1. The following Node class is modified slightly from the solution to topological sort that was discussed in class. Explain what the this pointer is doing in this code and how it is doing it. I am not looking for a quick answer. Instead I would like to be convinced that you understand why it is there. Put your answer on the back of this page. You get a brownie point if you can tell me what not_so_topo is doing (hint: it is not finding a topological sort).

    enum Color { BLUE, BROWN, YELLOW };  
    class Node {  
        Color color;  // BLUE, BROWN, or YELLOW  
        int ident;    // Number identifying this object  
        Node **depends; // Pointers to dependent objects  
        int ndepends;  // Number of dependent objects  
        public:  
            void Node (int id) {  
                ident = id;    // Identity of the node object  
                color = BLUE;  // Set the initial color to BLUE  
                ndepends = 0;  // Set the number of dependencies to 0  
                depends = NULL;  
            }  
            ~Node () { if (depends != NULL) delete depends; }  
            void setdeps (Node **deps, int num) { // Set the dependencies  
                ndepends = num; // Called by an object of the  
                depends = deps; // Supervisor class (not shown)  
            }  
            Node *not_so_topo () { // The modified topo method  
                if (color == YELLOW) return NULL;  
                if (color == BROWN) {  
                    cout << "Result: " << ident << " <- ";  
                    return this;  
                }  
                color = BROWN;  
                for (int i=0 ; i < ndepends ; i++) {  
                    Node *node;  
                    if ((node = depends[i]->not_so_topo()) != NULL) {  
                        if (node == this) {  
                            cout << ident << "\n";  
                            exit(0);  
                        } else {  
                            cout << ident << " <- ";  
                            return node;  
                        }  
                    }  
                }  
                color = YELLOW;  
                return NULL;  
            }  
    };
2. Objects of type A, given below, will be created “on-the-fly.” Each A object will be given a random value between 0 and 10000; We would like to maintain a list of A objects sorted by increasing value. The list should be updated every time a new A object is added to it.

    class A { public: int value; A(int v) { value = v; }};

Suppose we use an array of pointers to A objects, all of whose elements are initially NULL, to implement the list. Such an array could be defined and initialized as follows:

    A **lst = new A*[10000];
    for (int i=0 ; i < 10000 ; i++) lst[i] = NULL;

We can keep all created A objects sorted at the head of the array. That is, after n objects have been created,

    lst[i] != NULL for 0 ≤ i < n and
    lst[i]->value ≤ lst[j]->value for 0 ≤ i < j < n.

One simple way to do this is move pointers at the tail of the list up one element to free an element in which a pointer is set to a newly created A object. A class for doing this is as follows:

    class OnTheFlySorter {
        A **lst;
        int nobjs; // nobjs = No. objects in list, not size of the array
    public:
        OnTheFlySorter (int size) { lst=new A*[size]; nobjs=0; }
        void insert (A *obj) {
            // Locate the first place in lst[] where obj can be placed (at i)
            int i=0;
            for ( ; i < nobjs && obj->value > lst[i]->value ; i++);

            // Insert the new obj in position i
            for (int k=nobjs-1 ; k >= i ; k--) lst[k+1] = lst[k];
            lst[i] = obj;
            nobjs++;
        }
        void show () {
            for (int i=0 ; i < nobjs ; i++) cout << lst[i]->value << " ";
            cout << "\n";
        }
    };

This code can be really slow. If values are decided randomly, on average, $n/2$ elements in lst will have to be moved to make room for the $n$th element. Hence, sorting $m$ objects will take roughly

    1 + 0.5 + 2 + 2.5 + 3 + ... + (m - 1)/2 + m/2 \approx m^2/4

which means sorting 10000 objects will cause about $10^8/4$ moves!!! This is ridiculous!! On the back of this page, propose (that is, describe) a better scheme and reimplement insert in the OnTheFlySorter class based on your idea. We are looking for an average number of moves that is not much more than the number of objects!!!
3. Write one C++ or C function that contains no more than three semi-colons. The more interesting the function the more points you get. I am mainly interested in your thinking up an interesting function - the correctness of your syntax is not that important.

However, if you have an interesting, working function that uses lots of stars in a meaningful way I will be happier. If your function uses stars in two different ways I will be happier yet. If your function uses both stars and ampersands in different ways I will be ecstatic. But do not put stars where stars do not belong.

You should also tell me what your function is intended to do and describe any data structures it uses. If you use a function that is given on this exam, I will feel sorry for you but I will not yield.
4. The code below violates an important principle of object oriented programming: namely, class B’s private data is open to any class due to method putNumber(). Replace putNumber() in class B with a method having the following prototype:

   void smooth();

and which is used like this

   B b(10);
   b.smooth();

instead of like this (see the code below)

   B b(10);
   A a;
   b.putNumber(a.encode(b));

Use space on this page to write the smooth() method. Draw an arrow from the method you write to the point at which it should appear in the code. You should not require a change in any other method of either class A or B. Here is the code:

   #include <iostream>
   using namespace std;

   class B {
     friend ostream &operator<<(ostream&, B&);
     friend class A;
     int number;
   public:
     B (int n) { number = n; }  
     void putNumber(int n) { number = n; }
   };

   ostream&operator<<(ostream &out, B &b) {
     out << b.number;
     return out;
   }

   class A {
   public:
     int encode(B &b) { return b.number - 1; }
   };

   int main() {
     B b(10);
     A a;
     b.putNumber(a.encode(b));
     cout << b << "\n";
   }