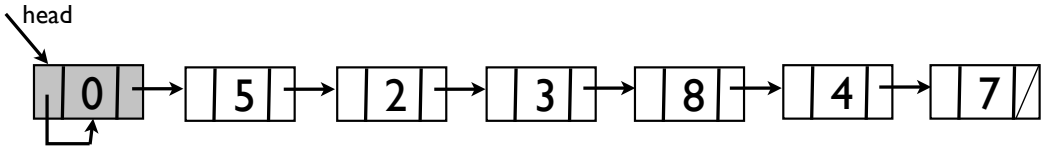
In this problem we will be implementing and investigating a variation of a linked list that we will call a `MaxList`. As illustrated in the figure below, this list has a `head` pointer and a head sentinel. In addition, each `MaxListNode` has non-negative integer `data`, a `next` pointer, and a `maxPrev` pointer that points to the maximum valued node before and including the current node. Look carefully at the picture to make sure you understand the structure. (Note: this problem has parts (a) through (l). (a)-(c) are straightforward, (d)-(f) are interdependent and challenging, and (g)-(l) are general knowledge and not deeply dependent on the previous parts.)



(a) (2 points) In this diagram, we have just invoked the function call `insertAt(3,8)` which places the value 8 in the third position in the list. Draw the accurate `maxPrev` pointers, notice which ones may change (depending on the value inserted), and which ones will certainly not change.



(b) (2 points) In this diagram, we have just invoked the function call `insertAt(3,3)` which now places the value 3 in the third position in the list. Draw the accurate `maxPrev` pointers. In this example, you should pay particularly close attention to how the `maxPrev` pointer is set for the new node.

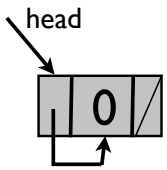


The following code is a partial definition of the `MaxList` class. (Continued on next page.)

```
class MaxList  {
public:
    MaxList();  // YOU'LL DEFINE
    // the big 3: (given) copy ctor, dtor, and op=
```

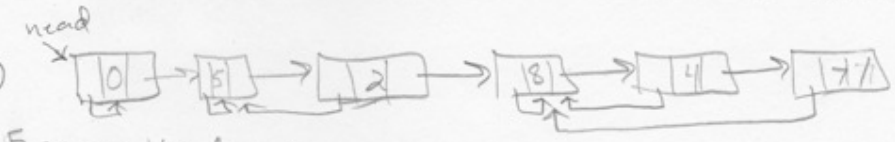
```
    int  size() const; // (given) the number of data elts in list
    void insertAt(int k, int ndata);  //  YOU'LL DEFINE
private:
    class MaxListNode  {
    public:
        MaxListNode(int ndata);
        MaxListNode * next;
        MaxListNode * maxPrev;
        int data;
    };

    MaxListNode * head;
    int length;

    //find: (given) walks k steps from curr
    MaxListNode * find(int k, MaxListNode * curr);

    //ptrToMaxData: (given) compares data in node1 and in node2 and returns
    //        the node whose data value is greatest (node2 if they're equal).
    MaxListNode * ptrToMaxData(MaxListNode * node1, MaxListNode * node2);

    void fixPtrsAfter(MaxListNode * curr);  //  YOU'LL DEFINE
    void insertAfter(MaxListNode * curr, int ndata);  //  YOU'LL DEFINE
};
```

(c) (3 points) The diagram below illustrates the state of memory after an empty `MaxList` is declared (i.e. `MaxList fancyList;`). Write the no-argument constructor for the `MaxList` class as it would appear in the maxlist.cpp file. Note that the `maxPrev` pointer is set to point to the sentinel itself, which means that the `maxPrev` pointer is set correctly for the sentinel node (whose data value is 0). The length of an empty `MaxList` is 0.



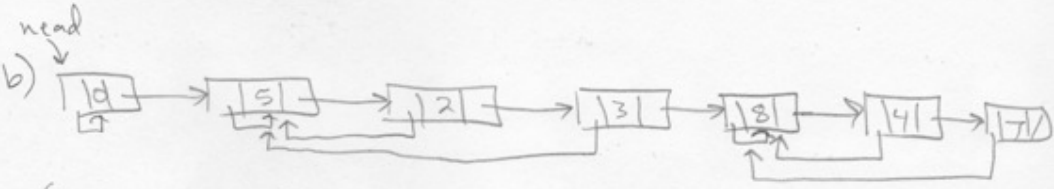

```
    ____________________ :: MaxList()
    {
```

4. a)



5 arrows to draw:
- 2 pts for all 5 correct
- 1 pt for 1-4 correct
- 0 pt for none correct

b)



6 arrows to draw:
- 2 pts for all 6 correct
- 1 pt for 1-5 correct
- 0 pt for none correct

c)

```
MaxList::MaxList()
{
  head = new MaxListNode(0);
  head->maxPrev = head;
  length = 0;
}
```

- 3 pts for correct MaxList::
- 2 pts for forgetting length or MaxList:: but otherwise correct
- 1 pt for assigning head correctly but not head->maxPrev
- 0 pt for not assigning head correctly

(d) (5 points) Please implement the `fixPtrsAfter` function. This function takes a pointer to a `MaxListNode` whose `maxPrev` pointer is set correctly, and it travels to the end of the list on valid `next` pointers, updating the `maxPrev` pointers as it goes. Assume that the input parameter is not `NULL`, that all `next` pointers are valid, and that the list is terminated by a `NULL` valued `next` pointer. You may find the `ptrToMaxData` comparison function to be useful. (As a reference, our solution has 3 lines of code and uses only the function parameters–no local or member variables. Remember that simple code is easier to grade, and everyone likes a happy grader!)
Solution:

```
void MaxList::fixPtrsAfter(MaxListNode *curr) {
  while (curr->next != NULL) {
    curr->next->maxPrev = ptrToMaxData(curr->maxPrev, curr->next);
    curr = curr->next;
  }
}
```

Rubric:
2pts for correctly setting up the loop
2pts for correct pointer manipulation
1pts for code clarity

(e) (5 points) Next, implement the `insertAfter` function. The function creates a new node with value `ndata`, and inserts it into the list *after* the node pointed to by parameter `curr` (which will not be `NULL`). This function is also responsible for making sure all `maxPrev` pointers are set appropriately. You may use `MaxList` member functions in your solution.

```
void MaxList::insertAfter(MaxListNode *curr, int ndata) {
  MaxListNode* newNode = new MaxListNode(ndata);
  newNode->next = curr->next
  curr->next = newNode;
  fixptsafter(curr);
}
```

Rubric:
1pt correctly allocating the memory
2pts for correctly inserting the value and giving it the correct max based on previous values
1pt fixing the values after the insert
1pt for code clarity

(f) (5 points) Finally, implement the public `insertAt` function so that it behaves as described in parts (a) and (b) of this problem. For simplicity, you may assume that the list is always long enough to accept a new value in position `k`. As illustrated above, the first data element occupies position 1, the second occupies position 2, etc. You may use any of the private member functions you would like. Don't forget to update the `length` data member!

```
void MaxList::insertAt(int k, int ndata) {
  MaxListNode* curr = find(k-1, head);
  insertAfter(curr, ndata);
  length++;
}
```

Rubric:

3 points for code clarity (since it really is all in the given functions)
1 pt for correct parameters
1 pt for the right idea.

(g) (2 points) Imagine an implementation of a function called `insertAtFront` that adds a new value to the front of the list. What is the running time of such a function in the worst case? (Circle the appropriate answer.)

constant time ($O(1)$) **linear time ($O(n)$)**

(h) (3 points) Briefly explain why the `find` function is in the `private:` section of the class definition.

Solution: The implementation of find is private because the client side does not know how MaxListNode is implement (it is private). [1 pt for this specifically] Up to 2 points for other good answers.

(i) (2 points) Briefly explain the role of the sentinel node in the implementation. Which functions were affected and how?

Solution: Sentinel nodes simplify going through the list in `fixPtrsAfter` because each node points to a non-NULL value. Also, inserting at the front of the list is not a special case.

(j) (2 points) Write the function signature for the copy constructor as it would appear in maxlist.cpp.

Solution: `MaxList::MaxList(const MaxList & orig)`
Rubric: -1 for no const, -1 for not by reference, -1 for other C++ errors (max of -2)

(k) (2 points) Does the `MaxList` class *require* a destructor? Briefly justify your answer.

Solution: yes(1pt), because there is dynamic memory (1pt)

(l) (2 points) Describe the purpose and effect of the keyword `const` in the declaration for the function `int size() const;`.

Solution: `const` is used because the function just returns an int, and leaves all of the member variables unchanged.

5. **[Miscellaneous – 5 points].**

(a) (5 points) For this problem, you will be a code critic. Please answer the questions below about the following `sphere` class member function. `setPicture` is supposed to change the value of member variable `thePicture`. The adapted `sphere` class appears at the end of the exam.

```
void  sphere::setPicture(PNG newPicture) {

   thePicture = newPicture;
   return;

}
```

i. (2 points) Comment on the type specification in the parameter list. Is there a better way to specify that parameter?
Solution: It would be better to pass by reference (or pointer) so a local copy of newPicture is not made. That reference should be `const` since the newPicture is not changed. (1pt for reference, 1pt for const)

ii. (3 points) What does the assignment `thePicture = newPicture;` assume about the `PNG` class? In a few words, describe the consequence if that assumption is false.
Solution: It assumes that operator= is overloaded for the PNG class. If that assumption is false, no deep copy will be made of the picture, and memory will be essentially tangled between the 2 items.

(b) (0 points) Please give us feedback about the course, entering your responses on items 11 thru 13 of the scantron form you used for your multiple choice responses:

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

(a) 1 (b) 2 (c) 3 (d) 4 (e) 5

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

(a) 1 (b) 2 (c) 3 (d) 4 (e) 5

iii. 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.

(a) 1 (b) 2 (c) 3 (d) 4 (e) 5

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

   double getDiameter() const;
   double getRadius() const;
   void setRadius(double r);
   void setPicture(PNG newPicture);

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

scratch paper