SYSC 2002D Winter 2009 - Midterm (February 12th, 2009)
10:05am-11:20am (75min)
Closed book. No aids permitted other than what is provided.
Sample Solutions

Question 1 [3 marks]

Note: Assume all required includes are present.

a) Give the output produced by the following: (2 marks)

```
int *h, i, j;
IntBag *g;
h = new int;
g = new IntBag[3];
*h = 8;
for ( i=0; i<3; i++ )
   for ( j=0; j<i; j++ )
      g[i].add(j);
cout << *h << " " << g[2].size() << "\n";
(*h)--;
g = new IntBag[6];
for ( i=0; i<6; i++ ) {
   for ( j=0; j<i; j++ ) {
      (*(g+i)).add(5*j);
   }
}
cout << *h << " " << g[4].size() << "\n";
```

8 2
7 4 (½ mark for each number)

b) There is a memory leak. **Write the missing code and clearly indicate with an arrow exactly where the code belongs**. (1 mark)

`delete [] g;` should be added before or after `(*h)--;`
(½ mark each for statement and its place – no marks for the statement if the [] is missing.)

Question 2 [4 marks]

In this question you are completing an application program that **uses** the `IntBag` class. In the space below, fill in the missing parts of `realMain` as per the directions. For example, if `b1` contained { 7, 2, 1, 3 }, `smallest` would be 1. **You must include comments to get full marks.** (There is a blank page at the end if you run out of space below. Clearly indicate on this page if your answer continues there.)

```
#include "stdstuff.h"
#include "IntBag.h"

void realMain () {
  IntBag b1;
  int smallest;
  // add any other declarations needed here:

  int value; // (½ mark for the appropriate declarations)
  bool gotValue;

  // code to add items to the bag omitted here (you do not have to write this code)
  // complete the missing portions of the code below:

  if (b1.size()==0) // or !b1.startWalk(value) (1 mark)

     cout << "Bag b1 is empty.\n";
  else { // store the smallest element of bag b1 in "smallest"

     b1.startWalk(smallest); // initialize smallest to the first element
     // (we know there is a first element)  (1 mark)
     // Note that initializing smallest to anything else may not work,
     // depending on what we've stored in b1 (deduct ½ mark).

     // check each successive element and if it's the smallest so far,
     // store it in smallest (1.5 marks)
     for ( gotValue = b1.continueWalk(value); gotValue;
           gotValue = b1.continueWalk(value)) {
        if (value < smallest) smallest = value;
     }

     // Note: subtract ½ mark if no comments.
     // As this is an application program using the IntBag class, you do
     // not have direct access to count, data, etc.

     cout << "The smallest element in b1 is: " << smallest << "\n";
  } // end if
} // end realMain

int main () {
  try {
    realMain ();
  }
  catch (exception &e) { // catches all uncaught exceptions
    cout << "\nException <" << e.what() << "> occurred.\n";
  }
  pause ();
  return 0;
}
```

Question 3 [4 marks]
In this question, you are extending the given `IntBag` class.

We are adding a new constructor to `IntBag`. Here is the addition to the header file (`IntBag.h`):

```
// This constructor creates a new IntBag that comes to life containing the
// first "count" elements of "array".  If count is too large, this method
// throws an invalid_argument exception.  (Notes: You need to figure out what
// "too large" means.  You may assume that the array given always contains at
// least "count" elements.)
IntBag ( int count, int array[] );
```

Write the corresponding addition to the implementation file (IntBag.cpp). **You may use other IntBag methods. You must include comments.**

```
// This constructor creates a new IntBag that comes to life containing the
// first "count" elements of "array".  If count exceeds MAXVALUES, this
// method throws an invalid_argument exception.  (We assume that the array
// given always contains at least "count" elements.)
IntBag::IntBag ( int count, int array[] ) {

   // if count exceeds MAXVALUES, initialize fields (as per default
   // constructor) and throw invalid_argument exception
   if (count>MAXVALUES) { // (½ mark)
      count = 0; // (½ mark)
      walkInProgress = false; // (½ mark)
      throw invalid_argument
         ("IntBag::IntBag - count exceeds maximum IntBag size");
         // string must start with "IntBag::IntBag" (½ mark for exception)
   }

   // otherwise, set count and data as per the arguments, and set
   // walkInProgress to false
   this->count = count; // or IntBag::count = count; (½ mark)
   for (int i=0;i<count;i++) { // (1 mark for loop and setting data)
       data[i] = array[i];
   }
   walkInProgress = false; // (½ mark)
}

// Notes:
// walkInProgress = false; could be written just once as long as it's at
// the beginning.

// The five lines starting with this->count = count; could be replaced
// with the following 3 lines (worth the same 2 marks as the above):
// for (int i=0;i<count;i++) {
//      add(array[i]);
// } // The add method updates count, data and walkInProgress.

// Deduct ½ mark if no comments.
```

Question 4 [4 marks]
For this question, you are again extending the given `IntBag` class.

Overload the "==" method in the `IntBag` class, i.e. write the implementation of "==" (is equal to) for `IntBags`. This method is available to application programs and has one argument (an `IntBag`). It returns true if the bags contain the same values, each occurring with the same frequency. Otherwise it returns false. **You must include comments. You may use other methods in the `IntBag` class in your solution.** (If you use the spare page, be sure to indicate that on this one.)

For example, if ib1 = { 1, 2, 1, 3, 4 }, ib2 = { 1, 2, 3, 4, 1 }, ib3 = { 1, 2, 2, 3, 4 }, and ib4 = { 1, 2, 3, 4 }then: ib1==ib2 returns true, ib1==ib3 returns false, and ib1==ib4 returns false.

Addition to Header File: (Does this go in the private or **public** part? Circle one or the other.) **(½ mark)**

```
  // returns true if two intbags contain the same values with the same
  // frequency (in any order). (½ mark for comment)
  bool operator== (const IntBag &otherIntBag) const;
  // (1 mark for signature: each minor omission or mistake is ½ mark
  // deduction.)
```

Addition to Implementation File:

```
// returns true if two IntBags contain the same values with the same
// frequency (in any order).
bool IntBag::operator== (const IntBag &otherIntBag) const {

   // if the sizes are different, the bags can't be the same
   // (not required for correctness, but more efficient)
   if (size()!=otherIntBag.size()) return false;

   // check that each element occurs the same number of times in
   // the other IntBag.  (We really need to check each element just once
   // but keeping track of the ones we've already checked is complicated.)
   for (int i=0; i<size(); i++) { // (1 mark for loop)
       if (countOccurrences(data[i]) !=
               otherIntBag.countOccurrences(data[i])) return false;
               // (1 mark for checking each item)
   }

   return true; // if we get here all values occur with same frequency
}

// Notes:
// ½ mark credit for checking size can be given if the rest was
// missing or incorrect.
// Deduct ½ mark if no comments in implementation file.
```

File: IntBag.h You may remove and keep the last two pages of the exam.

```
class IntBag {

private:
  enum { MAXVALUES = 10 };
  int data [MAXVALUES];
  int count;  // number of values currently in the bag
  bool walkInProgress;
  int walkPosition; // position of last value returned

public:

  // constructs an empty bag. For really trivial constructors and methods, the complete
  // implementation is often placed directly in the class definition.
  IntBag () { count = 0; walkInProgress = false; }

  // adds a value to the bag.  returns true on success (space available)
  // and false on failure (bag full)
  bool add (int value);

  // removes one occurrence of the specified value from the bag.  returns
  // true on success (value existed) and false on failure (value did not exist)
  bool remove (int value);

  // returns the number of times that the specified value occurs in the bag
  int countOccurrences (int value) const;

  // returns the number of values in the bag.
  int size () const { return count; }

  // These two methods make it possible to "walk through" the values in a bag. The values
  // in the bag may be returned in any order.  There is no guarantee that they will be
  // returned in numerical order, in the order in which they were entered, or any other
  // particular order.  "startWalk" may be called at any time.  If returns true on
  // success (value obtained) and false on failure (bag empty).  "continueWalk" may only
  // be called if
  //   1/. the bag has not been modified since a walk was begun by calling startWalk AND
  //   2/. the last call to startWalk or continueWalk returned true
  // If these conditions are not met, "continueWalk" throws a "range_error" exception.
  // It returns true on success (value obtained) and false on failure (no more values).
  bool startWalk (int &value);
  bool continueWalk (int &value);

  // randomly picks and returns one of the values in the bag.
  // throws a range_error exception if the bag is empty.
  int pickRandom () const;
};
```

File: IntBag.cpp

```
#include "stdstuff.h"
#include "IntBag.h"
// note that the constructor and size are not mentioned here as their
// implementations are completely specified in the header file

// adds a value to the bag.  returns true on success (space available)
// and false on failure (bag full)
bool IntBag::add (int value) {
  if (count == MAXVALUES) return false;
  data[count++] = value;
  walkInProgress = false; // scrub any ongoing walk
  return true;
}
```

```
// removes one occurrence of the specified value from the bag.  returns
// true on success (value existed) and false on failure (value did not exist)
bool IntBag::remove (int value) {
  int i;
  for (i = 0; i < count; i++) {
    if (data[i] == value) {
      // we've found the data value. overwrite it with the last data value and adjust the
      // count to reflect the fact that there are now one fewer values in the bag. Note
      // that this works even if the value being removed is the last value in the bag.
      data[i] = data[--count];
      walkInProgress = false; // scrub any ongoing walk
      return true;
    }
  }
  return false;
}

// returns the number of times that the specified value occurs in the bag
int IntBag::countOccurrences (int value) const {

  int i, occurrences = 0;
  for (i = 0; i < count; i++) {
    if (data[i] == value) {
      occurrences++;
    }
  }
  return occurrences;
}

bool IntBag::startWalk (int &value) { // comments removed to save trees - see header
// You have plenty of room for comments, so you must include them in your code!
  if (count == 0) return false;
  walkInProgress = true; walkPosition = 0;
  value = data[walkPosition];
  return true;
}

bool IntBag::continueWalk (int &value) { // comments removed to save trees - see header
// You have plenty of room for comments, so you must include them in your code!

  if (!walkInProgress) {
    throw range_error ("IntBag::continueWalk invalid call");
  }
  if (++walkPosition == count) {
    walkInProgress = false; // we've come to the end of the road
    return false;
  }
  value = data[walkPosition];
  return true;
}

// randomly picks and returns one of the values in the bag.
// throws a range_error exception if the bag is empty.
int IntBag::pickRandom () const {

  int i;
  if (count == 0) {
    throw range_error ("IntBag::pickRandom bag is empty");
  }
  i = (int) ((rand () / (RAND_MAX + 1.0)) * count);
  return data[i];
}
```