Object-oriented programming in C++

Proficiency: Mastery

Inheritance and polymorphism are fundamental concepts in computer science. Although similar, inheritance and polymorphism vary. Inheritance is a simple idea. Say you made a class, called class A. Then while creating another class, class B, you notice that class A and class B share some similar concepts and maybe some functions in class A could help you class B out. Then any function in class A that is public in class A can be used in class B. Public means variables and functions declared as public are available from anywhere, for other classes and instances of the object. Private narrows the scope so your private variables and functions are visible in its own class only. Another option is protected. Protected is only used with inheritance. Functions and variables that are given a protected scope can only be accessed within the class and to any other classes that extend the base class. An example of us using inheritance is when we implemented a minimum priority queue ADT. We inherited from a MinHeap class and used functions in the MinHeap class in our minimum priority queue. Inheriting saves time because you an simply inherit functions from other classes instead of rewriting code.

Unlike many computer science terms, “polymorphism” means exactly what it does. ‘Poly’ means ‘many’ and ‘morphism’ means ‘to change or form’, So polymorphism is the ability to present the same interface for differing data types. So a polymorphic type has the ability to work with many different types. an example of polymorphism is

generic programming. With generic programming, code is written with no mention of a specific type, therefore it can be used transparently with a variety of types.

Most of our projects this year incorporated inheritance, so a variety of projects would be acceptable to attach. In project 1000 we made a Dictionary ADT that inherited from our red-black tree template class. In this project we used the structure and functions of a red black tree to store dictionary values.

```
// dict.h
// header file for dictoinary class
// Clay Sarafin & Taylor Heilman

#ifndef DICT_H
#define DICT_H

#include <iostream>

#include "rb.h"

using namespace std;

template <class KeyType>
class Dictionary : public RBTree<KeyType>{
        public:
                Dictionary() : RBTree<KeyType>() {};

                using RBTree<KeyType>::get;
                using RBTree<KeyType>::insert;
                using RBTree<KeyType>::count;
                using RBTree<KeyType>::inOrder;
                using RBTree<KeyType>::empty;


                std::string toString();
};

#include "dict.cpp"

#endif
```

```
// dict.cpp
//
// Clay Sarafin & Taylor Heilman

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string>


using namespace std;


/*-------------------------------------------
 * toString()
 * PreConditions:       inOrder from BST class does not have brackets, every entry is followed
 by ", "
                                and does not end with std::endl
                      type KeyType's operator<< is overloaded, and is formatted as "key:valu
e"
 * PostConditions:     string representation of the entire dictionary is printed out
-------------------------------------------*/

template <class KeyType>
std::string Dictionary<KeyType>::toString(){
        std::string str;
        str += "{";
        str += inOrder();
        str += "}";

        return str;
}
```

```
// pq.h
// This MinPriorityQueue template class assumes that the class KeyType has
// overloaded the < operator and the << stream operator.

#ifndef PQ_H
#define PQ_H

#include <iostream>
#include "heap.h"

template <class KeyType>
class MinPriorityQueue : public MinHeap<KeyType>
{
  public:
    MinPriorityQueue();              // default constructor
    MinPriorityQueue(int n);         // construct an empty MPQ with capacity n
    MinPriorityQueue(const MinPriorityQueue<KeyType>& pq);  // copy constructor

    KeyType* minimum() const;                    // return the minimum element
    KeyType* extractMin();                       // delete the minimum element and return it
    void decreaseKey(int index, KeyType* key);   // decrease the value of an element
    void insert(KeyType* key);                   // insert a new element
    bool empty() const;                          // return whether the MPQ is empty
    int length() const;                          // return the number of keys
    std::string toString() const;                // return a string representation of the MPQ
    bool find(KeyType* key);
    string findCode(KeyType* key, int lenght);

    // Specify that MPQ will be referring to the following members of MinHeap<KeyType>.

    using MinHeap<KeyType>::A;
    using MinHeap<KeyType>::heapSize;
    using MinHeap<KeyType>::capacity;
    using MinHeap<KeyType>::parent;
    using MinHeap<KeyType>::swap;
    using MinHeap<KeyType>::heapify;


    /* The using statements are necessary to resolve ambiguity because
       these members do not refer to KeyType.  Alternatively, you could
       use this->heapify(0) or MinHeap<KeyType>::heapify(0).
    */
};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinPriorityQueue<KeyType>& pq);

class FullError { };    // MinPriorityQueue full exception
class EmptyError { };   // MinPriorityQueue empty exception
class KeyError { };     // MinPriorityQueue key exception
class IndexError { };     // MinPriorityQueue key exception

#include "pq.cpp"

#endif
```

88

call MinHeap constructors via initializer lists