1. The following is much longer than is necessary for a perfect score on this question so as to explain in great detail what is happening.

The code shown is intended to be run on a collection of Node objects, each of which has a list of dependent Node objects. The method not_so_topo, when invoked by one Node object, in turn invokes the not_so_topo method of all of its dependent objects. Each time this happens, all the necessary current information about execution of the invoking object’s not_so_topo method, including the object itself, is stored on the system stack. By the time a not_so_topo is invoked with color == BROWN the system stack contains a series of stacked Node objects and their running environments. The value of the this pointer in this bottom-level invocation of not_so_topo is the address of the Node object whose not_so_topo method was invoked when its color was BROWN, from here on we refer to this as the bottom-level object. The bottom-level object is printed and its address is immediately returned, via this, to the not_so_topo method of the invoking object which is popped from the top of the system stack. The value is saved in a pointer variable whose name is node. The freshly popped invoking object checks to see if it is the same object as the bottom-level object by means of the test

if (this == node) { ... }

If not, the invoking object prints itself and the address of the bottom-level object is returned to the not_so_topo method and object that is currently at the top of the system stack. That object is popped and compares itself similarly against the bottom-level object. This process continues until the test succeeds. At that point, the popped object prints itself and cause the program to terminate. In summary, the this pointer is used to

(a) record the address of a Node object whose color is BROWN
(b) compare the address of a stored object with this address to see if the stored object is the object whose color is BROWN

Here is what not_so_topo does: it prints one “cycle” in the given network of Node objects.
2. We wish to store incoming objects in increasing order of their value in such a way that storing objects and listing them in order can both be done very fast. One should look for exploitable properties of the problem: that is, properties of the problem that will make it really easy to solve. In this case, there is an exploitable property: namely, the fact that $A$ objects will have values between 0 and 10000. The latter number is not so great and we may use an array of pointers to $A$ objects, indexed on an object’s value, to store the objects. If all $A$ objects are distinct, then storing takes constant time and listing takes linear time. However, there is a wrinkle: since values for $A$ objects are determined randomly, there may exist one or more collections of $A$ objects such that all objects in a collection have the same value. To accommodate this we make each array element point to a linked list of $A$ objects where all objects in a linked list have the same value which is the index of the array element pointing to that list. Because $A$ objects do not know they are supposed to be in linked lists, a Cell class can be established where each Cell object points to an $A$ object and the next Cell object in a linked list or NULL. The array is then an array of pointers to Cell objects. The code below implements this idea. Storing time is still constant because a new object merely has to be attached to a Cell object and inserted at the head of a list.

```cpp
class Cell {
public:
    A *object;
    Cell *next;
    Cell(A *object, Cell *next) { this->object = object; this->next = next; }
};

class OnTheFlySorter {
    Cell **lst;

public:
    OnTheFlySorter (int size) {
        lst = new Cell*[10000];
        for (int i=0 ; i < 10000 ; i++) lst[i] = NULL;
    }

    void insert (A *obj) {
        lst[obj->value] = new Cell(obj, lst[obj->value]);
    }

    void show () {
        for (int i=0 ; i < 10000 ; i++)
            for (Cell *ptr = lst[i] ; ptr != NULL ; ptr=ptr->next)
                cout << ptr->object->value << " ";
        cout << "\n";
    }
};
```
3. The function `cmp` below is sufficient to make me happy. It is used by `qsort` to sort an array of `A` objects, where class `A` is given below.

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

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

int cmp (const void *obj1, const void *obj2) {
    if (*((A**)obj1).value < *((A**)obj2).value) return -1;
    if (*((A**)obj1).value > *((A**)obj2).value) return 1;
    return 0;
}

int main(int argc, char **argv) {
    if (argc != 2) {
        cerr << "Usage: " << argv[0] << " <number-of-A-objects>\n";
        exit(0);
    }

    int n = atoi(argv[1]);
    A **lst = new A*[n];
    for (int i=0 ; i < n ; i++)
        lst[i] = new A((int)(10000*(rand()/(RAND_MAX+1.0))));
    qsort(lst, n, sizeof(A*), cmp);
    for (int i=0 ; i < n ; i++) cout << *(lst[i]->value) << " ";
    cout << "\n";
}
```
4. We saw this one in class. The following is an acceptable solution

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

class B;

class A {
public:
    int encode(B &);
};

class B {
    friend ostream &operator<<(ostream&, B&);
    friend class A;
    int number;
public:
    B (int n) { number = n; }  
    void smooth() { number = (new A())->encode(*this);  }
};

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

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

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