Name Mark Vismonte

CS 32
Winter 2010
Midterm Exam
February 10, 2010
David Smallberg

Problem #	Possible Points	Actual Points
1	10	10
2	15	15
3	35	35
4	15	15
5	25	22
TOTAL	100	97 ✓

6 14 — P=15

11 11

STUDENT ID #: _____

SIGNATURE: _____

OPEN BOOK, OPEN NOTES
NO ELECTRONIC DEVICES

ENJOY!

1. **[10 points]**

What is the output of the following program?

```
class Wing
{
  public:
    Wing() { cout << "W "; }
    ~Wing() { cout << "~W "; }
};

class MagnificentTail
{
  public:
    MagnificentTail() { cout << "M "; }
    ~MagnificentTail() { cout << "~M "; }
};

class Bird
{
  public:
    Bird() { cout << "B "; }
    ~Bird() { cout << "~B "; }
  private:
    Wing m_wings[2];
};

class Peacock : public Bird
{
  public:
    Peacock() { cout << "P "; }
    ~Peacock() { cout << "~P "; }
  private:
    MagnificentTail m_tail;
};

int main()
{
  Peacock p;
  cout << endl;
  cout << "====" << endl;
  Peacock* pp = &p;
  cout << "====" << endl;
}
```

W W B M P 5

= = = = 2

= = = =

~P ~M ~B ~W ~W 3

2. [15 points in all]

Here is a Robot class. Every Robot contains a pointer to the Valley it is in:

```
class Valley;

class Robot
{
  public:
    Robot(Valley* vp, int r, int c)
     : m_valley(vp), m_row(r), m_col(c)
    {}
    int row() const { return m_row; }
    int col() const { return m_col; }
  private:
    Valley* m_valley;
    int     m_row;
    int     m_col;
};
```

Because we did not declare a destructor, copy constructor, or assignment operator for the Robot class, the compiler writes those functions for us.

Here is a Valley class. Every Valley contains a collection of pointers to the dynamically allocated Robots in that Valley:

```
class Valley
{
  public:
    Valley()
     : m_nRobots(0)
    {}
    // other functions not shown
  private:
    Robot* m_robots[100];
    int    m_nRobots;
};
```

The first `m_nRobots` elements of the `m_robots` array contain pointers to dynamically allocated robots; the remaining elements have no particular value.

The users of the Valley class will need to copy Valley objects and assign one Valley object to another.

For parts a, b, and c below, you may implement additional Valley class helper functions if you like. Make no changes or additions to the Robot class.

a. [3 points]

Complete the implementation of the destructor for the Valley class:

```
Valley::~Valley()
{
 for (int i=0; i < m_nRobots; i++)
     delete  m_robots[i];
 }
```

b. [6 points]

Implement the copy constructor for the Valley class. (Be careful: Robots must believe that they're in the Valley that they're actually in.)

```
Valley::Valley(const Valley& other) : m_nRobots(other.m_nRobots)
 {
 Robot* a;
 for(int i=0; i<m_nRobots; i++)
    {
    a= other.m_robots[i];
    m_robots[i]=new Robot(this, a->row(), a->col(),);
    }
 }
```

c. [6 points]

Implement the assignment operator for the Valley class:

```
Valley& Valley::operator=(const Valley& rhs)
{
  if (this != &rhs)
  {
    for (int i=0; i<m_nRobots; i++)
      delete m_robots[i];
    m_nRobots = rhs.m_nRobots;
    for(int i=0; i<m_nRobots; i++)
    {
      Robot* a = other.m_robots[i];
      m_robots[i] = new Robot(this, a->row(), a->col());
    }
  }
  return *this;
}
```

3. [35 points in all]

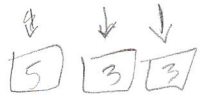
Consider the following class that implements a doubly-linked list of integers with no dummy node. The last node's `next` pointer is NULL; the first node's `prev` pointer is NULL. There is no tail pointer. When the list is empty, `head` is NULL.

```
class LinkedList
{
  public:
    ...
    int countAdjacentMatches() const;
    void eraseLast()
    {
        if (head != NULL)
            eraseLastAux(head);
    }
    void writeDiff(const LinkedList& other) const
    {
        wd(head, other.head);
    }
  private:
    struct Node
    {
      int value;
      Node* next;
      Node* prev;
    };
    Node* head;
    void eraseLastAux(Node* h);
    void wd(const Node* p1, const Node* p2) const;
};
```

a. [6 points]

The `countAdjacentMatches` member function counts how many nodes have a value that is equal to the value of the node that immediately follows it in the list. For example, if the LinkedList a contained nodes with the values 3 6 6 17 4 4 4 8 4 7, then the call `a.countAdjacentMatches()` returns 3, because it counted the first 6 and the first two 4s.

Write the `countAdjacentMatches` member function on the next page. *You will receive a score of zero on this problem if the body of your `countAdjacentMatches` member function is more than 15 statements long.*



[Write your `countAdjacentMatches` member function here.]

```
int LinkedList::countAdjacentMatches() const
{
 int count=0;
 if(head==NULL) return count;
 for(Node* curr=head; curr->next != NULL; curr = curr->next)
     if(curr->value == curr->next->value)
         count++;
 return count;
}
```

b. [7 points]

The `eraseLast` member function removes the last node, if any, from the linked list. To help it do its work, it calls `eraseLastAux`, which removes the last node from the linked list, but is guaranteed to be called on a list with one or more nodes; it will always be passed a non-NULL pointer.

In the space below, write a **non-recursive** implementation of the `eraseLastAux` member function. *You will receive a score of zero on this problem if the body of your `eraseLastAux` member function is more than 15 statements long.*

```
void LinkedList::eraseLastAux(Node* h)
{
 Node* toErase = h;
 for(; toErase->next != NULL; toErase = toErase->next) ;
 if(toErase->prev == NULL)
     head=NULL;
 else
     toErase->prev->next = NULL;
 delete toErase;
}
```

c. [7 points]

Now write a **recursive** implementation of the `eraseLastAux` member function. *You will receive a score of zero on this problem if the body of your `eraseLastAux` member function is more than 15 statements long or if it contains any occurrence of the keywords `while`, `for`, or `goto`.*

5 3 3

```
void LinkedList::eraseLastAux(Node* h)
{
if(h->next == NULL)
    {
    if(h->prev == NULL)
      head = NULL;
    else
      h->prev->next = NULL;
    delete h;
    }
else
    eraseLastAux(h->next);
}
```

p1 2 3 5 8 9 10
p2 3 4 5 6 7 8 9

d. [15 points]

A *strictly increasing list* is a list each of whose elements has a value that is less than the one that follows it. For example, 3 7 8 10 is a strictly increasing list, but 3 8 7 10 is not (8 is not less than 7), and 3 7 7 10 is not (7 is not less than 7). If `x` and `y` are LinkedLists whose nodes form two strictly increasing lists, calling `x.writeDiff(y)` writes out, one per line, all elements of `x` that are not in `y`. For example, if `x` contains 2 3 5 8 9 and `y` contains 3 5 6 7 8 10, then `x.writeDiff(y)` would write, one per line, the values 2 and 9.

The member function `writeDiff` calls a helper function `wd`, which accepts two Node pointers; if each points to a (possibly empty) strictly increasing linked list of Nodes, it writes out the values, one per line, that are in the first list but not the second.

In the space below, write a **recursive** implementation of the `wd` member function. You should assume that each of the lists is a (possibly empty) strictly increasing list. *You will receive a score of zero on this problem if the body of your `wd` member function is more than 20 statements long or if it contains any occurrence of the keywords `while`, `for`, or `goto`.*

```
void LinkedList::wd(const Node* p1, const Node* p2) const
{
 if(p1==NULL) return;
 if(p2 == NULL)
   {
    cout << p1->value << endl;
    wd(p1->next, p2);
    return;
   }
 if(p1->value < p2->value)
   {
    cout << p1->value << endl;
    wd(p1->next, p2);
   }
 else
   {
    wd(p1, p2->next);
   }
}
```

insert here sorry!

```
if(p1->value == p2->value)
{
 wd(p1->next, p2->next)
 return;
}
```

5

Ethel
Fred
Elaine
George
Ed
Jerry
Kramer
Ricky
Lucy
Ralph

5. [25 points in all]

Consider this attempt to write two classes that define an alarm clock type and a cuckoo clock type, and two functions that use these types:

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

     typedef int Time;

     class AlarmClock
     {
       public:
01:      AlarmClock(Time current) { m_time = current; }
02:      void setAlarm(Time alarm){ m_alarm = alarm; }
03:      void tick() {
04:          cout << "tick\n";
05:          if (++m_time == m_alarm) playAlarm();
06:      }
07:      void playAlarm()       { cout << "Buzz! Buzz!\n"; }
08:    private:
09:      Time m_time;
10:      Time m_alarm;
     };

11:  class CuckooClock : public AlarmClock
12:  {
13:    public:
14:      CuckooClock(Time t)     { m_time = t; }
15:      virtual ~CuckooClock() { cout << "Squawk!\n"; }
16:      void setAlarm(Time t) {
17:          cout << "Chirp!\n";
18:          m_alarm = t;
19:      }
20:      void playAlarm()  { cout << "Cuckoo! Cuckoo!\n"; }
     };

     void timePasses(AlarmClock& ac)
     {
          ac->setAlarm(503);
          ac->tick();
          ac->tick();
     }

     int main()
     {
         CuckooClock cc(501);
         timePasses(cc);
     }
```

When we run this program, we would like the output it produces to be

```
Chirp!
tick
tick
Cuckoo! Cuckoo!
Squawk!
```

a. [10 points]

Make the **minimal** changes **necessary** to correct the two classes so that the program builds successfully and produces the desired output above. You must in addition introduce const where appropriate. There are several restrictions:

1. You may modify only the lines in the program that we've indicated with numbers (so you must not modify the `timePasses` or `main` functions).
2. You must not add, remove, or change any statements that use `cout` (so you must not change the text of any quoted string literals).
3. You must not add or remove any member functions in either class.
4. You must not add or remove any data members in either class.
5. You must not add or remove the words `public`, `private`, or `protected`.

Notice that a correct solution may not necessarily use good C++ style, but it produces the required output.

List the line numbers and changes below that implement the correct solution, in a style like this example, where we suppose there were a line 21 with a function incorrectly declared:

```
21: void foo() ➔ int* foo()
```

Write your answer in the space below.

14: CuckooClock(Time t) : AlarmClock(t) { } +2

07: void playAlarm -> virtual void playAlarm +2

18: m_alarm = t; -> AlarmClock::setAlarm(t); +2

02: virtual void setAlarm(Time alarm) { m_alarm = alarm; }
+2

8

b. [3 points]

In part c, we'll define a new class named VoiceAlarm. A VoiceAlarm is a kind of AlarmClock that meets the following requirements:

1. When you create a VoiceAlarm, you must specify a string that will be the message played when it's time to play the alarm.
2. All VoiceAlarms start out with a current time of 1200.
3. Your VoiceAlarm constructor must use new to dynamically allocate a C++ string for the message and use a pointer data member to point to that dynamically allocated string.
4. When a VoiceAlarm's alarm is to be played, it must write the saved message instead of buzzing.
5. A VoiceAlarm must not leak memory.

Here's how a VoiceAlarm might be used:

```
int main()
{
    AlarmClock* a;
    ...
    a = new VoiceAlarm("Wake up, slacker!\n");
    ...
    delete a;
}
```

If you must make any changes to the AlarmClock base class to ensure your VoiceAlarm class works properly, state the changes in one or two sentences below; if you need not make any changes, write *No changes*.

AlarmClock must have the function
virtual ~AlarmClock() {}

3

c. [12 points]

On the next page, define and implement the VoiceAlarm class. Avoid needless redundancy; for example, the VoiceAlarm class must not declare a data member of type Time or int. (Also, for this problem you do not have to concern yourself with a copy constructor or an assignment operator.)

[Write your definition and implementation of VoiceAlarm here.]

```
struct VoiceAlarm : public AlarmClock
{
    VoiceAlarm(const string& alarm) : AlarmClock(1200)
        {
        m_alarmMes = new string(alarm); //uses string copy constructor
        }
    ~VoiceAlarm() { delete m_alarmMes; }
    virtual void playAlarm() { cout << *m_alarmMes; }
private:
    string* m_alarmMes;
};
```

Const -1