|  |  |  |
| --- | --- | --- |
| **American University of Sharjah**  **School of Engineering**  Computer Engineering Department  P. O. Box 26666 Sharjah, UAE |  | **Instructor:** Dr. Ghassan Z. Qadah  **Lab Instructor:** Ms.Praveena Kolli  **Office**: EB2-126  **Phone**: 971-6-515-2352  **e-mail**: pkolli@aus.edu  **Semester**: Spring 2018 |

**CMP305L Data Structures and Algorithms**

**Lab #1 – Review of C++**

**Exercise 1:** Implement the below given Set class,

class Set {

public:

Set (int size=5 ); //default constructor

Set (const Set & s); // copy constructor

~Set ( ); //destructor

int GetSize ( ) const ; // returns the max size of the Set

int GetCount ( ) const// returns the current number of elements within Set

bool InsertElement ( int value); // insert value

bool DeleteElement ( int value); // delete value

int GetNextElement ( ); //returns value at iter

void Reset(); //sets iter to 0;

private:

int eCount // current number of elements within the set.

int max\_size;

int iter; // variable used in reset and getNextElement function for iteration.

int \*arr ; // an array to store the Set elements.

};

Overload the below operators,

1. Assignment operator “=”.
2. A comparison operators “==” and “!=”.
3. Set addition “+” and subtraction “-“ operators.
4. Set addition assignment “+=” and subtraction assignment “- =” operators.
5. Pre and Post increments (++) and decrement s(--)
6. Set insertion “<<” and extraction “>>” operators.

**Hint**: Reset and GetNextElement functionality can tested as below codes,

setObject.Reset();

for (int i=0; i<setObject.GetCount();i++)

cout<<setObject.GetNextElement()<<endl;

**Exercise 2**: Transform the definition and implementation of Set into an equivalent template class for different data types such as integers, floats, doubles, etc. Test your program comprehensively to make sure that it does work with at least two data types, namely, integer and double.

**Note**: Write an appropriate main to test the functionality of each class.

#include<iostream>

using namespace std;

template<class T>

class Set {

public:

Set(int size=5 ); //default constructor

Set(const Set & s); // copy constructor

~Set( ); //destructor

int getSize( ) const; // returns the max size of the Set

int getCount( ) const;// returns the current number of elements within Set

bool insertElement( T value); // insert value

bool deleteElement( T value); // delete value

bool hasElement(T value) const; //checks if value belongs to Set

//Having iter as a field doesn't make sense. Better to remove it.

T getElementAt(int i) const; //returns value at i

//Assignment

Set& operator=(const Set& s);

//Set equality

bool operator==(Set& s) const;

bool operator!=(Set& s) const;

//These are implemented as non-member functions:

//Union

//Set operator+(const Set& s1, const Set& s2);

//Set Difference

//Set operator-(const Set& s1, const Set& s2);

Set& operator+=(Set& s);

Set& operator-=(Set& s);

//What would it mean to increment or decrement a set? Is it a map that increments each element?

Set& operator++();

Set& operator--();

Set operator++(int);

Set operator--(int);

template<class Y>

friend istream& operator>>(istream & cin, Set<Y>& s);

template<class Y>

friend ostream& operator<<(ostream & cout, const Set<Y>& s);

private:

int eCount; // current number of elements within the set.

int max\_size;

T \*arr ; // an array to store the Set elements.

};

template<class T>

Set<T>::Set(int size){

max\_size = size;

arr = new T[max\_size];

eCount = 0;

}

template<class T>

Set<T>::Set(const Set<T> & s){

max\_size = s.getSize();

arr = new T[max\_size];

eCount = s.getCount();

for(int i = 0; i < eCount; ++i){

arr[i] = s.getElementAt(i);

}

}

template<class T>

Set<T>::~Set(){

delete arr;

}

template<class T>

int Set<T>::getSize() const{

return max\_size;

}

template<class T>

int Set<T>::getCount() const{

return eCount;

}

template<class T>

bool Set<T>::insertElement(T value){

if(!hasElement(value)){

if(eCount < max\_size){

arr[eCount++] = value;

return true;

}

else{

return false;

}

}

}

template<class T>

bool Set<T>::deleteElement(T value){

if(!hasElement(value)){

return false;

}else{

//delete element

Set<T> temp(max\_size);

int j = 0;

for(int i = 0; i < eCount-1; ++i){

if(getElementAt(i) != value){

temp.arr[j++] = arr[i];

}

}

}

}

template<class T>

bool Set<T>::hasElement(T value) const{

for(int i = 0; i < eCount; ++i){

//would == fail on floats?

if(arr[i] == value){

return true;

}

}

return false;

}

template<class T>

T Set<T>::getElementAt(int i) const{

return arr[i];

}

template<class T>

Set<T>& Set<T>::operator=(const Set<T>& s){

max\_size = s.getSize();

arr = new T[max\_size];

eCount = s.getCount();

for(int i = 0; i < eCount; ++i){

arr[i] = s.getElementAt(i);

}

return \*this;

}

template<class T>

bool Set<T>::operator==(Set<T>& s) const{

if(s.getCount() != eCount){

return false;

}

for(int i = 0; i < eCount; ++i){

if(!s.hasElement(arr[i])){

return false;

}

}

return true;

}

template<class T>

bool Set<T>::operator!=(Set<T>& s) const{

return !(s==this);

}

template<class T>

Set<T> operator+(const Set<T>& s1, const Set<T>& s2){

Set<T> sum(s1.getSize() + s2.getSize());

for(int i = 0; i < s1.getCount(); ++i){

sum.insertElement(s1.getElementAt(i));

}

for(int i = 0; i < s2.getCount(); ++i){

sum.insertElement(s2.getElementAt(i));

}

return sum;

}

template<class T>

Set<T> operator-(const Set<T>& s1, const Set<T>& s2){

Set<T> difference(s1);

for(int i = 0; i < s2.getCount(); ++i){

if(s1.hasElement(s2.getElementAt(i))){

difference.deleteElement(s2.getElementAt(i));

}

}

return difference;

}

template<class T>

Set<T>& Set<T>::operator++(){

for(int i = 0; i < eCount; ++i){

++arr[i];

}

return \*this;

}

template<class T>

Set<T>& Set<T>::operator--(){

for(int i = 0; i < eCount; ++i){

--arr[i];

}

return \*this;

}

template<class T>

Set<T> Set<T>::operator++(int){

Set<T> temp(\*this);

++(\*this);

return temp;

}

template<class T>

Set<T> Set<T>::operator--(int){

Set temp(this);

--(\*this);

return temp;

}

template<class T>

Set<T>& Set<T>::operator+=(Set<T>& s){

\*this = \*this + s;

return \*this;

}

template<class T>

Set<T>& Set<T>::operator-=(Set<T>& s){

\*this = \*this - s;

return \*this;

}

template<class Y>

istream& operator>>(istream & cin, Set<Y>& s){

cout << "Enter " << s.getSize() << " elements.";

for(int i = 0; i < s.getSize(); ++i){

Y var;

cin >> var;

s.insertElement(var);

}

return cin;

}

template<class Y>

ostream& operator<<(ostream & cout, const Set<Y>& s){

for(int i = 0; i < s.getCount(); ++i){

cout << s.getElementAt(i) << " ";

}

return cout;

}

int main(int argc, char\* argv[]){

Set<double> mySet;

cout << "I have a set of size " << mySet.getSize() << " with " << mySet.getCount() << " elements," << endl;

cin >> mySet;

cout << mySet << endl;

if(mySet.hasElement(1)){

cout << "My set has the element 1" << endl;

}else{

cout << "My set does not have the element 1" << endl;

}

Set<double> Set2(mySet);

Set<double> Set3 = mySet;

if(Set2 == Set3){

cout << "The copy constructor and assignment operator do the same thing" << endl;

}

Set<double> testSet;

cin >> testSet;

cout << "Before union: Set 1 = " << mySet << "\n Set 2 = " << testSet;

Set<double> Set4(mySet.getSize() + testSet.getSize());

Set4 = mySet + testSet;

cout << "After union: " << Set4;

cout << "testSet - mySet = " << testSet - mySet;

cout << "Before incrementing values:" << Set2;

Set2++;

cout << "After incrementing values:" << Set2;

return 0;

}



