CSCI 104L Lecture 12: Polymorphism and STL Maps

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();
    // ...
}
STL’s map class

```cpp
#include <map>
#include "student.h"
int main() {
    map<string, Student> slist1;
    Student s1("Tommy", 86328);
    Student s2("Jill", 54982);
    ...
    slist1["Tommy"] = s1; // associate the string Tommy with his student record.
    string myname = "Jill";
    slist1[myname] = s2;
    ...
    Student s3 = slist1["Tommy"];  
    slist1.erase("Jill");
    return 0;
}
```

STL’s pair class

```
std::pair<string, int> mypair("Bill", 1);
cout << mypair.first << "\"" << mypair.second << endl;
std::pair<char, double> p2(‘c’, 2.3);
```

STL’s iterator class

```
map<int, string> m;
...
map<int, string>::iterator it;
for (it = m.begin(); it != m.end(); ++it) {
    cout << it->second << endl;
}
```

- The data structure has two public functions: begin(), which returns an iterator at the start of the data, and end() which returns an iterator at the end of the data.
- The iterator is a custom class defined within the scope of the data structure it iterates over. It overloads operator== and operator!= (so you can check if your iterator is at the end), operator++ (so you can get to the next piece of data), and operator* (so you can look at the value the iterator is currently sitting at).
- Every iterator in the STL is implemented in the same manner, so that you can always use an iterator for a data structure, even though you may not understand how the data structure works.
- Think of it like a pointer (it is not a pointer, but it has overloaded operator* to act like one).
- Also, think of the end() function as returning one PAST the end of the data structure, so the above for-loop works properly.