**Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования**

**"Уфимский государственный авиационный технический университет"**

**Кафедра** Высокопроизводительных вычислительных технологий и систем

**Дисциплина:** Программирование

**Отчет по лабораторной работе № 7**

**Тема: «Перегрузка операций»**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Группа ПМИ-148 | Фамилия И.О. | Подпись | Дата | Оценка |
| Студент | Фаизова А.Н. |  |  |  |
| Принял | Гайнетдинова А.А. |  |  |  |

**Уфа 2018**

**Цель:***изучить на практике создание операторов/операций, специфичных для объектов одного класса.*

**Теоретический материал**

Перегрузка операторов

В С++ можно создавать методы со специальными именами вида: operator \*, где \* — это один из существующих операторов языка (например, +, -, /, \*, = и т.д.).

При создании такого метода, говорят, что оператор \* был перегружен, т.е. теперь его можно использовать для взаимодействия с объектами создаваемого класса. Перегружать можно многие, но не любые операторы.

Вызывать этот метод можно обращаясь к нему по указанному символу (в данном случае, \*). При этом аргументом метода будет считаться объект стоящий справа от символа, а вызываться метод будет для объекта, стоящего слева от символа.

Так, например, удобно для объектов своего класса использовать символ + в качестве метода, складывающего объекты, вместо именованного метода (например, sum).

Перегрузка операторов, как и перегрузка функций в целом, реализует принцип полиморфизма.

=!Неявные приведения типа можно подавить, объявив конструктор с ключевым словом explicit. Такой конструктор можно вызывать только явно. В частности, там, где принципиально требуется копирующий конструктор конструктор с модификатором explicit вызываться неявно уже не будет.

=!Функцию-операцию operator[] определяют с целью придать смысл операциям индексации над объектами классов. Второй аргумент этой операции (индекс)может иметь любой тип. Это и позволяет определять такие типы стандартной библиотеки, как вектора, ассоциативные массивы и т.д.

=!Заметьте, что не стали делать доступными все средства из пространства имен std при помощи директивы using namespace std; Вместо этого воспользовались префиксом std: : для string и cout. Таким образом не стали засорять глобальное пространство имен и порождать ненужные зависимости.

=! Между композицией и агрегацией есть одно важное отличие: при использовании композиции, объект-содержимое не может существовать без своего контейнера, а случае агрегации, такое вполне возможно. Да еще и при использовании агрегации, объект-содержимое может принадлежать даже нескольким контейнерам (но тут нужно быть осторожным) или у одного объекта-контейнера, в течении «жизни», может быть разное содержимое (причем, агрегация позволяет использовать даже содержимое разных типов одном у тому же контейнеру).

Класс, например,МОЗГ, подключенный по композиции, является вложенным для класса ЧЕЛОВЕК, и его больше нигде не можем использовать. И в принципе,если мы уничтожаем объект класса ЧЕЛОВЕК, вместе с ним уничтожается и МОЗГ.{Делегирование: метод ДУМАТЬ класса МОЗГ}

Класс КЕПКА жестко к человеку не привязан. Поэтому этот класс мы можем использовать и с другими классами. В этом суть агрегации.

**Ответы на контрольные вопросы**

1. Что такое конструктор копирования и для чего он нужен?

Конструктор копирования позволяет создать новый объект данного класса на основании уже существующего. От простого копирования он отличается тем, что позволяет обработать ситуации, когда по некоторым правилам нельзя создать два абсолютно одинаковых объекта.

1. Что является входным параметром конструктора копирования? Почему входной параметр должен описываться со служебным словом const.

Конструктор копирования - это обычный конструктор, который в качестве входного параметра принимает ссылку на существующий объект:

<имя класса> (const <имя класса><название копируемого объект>)

Конструкторы копирования не должны изменять объект, который он копирует, поэтому *const* является предпочтительным для параметра *other*. Оба будут работать, но *const* является предпочтительным, поскольку в нем четко указано, что переданный объект не должен быть изменен функцией.

1. Какие функции называются дружетсвенными? Как описываются?
2. Для чего необходимы дружественные функции?

Инкапсуляция, как принцип объектно-ориентированного программирования, требует, чтобы компонентные данные (поля класса) были недоступны для прямого изменения в функциях, которые не являются частью класса. В некоторых случаях все-таки возникает необходимость предоставить сторонней функции доступ к полям класса для выполнения некоторых специфических операций. В этом случае функцию необходимо объявить в классе со служебным словом friend. Объявляя функцию таким способом, вы даете ей такие же привилегии, что методам класса.

Дружественная функция объявляется внутри класса, к элементам которого ей нужен доступ, с ключевым словом friend. В качестве параметра ей должен передаваться объект или ссылка на объект класса, поскольку указатель this ей не передается. Одна функция может "дружить" сразу с несколькими классами.

Дружественная функция может быть обычной функцией или методом другого ранее определенного класса. На нее не распространяется действие спецификаторов доступа, место размещения ее объявления в классе безразлично.

1. В чем заключается перегрузка операторов?

Перегрузка операторов в программировании — один из способов реализации полиморфизма, заключающийся в возможности одновременного существования в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

Иногда возникает потребность описывать и применять к созданным программистом типам данных операции, по смыслу эквивалентные уже имеющимся в языке. Классический пример — библиотека для работы с комплексными числами. Они, как и обычные числовые типы, поддерживают арифметические операции, и естественным было бы создать для данного типа операции «плюс», «минус», «умножить», «разделить», обозначив их теми же самыми знаками операций, что и для других числовых типов. Запрет на использование определённых в языке элементов вынуждает создавать множество функций с именами вида ComplexPlusComplex, IntegerPlusComplex, ComplexMinusFloat и так далее.

Когда одинаковые по смыслу операции применяются к операндам различных типов, их вынужденно приходится называть по-разному. Невозможность применять для разных типов функции с одним именем приводит к необходимости выдумывать различные имена для одного и того же, что создаёт путаницу, а может и приводить к ошибкам.

Для перегрузки операции используется специальная форма функции, называемая функцией операции. Функция операции выглядит следующим образом:

Ореrаtоr ор ( список-аргументов )

здесь ор – символ перегружаемой операции. Например, operator+ ( ) перегружает операцию +, а ореrаtоr\* ( ) – операцию \*.

В C++ можно выделить четыре типа перегрузок операторов:

* Перегрузка обычных операторов + - \* / % ˆ & | ~ ! = <> += -= \*= /=%= ˆ= &= |= <<>>>>= <<= == != <= >= && || ++ -- , ->\* -> ( ) [ ]
* Перегрузка операторов преобразования типа
* Перегрузка операторов аллокации и деаллокации new и delete
* Перегрузка литералов operator""

1. Какие операторы нельзя перегружать?

Нельзя перегружать операции:

:: (разрешение области видимости;),

. (выбор члена класса;) и

.\* (выбор члена класса через указатель на классовые члены;).

Тернарная условная операция ?: также запрещена к перегрузке. Также нельзя перегружать именованные операции sizeofи typeid.

Невозможно использовать не существующие в языке C++ знаки операций, но можно воспользоваться функциями в случаях, когда существующие знаки операций не подходят.

1. Можно ли создать свой собственный оператор?

К сожалению, свои собственные операторы в C++ создавать нельзя, а можно только перегружать существующие (не все, но многие). Такое ограничение вызвано прежде всего тем, что синтаксический анализатор языка с созданием каждого нового оператора пришлось бы тоже учить как-то его распознавать (проверять корректность использования и т.д.).

1. В каких случаях перегрузка оператора может осуществляться только при помощи дружественной функции? Приведите примеры.

1) Когда мы реализуем интерфейс (.h файл) в который помещаются только объявления методов, а реализация выносится в скрытый .dll файл

// Перегрузка оператора потокового ввода/вывода

/\*\*Друж. фун-я для вывода получает ссылку на поток

и ссылку на переменную, которую нужно вывести\*\*/

friend std::ostream& operator<<(std::ostream& s\_out, const CHousing& OutNum);

/\*\* Возвращаем ссылку на поток ввода. Оператор ввода в поток

получает два параметра: ссылку на поток ввода

и ссылку на элемент, который нужно выводить \*\*/

friend std::istream& operator>>(std::istream& s\_in, CHousing& InNum);

2) Когда операция производится над объектами разных классов. Пример:

structVector2

{

//СкладываемVector2 иVector3

Vector2 operator + (Vector3 v3) {/\*...\*/}

}

//Объекту Vector2 присваиваем сумму объектов Vector2 и Vector3

vec2 = vec2 + vec3; //Ok

vec2 = vec3 + vec2; //Ошибка

Ошибка произойдет по следующей причине, в структуре Vector2 мы перегрузили оператор +, который в качестве значения справа принимает тип Vector3, поэтому первый вариант работает. Но во втором случае, необходимо писать перегрузку уже для структуры Vector3, а не 2. Чтобы не лезть в реализацию класса Vector3, мы можем написать такую дружественную функцию:

struct Vector2

//Складываем Vector2 и Vector3

Vector2 operator + (Vector3 v3) {/\*...\*/}

//Дружественность необходима для того, чтобы мы имели доступ к приватным членам класса Vector3

friend Vector2 operator + (Vector3 v3, Vector2 v2) {/\*...\*/}

}

vec2 = vec2 + vec3; //Ok

vec2 = vec3 + vec2; //Ok

Функция ввода и функция вывода не могут быть членами класса, потому что если функция operator является членом класса, её левый операнд (передаваемый неявно посредством указателя this) должен быть членом этого класса, который вызывает эту функцию.

1. Почему при перегрузке операторов ввода из потока и вывода в поток необходимо возвращать ссылку на поток?

Таким образом, возвращаем ссылку на поток (ostream/istream) для сцепленного вызова. То есть это даёт нам возможность использовать оператор ввода/вывода несколько раз. Это как cout.

cout<<a<<b<<c<<endl;

Мы 4 раза использовали операцию "поместить в поток"(<<) - это и есть сцепленный вызов

Перегруженный оператор возвращает ссылку на исходный объект ostream, что позволяет объединять вставки:

cout << "The date is" << dt << flush;

**Индивидуальное задание №1**

1. Разработать полный класс **Многочлен**, содержащий необходимые

функции конструктора и деструктора, а также функции get и set. Класс

содержит поле, хранящее степень многочлена, и массив коэффициентов при

членах. Например, многочлен имеет степень 4 и массивкоэффициентов {0.,0.,1.,0.,2.}.

Определить для класса конструктор копирования, перегрузитьстандартные операции:

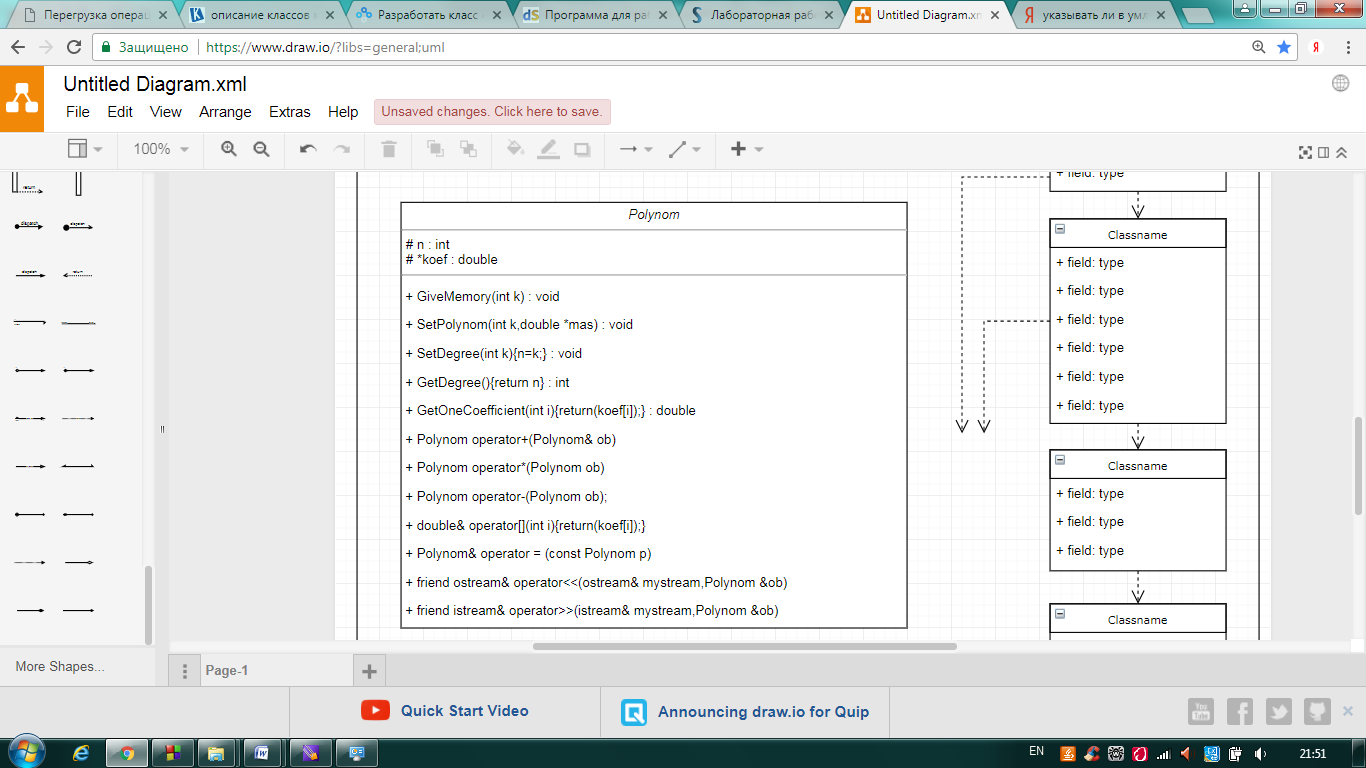
- присваивания,

- вывода в поток и ввода из потока,

- сложения и вычитания двух многочленов.

Создать объекты данного класса в функции main() и продемонстрировать работу всех перегруженных операций.

Блок-схема



Исходный код программы

**Polynom.h**

#ifndefPOLYNOM\_H

#define POLYNOM\_H

#include <iostream>

#include <conio.h>

#include <math.h>

#include <stdlib.h>

using namespace std;

class Polynom {

public:

Polynom(); //конструкторы

Polynom(int k);

Polynom(int k,double \*mas);

Polynom(constPolynom&ob); //конструктор копирования

~Polynom(){delete[]koef;}

voidGiveMemory(intk); //освобождает память под новый полином

void SetPolynom(int k,double \*mas);

void SetDegree(int k){n=k;}; //установить степень

int GetDegree(){return n;}; //получить степень

double GetCoefficient(int i){return(koef[i]);};

Polynom operator+(Polynom& ob); //перегрузка операторов

Polynom operator\*(Polynom ob);

Polynom operator-(Polynom ob);

double& operator[](int i){return(koef[i]);}//перегрузка []

Polynom& operator = (const Polynom p) {

if(&p==this) return \*this;

if(koef) delete [] koef;

n=p.n;

koef=new double [p.n+1];

for(int i=0;i<=p.n;i++)

koef[i]=p.koef[i];

return \*this;

}

friend ostream& operator<<(ostream& mystream,Polynom &ob);

friend istream& operator>>(istream& mystream,Polynom &ob);

int min(int n,int m)

{return (n<m)? n:m; }

int max(int n,int m)

{return (n>m)? n:m; }

/\*\*

Binary function that accepts two values of type T as arguments, and returns a value convertible to bool. The value returned indicates whether the element passed as first argument is considered less than the second.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

\*\*/

protected:

intn; // степеньмногочлена

double \*koef; // коэффициентымногочлена

private:

};

#endif // POLYNOM\_H

**Polynom.cpp**

#include "Polynom.h"

#include <iostream>

#include <conio.h>

#include <math.h>

#include <stdlib.h>

using namespace std;

Polynom::Polynom()

{

n=rand()%5;

koef=newdouble[n+1]; // Выделяемпамятьдляn+1-элементногомассива.

for(int i=n;i>=0;i--)

koef[i]=rand()%20;

}

Polynom::Polynom(int k)

{ n=k;

koef=new double[n+1];

for(int i=n;i>=0;i--)

koef[i]=rand()%20;

}

Polynom::Polynom(int k,double mas[])

{n=k;

koef=newdouble[n+1]; // выделение памяти под массив

for(int i=n;i>=0;i--)

koef[i]=mas[i];

}

Polynom::Polynom(const Polynom&ob)

{n=ob.n;

koef=new double[n+1];

for(int i=0;i<=n;i++)

koef[i]=ob.koef[i];

}

void Polynom::GiveMemory(int k)

{

if(koef) delete [] koef;

koef=newdouble[k+1];

}

/\*\*

C++ поддерживает динамическое выделение и освобождение памяти объектов с использованием операторов new и delete. Эти операторы выделяют память для объектов из пула, называемого свободным хранилищем. Оператор new вызывает специальную функцию operatornew, а оператор delete вызывает специальную функцию operatordelete.

\*\*/

/\*\*Удаляем весь массив, «delete[]» получает только адрес первого элемента. определяет адрес начала блока (вычитая ровно столько же, сколько было прибавлено при выполнении «new[]»), читает число элементов из начала блока, вызывает нужное число деструкторов, затем – вызывает функцию «operator delete[]()», передав ей адрес начала блока.\*\*/

void Polynom::SetPolynom(int k,double \*mas)

{ n=k;

if(koef) delete [] koef;

koef = new double [n+1];

for(int i=n;i>=0;i--)

koef[i]=mas[i];

}

Polynom Polynom::operator+(Polynom& ob)

{ int i;

Polynom rab;

rab.GiveMemory(max(n,ob.GetDegree()));

for(i=0;i<=min(n,ob.GetDegree());i++)

rab.koef[i]=koef[i]+ob.GetCoefficient(i);

if(n<ob.GetDegree())

{

for(i=min(n,ob.GetDegree())+1;i<=ob.GetDegree();i++)

{rab.koef[i]=ob.GetCoefficient(i);}

rab.n=ob.GetDegree();

}

else

{

for(i=min(n,ob.GetDegree())+1;i<=n;i++) rab.koef[i]=koef[i];

rab.n=n;

}

return rab;

}

Polynom Polynom::operator-(Polynom ob)

{ int i;

Polynom rab;

rab.GiveMemory(max(n,ob.GetDegree()));

for(i=0;i<=min(n,ob.GetDegree());i++)

rab.koef[i]=koef[i]-ob.GetCoefficient(i);

if(n<ob.GetDegree())

{

for(i=min(n,ob.GetDegree())+1;i<=ob.GetDegree();i++)

{rab.koef[i]=-ob.GetCoefficient(i);}

rab.n=ob.GetDegree();

}

else

{

for(i=min(n,ob.GetDegree())+1;i<=n;i++) rab.koef[i]=koef[i];

rab.n=n;

}

return rab;

}

// Перегрузка оператора потокового ввода/вывода

/\*\*Друж. фун-я для вывода получает ссылку на поток

и ссылку на переменную, которую нужно вывести\*\*/

ostream& operator<<(ostream& mystream,Polynom &ob)

{ charc=' '; //пропустим “+” перед первым коэффициентом

for(int i=ob.n;i>=0;i--)

{ double ai=ob.koef[i];

if(ai==0) continue;

else {if(ai>0) mystream<<c; mystream<<ai;}

if(i==0) continue; else mystream<<"x";

if(i==1) continue; else mystream<<"^"<<i;

if(ai!=0)c='+';

}

if(c==' ')mystream<<0;

mystream<<endl;

return mystream;

}

/\*\* Возвращаемссылкунапотокввода. Оператор ввода в поток

получает два параметра: ссылку на поток ввода

и ссылку на элемент, который нужно выводить \*\*/

istream& operator>>(istream& mystream,Polynom &ob)

{

int i;

cout<<"Enter Degree:"; mystream>>ob.n; cout<<endl;

for(i=ob.n;i>=0;i--)

{

cout<<"Enter Coefficient "<<i<<":"; mystream>>ob.koef[i];

}

return mystream;

}

**Main.cpp**

#include <iostream>

#include <conio.h>

#include <math.h>

#include <stdlib.h>

#include <Polynom.h>

int main()

{

Polynom f,g,\*G;

Polynom F;

std::cout << " Default constructor\n F(x) = " << F;

int n=5;

double C[n+1]={1.0,3.2,0.0,4.1,0.0,1.1};

G=new Polynom(n,C);

std::cout<< " Fully populated constructor\n G(x) = " << \*G;

std::cout << "\n======================================\n";

std::cout<<" If you want to calculate the sum and difference of polynomials:\n ";

std::cout << " First polynomial\n";

std::cin>>f;

Polynom f1(&f); // Calling the copy constructor

std::cout<<" f(x)= "; std::cout<<f;

std::cout << "\*Calling the copy constructor\n";

std::cout<<" f1(x)= "; std::cout<<f1;

std::cout << " Second polynomial\n";

Polynom h;

std::cin>>h;

Polynom sum=f+h;

std::cout<<"f(x)+h(x) = "; std::cout<<sum;

Polynom dif=f-h;

std::cout<<"f(x)-h(x) = "; std::cout<<dif;

Polynom s=f+\*G;

std::cout<<"f(x)+G(x) = "; std::cout<<s;

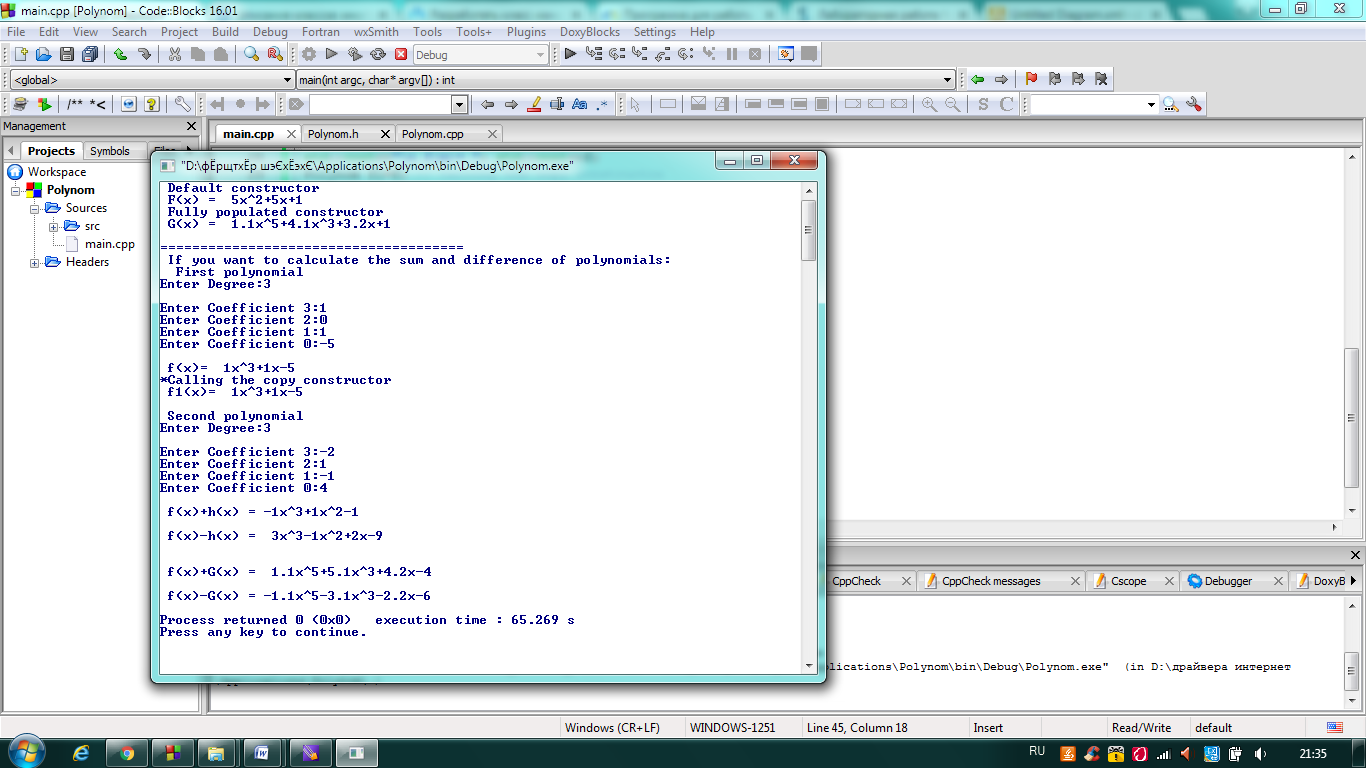
Polynom r=f-\*G;

std::cout<<"f(x)-G(x) = "; std::cout<<r;

return 0;

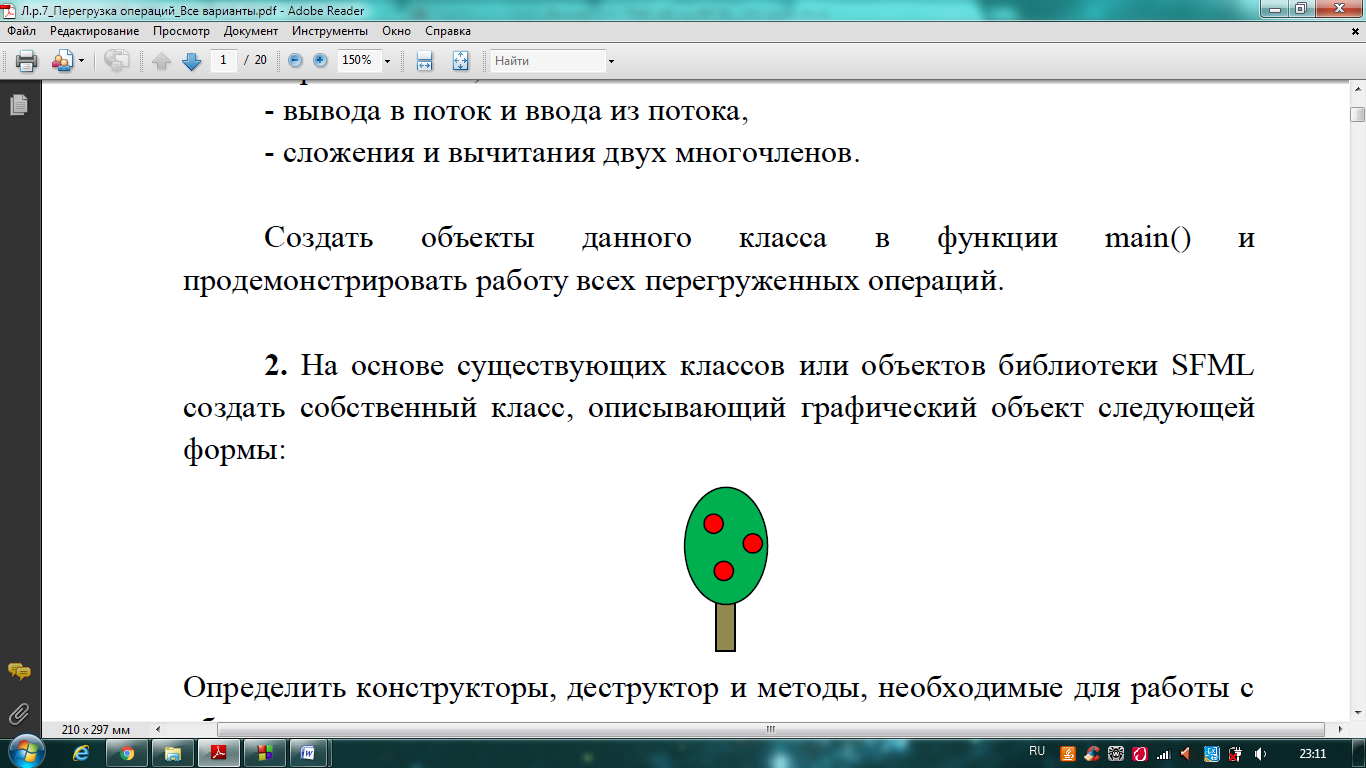
}

Пример выполнения программы



**Индивидуальноезадание №2**

2. На основе существующих классов или объектов библиотеки SFMLсоздать собственный класс, описывающий графический объект следующейформы:



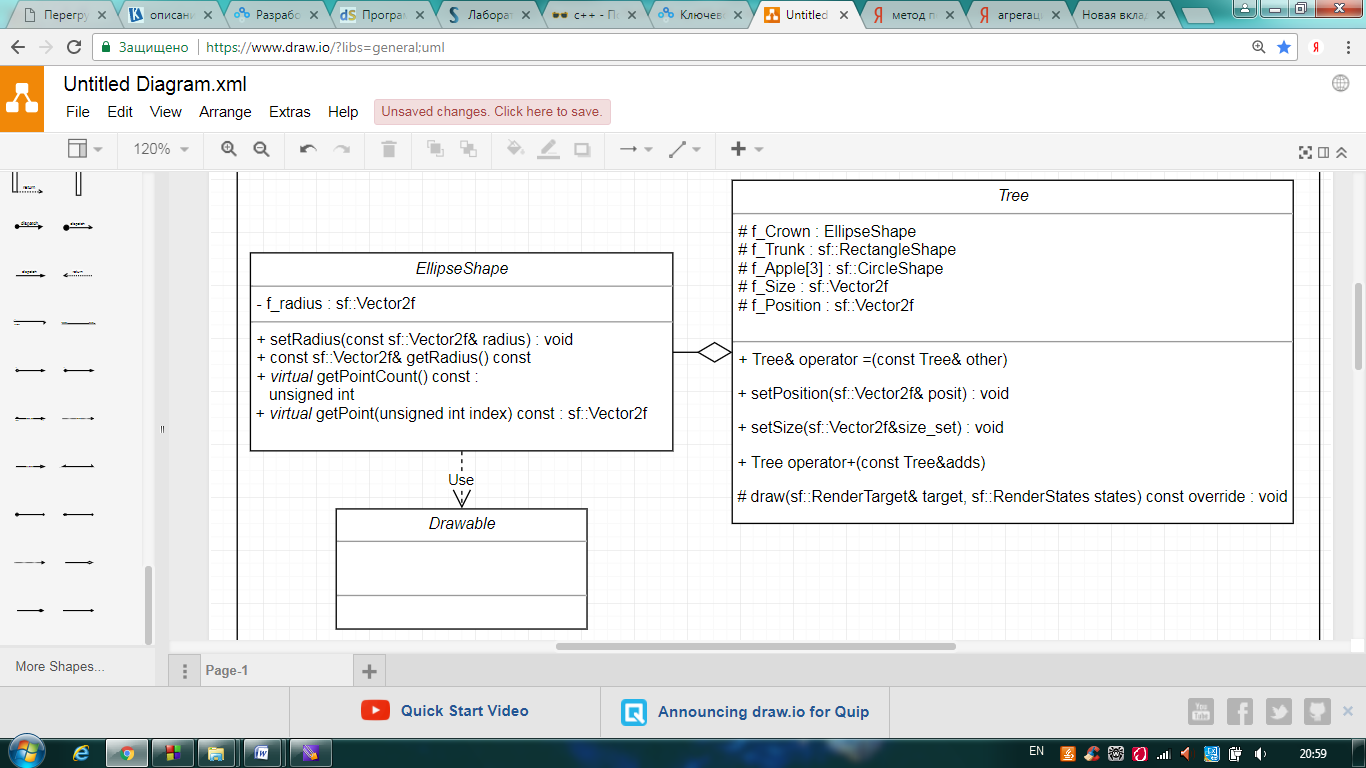
Определить конструкторы, деструктор и методы, необходимые для работы собъектом.

Перегрузить для созданного класса операцию сложения двух объектов,

результатом которой будет объект, координаты и размеры которого естьсреднее арифметическое соответствующих полей двух операндов.

Создать программу, демонстрирующую работу перегруженнойоперации.

Блок-схема



Исходный код программы

**EllipseShape.h**

#ifndefELLIPSESHAPE\_H

#define ELLIPSESHAPE\_H

#include <SFML/Graphics.hpp>

#include <cmath>

#include <time.h>

class EllipseShape : public sf::Shape

{

public :

explicit EllipseShape(const sf::Vector2f& radius = sf::Vector2f(0, 0)) :

f\_radius(radius)

{

update();

}

void setRadius(const sf::Vector2f& radius)

{

f\_radius = radius;

update();

}

const sf::Vector2f& getRadius() const

{

return f\_radius;

}

virtual unsigned int getPointCount() const

{

return 90;

}

virtual sf::Vector2f getPoint(unsigned int index) const

{

static const float pi = 3.141592654f;

float angle = index \* 2 \* pi / getPointCount() - pi / 2;

float x = std::cos(angle) \* f\_radius.x;

float y = std::sin(angle) \* f\_radius.y;

return sf::Vector2f(f\_radius.x + x, f\_radius.y + y);

}

private :

sf::Vector2f f\_radius;

};

#endif // ELLIPSESHAPE\_H

**Tree.h**

#ifndef TREE\_H

#define TREE\_H

#include <EllipseShape.h>

#include <SFML/Graphics.hpp>

class Tree : public sf::Drawable

{

public:

Tree();

virtual ~Tree();

Tree(const Tree& other);

Tree(const sf::Vector2f& position, const sf::Vector2f& size);

Tree& operator =(const Tree& other);

void setPosition(sf::Vector2f& posit);

void setSize(sf::Vector2f& size\_set);

Tree operator+(const Tree& adds) ;

protected:

void draw(sf::RenderTarget& target, sf::RenderStates states) const override;/\*\*идентификатор overrideдобавлендляуказаниятого, чтометодяв-сяпереопределениемвирт.методавбазовомклассе \*\*/

EllipseShape f\_Crown;

sf::RectangleShape f\_Trunk;

sf::CircleShape f\_Apple[3];

sf::Vector2f f\_Size;

sf::Vector2f f\_Position;

private:

};

#endif // TREE\_H

**Tree.cpp**

#include "Tree.h"

#include <SFML/Graphics.hpp>

#include <EllipseShape.h>

Tree::Tree()

{

f\_Size = {100,250};

f\_Position = {10,4};

sf::Vector2f razmer (f\_Size.x, f\_Size.y / 2.f);

f\_Crown.setRadius(razmer);

f\_Crown.setPosition(f\_Position);

f\_Crown.setFillColor(sf::Color::Green);

f\_Crown.setOutlineThickness(2);

f\_Crown.setOutlineColor(sf::Color::Black);

f\_Trunk.setSize({0.5\*razmer.x, 1.25\*razmer.y});

f\_Trunk.setPosition(f\_Position + sf::Vector2f{ 0.75\*razmer.x, 2\*razmer.y - 0.1\*razmer.y});

f\_Trunk.setFillColor(sf::Color(128,64,0,255));

f\_Trunk.setOutlineThickness(2);

f\_Trunk.setOutlineColor(sf::Color::Black);

for (int i = 0 ; i < 3 ; i++)

{

f\_Apple[i].setRadius(f\_Size.x / 6.f);

f\_Apple[i].setFillColor(sf::Color::Red);

f\_Apple[i].setOutlineThickness(1);

f\_Apple[i].setOutlineColor(sf::Color::Black);

}

f\_Apple[0].setPosition(f\_Position + sf::Vector2f{0.42\*f\_Size.x, 0.3\*f\_Size.y});

f\_Apple[1].setPosition(f\_Position + sf::Vector2f{0.77\*f\_Size.x, 0.6\*f\_Size.y});

f\_Apple[2].setPosition(f\_Position + sf::Vector2f{1.17\*f\_Size.x, 0.4\*f\_Size.y});

}

Tree::~Tree()

{

//dtor

}

Tree::Tree(const Tree& other)

{

f\_Size = other.f\_Size + sf::Vector2f{40,70};

f\_Position = other.f\_Position + sf::Vector2f{400,100};

sf::Vector2f razmer (f\_Size.x, f\_Size.y / 2.f);

f\_Crown.setRadius(razmer);

f\_Crown.setPosition(f\_Position);

f\_Crown.setFillColor(sf::Color::Green);

f\_Crown.setOutlineThickness(2);

f\_Crown.setOutlineColor(sf::Color::Black);

f\_Trunk.setSize({0.5\*razmer.x, 1.25\*razmer.y});

f\_Trunk.setPosition(f\_Position + sf::Vector2f{ 0.75\*razmer.x, 2\*razmer.y - 0.1\*razmer.y});

f\_Trunk.setFillColor(sf::Color(128,64,0,255));

f\_Trunk.setOutlineThickness(2);

f\_Trunk.setOutlineColor(sf::Color::Black);

for (int i = 0 ; i < 3 ; i++)

{

f\_Apple[i].setRadius(f\_Size.x / 6.f);

f\_Apple[i].setFillColor(sf::Color::Red);

f\_Apple[i].setOutlineThickness(1);

f\_Apple[i].setOutlineColor(sf::Color::Black);

}

f\_Apple[0].setPosition(f\_Position + sf::Vector2f{0.42\*f\_Size.x, 0.3\*f\_Size.y});

f\_Apple[1].setPosition(f\_Position + sf::Vector2f{0.77\*f\_Size.x, 0.6\*f\_Size.y});

f\_Apple[2].setPosition(f\_Position + sf::Vector2f{1.17\*f\_Size.x, 0.4\*f\_Size.y});

}

Tree::Tree(const sf::Vector2f& position, const sf::Vector2f& sie)

{

sf::Vector2f razmer (sie.x, sie.y / 2.f);

f\_Crown.setRadius(razmer);

f\_Crown.setPosition(position);

f\_Crown.setFillColor(sf::Color::Green);

f\_Crown.setOutlineThickness(2);

f\_Crown.setOutlineColor(sf::Color::Black);

f\_Trunk.setSize({0.5\*razmer.x, 1.25\*razmer.y});

f\_Trunk.setPosition(position + sf::Vector2f{ 0.75\*razmer.x, 2\*razmer.y - 0.1\*razmer.y});

f\_Trunk.setFillColor(sf::Color(128,64,0,255));

f\_Trunk.setOutlineThickness(2);

f\_Trunk.setOutlineColor(sf::Color::Black);

for (int i = 0 ; i < 3 ; i++)

{

f\_Apple[i].setRadius(sie.x / 6.f);

f\_Apple[i].setFillColor(sf::Color::Red);

f\_Apple[i].setOutlineThickness(1);

f\_Apple[i].setOutlineColor(sf::Color::Black);

}

f\_Apple[0].setPosition(position + sf::Vector2f{0.42\*sie.x, 0.3\*sie.y});

f\_Apple[1].setPosition(position + sf::Vector2f{0.77\* sie.x, 0.6\* sie.y});

f\_Apple[2].setPosition(position + sf::Vector2f{1.17\* sie.x, 0.4\* sie.y});

}

void Tree::setSize(sf::Vector2f& size\_set)

{

f\_Size = size\_set;

sf::Vector2f razmer (size\_set.x, size\_set.y / 2.f);

f\_Crown.setRadius(razmer);

f\_Trunk.setSize({0.5\*razmer.x, 1.25\*razmer.y});

for (int i=0; i<3; i++)

{

f\_Apple[i].setRadius(size\_set.x / 6.f);

}

}

void Tree::setPosition(sf::Vector2f& posit)

{

sf::Vector2f razmer (f\_Size.x, f\_Size.y / 2.f);

f\_Crown.setPosition(posit);

f\_Trunk.setPosition(posit + sf::Vector2f{ 0.75\*razmer.x, 2\*razmer.y - 0.1\*razmer.y});

f\_Apple[0].setPosition(posit + sf::Vector2f{0.42\*f\_Size.x, 0.3\*f\_Size.y});

f\_Apple[1].setPosition(posit + sf::Vector2f{0.77\*f\_Size.x, 0.6\*f\_Size.y});

f\_Apple[2].setPosition(posit + sf::Vector2f{1.17\*f\_Size.x, 0.4\*f\_Size.y});

}

Tree& Tree::operator=(const Tree& other)

{

if (this == &other) return \*this;

return \*this;

}

Tree Tree::operator+(const Tree& adds)

{

Tree result;

sf::Vector2f vec;

vec.x=(f\_Position.x+adds.f\_Position.x)/2.f;

vec.y=(f\_Position.y+adds.f\_Position.y)/2.f;

sf::Vector2f size\_mid;

size\_mid.x=(f\_Size.x + adds.f\_Size.x)/2.f;

size\_mid.y=(f\_Size.y + adds.f\_Size.y)/2.f;

result.setSize(size\_mid);

result.setPosition(vec);

return result;

}

**Main.cpp**

#include <SFML/Graphics.hpp>

#include <iostream>

#include <time.h>

#include <EllipseShape.h>

#include <Tree.h>

void Tree::draw(sf::RenderTarget& target, sf::RenderStates states) const

{

target.draw(f\_Trunk, states);

target.draw(f\_Crown, states);

for (int i = 0; i<3; i++)

{

target.draw(f\_Apple[i], states);

}

}

// Функция создаёт окно определённого размера с определённым заголовком.

void initWindow(sf::RenderWindow& window)

{

sf::VideoMode videoMode(800, 600);

const std::string title = "Tree + class derived from sf::Drawable";

sf::ContextSettings settings;

settings.antialiasingLevel = 8;

window.create(videoMode,title, sf::Style::Default, settings);

}

int main()

{

sf::RenderWindow window;

initWindow(window);

Tree tree;

Tree derevo=tree;

Tree apple\_tree=tree+derevo;

while (window.isOpen())

{

sf::Event event;

while (window.pollEvent(event))

{

if (event.type == sf::Event::Closed)

{

window.close();

}

}

// Рисуем tree как обычную фигуру: вызовом window.draw.

window.clear(sf::Color(240,240,255,255));

window.draw(tree);

window.draw(derevo);

window.draw(apple\_tree);

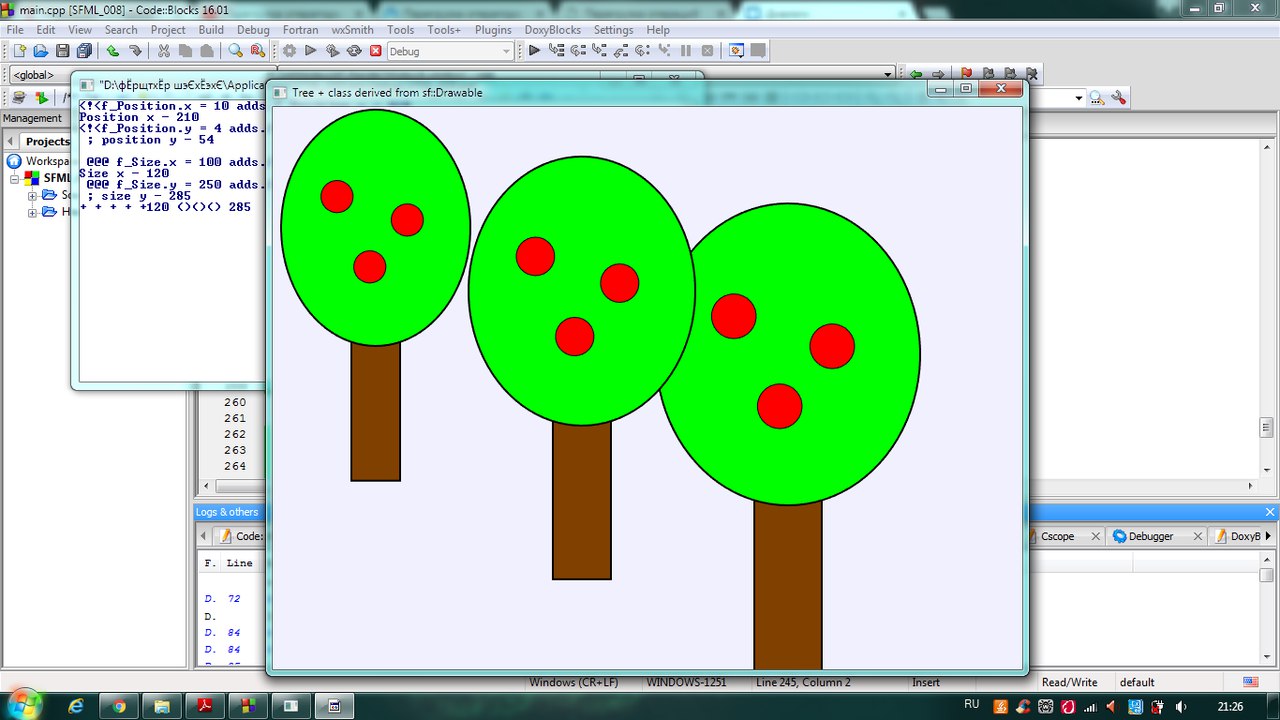
window.display();

}

return 0;

}

Пример выполнения программы



**Вывод**

На выполненной лабораторной работе мы изучили создание операторов/операций, специфичных для объектов одного класса

**Список использованной литературы**

1. Перегрузка операций[Электронный ресурс] / БикмеевА. Т. // <http://bikmeyev-at.ugatu.su/students/CPP/LR203/index17.html> (Дата обращения 11.04.2018).
2. Бьерн Страуструп. Язык программирования C++. Специальное издание. Пер. с англ. — М.: Издательство Бином, 2011 г. — 1136 с: ил.
3. Алгоритм: Вернуть наименьший/наибольшийэлемент[Электронный ресурс]

<http://www.cplusplus.com/reference/algorithm/min/>

(Дата обращения 10.04.2018).