University of Illinois at Urbana-Champaign
Department of Computer Science

First Examination

CS 225 Data Structures and Software Principles
Summer 2005
3:00pm – 4:15pm Tuesday, July 5

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

- This is a **closed book** and **closed notes** exam. No electronic aids are allowed, either.
- You should have 7 sheets total (the cover sheet, plus numbered pages 1-13). 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 back of the second-to-last page contains some class and function declarations you have seen before; you can use this sheet as reference while taking the exam.
- 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.

Problem	Points	Score	Grader
1	20		
2	15		
3	20		
4	20		
5	15		
Total	90		

1. **[Classes in C++ – 20 points].**

   Consider the following code, which contains a class representing an exam for a course of at most 20 students. The code below would appear in the header file for this class, which we'll assume is named `examdata.h`.

```
#ifndef EXAMDATA_225_H
#define EXAMDATA_225_H

#include "string.h"   // we will assume the string.h file here
                      //  is the same one you've seen on the MPs

class ExamData {
public:

   // Assigns the internal String to be equal to the
   // parameter String, and sets this collection of
   // exam scores to be of size zero
   ExamData(String theName);

   // If there are less than 20 exam scores so far,
   // add the parameter score in the next available cell,
   // (thus incrementing the number of scores by 1) and
   // return true. Otherwise, do not change the member variables
   // at all, and return false.
   bool addValue(double examScore);

private:

   String examName;
   int numScoresSoFar;
   double theValues[20];
};

#endif
```

   On the next page, write the `examdata.cpp` file for this class. The comments on the two functions above indicate what needs to be done by those two functions. You do not need to have any comments in your answer.

(Classes in C++, continued)

```
#include "examdata.h"

ExamData::ExamData(String theName)
{
   examName = theName;
   numScoresSoFar = 0;
}

bool ExamData::addValue(double examScore)
{
   if (numScoresSoFar < 20)
   {
      theValues[numScoresSoFar] = examScore;
      numScoresSoFar++;
      return true;
   }
   else
      return false;
}
```

2. [**Pointers and References – 15 points**].

   Consider the following class:

```
class Circle
{
public:

   // initializes member variables to parameter values
   Circle(double theRadius, double theX, double theY);

   // calculates and returns area of circle
   double area();

   // moves the center of the circle by the first parameter's
   // value in the positive X direction and by the second
   // parameter's value in the positive Y direction.
   void translate(int xMove, int yMove);

private:

   double radius;     // radius of the circle
   double xCenter;    // x-coordinate of the center of the circle
   double yCenter;    // y-coordinate of the center of the circle
};
```

(a) Consider the following code:

```
   Circle* cPtr;
```

Given the variable `cPtr` above, write additional code that will (1) create a *local* `Circle` object with radius `3.1`, x-coordinate `5.6`, and y-coordinate `2.3`, (2) assign the pointer above to point to that `Circle` object, and (3) using the pointer, obtain the area of the circle and print it out.

```
   Circle myLocal(3.1, 5.6, 2.3);
   cPtr = &myLocal;
   cout << cPtr->area() << endl;
```

(b) Imagine you have the following function:

```
double Foo(Circle const & theParam)
{
   Circle c2(theParam);
   double theArea = theParam.area();
   return (theArea + c2.area());
}
```

The above function will not compile with the given `Circle` class. Explain what change you would need to make to the `Circle` class so that the above code and the `Circle` class would both compile correctly.

Since `theParam` is const, you cannot call `area()` on `theParam` in the second line. In order to make the above code work correctly, after `area()` in both the `.h` and `.cpp` file, you must say `const`. That is,

```
   double area() const;
```

3. **[The Big Three – 20 points].**

   Consider the following code:

```
// this would be in the .h file
class Foo {
   public:
      Foo();         // no-argument constructor

      // As part of the question on the next page, you'll add a
      // function declaration here. Other public functions (functions we
      // don't care about for this problem) would go here too.

      Foo const & operator=(Foo const & origVal);





   private:
     String labels[5];
     Array<String>* names;
     Array<int>* values;
     Array<String*> namesPtr;
};
```

   Assume the member functions assign the above member variables so that:

   - every array is of size 5, indexed from 0 through 4
   - all pointers are pointing to legitimate objects, not to NULL or to garbage memory

   Add a correctly-written assignment operator to the class `Foo`. The declaration should go in the code on this page, and the definition should appear on the next page in the way in which you'd write it in the `.cpp` file. Do not worry about `#include` macros or any other such things; just write the assignment operator function.

(The Big Three, continued)

```
Foo const & Foo::operator=(Foo const & origVal)
{
   if (this != &origVal)
   {
      delete names;
      delete values;
      for (int i = 0; i <= 4; i++)
         delete namesPtr[i];

      for (int i = 0; i <= 4; i++)
      {
         labels[i] = origVal.labels[i];
         namesPtr[i] = new String(*((origVal.namesPtr)[i]));
      }

      names = new Array<String>(*(origVal.names));
      values = new Array<int>(*(origVal.values));
   }
   return *this;
}
```

4. **[Generic Programming – 20 points].**

   (a) You want to write a class whose instances are function objects. The class should be named `XCharsInRange`, and the `operator()` for the class has a return value of type `bool` and has three parameters. The first two parameters are of type `int`, and the third parameter is of type `String`. You can assume the first parameter is less than or equal to the second parameter. The function will return `true` if, in the parameter `String`, the total number of times the `'X'` and `'x'` characters appear, is between the two integers inclusive, return `true`; otherwise, return `false`.

   For example, if the first two arguments to the function were `5` and `7`, then the function returns true if the `'X'` and `'x'` characters appear between 5 and 7 times, inclusive. in the parameter `String`, total. If the `String` had two `'X'` characters and five `'x'` characters, for example, then the total is seven and so you would return `true`.

   (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`, though you can if you want to.)

```
class XCharsInRange {
public:
   bool operator()(int first, int second, String foo);
};


bool XCharsInRange::operator()(int first, int second, String foo)
{
   int total = 0;
   for (int i = 0; i < foo.length(); i++)
      if ((foo[i] == 'X') || (foo[i] == 'x'))
         total++;

   if ((total >= first) && (total <= second))
      return true;
   else
      return false;
}
```

(b) You want to write a generic function called `IsSorted`. This function has two template types, `Iterator` and `Comparer`. We will assume that instances of type `Comparer` are function objects that take two values of the type the `Iterator` type points to (i.e., two values of the type you'd get when you dereference something of type `Iterator`), and returns `true` if the first value is "less than" the second value (by some definition of "less than") and returns `false` otherwise. The generic function accepts two iterators as parameters, defining a range `[first, last)` as we have discussed before, and also has a third parameter, of the `Comparer` type discussed above. The generic function returns `true` if the values in the range are sorted from lowest to highest according to the `Comparer` object, and returns `false` otherwise.

For example, if the range held integers, and the function object defined "less than" using the integer `operator<`, and our range was:

```
 first                                                  last
  5    6   9    13    18   28   33   43   53   61  79    49
```

then the function should return `true`, since for each pair of consecutive values, the first is less than the second, according to the given comparison function. (The `49` at the end does not count since the value is not actually in the range). Assume the iterators implement the full iterator interface we have discussed in lecture, section, and the sample code.

```
template <typename Iterator, typename Comparer>
bool IsSorted(Iterator first, Iterator last, Comparer isLessThan) {
   // your code goes here

   while (first != last) {
      Iterator temp = first;
      temp++;
      if (temp == last)  // range is size 1
         return true;
      else {             // range of real values is at least two
         if (isLessThan(*first, *temp))
            first++;
         else
            return false;
      }
   }
   return true;
}
```

5. **[Inheritance – 15 points].**

   Consider the following class:

```
class Window {
public:

   // initializes member variables to be equal to parameter values
   Window(String theTitle, int theWidth, int theHeight);

   virtual ~Window() { }

   // prints out
   //    This window has title T, width W, and height H
   // where T, W, and H are the values of the member variables
   virtual void print();

   String getTitle(); // returns title of window
   int getWidth();   // returns width of window
   int getHeight();  // returns height of window

private:

   String title;
   int width, height;
};
```

   You want to write a class `DialogWindow` that is a derived class of `Window`, and whose specification is as follows:

   - There is one additional member variable, a `String` to hold a message to the user
   - There is a constructor that has four parameters, which hold the title, the width, the height, and the message string of the window, respectively. The constructor should initialize the object's member variables to be equal to the parameter values.
   - There is a method `print()` which has no parameters, returns nothing, and which prints:

     ```
     This window tells the user D while having title T, width W, and height H
     ```

     where D is the dialog message stored in one of this class's member variables, and T, W, and H are the title, width, and height stored in the other member variables.

(Inheritance, continued)

```
class DialogWindow : public Window {
private:
   String message;
public:
   DialogWindow(String theTitle, int theWidth, int theHeight, String theMessage) :
           Window(theTitle, theWidth, theHeight), message(theMessage)
   {
      // no code needed here
   }

   virtual void print() {
      cout << "This window tells the user " << message << " while having";
      cout << " title " << getTitle() << ", width " << getWidth();
      cout << ", and height " << getHeight() << endl;
   }
};
```

```
class String
   // here are the member function declarations for the String class
   String();                // initializes to empty string
   String(char const * initString);  // initializes to "literal" parameter
   String(String const & origVal);   // copy constructor
   ~String();                        // destructor
   String const & operator=(String const & origVal);  // assignment op
   char operator[](int index) const;   // accesses char at parameter index
   char & operator[](int index);       // accesses char at parameter index
   String substring(int startIndex, int substringLength) const;
       // piece of this beginning at startIndex; substringLength chars long
   String concat(String const & secondString) const; // appends param to copy
                                                     //  of this String
   int length() const;         // length of string
   // we've left the six relational operators (operator==, operator<,
   // etc.) out; you don't need them.
   friend std::ostream & operator<<(std::ostream & Out,
                              String const & outputString); // prints


template <typename Etype>
class Array
   // Here are the member function declarations for the Array class;
   // we've left the declarations for the iterators and iterator
   // support functions (begin(), end(), etc.) out; you don't need them.
   Array();   // size 0 array, indiced 0 through -1
   Array(int low, int high);   // indices low through high
   Array(Array<Etype> const & origVal);  // copy constructor
   ~Array();                             // destructor
   Array<Etype> const & operator=(Array<Etype> const & origVal);//assignment op
   Etype const & operator[](int index) const;   // accesses cell at param index

   Etype & operator[](int index);   // accesses cell at param index
   void initialize(Etype const & initElement); // inits all cells to param
   void setBounds(int theLow, int theHigh);  // changes bounds of array,
   int size() const;   // returns number of cells in array
   int lower() const;  // returns lowest index
   int upper() const;  // returns upper index
```

(scratch paper, page 1)

(scratch paper, page 2)