Consider the following code; which print function will be called?

```c++
DeluxeLinkedList *q = new DeluxeLinkedList;
LinkedList *p = q;
p->print();
```

The compiler only knows that p points to an object of type LinkedList. Since it can’t figure out more than this, it will just call the version of print in LinkedList. This is called **static binding**.

The compiler may not know it, but the program DOES know it at runtime. When it gets to the function call, it knows whether the object is of the Deluxe version or not, and can thus call the correct print function. This is called **dynamic binding**; here’s how to get it:

```c++
class LinkedList {
    virtual void print();
};
```

The concept of waiting until runtime to determine which class function to call is referred to as **polymorphism**, meaning “many forms.”
class Shape {
public:
    Shape();
    virtual ~Shape();
    virtual void draw() = 0;
...
};
class Triangle : public Shape { ... };
class Square : public Shape { ... };

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);
}
IncompleteList::prepend(const int& item) {
    insert(0, 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();
    // ...
}