Министерство образования Республики Беларусь

Учреждение образования

«Брестский государственный технический университет»

Кафедра ИИТ

Лабораторная работа №4

за 1 семестр

По дисциплине: «ООТПиСП»

Тема: «Шаблоны функций и классов»

Выполнил:

Студент 2 курса

Группы ПО-4(1)

Иваненко И. Л.

Проверил:

Миндер А. В.

2021

Лабораторная работа №4

Шаблоны функций и классов

Цель работы: получить практические навыки навыки создания шаблонов и использования их в программах С++.

Вариант 10

Задание:

1. Создать шаблон заданного класса. Определить конструкторы, деструктор, перегруженную операцию присваивания (“=”) и операции, заданные в варианте задания.

2. Написать программу тестирования, в которой проверяется использование шаблона для стандартных типов данных.

3. Выполнить тестирование.

4. Определить пользовательский класс, который будет использоваться в качестве параметра шаблона. Определить в классе необходимые функции и перегруженные операции.

5. Написать программу тестирования, в которой проверяется использование шаблона для пользовательского типа.

6. Выполнить тестирование.

1) Постановка задачи

10. Класс − множество set. Дополнительно перегрузить следующие операции:

> − проверка на принадлежность (типа операции in множественного типа в языке Pascal);

\* − пересечение множеств;

 < − проверка на подмножество.

2) Код программы

Tree.h:

#pragma once

#include <stack>

template<class T>

class Tree

{

private:

struct Node

{

T data;

Node\* left;

Node\* right;

};

Node\* root;

class TreeIterator

{

private:

std::stack<T> nodes;

void fillUpNodes(Node\* node)

{

if (node == nullptr)

{

return;

}

fillUpNodes(node->right);

nodes.push(node->data);

fillUpNodes(node->left);

}

void removeNodes()

{

while (hasNext())

{

nodes.pop();

}

}

public:

TreeIterator(Node\* node)

{

fillUpNodes(node);

}

bool hasNext()

{

if (nodes.size() > 0)

{

return true;

}

else

{

return false;

}

}

T next()

{

T returnData = nodes.top();

nodes.pop();

return returnData;

}

void toStartOf(Node\* node)

{

removeNodes();

fillUpNodes(node);

}

};

TreeIterator\* treeIterator;

Node\* makeNodeEmpty(Node\* node);

Node\* insert(T data, Node\* node);

Node\* getNodeWithMinData(Node\* node);

Node\* getNodeWithMaxData(Node\* node);

Node\* remove(T data, Node\* node);

bool inTree(T data, Node\* node);

public:

Tree();

~Tree();

void insert(T data);

void remove(T data);

bool inTree(T data);

bool hasNext();

T next();

void toStart();

};

template<class T>

Tree<T>::Tree()

{

root = nullptr;

treeIterator = new TreeIterator(root);

}

template<class T>

Tree<T>::~Tree()

{

root = makeNodeEmpty(root);

delete treeIterator;

}

template<class T>

typename Tree<T>::Node\* Tree<T>::makeNodeEmpty(Tree<T>::Node\* node)

{

if (node == nullptr)

{

return nullptr;

}

else

{

makeNodeEmpty(node->left);

makeNodeEmpty(node->right);

delete node;

}

return nullptr;

}

template<class T>

void Tree<T>::insert(T data)

{

root = insert(data, root);

treeIterator = new TreeIterator(root);

}

template<class T>

typename Tree<T>::Node\* Tree<T>::insert(T data, Tree<T>::Node\* node)

{

if (node == nullptr)

{

node = new Node;

node->data = data;

node->left = nullptr;

node->right = nullptr;

}

else if (data < node->data)

{

node->left = insert(data, node->left);

}

else if (data > node->data)

{

node->right = insert(data, node->right);

}

return node;

}

template<class T>

void Tree<T>::remove(T data)

{

root = remove(data, root);

treeIterator = new TreeIterator(root);

}

template<class T>

typename Tree<T>::Node\* Tree<T>::remove(T data, Tree<T>::Node\* node)

{

Node\* temp;

if (node == nullptr)

{

return nullptr;

}

else if (data < node->data)

{

node->left = remove(data, node->left);

}

else if (data > node->data)

{

node->right = remove(data, node->right);

}

else if (node->left && node->right)

{

temp = getNodeWithMinData(node->right);

node->data = temp->data;

node->right = remove(node->data, node->right);

}

else

{

temp = node;

if (node->left == nullptr)

{

node = node->right;

}

else if (node->right == nullptr)

{

node = node->left;

}

delete temp;

}

return node;

}

template<class T>

typename Tree<T>::Node\* Tree<T>::getNodeWithMinData(Tree<T>::Node\* node)

{

if (node == nullptr)

{

return nullptr;

}

else if (node->left == nullptr)

{

return node;

}

else

{

return getNodeWithMinData(node->left);

}

}

template<class T>

typename Tree<T>::Node\* Tree<T>::getNodeWithMaxData(Tree<T>::Node\* node)

{

if (node == nullptr)

{

return nullptr;

}

else if (node->right == nullptr)

{

return node;

}

else

{

return getNodeWithMaxData(node->right);

}

}

template<class T>

bool Tree<T>::inTree(T data)

{

return inTree(data, root);

}

template<class T>

bool Tree<T>::inTree(T data, Tree<T>::Node\* node)

{

if (node == nullptr)

{

return false;

}

else if (data < node->data)

{

return inTree(data, node->left);

}

else if (data > node->data)

{

return inTree(data, node->right);

}

else

{

return true;

}

}

template<class T>

bool Tree<T>::hasNext()

{

return treeIterator->hasNext();

}

template<class T>

T Tree<T>::next()

{

return treeIterator->next();

}

template<class T>

void Tree<T>::toStart()

{

treeIterator->toStartOf(root);

}

Set.h:

#pragma once

#include "Tree.h"

#include <initializer\_list>

template <class T>

class Set

{

private:

Tree<T>\* tree;

int length;

void insertIfDoesNotExist(T data);

void eraseIfExists(T data);

public:

Set(std::initializer\_list<T> elements);

Set(const Set<T>& other);

Set();

~Set();

void insert(T element);

void erase(T element);

void clear();

int size();

bool operator > (T data);

bool operator < (Set<T>& other);

Set<T>& operator \* (Set<T>& other);

void operator = (std::initializer\_list<T> elements);

Set<T>& operator = (Set<T>& other);

bool hasNext();

T next();

void toStart();

};

template<class T>

Set<T>::Set(const Set<T>& other)

{

tree = other.tree;

length = other.length;

}

template<class T>

Set<T>::Set(std::initializer\_list<T> elements)

{

tree = new Tree<T>();

length = 0;

this->operator=(elements);

}

template<class T>

Set<T>::Set()

{

tree = new Tree<T>();

length = 0;

}

template<class T>

bool Set<T>::operator < (Set<T>& other)

{

while (other.hasNext())

{

if (!(this->operator>(other.next())))

{

return false;

}

}

other.toStart();

return true;

}

template<class T>

bool Set<T>::operator > (T data)

{

return tree->inTree(data);

}

template<class T>

typename Set<T>& Set<T>::operator \* (Set<T>& other)

{

Set<T>\* crossingSet = new Set<T>();

while (other.hasNext())

{

T data = other.next();

if (this->operator>(data))

{

crossingSet->insert(data);

}

}

other.toStart();

return \*crossingSet;

}

template<class T>

void Set<T>::operator = (std::initializer\_list<T> elements)

{

for (auto element : elements)

{

insertIfDoesNotExist(element);

}

}

template<class T>

typename Set<T>& Set<T>::operator = (Set<T>& other)

{

if (this == &other)

{

return \*this;

}

while (other.hasNext())

{

insert(other.next());

}

other.toStart();

return \*this;

}

template<class T>

void Set<T>::insert(T data)

{

insertIfDoesNotExist(data);

}

template<class T>

void Set<T>::insertIfDoesNotExist(T data)

{

if (!this->operator>(data))

{

tree->insert(data);

length++;

}

}

template<class T>

void Set<T>::clear()

{

while (hasNext())

{

erase(next());

}

toStart();

length = 0;

}

template<class T>

void Set<T>::erase(T data)

{

eraseIfExists(data);

}

template<class T>

void Set<T>::eraseIfExists(T data)

{

if (this->operator>(data))

{

tree->remove(data);

length--;

}

}

template<class T>

int Set<T>::size()

{

return length;

}

template<class T>

bool Set<T>::hasNext()

{

return tree->hasNext();

}

template<class T>

T Set<T>::next()

{

return tree->next();

}

template<class T>

void Set<T>::toStart()

{

tree->toStart();

}

template<class T>

Set<T>::~Set()

{

delete tree;

}

Complex.h:

#pragma once

#include <iostream>

#include <string>

class Complex

{

private:

int re;

int im;

public:

Complex();

Complex(int \_re, int \_im);

std::string getComplex();

std::ostream& operator<<(std::ostream& outstream);

bool operator == (const Complex& other);

bool operator != (const Complex& other);

bool operator > (const Complex& other);

bool operator < (const Complex& other);

};

Complex.cpp:

#include "Complex.h"

Complex::Complex()

{

re = 0;

im = 0;

}

Complex::Complex(int \_re, int \_im)

{

re = \_re;

im = \_im;

}

std::string Complex::getComplex()

{

std::string complex;

complex += std::to\_string(re);

if (im >= 0) {

complex += ('+' + std::to\_string(im) + 'i');

}

else {

complex += (std::to\_string(im) + 'i');

}

return complex;

}

std::ostream& Complex::operator<<(std::ostream& outstream)

{

outstream << getComplex();

return outstream;

}

bool Complex::operator==(const Complex& other)

{

return(re == other.re && im == other.im);

}

bool Complex::operator!=(const Complex& other)

{

return !(re == other.re && im == other.im);

}

bool Complex::operator<(const Complex& other)

{

return (re < other.re);

}

bool Complex::operator>(const Complex& other)

{

return (re > other.re);

}

Source.cpp:

#include "Complex.h"

#include "Set.h"

using namespace std;

int main()

{

cout << "========== INT ==========" << endl;

Set<int> set1 = { 4, 2, 7, 11, 0 };

Set<int> set2 = {7, 0, 4, 1, 2, 3, 4, 5, 8, 11 };

Set<int> set3;

set3 = set1;

Set<int> set4 = {-1, 0, 3, 2, 1, 6, 7 };

Set<int> set5 = { 3, 6, 7, 2, 1, 9, 10 };

Set<int> set6 = (set4 \* set5);

cout << "Set1: ";

while (set1.hasNext())

{

cout << set1.next() << " ";

}

set1.toStart();

cout << endl;

cout << "Set2: ";

while (set2.hasNext())

{

cout << set2.next() << " ";

}

set2.toStart();

cout << endl;

cout << "Set3: ";

while (set3.hasNext())

{

cout << set3.next() << " ";

}

set3.toStart();

cout << endl;

cout << "Set4: ";

while (set4.hasNext())

{

cout << set4.next() << " ";

}

set4.toStart();

cout << endl;

cout << "Set5: ";

while (set5.hasNext())

{

cout << set5.next() << " ";

}

set5.toStart();

cout << endl;

cout << "======== OPERATOR '>' ========" << endl;

cout << "Set1 > 0: " << (set1 > 0) << endl << "Set1 > 4: " << (set1 > 4) << endl

<< "Set2 > 3: " << (set2 > 3) << endl << "Set2 > 6: " << (set2 > 6) << endl

<< "Set3 > 2: " << (set3 > 2) << endl << "Set3 > 4: " << (set3 > 4) << endl;

cout << "======== OPERATOR '<' ========" << endl;

cout << "Set1 < Set2: " << (set1 < set2) << endl << "Set2 < Set1: " << (set2 < set1) << endl

<< "Set2 < Set3: " << (set2 < set3) << endl << "Set3 < Set2: " << (set3 < set2) << endl

<< "Set3 < Set1: " << (set3 < set1) << endl << "Set1 < Set3: " << (set1 < set3) << endl;

cout << "======== OPERATOR '\*' ========" << endl;

cout << "Set6(Set4 \* Set5): ";

while (set6.hasNext())

{

cout << set6.next() << " ";

}

set6.toStart();

cout << endl << endl << "========== CHAR ==========" << endl;

Set<char> chset1 = { 'a', 'b', 'e', 'c', 'd' };

Set<char> chset2 = { 'b', 'e', 'a', 'g', 'f', 'h', 'c', 'd' };

Set<char> chset3;

chset3 = chset1;

Set<char> chset4 = { 'x', 'y', 'z', 'm', 'n', 'a' };

Set<char> chset5 = { 'z', 'y', 'x', 'k', 'n', 'f', 'i' };

Set<char> chset6 = (chset4 \* chset5);

cout << "CharSet1: ";

while (chset1.hasNext())

{

cout << chset1.next() << " ";

}

chset1.toStart();

cout << endl;

cout << "CharSet2: ";

while (chset2.hasNext())

{

cout << chset2.next() << " ";

}

chset2.toStart();

cout << endl;

cout << "CharSet3: ";

while (chset3.hasNext())

{

cout << chset3.next() << " ";

}

chset3.toStart();

cout << endl;

cout << "CharSet4: ";

while (chset4.hasNext())

{

cout << chset4.next() << " ";

}

chset4.toStart();

cout << endl;

cout << "CharSet5: ";

while (chset5.hasNext())

{

cout << chset5.next() << " ";

}

chset5.toStart();

cout << endl;

cout << "======== OPERATOR '>' ========" << endl;

cout << "CharSet1 > a: " << (chset1 > 'a') << endl << "CharSet1 > k: " << (chset1 > 'k') << endl

<< "CharSet2 > g: " << (chset2 > 'g') << endl << "CharSet2 > y: " << (chset2 > 'y') << endl

<< "CharSet3 > k: " << (chset3 > 'k') << endl << "CharSet3 > a: " << (chset3 > 'a') << endl;

cout << "======== OPERATOR '<' ========" << endl;

cout << "CharSet1 < CharSet2: " << (chset1 < chset2) << endl << "CharSet2 < CharSet1: " << (chset2 < chset1) << endl

<< "CharSet2 < CharSet3: " << (chset2 < chset3) << endl << "CharSet3 < CharSet2: " << (chset3 < chset2) << endl

<< "CharSet3 < CharSet1: " << (chset3 < chset1) << endl << "CharSet1 < CharSet3: " << (chset1 < chset3) << endl;

cout << "======== OPERATOR '\*' ========" << endl;

cout << "CharSet6(CharSet4 \* CharSet5): ";

while (chset6.hasNext())

{

cout << chset6.next() << " ";

}

chset6.toStart();

cout << endl << endl << "========== COMPLEX ==========" << endl;

Complex complex1(1, 2);

Complex complex2(2, 1);

Complex complex3(3, 4);

Complex complex4(6, 10);

Complex complex5(7, 7);

Complex complex6(5, 3);

Complex complex7(4, 0);

Set<Complex> cmset1 = { complex7, complex1, complex2, complex3, complex6 };

Set<Complex> cmset2 = { complex2, complex3, complex4, complex7, complex1, complex6, complex4, complex5 };

Set<Complex> cmset3;

cmset3 = cmset1;

Set<Complex> cmset4 = (cmset1 \* cmset2);

cout << "ComplexSet1: ";

while (cmset1.hasNext())

{

cout << cmset1.next().getComplex() << " ";

}

cmset1.toStart();

cout << endl;

cout << "ComplexSet2: ";

while (cmset2.hasNext())

{

cout << cmset2.next().getComplex() << " ";

}

cmset2.toStart();

cout << endl;

cout << "ComplexSet3: ";

while (cmset3.hasNext())

{

cout << cmset3.next().getComplex() << " ";

}

cmset3.toStart();

cout << endl;

cout << "======== OPERATOR '>' ========" << endl;

cout << "ComplexSet1 > 1+2i: " << (cmset1 > complex1) << endl << "ComplexSet1 > 7+7i: " << (cmset1 > complex5) << endl

<< "ComplexSet2 > 5+3i: " << (cmset2 > complex6) << endl << "ComplexSet2 > 3+4i: " << (cmset2 > complex3) << endl

<< "ComplexSet3 > 6+10i: " << (cmset3 > complex4) << endl << "ComplexSet3 > 4: " << (cmset3 > complex7) << endl;

cout << "======== OPERATOR '<' ========" << endl;

cout << "ComplexSet1 < ComplexSet2: " << (cmset1 < cmset2) << endl << "ComplexSet2 < ComplexSet1: " << (cmset2 < cmset1) << endl

<< "ComplexSet2 < ComplexSet3: " << (cmset2 < cmset3) << endl << "ComplexSet3 < ComplexSet2: " << (cmset3 < cmset2) << endl

<< "ComplexSet3 < ComplexSet1: " << (cmset3 < cmset1) << endl << "ComplexSet1 < ComplexSet3: " << (cmset1 < cmset3) << endl;

cout << "======== OPERATOR '\*' ========" << endl;

cout << "ComplexSet4(ComplexSet1 \* ComplexSet2): ";

while (cmset4.hasNext())

{

cout << cmset4.next().getComplex() << " ";

}

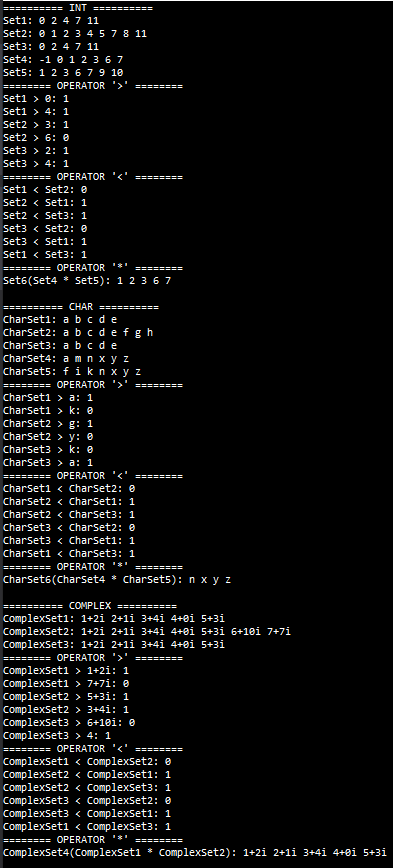
cmset4.toStart();

cout << endl;

return 0;

}

Результат выполнения:



Вывод: получил практические навыки создания шаблонов и использования их в программах С++.