University of Illinois at Urbana-Champaign
Department of Computer Science

First Examination

CS 225 Data Structures and Software Principles
Spring 2007
7p-9p, Thursday, March 1

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	25		
3	15		
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;

void myFun(int * x) {
    x = new int;
    *x = 12;
}

int main(){
   int v = 10;
   myFun(&v);
   cout << v << endl;
   return 0;
}
```

What is the result when this code is compiled and run?

(a) Nothing. This code does not compile because of a type mismatch.

(b) 12 is sent to standard out

(c) 10 is sent to standard out

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

(e) None of these answers is accurate.

MC2 (2.5pts)

Consider the following C++ statements:

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

void myFun(int & x) {
    x = 12;
}

int main(){
   int v = 10;
   myFun(v);
   cout << v << endl;
   return 0;
}
```

What is the result when this code is compiled and run?

(a) Nothing. This code does not compile because of a type mismatch.

(b) 12 is sent to standard out

(c) 10 is sent to standard out

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

(e) None of these answers is accurate.

MC3 (2.5pts)

Which situation does not use the copy constructor?

(a) Calling a function that has only a reference parameter.

(b) Calling a function that has only a value parameter.

(c) Declaring a variable to be a copy of another existing object.

(d) Returning a value from a function.

(e) All of these situations use the copy constructor.

MC4 (2.5pts)

When should a pointer parameter `p` be a reference parameter? (That is, when would it be more appropriate for a parameter list to be `(myType * & p)` rather than `(myType * p)`?)

(a) When the function changes `p`, and you want the change to affect the actual pointer argument.

(b) When the function changes `p`, and you do NOT want the change to affect the actual pointer argument.

(c) When the function changes `*p`, and you want the change to affect the object that is pointed at.

(d) When the function changes `*p`, and you do NOT want the change to affect the object that is pointed at.

(e) When the pointer points to a large object.

MC5 (2.5pts)

Consider the following code:

```
class flower {
private:
  int petals;
public:
  flower(){petals = 6;}
  void setPetals(int newPetals) {petals = newPetals;}
  virtual int getPetals() { return petals; }
  void displayPetals();
};

class daisy : public flower {
public:
  daisy(){setPetals(20);}
  int getPetals(){return 6 * flower::getPetals();}
};

void flower::displayPetals() {
  cout << getPetals() << endl;
}

int main() {
  flower f;
  daisy d;

  f.displayPetals();
  d.displayPetals();

  return 0;
}
```

Which of the following best describes the output?

(a) 6 and then 20

(b) 6 and then 120

(c) 36 and then 120

(d) The call to `getPetals()` within `displayPetals()` is ambiguous, so there is a compile error.

(e) This code will not compile because `displayPetals()` is not defined for `daisy` objects.

MC6 (2.5pts)

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

Which of the following best describes the running time of the `push` and `pop` operations if the bottom of the Stack must be at the head of the linked memory structure? (n is the number of elements in the stack.)

(a) $O(1)$ for both `push` and `pop`

(b) $O(n)$ for both `push` and `pop`

(c) $O(1)$ for `push` and $O(n)$ for `pop`

(d) $O(n)$ for `push` and $O(1)$ for `pop`

(e) None of these is the correct choice.

MC7 (2.5pts)

Which of the options below is the appropriate function signature for the definition of the copy constructor for the train class?

```
template<class T>
class train {
private:
   // private portion of class definition here
public:
   // public portion of class definition here
};
```

(a) `train<T>::train(const train & oldTrain)`

(b) `train<T>::train(const train<T> & oldTrain)`

(c) `template<class T> train<T>::train(const train<T> & oldTrain)`

(d) `template<class T> train<T>::train<T>(const train & oldTrain)`

(e) None of these is correct.

MC8 (2.5pts)

What would be the result if the following code was compiled?

```
int  * const p = new int;   // line 1
const int q = 5;
*p = q + q;   // line 3

cout <<  *p  << endl;

delete p;
p = NULL;   // line 6
```

(a) A single compiler error, on line 1

(b) A single compiler error, on line 3

(c) A single compiler error, on line 6

(d) Multiple compiler errors

(e) The code will compile without errors

2. **[The Big Three – 25 points].** Consider the following partial class definition:

```
class Scene
{
        private:
                Image** pictures;
                int* xcoords;
                int* ycoords;
                int maxlayers;

               // perhaps some helper functions

        public:
               // constructors and destructor

               const Scene & operator=(const Scene & source);

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

The `pictures` structure is a dynamically allocated array of `Image` pointers. `xcoords` and `ycoords` are dynamically allocated arrays of integers. Each array has `maxlayers` elements and `maxlayers` is greater than zero.

In this question you will write the overloaded assignment operator for the `Scene` class that you would include in the `scene.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 write your answer on the following page.

problem 2 continued...

```
Scene const & Scene::operator=(Scene const & source) {
   if (this != &source) {
      for (int i = 0; i < maxlayers; i++)
         delete pictures[i];
      delete [] pictures;
      delete [] xcoords;
      delete [] ycoords;

      maxlayers = source.maxlayers;
      pictures = new Image *[maxlayers];
      xcoords = new int[maxlayers];
      ycoords = new int[maxlayers];
      for (int j = 0; j < maxlayers; j++) {
         xcoords[j] = source.xcoords[j];
         ycoords[j] = source.ycoords[j];
         if (source.pictures[j] != NULL)
            pictures[j] = new Image(*(source.pictures[j]));
         else
            pictures[j] = NULL;
      }
   }
   return *this;
}
```

Grading scheme:

- 2 points for getting the method signature correct (return type, scoping, method name, parameter type)
- 1 point for performing a self-assignment check
- 7 points for the "destructor" portion of the assignment operator
  - 3 points for deallocating the Image objects
  - 3 points for deallocating the actual arrays
  - 1 point for remembering to use "delete []" to deallocate arrays
- 14 points for the "copy constructor" portion of the assignment operator
  - 1 point for copying over maxlayers
  - 3 points for correctly allocating new arrays
  - 3 points for correctly copying the contents of xcoords and ycoords
  - 7 points for correctly copying the contents of pictures (this includes making hard copies of the Image objects)
- 1 point for getting the return value correct

3. **[Generic Programming – 15 points].**

(a) (5 points)

You are given the following generic function:

```
template <class Iter>
bool isValid(Iter first, Iter last) {

   Iter i1, i2;
   i1 = first;
   i2 = last;

   while (i1 != i2) { // position i2 at the second half of the list
      i1++;
      if (i1 != i2)
          i2--;
   }
   // now i1 == i2 just to the right of the middle of the list
   i1 = first;   // leave i2 there, and move i1 back to the front

   bool badPair = false;  // look for a bad pair
   if (i1 != i2) {
      while ((i2 != last) && !badPair) {
         badPair = (*i1 != *i2);
         i1++; i2++;
      }
   }
   return !badPair;
}
```

Furthermore, you have included the standard template library `list` class as seen in lecture, with a nested `iterator` class, and you have made the declaration:

```
list<int> theList;
```

and then inserted some values.

Write some code that uses iterators for the list `theList` that we declared above, and the template function above, to print a `1` (true) if `theList`'s second half is a duplicate of its first, and `0` (false) if it is not. If the list has odd length, the center element is ignored. Note that no iterators are declared yet; you will need to do that yourself, if you decide you need iterator variables.

```
cout<<isValid(theList.begin(),theList.end())<<endl;
```

Grading scheme:

- 2 points for getting the first and last iterator correct
- 2 points for calling the isValid function correctly
- 1 point for printing out the result

(b) (10 points) Now, we want to change the generic function from part (a) to the following:

```
template <class Iter, class Detector>
bool isValid(Iter first, Iter last, Detector test) {
   Iter i1, i2;
   i1 = first;
   i2 = last;

   while (i1 != i2) { // position i2 at the second half of the list
      i1++;
      if (i1 != i2)
          i2--;
   }
   // now i1 == i2 just to the right of the middle of the list
   i1 = first;  // leave i2 there, and move i1 back to the front

   bool badPair = false; // look for a bad pair
   if (i1 != i2) {
      while ((i2 != last) && !badPair) {
         badPair = (!(test(*i1,*i2));
         i1++; i2++;
      }
   }
   return !badPair;
}
```

For this problem you will write a class, objects of whose type can be passed as the third parameter to the above function when the first two parameters are iterators that point to collections of integers (for example, iterators to lists of integers, or iterators to vectors of integers, or whatever). The class should be such that the `test(*i1, *i2)` expression above evaluates to `1` (true) if either one of the two integer arguments is exactly 10 times the other, and evaluates to `0` (false) otherwise. It is okay to write the definition for this class right into the class declaration itself (i.e. you don't need to divide things up into a `.h` and `.cpp`). Note that we are asking you to write a function object whose only member function is an overloaded operator (which one?).

problem 3 continued...

```
class Detector  {
public:
    bool operator ()(int x, int y) {
        return x==10*y || y==10*x;
    }
};
```

Grading scheme:

- 2 points for writing the function inside a class
- 2 points for declaring the function as public
- 4 points for correctly overriding the operator (operator name, parameter type and return type)
- 2 points for correct implementation of the operator

4. **[skippyLists – 20 points].**

Suppose you have implemented a class called a `skippyList` that is a linked memory structure (with 6 sentinels!) containing nodes which have a previous pointer `prev` and a next pointer `next` arranged as shown in the diagram above. (Note that the nodes labeled with positive data in the diagram have null previous pointers.)

The following node structure is defined as a member of the skippyList class:

```
struct node{
     int value;
     node * prev;
     node * next;
     node(int  v = 0, node * p = NULL, node * n = NULL)
          : value(v), prev(p), next(n) { }
};
```

Write the `skippyList` member function `remove(node * p)` which removes both the node pointed to by `p`, and the node parallel to `p` from the structure (in our example, these are nodes whose data have the same absolute value). You may assume that `p` is a valid and existing data node (not a sentinel), but note that `p` could be either *type* of node. If you need to use names for the sentinel nodes, just tell us which is which by labeling the picture above. `remove` returns nothing. Be sure not to leak the nodes you remove.

```
void remove(node * p){  // 1 pt for void
    node * temp;    // 1 pt for declaring node ptr
    if(p->prev == NULL){    // 1 pt for checking what type of node
                            // 1 pt for checking prev is Null
        temp = p->next->prev;   // 1 pt for setting up temp node
        p->next->prev->prev->next = p->next->prev->next;    // 2 pts
        p->next->prev->prev->prev->next = p->next;  // 2 pts
        p->next->prev = temp->prev; // 1 pt
    }
    else{
        temp = p->prev->next;   // 1 pt for setting temp in 2nd case
        p->prev->prev->next->next = temp->next; // 2 pts
        p->prev->next = p->next;    // 1 pt
        temp->next->prev = p->prev; // 2 pts
    }
    delete temp;    // 1 pt for deleting opposite node
    delete p;   // 1 pt for deleting p
}
// Additional Grading:
// 2 additional points for syntax
// also you received 1 pt if you tried to fix the ptrs but messed up
// and you lost 2 pts if you reversed prev and next on all uses with
// a type of node
```

5. **[It's a Deal – 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 `Deal` that takes two arguments: a reference to a `Queue` called `deck` and an integer called `numPlayers`. The function should select from the `deck` every `numPlayers`th integer, and group them together, and it should do this repeatedly, until no integers are left. (This is intended to be very much like the process of dealing out cards to `numPlayers` of players.) Then these groups should be requeued into `deck` so that the groups appear in sequence, but within the groups the integers appear in the reverse of their original ordering.

For example, given the following queue, and input parameter 3

```
front             rear
 1 2 3 4 5 6 7 8 9 10
```

we get the following arrangement:

```
front                 rear
 10 7 4 1   8 5 2    9 6 3
 --------   ------   ------
```

Note that the blocks are for clarity. They are not apparent in the final version of `deck`.

Now of course, there are constraints on your implementation. Specifically, you must write the function using only the variable declarations we provide, except that you may declare `Stack`s and/or `Stack` pointers, as you need them. We are allowing you one `int` local variable, and it must be used only as a loop counter. (Hint1: for this problem you may use a `Stack` pointer to declare a dynamic array of stacks, Hint2: you may find the "%" operator useful–`a%b` returns the remainder when `a` is divided by `b`, and Comment: your solution will be graded in part by its elegance.)

Here's a start:

```
void Deal(Queue & deck, int numPlayers) {
  int i;
  Stack *players = new Stack[numPlayers];
  Stack temp;
  i=0;
  while (!deck.isEmpty()) {
    players[i % numPlayers].push(deck.dequeue());
    i++;
  }

  for (i=0; i<numPlayers; i++) {
    while (!players[i].isEmpty()) {
      temp.push(players[i].pop());
```

```
    }
    while (!temp.isEmpty()) {
      deck.enqueue(temp.pop());
    }
  }

  delete[] players;
}

// each syntax error takes off one-two point (depending on how serious it is).
// if the logic is partially correct, each
// ‘‘basic partial correct action’’ will get 5 points.
```

problem 5 continued...

(scratch paper, page 1)

(scratch paper, page 2)

```
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
};
```