CS 7B - Spring 2014 - Final Exam 5/20/14

Write responses on separate paper.

1. Define each of the following terms in the context of this course. If it helps your definition, give an example.

   (a) *pointer* A **pointer**, also called a link or a reference, is an object, often a variable, that stores the location (the machine address) of some other object, typically of a structure containing data that we wish to manipulate. If we use pointers to locate all the data in which we are interested, then we need not be concerned about where the data themselves are actually stored, since by using a pointer, we can let the computer system locate the data when required.

   (b) *dereferencing a pointer* We use a star ***** to denote a pointer not only in the declarations of a C++ program, but also to access the object referenced by a pointer. In this context, the star appears not to the right of a type, but to the left of a pointer. Thus ***p** denotes the object to which `p` points. The action of taking `*p` is called "dereferencing the pointer p."

   (c) *dangling pointer* A links or pointer that is left undefined at the conclusion of a method of a linked structure, either because they have never been assigned or because they point to nodes that are no longer are used. Such links should either be reassigned to nodes still in use or set to the value 0 (NULL.) What makes the situation tricky is that we may have multiple variables pointing to the same memory. If multiple variables point to the same memory and we call delete on any of those variables, we have effectively deallocated the memory for all of the variables. If we don't explicitly clear the variables to 0 (NULL), they will be dangling references, and calling delete on any of them will produce a runtime error.

   (d) *dynamic array* A dynamically allocated array is represented by a pointer to its initial element.

   We can create objects dynamically (during run time) as the need arises. The C++ run-time system reserves a large block of memory called the "free store," for this reason. (This memory is also sometimes called heap memory, but this should not be confused with the heap data structure.) The operator `new` dynamically allocates the correct amount of storage for an object of a given type from the free store and returns a pointer to this object. That is, the value of this pointer is the address where this object resides in memory.
   Indeed, C++ allows for pointer variables to any data type, even to other pointers or to individual cells in an array.

   For example, we could allocate a new person object and initialize its members as follows.

```
Person *p;
// ...
p = new Person; // p points to the new Person
p->fname = "Pocahontas"; // set the structure members
p->id = 04564443;
p->dob = 1692-01-01;
```

   Consider a class `Vector`, shown in the following code fragment, which stores a vector by dynamically allocating an array of integers. The dynamic array is referenced by the member variable data. A dynamically allocated array is represented by a pointer to its initial element. The member variable `size` stores the number of elements in the vector. The constructor for this class allocates an array of the desired size. In order to return this space to the system when a `Vector` object is removed, we need to provide a destructor to deallocate this memory. (Recall that when an array is deleted we use "delete[ ]," rather than "delete.")

```
class Vector { // a vector class
```

```
public:
    Vector(int n); // constructor, given size
    Vector(); // destructor
    // ...  other public members omitted
private:
    int* data; // an array holding the vector
    int size; // number of array entries
};
Vector::Vector(int n) { // constructor
    size = n;
    data = new int[n]; // allocate array
}
Vector::Vector() { // destructor
    delete [] data; // free the allocated array
}
```

(e) *abstract data type* An abstract data type is a type defined by its operations, not by how those operations are implemented. The stack type is a good example. Abstract data types are like patterns in that they define the effects of operations, but they do not specifically define how those operations are implemented. As with algorithms, however, there are well-known implementation techniques for these operations. For example, a stack can be implemented using any number of underlying data structures, such as a linked list or an array. Once we make the decision to use a particular data structure, though, the implementation decisions are sometimes already made. Suppose we implemented a stack using a linked list and are unable to wrap around an existing linked list, but we must write our own list code. Because the stack is a last-in-first-out structure, it only makes sense for us to insert and remove items at one end of the linked list. Furthermore, it only makes sense to insert and remove at the front of the list. Theoretically, you could insert and remove at the end, but this would result in an inefficient traversal of the entire list for every insertion or removal. To avoid those traversals would require a doubly linked list with a separate pointer to the last node in the list. Inserting and removing at the beginning of the list allows the simplest, most efficient implementation, so linked-list implementations of stacks are almost all implemented the same way.

Thus, even though the abstract in abstract data type means the type is conceptual and without implementation detail, in practice, when you choose to implement an abstract data type in your code, you won't be figuring out the implementation from scratch. Rather, you will have existing implementations of the type as guides.

An ADT specifies what each operation does, but not how it does it. In C++, the functionality of a data structure is expressed through the public interface of the associated class or classes that define the data structure. By public interface, we mean the signatures (names, return types, and argument types) of a class's public member functions. This is the only part of the class that can be accessed by a user of the class.

An ADT is realized by a concrete data structure, which is modeled in C++ by a class. A class defines the data being stored and the operations supported by the objects that are instances of the class. Also, unlike interfaces, classes specify how the operations are performed in the body of each function. A C++ class is said to implement an interface if its functions include all the functions declared in the interface, thus providing a body for them. However, a class can have more functions than those of the interface.

(f) *template function* Functions can be overloaded for a lot of different parameter list types, and it could make sense for all of them to have essentially the same body of code. For cases such as this, C++ has the ability to define functions with generic types, known as *function templates*. Defining a function template

follows the same syntax than a regular function, except that it is preceded by the keyword `template` and a series of template parameters enclosed in angle-brackets ¡¿:

```
template <template-parameters> function-declaration
```

The template parameters are a series of parameters separated by commas. These parameters can be generic template types by specifying either the `class` or `typename` keyword followed by an identifier. This identifier can then be used in the function declaration as if it was a regular type. For example, a generic sum function could be defined as:

```
template <class GenericType>
GenericType sum (GenericType a, GenericType b)
{
    return a+b;
}
```

It makes no difference whether the generic type is specified with keyword `class` or keyword `typename` in the template argument list (they are complete synonyms in template declarations).

In the code above, declaring GenericType (a generic type within the template parameters enclosed in angle-brackets) allows GenericType to be used anywhere in the function definition, just as any other type; it can be used as the type for parameters, as return type, or to declare new variables of this type. In all cases, it represents a generic type that will be determined on the moment the template is instantiated.

Instantiating a template is applying the template to create a function using particular types or values for its template parameters. This is done by calling the function template, with the same syntax as calling a regular function, but specifying the template arguments enclosed in angle brackets:

```
name <template-arguments> (function-arguments)
```

For example, the sum function template defined above can be called with:

```
x = sum<int>(10,20);
```

The function `sum<int>` is just one of the possible instantiations of function template sum. In this case, by using int as template argument in the call, the compiler automatically instantiates a version of sum where each occurrence of GenericType is replaced by `int`, as if it was defined that way.

(g) *template class* In general, a template is an outline for the definition of a function or class that uses parameters in place of types or objects. The template can then be instantiated by substituting types and objects for the template parameters. In C++, both functions and classes can be defined from templates.

(h) *standard template library* The C++ Standard Library is the result of the 1998 ISO standardization of the C++ programming language, which expanded its library by including a collection of classes and functions that were developed in the early 1990s by Alexander Stepanov and Meng Lee. Before the conclusion of the ISO standardization process, this separate collection was knows as the Standard Template Library. Because it is primarily a collection of classes, iterators, and implemented algorithms, Bjarne Stroustrup suggested the acronym CIA for the library. Regardless of the name, the implementation of the generic algorithms in the library are one of its most distinctive features.

The Standard Template Library (STL) is a collection of useful classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following standard containers.

`stack` Container with last-in, first-out access
`queue` Container with first-in, first-out access

`deque` Double-ended queue
`vector` Resizeable array
`list` Doubly linked list
`priority queue` Queue ordered by value
`set` Set
`map` Associative array (dictionary)

(i) *contiguous memory* This is a sequence of memory addresses without a gap from one to the next. Unfragmented memory.

(j) *overloading* You can have multiple definitions for the same function name in the same scope. The definitions of the function must differ from each other by the types and/or the number of arguments in the argument list. You can not overload function declarations that differ only by return type.

You can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

(k) *object*

(l) *instance of an class* This is a variable of a user-defined class type. For example:
`Foo* foo1 = new Foo ();`
creates an instance of the class Foo pointed to by foo1. Alternatively, you can create an instance of the class Foo by just declaring,
`Foo instanceOfFoo;`
With the pointer declaration, it will persist until you diliberately delete it. If you use the second method, it will be created on the runtime stack and be deleted when it goes out of scope.
Also, note that you can only use polymorphsim with instance pointers or references, not with plain instances.

(m) *data hiding*

(n) *initialization list* There is a subtlety in using constructors involving the assumption that whatever class maybe involved in the constructor has an assignment operator defined for it. If the type of name is a class without an assignment operator, this type of initialization might not be possible. In order to deal with the issue of initializing member variables that are themselves classes, C++ provides an alternate method of initialization called an initializer list. This list is placed between the constructor's argument list and its body. It consists of a colon (:) followed by a comma-separated list of the form `member name(initial value)`. To illustrate the feature, consider the Person constructor with its first three members initialized by an initializer list. The initializer list is executed before the body of the constructor.

```
// constructor using an initializer list
Person::Person(const string& fn, int id, Date dob)
      : _fname(fn), _id(id), _dob(dob) {}
```

(o) *friend* Complex data structures typically involve the interaction of many different classes, which often raises issues about coordinating the actions of these classes to allow sharing of information. Private members of a class may only be accessed from within the class, with the exception that we can declare a function as a `friend`, which means that this function may access the class's private data.

There are a number of reasons for defining `friend` functions. One is that syntax requirements may forbid us from defining a member function. For example, consider a class `Foo`. Suppose that we want to define an overloaded output operator for this class, and this output operator needs access to private member data. To handle this, the class declares that the output operator is a friend of the class as shown below.

```
class Foo { private:
    int secret;
public:
// ...  // give << operator access to secret
friend ostream& operator<<(ostream& out, const Foo& x);
};
ostream& operator<<(ostream& out, const Foo& x)
     {cout << x.secret; }
```

Another time when it is appropriate to use friends is when two different classes are closely related. For example, the code fragment below shows two cooperating classes `Vector` and `Matrix`. The former stores a three-dimensional vector and the latter stores a 3×3 matrix. This illustrates just one example of the usefulness of class friendship.

The class `Vector` stores is coordinates in a private array, called `coord`. The `Matrix` class defines a function that multiplies a matrix times a vector. Because `coord` is a private member of `Vector`, members of the class `Matrix` would not have access to `coord`. However, because `Vector` has declared `Matrix` to be a friend, class `Matrix` can access all the private members of class `Vector`. The ability to declare friendship relationships between classes is useful, but the extensive use of friends often indicates a poor class structure design. For example, a better solution would be to have class `Vector` define a public subscripting operator. Then the multiply function could use this public member to access the vector class, rather than access private member data. Note that "friendship" is not transitive. For example, if a new class `Tensor` was made a friend of `Matrix`, `Tensor` would not be a friend of `Vector`, unless class `Vector` were to explicitly declare it to be so.

```
class Vector { // a 3-element vector
public:  // ...  public members here
private:
    double coord[3]; // storage for coordinates
friend class Matrix; // give Matrix access to coord
}; class Matrix { // a 3x3 matrix
public:
Vector multiply(const Vector& v); // multiply by vector v
// ...  other public members here
private:
double a[3][3]; // matrix entries
};
Vector Matrix::multiply(const Vector& v) { // multiply by vector v
Vector w;
     for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            w.coord[i] += a[i][j] * v.coord[j]; // access to coord allowed
    return w;
}
```

(p) *protected* One important feature of classes is the notion of access control. Members may be declared to be `public`, which means that they are accessible from outside the class, or `private`, which means that they are accessible only from within the class. Private members cannot be accessed from outside the class definition.

It is often useful for a derived class to explicitly invoke a member function of a base class. For example, in the process of printing information for a `Student` derived from a `Person`, it is natural to first print the information of the `Person` base class, and then print information particular to the student. Performing this task is done using the class scope operator.

```
void Person::print() { // definition of Person print
    cout << "Name " << _fname << endl;
    cout << "ID " << _id << endl;
}
void Student::print() { // definition of Student print
Person::print(); // first print Person information
    cout << "Major " << major << endl;
    cout << "Year " << gradYear << endl;
}
```

Without the "Person::" specifier used above, the `Student::print` function would call itself recursively, which is not what we want.

Even though class `Student` is inherited from class `Person`, member functions of `Student` do not have access to private members of `Person`. For example, the following is illegal.

```
void Student::printName() {
    cout << _fname << '\n'; // ERROR! _fname is private to Person
}
```

(q) *inheritance* Consider defining a type `Shape` for use in a graphics system that needs to support circles, triangles, and squares. Assume also that we have

```
    class Point /* ...  */ ;
    class Color /* ...  */ ;
```

We might define a shape like this:

```
enum Kind {circle, triangle, square}; // enumeration
class Shape {
    Kind k; // type field
    Point center;
    Color col;
    / / ...
public:
    void draw();
    void rotate(int);
    // ...
};
```

The "type field" `k` is necessary to allow operations such as `draw()` and `rotate()` to determine what kind of shape they have. The function `draw()` might be defined like this:

```
void Shape::draw()
{
    switch(k) {
    case circle:
        // draw a circle
        break;
    case triangle:
        // draw a triangle
        break;
    case square:
        // draw a square
        break;     }
}
```

This is a mess. Functions such as `draw()` must "know about" all the kinds of shapes there are. Therefore, the code for any such function grows each time a new shape is added to the system. If we define a new shape, every operation on a shape must be examined and (possibly) modified. We are not able to add a new shape to a system unless we have access to the source code for every operation. Because adding a new shape involves "touching" the code of every important operation on shapes, doing so requires great skill and potentially introduces bugs into the code that handles other (older) shapes. The choice of representation of particular shapes can get severely cramped by the requirement that (at least some of) their representation must fit into the typically fixed-sized framework presented by the definition of the general type Shape.

The problem is that there is no distinction between the general properties of every shape (that is, a shape has a color, it can be drawn, etc.) and the properties of a specific kind of shape (a circle is a shape that has a radius, is drawn by a circle-drawing function, etc.). Expressing this distinction and taking advantage of it defines object-oriented programming. Languages with constructs that allow this distinction to be expressed and used support object-oriented programming. Other languages don't.
The inheritance mechanism (borrowed for C++ from Simula) provides a solution. First, we specify a class that defines the general properties of all shapes:

```
class Shape {
    Point center;
    Color col;
    // ...
public:
    Point where() { return center; }
    void move(Point to) { center = to; /* ...  */ draw(); }
    virtual void draw() = 0;
    virtual void rotate(int angle) = 0;
    // ...
};
```

This is now an abstract base class where the functions for which the calling interface can be defined but where the implementation cannot be defined yet are virtual. In particular, the functions `draw()` and `rotate()` can be defined only for specific shapes, so they are declared virtual.

Given this definition, we can write general functions manipulating vectors of pointers to shapes:

```
void rotate_all(vector<Shape*>& v, int angle) // rotate v's elements angle degrees
{
    for(int i = 0; i<v.size(); ++i) v[i]->rotate(angle);
}
```

To define a particular shape, we must say that it is a shape and specify its particular properties (including the virtual functions):

```
class Circle :  public Shape {
    int radius;
public:
    void draw() { /* ...  */ }
    void rotate(int) {} // yes, the null function };
```

In C++, class `Circle` is said to be derived from class `Shape`, and class `Shape` is said to be a base of class `Circle`. An alternative terminology calls `Circle` and Shape subclass and superclass, respectively. The derived class is said to inherit members from its base class, so the use of base and derived classes is commonly referred to as inheritance.

The programming paradigm is:

> Decide which classes you want;
> provide a full set of operations for each class;
> make commonality explicit by using inheritance.

Where there is no such commonality, data abstraction suffices. The amount of commonality between types that can be exploited by using inheritance and virtual functions is the litmus test of the applicability of object-oriented programming to a problem. In some areas, such as interactive graphics, there is clearly enormous scope for object-oriented programming. In other areas, such as classical arithmetic types and computations based on them, there appears to be hardly any scope for more than data abstraction, and the facilities needed for the support of object-oriented programming seem unnecessary.

Finding commonality among types in a system is not a trivial process. The amount of commonality to be exploited is affected by the way the system is designed. When a system is designed and even when the requirements for the system are written commonality must be actively sought. Classes can be designed specifically as building blocks for other types, and existing classes can be examined to see if they exhibit similarities that can be exploited in a common base class.

Class hierarchies and abstract classes complement each other instead of being mutually exclusive. In general, the paradigms listed here tend to be complementary and often mutually supportive. For example, classes and modules contain functions, while modules contain classes and functions. The experienced designer applies a variety of paradigms as need dictates.

(r) *polymorphism* In programming languages, polymorphism means that some code or operations or objects behave differently in different contexts.

For example, the + (plus) operator in C++:

4 + 5 ¡– integer addition
3.14 + 2.0 ¡– floating point addition
s1 + "aroo" ¡– string concatenation!

In C++, that type of polymorphism is called overloading.
Typically, when the term polymorphism is used with C++, however, it refers to using virtual methods involving a hierarchy of classes that are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes:

```
1 class Shape {
  protected:
3    int width, height;
  public:
5    Shape( int a=0, int b=0) {
        width = a;
```

```
7        height = b;
   }
9    int area() {
        cout << "Parent class area :" <<endl;
11      return 0;
   }
13 };

15 class Rectangle: public Shape{
   public:
17    Rectangle( int a=0, int b=0):Shape(a, b) { }
      int area () {
19       cout << "Rectangle class area :" <<endl;
         return (width * height);
21    }
   };
23
   class Triangle: public Shape{
25 public:
      Triangle( int a=0, int b=0):Shape(a, b) { }
27    int area () {
         cout << "Triangle class area :" <<endl;
29       return (width * height / 2);
      }
31 };

33 // Main function for the program
   int main() {
35    Shape* shape;
      Rectangle rec(10,7);
37    Triangle  tri(10,5);

39    // store the address of Rectangle
      shape = &rec;
41    // call rectangle area.
      shape->area();
43
      // store the address of Triangle
45    shape = &tri;
      // call triangle area.
47    shape->area();

49    return 0;
   }
```

The output of this program is

```
Parent class area
Parent class area
```

The reason for the base class' output is produced is that the call of the function `area()` is being set once by the compiler as the version defined in the base class. This is called static resolution of the function call, or static linkage - the function call is fixed before the program is executed. This is also sometimes called early binding because the `area()` function is set during the compilation of the program.

Making the `area()` in the `Shape` class virtual so that it looks like this:

```
virtual int area()
```

gets it right and the code now produces the output

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer and sees addresses of objects of tri and rec classes are stored in *shape their respective `area()` function is called.

Each of the child classes has a separate implementation for the function `area()`. This is how polymorphism is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

(s) *virtual function* A virtual function is a function in a base class that is declared using the keyword virtual. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

Instead, the function to be called is based on the kind of object for which it is called. This sort of operation is referred to as dynamic linkage, or late binding.

(t) *abstract base class* It's possible that you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function `area()` in the base class to the following:

```
class Shape {
protected:
   int width, height;
public:
   Shape( int a=0, int b=0) {
      width = a;
      height = b;
   }
   // pure virtual function
   virtual int area() = 0;
};
```

The "= 0" tells the compiler that the function has no body, making this virtual function a pure virtual function and the base class `Shape` an abstract base class which can't be instantiated.

2. Carefully analyze the following mystery program. What is output to the console?

```
#include <iostream>
using namespace std;
void  mystery(int v) {
    cout << " " << v;
    if(v == 4) return;
    if(v < 4) mystery(7-v);
    else mystery(9-v);
    cout << " " << 10*v;
}
int main() {
    mystery(1);
    return 0;
}
```

Solution: When `mystery(1)` is called by `main()`, the first thing it does is print out " 1". Then, since v ¡ 4, it calls `mystery(6)` which prints out " 6" and since 6 ¿ 4, it calls `mystery(3)` which prints " 3" and calls `mystery(4)` which prints " 4". At this point, we have 4 activation records (or frames) on the runtime stack. The one on top `mystery(4)` just returns, leaving `mystery(3)` to print out " 30", `mystery(6)` prints out " 60" and `mystery(1)` prints out " 10" for the complete output of " 1 6 3 4 30 60 10".

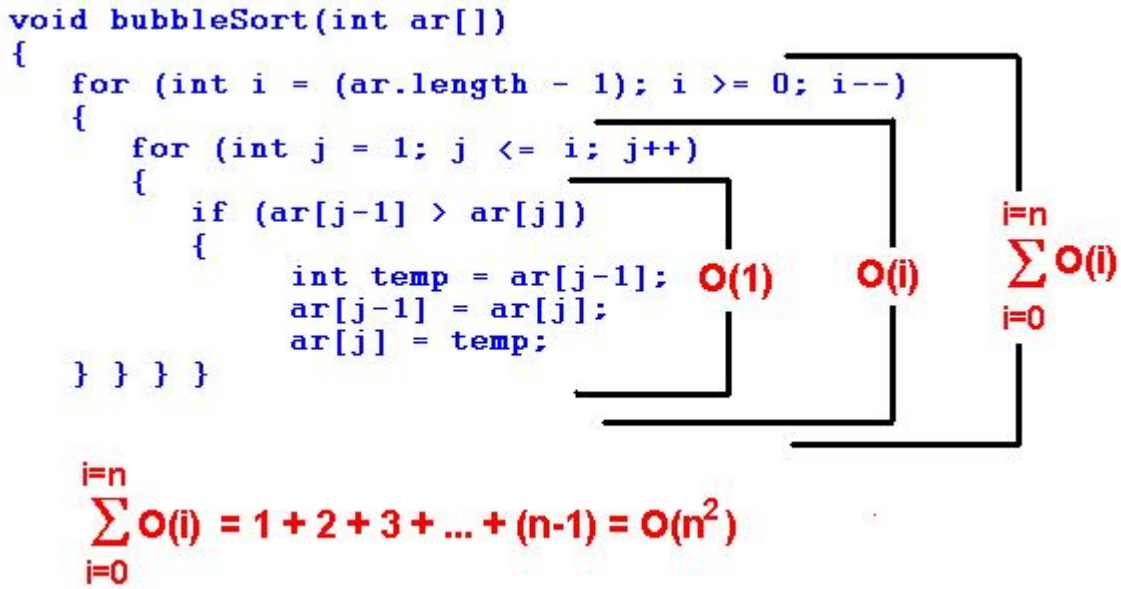
3. The running time of particular sorting algorithm for different sizes of arrays of randomly ordered integers is approximated by the formula (N is the number of integers to sort) time (milliseconds) $= 0.12+0.03N+0.24N^2$ Circle each of the following sorting algorithms that are consistent with the above formula.

   (a) merge sort

   (b) insertion sort

   (c) bubble sort

Solution: As this diagram illustrates, bubble sort is a quadratic time algorithm:



With insertion sort, an unordered list of elements is sorted by removing its entries one at a time and then inserting each of them into a sorted part (initially empty):

```
void insertionSort(int[] ar) {
   for (int i=1; i      ar.length; i++) { // for each element of the array
      int index = ar[i]; int j = i;
      while (j > 0 && ar[j-1] > index) {
           ar[j] = ar[j-1];  // find where it goes in the list
           j--;
```

```
7           }
            ar[j] = index; // and put it there
9       }
    }
```

Example. We color a sorted part in green, and an unsorted part in black. Here is an insertion sort step by step. We take an element from unsorted part and compare it with elements in sorted part, moving form right to left.

29, 20, 73, 34, 64
29, 20, 73, 34, 64
20, 29, 73, 34, 64
20, 29, 73, 34, 64
20, 29, 34, 73, 64
20, 29, 34, 64, 73

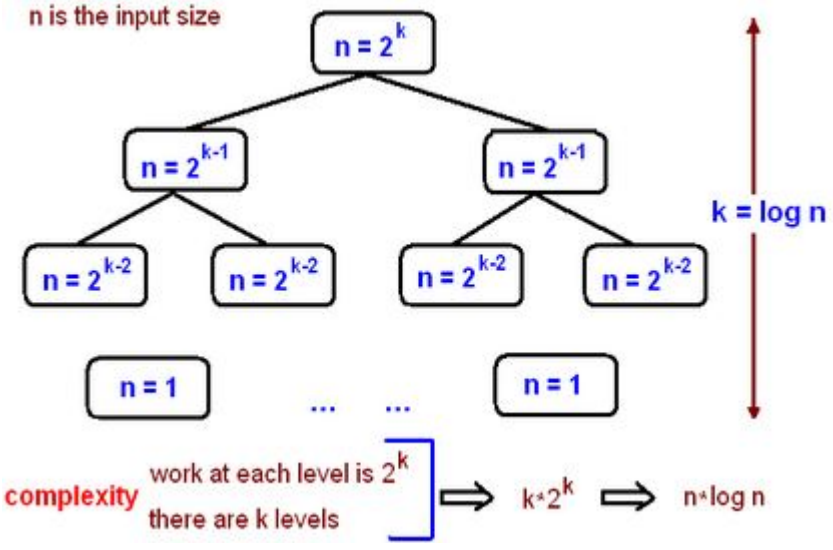
Calculate the number of comparisons of an array of N elements:

We need 0 comparisons to insert the first element.
We need 1 comparison to insert the second element.
We need 2 comparisons to insert the third element.
...
We need (N-1) comparisons (at most) to insert the last element.

All together,

$1 + 2 + 3 + \cdots + (N - 1) = O(N^2)$

The worst-case runtime complexity is $O(N^2)$. So it could be bubble sort or insertion sort, but merge sort takes $N \ln(n)$ time, so that doesn't fit the bill.



4. Which one of the following must be true if the statement "`this->foo++;`" is part of a valid C++ program?
   (a) `foo` must be defined as part of a class.
   (b) The statement must be inside a class definition.
   (c) The class being defined contains a method named "`foo`".

(d) The above code must be inside an class method.

(e) The local variable named "foo" is a `const`.

Solution: (a), (b) and (d) are certainly true. (c) is not true, since you can't increment a method. (e) is not true since you can't increment a `const`.

5. **Fast Flood Simulation**

Consider the complete program below which uses recursion simulate fast flooding of a geographic area. The terrain is represented as an evenly-spaced square grid of _N by _N miles (_N is odd), each grid square represents 1 square mile.

Flooding starts at the upper left region and will recursively attempt to flood the four nearest neighbors (North, South East and West). Water will flow into a neighbor if it is lower and there's water in the source square.

```
#include <ctime>
#include <cstdlib>
#include <iostream>
#include <vector>
#define N 5

using namespace std;

class Cell {
    friend ostream& operator<<(ostream& ostr, const Cell& x) {
        ostr << '(' << x._elev << ", " << x._waterLevel << ')';
        return ostr;
    }
public:
    Cell(unsigned elev, unsigned waterLevel) :
        _elev(elev), _waterLevel(waterLevel) {}
private:
    unsigned _elev; // elevation of cell
    unsigned _waterLevel; // water level
friend class Grid;
};

class Grid {
public:
    Grid(int n) : _N(n) {
        createGrid();
    }
    void createGrid() {
        for(int i = 0; i != _N*_N; ++i) {
            if(i/_N==0 || i/_N==_N-1 ||
               i%_N==0 || i%_N==_N-1) {
                    Cell c(0,0);
                    _grid.push_back(c);
               }
            else {
                 Cell c(40+_N*(rand()%(_N+1)),1);
                 _grid.push_back(c);
            }
        }
    }
    void updateGrid(int x) {
        //int row = x/_N, col = x%_N;
        // look N
        if(_grid[x-_N]._elev != 0 && _grid[x]._elev > _grid[x-_N]._elev && _grid[x].
    _waterLevel) {
            _grid[x]._waterLevel-=1;
            _grid[x-_N]._waterLevel+=1;
            updateGrid(x-_N);
        }
        // look E
```

```
50          if(_grid[x+1]._elev != 0 && _grid[x]._elev>_grid[x+1]._elev && _grid[x]._waterLevel) {
                _grid[x]._waterLevel-=1;
52              _grid[x+1]._waterLevel+=1;
                updateGrid(x+1);
54          }
            // look S
56          if(_grid[x+_N]._elev != 0 && _grid[x]._elev>_grid[x+_N]._elev && _grid[x]._waterLevel)
        {
                _grid[x]._waterLevel-=1;
58              _grid[x+_N]._waterLevel+=1;
                updateGrid(x+_N);
60          }
            // look W
62          if(_grid[x-1]._elev != 0 && _grid[x]._elev>_grid[x-1]._elev && _grid[x]._waterLevel) {
                _grid[x]._waterLevel-=1;
64              _grid[x-1]._waterLevel+=1;
                updateGrid(x-1);
66          }
        }
68
        void updateGrid() {
70          for(vector<Cell>::iterator i = ++_grid.begin(); i != --_grid.end(); ++i)
                ++(i->_waterLevel);
72          updateGrid(_N*_N/2);
        }
74      void printGrid() {
            for(vector<Cell>::iterator i = _grid.begin(); i != _grid.end(); ++i) {
76              cout << *i << " ";
                if((i - _grid.begin())%_N==_N-1) cout << endl;
78          }
        }
80  private:
        int _N; // grid is _N by _N square
82      vector<Cell> _grid; //each cell has an elevation and a water level
    };
84
    int main()
86  {
        srand(unsigned(time(0)));
88      bool keepGoing = 1;
        Grid g(5);
90      while(keepGoing) {
            g.printGrid();
92          g.updateGrid();
            cin >> keepGoing;
94      }

96      return 0;
    }
```

(a) Give a detailed description for how the constructor for `Grid` works.

Solution: The constructor (lines 25-40) takes an `int` parameter as an initializer for the private data member `_N` and then calls `createGrid()` which builds the `vector<Cell>` _grid as a vector of Cells with (0,0) Cells around the perimeter of an $N \times N$ grid. This is done by looking at the condition

```
if(i/_N==0 || i/_N==_N-1 || i%_N==0 || i%_N==_N-1)
```

which says if i is the index of a Cell on the top row (`i/_N==0`) or the bottom row (`i/_N==_N-1`) or the left margin (`i%_N==0`) or the right margin (i%_N==_N-1) then push the cell (0,0) on the vector. Otherwise, pick a random number between 40 and 40+_N for the elevation and set the waterlevel to 1 and push the Cell on the vector with

```
Cell c(40+_N*(rand()%(_N+1)),1);
```

(b) On what line of the code is a Cell instantiated?
Solution: Cells are instantiated on lines 32 and 36.

(c) How is `Cell` made to be a friend of `Grid`? Why is that necessary?
Solution: On line 20 we have `friend class Grid;` in the definition for `Cell`. This is needed so that Grid can access Cell's private data members, `_elev` and `_waterlevel`.

(d) How is `updateGrid()` overloaded? Why is that necessary?
Solution: The method `updateGrid()` without any parameter is defined on lines as

```
1    void updateGrid() {
         for(vector<Cell>::iterator i = ++_grid.begin(); i != --_grid.end(); ++i)
3            ++(i->_waterLevel);
         updateGrid(_N*_N/2);
5    }
```

The for loop rains on the grid 1 unit per Cell and then calls `updateGrid(_N*_N/2)`. This method of accessing a private data member via an overloaded function is called using a "wrapper." When `updateGrid()` is called in `main()`, it can't access `_N` directly, but the overloaded function can get around that this way.

Suppose `_N = 5` (as it is here.) Then `updateGrid(_N*_N/2) = updateGrid(12)` starts at the 13th Cell in the 5×5 array of Cells (in the middle.) From there it looks at adjacent Cells North (up), East (right), South (down), and West (left), checking first that it's not looking at a boundary Cell (`_grid[neighbor]._elev!=0`), then checking to see if the neighbor Cell has a lower elevation (`_grid[x]._elev>_gr`) and finally checking to see if there's water in the current Cell to flow out. If all three of these conditions are met, then the water level of the current Cell is lowered by 1 and the water level of the neighboring Cell is raised by one.

(e) Suppose the original `g` looks like this:

```
(0, 0)  (0, 0)  (0, 0)  (0, 0) (0, 0)
(0, 0) (45, 1) (40, 1) (45, 1) (0, 0)
(0, 0) (65, 1) (55, 1) (60, 1) (0, 0)
(0, 0) (65, 1) (60, 1) (50, 1) (0, 0)
(0, 0)  (0, 0)  (0, 0)  (0, 0) (0, 0)
```

Then after one `updateGrid()` `g` looks like this:

```
(0, 0)  (0, 1)  (0, 1)  (0, 1) (0, 1)
(0, 1) (45, 2) (40, 3) (45, 2) (0, 1)
(0, 1) (65, 2) (55, 1) (60, 2) (0, 1)
(0, 1) (65, 2) (60, 2) (50, 2) (0, 1)
(0, 1)  (0, 1)  (0, 1)  (0, 1) (0, 0)
```

Give a detailed explanation of how `updateGrid()` produces this change.
Solution: First, each Cell, except for the corner ones, has its waterlevel increased by 1. Then the `updateGrid(12)` calls on the Cell in the middle ((55,2), at this point) and looks North where the Cell (`_grid[7] = (40,2)` at this point) and since it's not a boundary Cell, its elevation is lower a (40¡55) and current Cell has water to spare, it gives up a unit of water level to it making (`_grid[12] = (55,1)` and (`_grid[7] = (40,3)`. Now the E, S, and W directions are checked, but they're all at a higher water level, so `updateGrid(12)` is done.

(f) How might you improve this model for fast flooding?
Here are some suggestions for improvement:

* Have the flow go more uniformly around the current square by choosing direction permutations more equitably (randomly?) and including the NE,NW,SW and SE corners.

  Add waterlevel and elevation to get a combined elevation for comparison.

* Have the starting cell begin not just at the very middle.
* Change the intensity of the rain to be greater in some areas than others.
* Allow for mudflow.
* Include snakes.
* Include wind.
* Use a real topo map for elevations.
* Did I say snakes already?