Polymorphism

class Shape {
  public:
    Shape();
    virtual ~Shape();
    virtual void draw() = 0;
  ...
};

A pure virtual function is a stub. It is you asserting that this function WILL be implemented by all subclasses.
The "function stub" will never be called itself, because it won't be written.

class IncompleteList {
  public:
    void prepend(const int& item);
    void append(const int& item);
    virtual void insert(int n, const int& item) = 0;
  protected:
    int size;
};

IncompleteList::append(const int& item) {
  insert(size, item);
}

You are making a game. The game will involve a hero, which will get its own class. The game will have three
monster types: Instructors, TAs, and CPs. Different monster types are worth differing amounts of points.
Your hero goes around slaying the vile monsters and gaining points as she does so.

Instructor *bosses = new Instructor[x];
TA *minions = new TA[y];
CP *flunkies = new CP[z];

while(true) {
  for(int i = 0; i < x; i++) bosses[i].monsterMove();
  for(int j = 0; j < y; j++) minions[j].monsterMove();
  for(int k = 0; k < z; k++) flunkies[k].monsterMove();
  // ...
}

This is awkward. It would be more convenient to loop over a single array.

Monster **monsters = new Monster*[x+y+z];
for(int i = 0; i < x; i++) monsters[i] = new Instructor();
  //...
while(true) {
  for(int i = 0; i < x+y+z; i++) monsters[i]->monsterMove();//
  // ...
}
Exceptions

```c++
#include<exception>
#include<stdexcept>

... if (position >= this->size()) throw logic_error("position was too large!");
```

A thrown exception will propagate up through the program stack until it reaches a piece of code designed to handle it. If no such code is found, the program terminates.

The user should do this:

```c++
try {
    cout << LL->get(15) << endl;
    cout << "Printed successfully!" << endl;
} catch (logic_error &e) {
    cout << "A logic error occurred!" << endl;
    cout << e.what();
} catch (exception &e) {
    cout << "General exception" << endl;
}
```

Abstract Data Types

- If we are precise about what we want to do (the operations we want to implement), then we have specified an **Abstract Data Type** or ADT.

- A **List** is defined by the following operations, where T denotes any one type (such as int, string, etc).
  1. void insert (int position, T value): inserts value at the specified position, moving all later elements one position to the right.
  2. void remove (int position): removes the value at the specified position, moving all later elements one position to the left.
  3. void set (int position, T value): overwrites the specified position with the given value.
  4. T get (int position): returns the value at the specified position.

- A **Set** (sometimes referred to as a Bag) supports the following:
  1. void add (T item): adds item to the set.
  2. void remove (T item): removes item from the set.
  3. bool contains (T item): determines whether the set contains item.

- A **Map** (sometimes referred to as a Dictionary) associates values with keys. keyType can be any individual data type, as can valueType.
  1. void add (keyType key, valueType value): adds a mapping from key to value.
  2. void remove (keyType key): removes the mapping for key.
  3. valueType get (keyType key): returns the value that key maps to.

- A List cares about order, a map associates keys and values, and a set only determines whether a thing is contained inside or not.
Array Lists

Analyze the runtime analysis for each of the operations of a List, when implemented with a Linked List.

Now instead consider implementing a List with an Array.

**Question 1.** What is the runtime for insert/remove/get on a sorted array? On a sorted linked list?

Stacks and Queues

class QueueADT {
    public:
        void enqueue(const T & data);
        const T & peekfront() const; // look at the oldest element
        void dequeue(); // remove the oldest element
    }

Notice the following:
- Enqueue cannot change the input parameter.
- Whomever called peekfront cannot change the return parameter they received.
- Peekfront cannot change any data members in the Queue (this cannot change).
- Passing a parameter by const reference allows you to avoid copying the input parameter, while promising the user you won’t change their data.

**Question 2.** What data structure should you use to implement a queue? How would you implement it?

**Question 3.** Is there anyway to do this efficiently with an array?

class StackADT {
    void push(const T & data);
    const T & top() const; // look at the newest element
    void pop(); // remove the newest element
}

**Question 4.** What data structure should you use to implement a stack? How would you implement it?