Оглавление

Предисловие

В цикл входят 6 работ, затрагивающих основные концепции ООП: инкапсуляцию (1-я работа), наследование и виртуальные функции (2-я и 3-я). В 4-й рассматривается использование в объектно-ориентированных программах контейнеров библиотеки STL. Рекомендуемый язык программирования при выполнении этих работ – C++, рекомендуемая система программирования – MS Visual Studio .Net.

Пятая работа выполняется в Visual Studio .Net на языке C#. Целью ее включения в цикл является демонстрация того, что основные концепции ООП поддерживаются в разных языках похожими средствами.

В последней, 6-й работе студенты знакомятся с основами построения графического интерфейса.

В каждой работе приведены краткие сведения, листинги программ, варианты заданий и контрольные вопросы. Для ответа на последние достаточно изучить приведенные сведения или запустить соответствующую программу.

В приложениях приведены примеры UML - диаграмм классов, краткие сведения по обработке исключений и пример проектирования класса с полями-указателями.

Приведенные в работах программы выполнены на Visual Studio .Net 2022, а UML-диаграммы – на свободно распространяемом case-средстве Umbrello.

Защита работ и оформление отчета

Содержание отчета :

1. Задание.
2. Спецификация класса (ов) с комментариями (см. далее).
3. Используемые математические зависимости и алгоритмы, например, вычисления дня недели по дате или алгоритма пересечения фигур.
4. Диаграмма классов
5. Листинг программы.

Этот перечень обязателен для работ на C++. Для C# спецификация классов не предусмотрена.

Комментарии должны быть на русском языке короткими, но содержательными. Не имеет смысла комментарий к конструктору по умолчанию или к такому же деструктору.

Обязательны пояснения для кода, реализующего сложные алгоритмы (сравнение, включение, пересечение в третьей работе); можно привести математическую или словесную формулировку или ссылку на соответствующую литературу.

На диаграмме классов должны присутствовать все классы, имеющиеся в программе, и все они должны быть связаны между собой. Если на диаграмме есть какой-либо изолированный класс, то можно сделать вывод, что он не нужен в программе.

Постановка задачи и основные детали ее реализации должны быть понятны без обращения к коду программы.

На последней странице работы должны быть приведены контрольные вопросы, знать ответы на которые обязан каждый студент.

Вопрос о форме отчета – бумажный/электронный – решается преподавателем.

По умолчанию перед началом защиты оценка работы и знаний составляет 100%. В процессе защиты она уменьшается на:

- 100% за непонимание содержания предъявляемой работы;
- 50% за нечеткие ответы на вопросы по коду программы, что может свидетельствовать о несамостоятельности ее выполнения;
- 5% за неправильный ответ на один контрольный вопрос;
- 10% за отсутствие комментариев или диаграммы классов;
- до 10% за орфографические ошибки в русском тексте. (Наличие ошибок является свидетельством либо лени, либо небрежности; и то, и другое преследуется.)
  При наличие дробной величины штрафа его величина округляется до 0,5 или 1.

Работа принимается, если студент набрал не менее 60% баллов.

В первом полусеместре необходимо сдать 3 первых лабораторных работы. Штраф за опоздание – 10%.

Максимальная оценка работы, защищаемой после окончания семестра, 75%.

Ограничение на количество работ, предлагаемых студентом к защите в течение одного занятия – 2.

Чисто технические сведения, не имеющие отношения к проектированию программ (римская система счисления, площади и объемы геометрических фигур и др.) добываются студентами самостоятельно.

Замечания по приему работы.

1. Программа должна быть правильно спроектирована; критерий правильности определяет преподаватель. Аргумент студента “Но она же работает” не является достаточным.
2. Обязательным в проектах на C++ должно быть наличие средства определения утечек памяти, в частности, утилиты Visual Leak Detector

https://pro-prof.com/forums/topic/cpp_memory_leaks

Лабораторная работа №1. Классы и объекты в C++

Цель работы

Ознакомление с технологией построения классов, разработки методов, перегрузки операций и обработки исключений.

Основные сведения

Классы и объекты

Базовыми понятиями объектно-ориентированной программы являются «объект класса» и «класс». Эти понятия аналогичны понятиям переменной и её типа. Членами класса являются переменные базовых типов языка, объекты классов, объявленных в программе (последние называют пользовательскими типами), и функции, работающие с этими переменными. Обычно переменные, определенные в классе, называют полями или данными, а функции – методами класса. Говорят, что значения полей объекта класса определяют его состояние, а методы класса – поведение. В C++ имеется возможность управлять доступом к полям и методам класса, в частности, установление такого порядка, при котором доступ к полям класса имеют только его методы. Объединение данных и методов работы с ними в одну языковую конструкцию с возможностью управления доступом к данным называется инкапсуляцией.

Тип класса может быть задан одним из 3-х ключевых слов: `class, struct, union`. Имя класса становится идентификатором нового типа данных. Определение класса выглядит следующим образом:

```
class Alpha { // Начало определения
// Поля класса
int _x;
double _y;
//Конструктор
Alpha (){_x=0; _y=0;}
```

```
//Более распространенный способ записи подобного
//конструктора
//Alpha():_x(0), _y(0){}
//Метод
Alpha  setAlpha( int a, int b){_x=a; _y=b;}
//Деструктор
~Alpha(){};
}; //Конец определения класса
```

Замечание.

Стандарт C++11 разрешает инициализировать поля класса при их объявлении, так что в VS 2019 можно записывать в определении класса:

```
int _x=0;
int _y=0;
```

В этом случае приведенный конструктор, инициализирующий поля класса какими-либо константами, не нужен.

Для создания объекта используется конструктор класса. Конструктор похож на обычную функцию с именем, совпадающим с именем класса, но без указания типа возвращаемого значения.

Объект класса может быть создан в памяти, управляемой компилятором,

```
Alpha a1;
```

или в свободной памяти с помощью функции `new`.

```
Alpha *a2=new Alpha;
```

В первом случае конструктор инициализирует поля объекта, созданного компилятором, во втором создает объект во время выполнения программы. В классе может быть записано несколько конструкторов; отличаться друг от друга они должны списком параметров.

Деструктор класса автоматически разрушает объект при выходе последнего из области видимости; деструктор имеет имя класса, которому предшествует символ `~`. Деструктор не

имеет параметров и не возвращает значения; в классе может быть только один деструктор.

Конструктор класса `X`, создаваемого пользователем, может иметь параметры любого типа, кроме X; допускается параметр типа const `X&` (константная ссылка на существующий объект типа `X`). Конструктор с параметром-ссылкой называется конструктором копирования; он подразумевает создание нового объекта и копирование в него значений полей уже существующего объекта-параметра. Конструкторы и деструктор называются специальными методами.

Примеры приведены в листинге 1.1.

Если пользователь не определяет в классе `X` ни одного из специальных методов, то компилятор автоматически создает:

1. конструктор с пустым телом и пустым списком параметров (конструктор по умолчанию);
2. деструктор с пустым телом;
3. конструктор копирования.

Кроме этого, компилятор автоматически предоставляет пользователю оператор присваивания.

Явное определение пользователем в классе конструктора с параметрами отменяет автоматическое создание конструктора по умолчанию, определение конструктора копирования отменяет 3, определение деструктора отменяет 2 .

Объявление объекта класса должно соответствовать имеющемуся конструктору. Способы создания объектов те же, что и в языке `C`: объект может объявляться как простая переменная, массив, указатель. Для создания массива объектов следует иметь в классе конструктор без параметров или конструктор, у которого все параметры заданы по умолчанию.

Доступ к члену класса из точки программы вне определения класса определяется наличием в классе модификаторов доступа `public, private, protected`.Модификатор `public` определяет все следующие за ним члены класса как открытые,

к которым разрешен доступ из любой точки программы; `private` запрещает прямой доступ к соответствующим членам из точки программы вне класса (закрытые члены). В классе типа `class` все члены по умолчанию `private`, в классах типа `struct - public`. Каждый атрибут, записанный в классе, отменяет для следующих за ним членов действие предыдущего. Атрибут `protected`(защищенный) используется в иерархии классов.

Принятый способ использования атрибутов: данные объявляются типа `private` (`protected` в базовых классах), методы, конструкторы и деструктор – типа `public`.

Форма обращения к членам класса через его объект может быть двоякой:

```
имя_объекта.имя_члена
или
указатель_на_объект->имя_члена.
```

Во втором случае объект должен быть создан в свободной памяти.

Наряду с методами класса доступ к закрытым членам имеют так называемые дружественные функции. Они не являются членами класса, хотя и объявляются в его теле в следующем виде:

`friend` тип имя (параметры);

Дружественные функции рекомендуется использовать в следующих случаях:

– при необходимости упростить форму обращения к компонентам класса: не будучи компонентом класса, дружественная функция при вызове не требует имени объекта;

– при необходимости использовать методы одного класса для обработки закрытых данных другого класса.

Перегрузка операций

К перегрузке стандартных операций прибегают в тех случаях, когда хотят расширить действие операции на типы операндов, отличающихся от заданных для этой операции в

стандарте языка, или вообще изменить смысл знака операции. Перегрузка должна быть определена в классах, которые создает пользователь и действует только для объектов этого класса. Перегрузка операций производится с помощью специальной функции operator согласно следующей форме:

тип **operator** знак_операции (параметр) {... }.

При этом оператор-функция может быть как членом класса, так и дружественной функцией.

Компилятор анализирует типы операндов, используемых в операции, и вызывает стандартную или перегруженную версию операции класса, в котором определена перегрузка.

Перегружаться могут практически все, за небольшим исключением, операции. При этом функция operator *не изменяет приоритет операции и число операндов*, определенных в стандартной операции.

Примеры дружественной функции и перегруженных операций приведены в листинге 1.1.

Включение и делегирование

Под включением понимается использование в классе X объекта класса Y или указателя на него. Говорят, что эти два класса связаны отношением ассоциации (см. Приложение 1).

Включение позволяет использовать в классе X методы класса Y. В частности, широко применяется т.н. делегирование

```
class Y {
public: int fooY() { return 5; }
};
class X {
  Y y1[10];
public: int fooX() { return   y1.fooY(); }
};
```

```
X x1;
cout << x1.fooX(); //5
```

Здесь класс Y делегирует свою функциональность (метод fooY) классу X. Метод fooX становится делегатом.

Положим, что метод fooY изменяет значение какого-либо поля класса Y. Тогда говорят, что класс X посылает сообщение классу Y с тем, чтобы последний изменил свое состояние.

Обработка исключений в методах класса

см. Приложение 2

В качестве примера в листинге 1.1 приведено применение try, throw, catch для предотвращения деления на 0 в перегруженной операции деления.

Ниже приведена программа определения и использования класса Complex; его UML-диаграмма (см. Приложение 1) приведена рис. 1.1

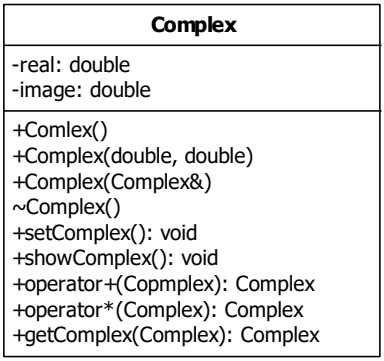

Рисунок 1.1

Проект включает 3 файла: заголовочный - Complex.h , реализации - Complex.cpp и функции MyComplex.cpp для main. В main приведены примеры создания объектов с помощью

различных конструкторов, перегрузка операций и дружественная функция, а также активизация обработки исключения.

```
Листинг 1.1
//complex.h
#pragma once
//Спецификация класса
class Complex
{    //Поля
      double _real=0;
      double _image=0;
public:
      //Конструкторы
      Complex(); //Пустой
      Complex(double, double);//С параметрами
      Complex(const Complex&); //Копирования
      // Методы
      void setComplex(); // Ввод  числа
      void showComplex() const; //Вывод на консоль
      Complex addComplex(Complex&); //сложения
      //Перегруженные операции
      Complex operator = (Complex&);//присваивания
      Complex& operator + (Complex&);// сложения
      Complex& operator * (Complex&); // умножения
      Complex& operator / (Complex&);//деления
      Complex& operator-();//Инверсии знака
 //Дружественная функция вывода на консоль
      friend void getComplex(Complex);
      //Деструктор
      ~Complex();
};

//complex.cpp
#include "Complex.h"
#include <iostream>
using namespace std;
//Определения конструкторов и методов
Complex::Complex() {}
Complex::Complex(const Complex& c1) {
      this->_real = c1._real; this->_image = c1._im-
age;
```

```
}
Complex::Complex(double r, double im) {
      this->_real = r; this->_image = im;
}
Complex::~Complex() { }

void Complex::setComplex() {

      cout << "Действительная часть ? "; cin >>
_real;
      cout << "Мнимая часть ? "; cin >> _image;

}
void Complex::showComplex()const {
      cout << this->_real << " +i" << this->_image <<
endl;
}
Complex Complex::addComplex(Complex& t) {
      Complex tmp;
      tmp._real = this->_real + t._real;
      tmp._image = this->_image + t._image;
            return tmp;
//или
//return Complex(_real + t._real, _image + t._image);

}
void getComplex(Complex c1) {
      cout << c1._real << " +i" << c1._image << endl;
}

Complex  Complex:: operator = (Complex& c1)
{
       this->_real =c1._real;
      this->_image = c1._image;
      return *this;}

Complex& Complex:: operator + (Complex t)
{
      Complex tmp;
      tmp.__real = this->_real + t._real;
      tmp._image = this->_image + t._image;
```

```
      return tmp;
}
Complex& Complex::operator *(Complex& t) {
      Complex  tmp;
      tmp._real = this->_real*t._real - this->_im-
age*t._image;
      tmp._image = this->_image*t._real + this-
>_real*t._image;
      return tmp;
}
Complex& Complex::operator / (Complex& t) {
      Complex  tmp;
      double buf= t._real*t._real + t._image*t._im-
age;
      //выброс исключения
      if (buf == 0) throw 1;

      tmp._real= (this->_real*t._real + this->_im-
age*t._image)/ buf;
      tmp._image = (this->_image*t._real - this-
>_real*t._image)/buf;
      return tmp;
}

//MyComplex.cpp
#include "Complex.h"
#include <iostream>
using namespace std;
int main()
{
      setlocale(LC_ALL, "Russian");
// Конструктор
      Complex cm1, cm2, cm3;
      try {
            cout << "1-е число " << endl;
            cm1.setComplex();
            cout << "2-е число " << endl;
            cm2.setComplex();
            cout << "Сумма " << endl;
            //cm3 = cm1 + cm2;
            cm3.showComplex();
            cout << "Произведение " << endl;
```

```
        cm3 = cm1*cm2;
        cm3.showComplex();
        cout << "Частное" << endl;
        cm3 = cm1 / cm2;
        cm3.showComplex();
        Complex cm4(34.7, 21.8);
        cout << "Массив из 2-х чисел" << endl;
        Complex cm5[2];
        for (int i = 0; i < 2; i++)
              cm5[i].setComplex();
        cout << "Объект-указатель" << endl;
        Complex *cm6 = new Complex(3, 8);
        cm6->showComplex();
        //Копирование объектов
        cout << "Объект-копия cm1 " << endl;
        Complex cm7(cm1);
        //или
        cout << "Объект-копия cm3 " << endl;
        Complex cm8 = cm3;
        getComplex(cm8);
        delete cm6;
  }//try
  catch (int) {
        cout << "Деление на ноль!" << endl;
        return 1;
  }
  return 0;
}
```

Замечания к программе:

1. Стандарт C++11 позволяет наряду с обычным объявлением методов вместо типа возвращаемого значения использовать слово auto; предполагается, что компилятор определит тип по выражению в return (см. перегрузку operator +).

2. ключевое слово this, используемое в листинге, называется скрытым указателем на объект; this создается автоматически вместе с объектом и не требует объявления. Употребление его в программе, кроме как в операторе return перегруженного оператора «=», необязательно.

3. В программе выделена спецификация класса (объявления переменных и методов). Она помещена в заголовочный файл. Спецификацию можно разместить и в том же файле, где расположены определения методов и функция main. Можно использовать смешанный стиль: если тело метода содержит немного кода или вообще кода нет, то тело можно записать в разделе спецификации, а тела сложных методов вынести за пределы определения класса.

Вообще говоря, спецификацию выделять необязательно – в определении класса можно записать определения методов без их отдельного объявления и поместить определение класса наряду с функцией main в один .cpp файл. Такой стиль написания программы называется inline. Далее в тексте руководства программы написаны именно так. Но в лабораторных работах выделять спецификацию классов в C++ программах обязательно.

4. В программе определены конструктор копирования Complex (const Complex&) и операция присваивания =.

Эти объявления и определения можно не делать, так как созданные автоматически эти члены будут работать точно так же, как и записано – копировать поля объекта-параметра. Аналогично можно не объявлять и деструктор.

Однако, если в классе есть поле-указатель и в конструкторе работает функция new, то в обязательном порядке деструктор определяется явно и в него вводятся функции delete или delete[]. Если же объекты класса участвуют в операциях копирования и присваивания, то нужно явно определять и последние. Коротко эти требования формулируются как «правило 3-х». (см. Приложение 3).

Варианты задания

При выполнении работы необходимо:

– разработать соответствующие классы, конструкторы, поля и методы;

– поля класса сделать закрытыми; для чтения и изменения их значений определить открытые методы;

– предусмотреть во всех вариантах консольный ввод данных для создания объектов и консольный вывод результатов.

– во всех вариантах необходимо использовать хотя бы один раз блоки try catch. Это можно сделать для контроля арифметических ошибок, для проверки существования файлов и т.п. Отсутствие такового карается дополнительным уменьшением оценки.

– во всех перечисленных вариантах не рекомендуется пользоваться контейнерами STL, кроме string.

1. Определить класс, создающий объект типа Дата (Date). Методы: ввод и вывод значений полей объекта на консоль, определения соответствующего дате дня недели, прибавления к дате целой константы без использования библиотечных классов, таких как ctime.

2. Определить класс, создающий объект типа обыкновенная дробь (Fraction) с методами сравнения объектов, изменения, сложения, умножения, деления, сокращения.

3. Определить класс римских чисел RomanFigure. Методы: ввода и вывода римских чисел, сложение и вычитание их. Вычисления над римскими числами (не более 3-х символов) должны выполняться как в римской системе, так и (для контроля) с их эквивалентами в десятичной системе. В качестве поля для представления числа можно использовать String (StringBuilder) или символьный массив.

4. Определить класс Линия в трехмерном пространстве (Line3D), объекты которого задаются координатами концов в

прямоугольной системе координат. Методы: определение длины, параллельный перенос, построение замкнутой фигуры.

5. Определить класс Прямоугольник (Rectangle) со сторонами, параллельными осям координат на плоскости. Методы: перемещение, определение площади, построение прямоугольника, являющегося пересечением двух других.

6. Определить класс многочленов Polinom от одной переменной, задаваемых степенью многочлена и массивом коэффициентов, с операциями вычисления значения многочлена для заданного аргумента, сложения и вычитания многочленов.

7. Определить класс Множество (Set) с методами: добавление элементов в множество и проверкой пересечения двух множеств. Мощность множества задается при ее создании.

8. Определить класс вещественных матриц (Matrix) с методами, реализующими сложение и умножение матриц, транспонирование. Размер матрицы задается при ее создании.

9. Реализовать класс Стек (Stack) для какого-либо типа данных с методами push, pop, isEmpty и back (показывается последний элемент без его извлечения), работающими согласно соответствующей дисциплине обслуживания. Размер стека задается при его создании.

10. Описать класс Предметный указатель (Index). Указатель содержит слово и номера страниц некоторого текста, в которых это слово встречается. Количество номеров страниц, относящихся к одному слову, не превышает десяти. В классе может

быть массив указателей. Анализируемый текст содержится в файле.

11. Создать класс Телефон (Telephon) с полями: марка, собственный номер, массив запомненных номеров, количество звонков, тариф. Создать методы Позвонить и Ответить. Создав 2 объекта, смоделировать вызов-ответ, после чего подсчитать стоимость разговора. Если звонок новый, то его номер заносится в массив номеров (не менее 10).

12. Создать класс Треугольник (Triangle) с полями, в которых фиксируются координаты ее вершин (на плоскости). Методы: определения центра тяжести, площади, периметра, перемещения, изменения размеров, сравнения площадей двух объектов.

В следующих вариантах при необходимости рекомендуется создавать 2 класса и объекты или ссылки на один из них включить в другой.

13. Определить классы Автостоянка (Parking) и Автомобиль (Car). Для каждого автомобиля во втором классе задается госномер, марка, цвет и методы Парковаться и Покинуть парковку. Если авто паркуется, то его данные заносятся в массив Автостоянки с соответствующей отметкой. При выезде эта отметка снимается, но данные остаются. Программа должна отвечать на запрос, присутствует ли конкретный автомобиль на парковке, а также по запросу вывести список присутствующих автомобилей. (В данном случае объект Автостоянки целесообразно включить в Автомобиль.)

14. Определить классы Карта (Card) и Колода_карт (Cardatch). Поля первого – масть (suit) и достоинство (rank).

Методы класса возвращают масть и достоинство. Второй включает массив (32) объектов первого. Методы класса:

- перемешивания колоды;
- сравнения 2-х карт по достоинству при условии, что масти одинаковы;
- создания 4-х мест и раздачи равного количества карт;
- моделирования упрощенного розыгрыша взятки: на стол выкладываются по одной карте от каждого из 4-х игроков; первая выложенная карта определяет масть; выигрывает карта, старшая по достоинству (картинки старше простых карт; козырной масти нет).

Список карт для инициализации программы хранить в файле.

15. Определить классы Книга (Book), Библиотека (Library) и читатели (Readers). Поля книги: ФИО автора (ов), название (Title), год издания (Year), издательство (Publishers), Id, выдана/на полке. Библиотека содержит массив книг и методы выдачи и возвращения книги. Читатели – свой Id и Id книги, которая им выдана. Предусмотреть выдачу и возвращение книг, а также поиск книги по Id с сообщением выдана она или нет и если выдана, то кому. Список книг для инициализации программы хранить в файле (не менее 10).

16. Определить классы Студент (Student) и Институт (Institute). Поля первого: ФИО, дата рождения, адрес, учится/отчислен, стипендия/нет, экзаменационная оценка. Студент сдает экзамены (метод), по результатам которых поле оценки заполняется случайными значениями 5/4/3/2. В Институт включен массив студентов. Метод этого класса (Провести сессию) инициирует методы сдачи экзамена для каждого студента. По полученным оценкам институт издает приказы (методы) об отчислении или назначении стипендии. Список

студентов для инициализации программы хранить в файле. (не менее 10)

Контрольные вопросы

1. Для чего служит конструктор ? Может ли в классе быть несколько конструкторов? Чем должны отличаться различные конструкторы одного и того же класса?
2. Для чего служит деструктор класса? Имеет ли деструктор параметры?
3. Какие сообщения и в какой последовательности будут выведены на монитор после выполнения нижеследующей программы?

```
class Alpha {
public:
int x, y;
Alpha(){cout<<"Конструктор  #1\n";}
Alpha(int m){cout<<"Конструктор  #2\n";
 x=y=m;
}
Alpha (const Alpha& other){
 cout<<”Конструктор #3\n”;
  x=other.x; y=other.y;
}
~Alpha(){cout<<"Деструктор "<<endl;}
};
void main()
{
Alpha a1;
a1.x=1;
Alpha *a2;
a2=new Alpha;
Alpha a3[2];
Alpha a4(4);
Alpha a6(a1);
}
```

4. Имеется следующий класс

```
class Alpha {
Alpha(const Alpha& a){/*...*/}
```

```
};
```

Какие члены класса в данном случае будут автоматически определены при создании класса?

5. Определен следующий класс:

   class Alhpa { public: int abc; };

   Запишите обращения к члену abc с использованием точки и стрелки.
6. Каким образом компилятор отличает вызов стандартной операции от вызова перегруженной? Вспомните язык С и его операции <<, >> и запишите результат второго выражения

   ```
   int a=4;
   cout<<(a<<3);
   ```
7. Что такое отношение включения и как оно изображается на диаграммах? Что такое делегирование?

Лабораторная работа №2. Наследование в C++

Цель работы

Ознакомление с отношением наследования и иерархией классов.

Основные сведения

Базовый и производные классы

Наследование – механизм языка, позволяющий написать новый класс на основе уже существующего (базового, родительского) класса. Формат определения производного класса следующий:

тип_класса имя_производного_класса : список[модификатор_доступа имя_базового_класса] {определение_производного_класса};

В объекте производного класса наряду с собственными полями создаются поля, имя и тип которых определены в базовом классе, а сам объект производного класса получает доступ к методам базового. В этом, собственно, и заключается реализация механизма наследования.

При создании объектов производного класса автоматически вызываются конструкторы базовых классов согласно списку базовых классов в объявлении производного класса, а затем конструктор производного класса. Объекты разрушаются в порядке, обратном их созданию, т.е. вначале вызывается деструктор производного класса, а затем базового.

Использование наследования позволяет строить иерархии классов: один класс может быть базовым для нескольких производных, производный класс может быть, в свою очередь, базовым для какого-либо класса и т.д. Если производный класс имеет несколько базовых, то такое наследование называется множественным.

Если доступ к собственным членам производных классов определяется обычным образом, то на доступ к наследуемым влияет, во-первых, атрибут доступа в базовом классе и, во-вторых, модификатор доступа (public / private), указанный перед именем базового класса в конструкции определения производного класса. Общепринятое (но необязательное) правило – поля базового класса определяются как защищенные (protected).

При создании иерархии классов первостепенное значение имеет умение выделить существенные характеристики некоторого объекта в анализируемой предметной области, отличающие его от всех других объектов и, таким образом, четко описать его концептуальные границы с точки зрения наблюдателя. Этот процесс называется абстракцией.

В процессе проектирования рекомендуется придерживаться принципа "это есть (является)" (is a), выделяя общие черты объектов и инкапсулируя их в базовом классе. Положим, например, что нам надо создать классы Car (автомобиль) и Lorry (грузовик). Что общего у объектов этих классов и чем они разнятся? На самом деле у них много общих черт и много различий, но для упрощения примем следующие формулировки:

– грузовик есть средство передвижения, имеющее марку, цену и год выпуска, а также характеризующееся грузоподъемностью;

– автомобиль есть средство передвижения, имеющее марку, цену и год выпуска, а также характеризующееся скоростью.

В соответствии с этим создадим базовый класс "Средство передвижения "(Vehicle). В нем определим три поля для марки, цены и года и конструктор с соответствующими параметрами. В производных классах Car (автомобиль) и Lorry (грузовик) определим по одному дополнительному полю и соответствующие конструкторы.

Замечания.

- В нижеприведенных листингах в отличие от предыдущих ввод данных осуществляется за пределами класса. Это удобно, если предполагается ввод информации из файла или ваша программа является фрагментом большого проекта. По-хорошему, в классе надо избавиться и от cout – в методе displayVehicle возвращать массив значений и выводить их потом в файл или на консоль.
- Обратите внимание на using. Так делать правильнее, чем using namespace std;

```
Листинг 2.1
//Базовый класс
#include <iostream>
#include <string>
#include <vld.h>
using std::cin;
using std::cout;
using std::string;

//Базовый класс
class Vehicle {
protected:
	string brand = "";//Марка
	int price=0;//цена
	int year=0; //год выпуска
public:
Vehicle()=default;//Нужен для создания массива в
                  //классе Garage
	Vehicle(string aBrand,int aPrice, int aYear) {

		brand = aBrand;
		price = aPrice;
		year = aYear;
	}
//Виртуальная функция
	virtual void displayVehicle(){}
//Виртуальный деструктор
virtual ~Vehicle() = default;//По умолчанию
};
//Производный класс
```

```
class Lorry :public Vehicle {
	int carrying=0; // грузоподъемность
public:
	//Конструктор
	Lorry(string aBrand, int aPrice, int aYear, int
aCarring)
		:Vehicle(aBrand, aPrice, aYear)//Вызов
базового
	    {
			carrying = aCarring;
	}

	void displayVehicle() override {
		cout << "\n  Марка "<<brand;
		cout << " Цена " << price;
		cout << " Год выпуска " << year;
		cout << " Грузоподъемность "<< carring;
	}
};

// Производный класс
class Car : public Vehicle {
	int speed=0; // скорость
public:
	Car(string aBrand, int aPrice, int aYear, int
aSpeed)
		:Vehicle(aBrand, aPrice, aYear)
	   {
		speed=aSpeed;
	}
	//Переопределение виртуальной функции
	void displayVehicle() override {
		cout << "\n  Марка  "<<brand;
		cout << "  Цена " << price;
		cout << "  Год выпуска  " << year;
		cout << "  Скорость " <<speed;
	}
};

int main()
{
	setlocale(LC_ALL, "Russian");
```

```
        string aBrand=””;
        int aPrice=0,aYear=0,aCarring=0;
        cin>>aBr>>aPr>>aYea>>aCarr;
        Vehicle* v=new Lorry aBr, aPr,aYea,aCarr);
        v->displayVehicle();
        delete v;
    v = new Car("Camaro", .., .., ..);
        v->displayVehicle();
        delete v;
return 0;}
```

Предположим теперь, что нам надо добавить к проекту автобусы. Рассматривая автобус как средство передвижения для перевозки пассажиров, добавляем производный класс Bus с полем "вместимость", создаем в нем соответствующий конструктор и изменяем метод displayVehicle. При этом уже имеющиеся классы не трогаем, их можно даже не перекомпилировать; мы просто повторяем их в новой программе. Таким образом, расширение предметной области происходит гораздо безболезненнее, чем если бы все объекты были сосредоточены в одном классе и нам бы пришлось его переделывать. Это обстоятельство является одной из причин применения наследования. Говорят, что при изменении правильно спроектированной программы процент повторно используемого кода высок.

В классе Vehicle определена виртуальная функция displayVehicle. Виртуальной называется функция, определенная с атрибутом virtual в базовом классе и имеющая в каждом производном классе такую же сигнатуру. При этом реализация функции (ее тело) в конкретном производном классе может отличаться от ее реализации в базовом и других производных классах. Например,

```
     class  Based {
/* ...*/
       public:
        virtual int fb(){return 3;}
       /* ...*/
     };
```

```
    class Derived1:public Based {
       /* ...*/
      int  fb() {return 5;}
       /* ...*/
      };
   class Derived2:public Based {
     int  fb() {return 7;}
     /* . . . */
};
```

Виртуальная функция класса может вызываться обычным образом – как член производного класса через его объект. Но тогда никаких преимуществ по сравнению с обычной функцией ее применение не приносит.

Правильный вызов виртуальной функции заключается в последовательном применении следующих 2-х инструкций:

```
    указатель_ на_базовый_ класс =
    указатель_на__объект_производного_ класса;
    указатель_на базовый_класс →
вызов_виртуальной_функции;
```

(Инструкции в тексте программы могут быть разделены. Заметьте, что в первой инструкции никакого явного преобразования типа указателей не нужно.)

Для нашего примера вызов функции fb из производного класса:

```
Based* ptr=new Derived1;
ptr->fb(); // возвращает 5
. . .
ptr=new Derived2;
ptr->fb();// возвращает 7
```

Говорят, что если виртуальная функция вызывается из производного класса, то она перекрывает базовую версию.

Класс, содержащей виртуальный метод, называют полиморфным классом. В нашем случае это означает, что функции fb() в разных точках программы могут действовать по-разному. При наличии виртуальных функций деструктор в базовом классе должен быть виртуальным.

Еще раз. Механизм виртуальных функций правильно работает только для указателей и ссылок на объекты. Полиморфизм проявляется только тогда, когда объект производного класса адресуется косвенно, через указатель или ссылку на базовый.

Таким образом, в нашей иерархии существует метод, имеющий доступ к объекту любого класса и использующий при этом только указатель на базовый класс.

Замечания.

1.В листинге виртуальная функция приведена с пустым телом. Вообще говоря, предпочтительной формой виртуальной функции в базовом классе является т.н. чистая (pure) функция. Для нашего случая

```
virtual void displayVehicle() =0;
```

Класс с чистой виртуальной функцией называется абстрактным; объект такого класса создать нельзя, только указатель на объект.

2.Вызов виртуальной функции осуществляется медленее вызова обычной. Поэтому записывать virtual просто так в C++ не стоит.

UML-диаграмма иерархии классов приведена на рис. 2.1.

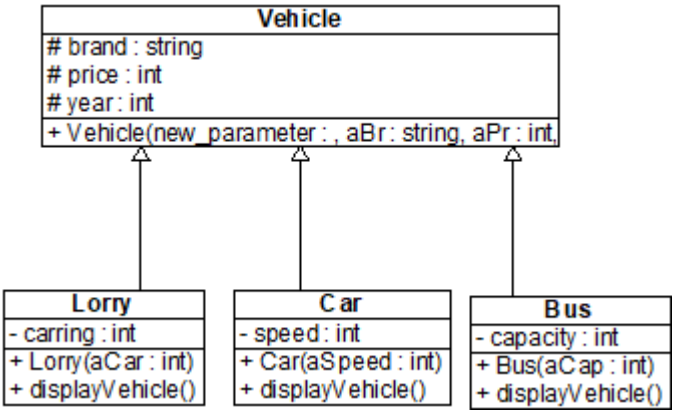

Рисунок 2.1.

Продолжаем обсуждение программы.

Положим теперь, что при использовании разработанных классов мы будем иметь дело с массивами их объектов, которые надо создавать и изменять их состояния. Чтобы не заставлять клиента (в данном случае main) заниматься этим, введем дополнительный класс Garage, включив в него разработанные нами типы, сформируем в нем массивы объектов и определим соответствующие методы.

Вот как он может выглядеть (продолжаем программу):

Листинг 2.1 (продолжение)

```
class Garage {
      int nL, nC; // Кол-во грузовиков и автомобилей
      Vehicle** veh;

public:
      Garage(int n, int m)// Конструктор
      {
            nL = n; nC = m;
            veh = new  Vehicle* [nL + nC];
            string b; int p, ye, c, s;
            cout << "Грузовики\n";
            for (int i = 0; i < nL; i++)
            {

                  cin >> b; cin >> p; cin >> ye;cin>>
c;
                  veh[i] = new Lorry(b,p,ye,c);
            }
            cout << "Автомобили\n";
            for (int i = 0; i < nC; i++)
            {
                  cin >> b; cin >> p; cin >> ye; cin
>> s;
                  veh[i+nL] = new Car(b, p, ye, s);
            }

      }
```

```
	//Вывод
	void printGarage()
	{
		if (nL + nC == 0) { cout << "Машин нет
\n"; return; }

		for (int i = 0; i < nL+nC; i++)
		{
			veh[i]->displayVehicle();
			}
	}

	// Деструктор
	~Garage() {
		for (int i = 0; i < nL + nC; i++)
			delete veh[i];
		delete[] veh;

	}

};


int main()
{
	setlocale(LC_ALL, "Russia");
	Garage g(2, 3);
	g.printGarage()
	return 0;
}
```

Обратите внимание, что main не знает о существовании автомобилей и грузовиков.. Единственное, что он делает – задает соответствующие константы в конструкторе Garage.

Иерархия классов программы приведена на рис.2.2. Обратите внимание на связь между Vehicle и Garage.

В класс Garage можно инкапсулировать все требуемые методы работы с массивами объектов: сортировку, ввод-вывод в файл и т.п.

Упрощение клиентской части программы за счет введения некоторого дополнительного класса является содержанием паттерна Фасад (Façade) [2].

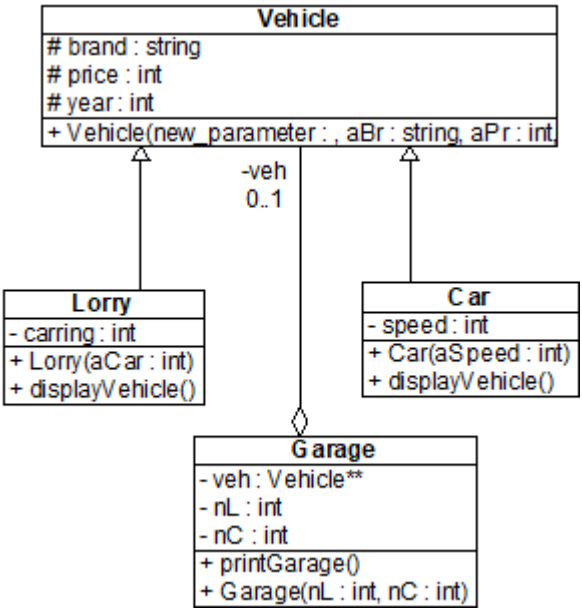

Рисунок 2.2.

Варианты задания

Нижеследующие варианты предполагают создание иерархии классов с виртуальными функциями. Если есть необходимость, то в производном классе можно объявлять

дополнительные компоненты. **Проектировать иерархию надо так, чтобы добавление предполагаемого нового производного класса (типа) происходило бы без существенного изменения уже существующих классов, т.е. процент повторного использования уже созданного кода к моменту модификации проекта должно быть большим. Почаще вспоминайте известный слоган ООП – «Добавляй (класс) но не изменяй (существующие)».**

Безусловно, виртуальные функции должны вызываться правильным образом.

По-прежнему не рекомендуется использование контейнеров STL.

Функция main должна иметь как можно более простой вид. Предполагайте, что клиент (main) не знает и не хочет знать C++, а читает руководство пользователя, в котором всего лишь перечислены формы вызова функций интерфейса вашей программы. С этой точки зрения целесообразно (но не обязательно) ввести класс-фасад, имя которого упоминается в задании. Если вы работаете с массивами объектов, то фасад обязателен.

1. Определить базовый класс Number с методами сложения и вычитания и полем типа char* или string. (Первый вариант труднее). В качестве производных классов создать NumberBit и NumberHex. Объекты первого подкласса могут содержать только двоичные символы, второго – только шестнадцатеричные, что обеспечивается проверкой допустимых символов при вводе данных. При реализации операций объекты подклассов рассматривается как двоичное или шестнадцатеричное число и операции выполняются в соответствующей системе.

2. Создать класс Figure, объект которого задается координатами точек, с вычислением площади, центра тяжести и

периметра. На его основе реализовать классы Rectangle (прямоугольник) и Pentagon (пятиугольник),

3. Создать класс Line (строка). с методами, реализующими операции сложения и вычитания. На его основе реализовать классы Integer и Row. Объект первого рассматривается как обычное целое число, объект второго как строка, для которой сложением является конкатенация, а вычитание двух строк сводится к удалению из первой символов, входящих во вторую строку (воспользуйтесь методом erase).

4. Создать класс Body. На его основе реализовать классы Parallelepiped (прямоугольный параллелепипед), Cone (конус) и Ball (шар) с возможностью вычисления площади поверхности и объема.

5. Создать класс кошелек (Wallet) с тремя отделениями (подклассами) для долларов, евро и рублей. Предусмотреть покупку какого-либо товара в той валюте, какая написана на ценнике. При этом следует иметь в виду, что в кошельке находятся купюры достоинством 1, 5, 10 и 100 денежных единиц, в то время как цена товара может быть целой, но не совпадать с достоинством имеющихся купюр и вообще быть больше. Сдача должна возвращаться в соответствующее отделение кошелька в купюрах 1/5/10. Предусмотреть перевод валютной суммы в рубли.

6. Создать класс Triangle (треугольник), задав в нем координаты одной стороны и угол между ней и прилежащей. Реализовать методы вычисления площади и периметра. На его основе создать классы, описывающие равносторонний (угол равен 60), равнобедренный и прямоугольный треугольники (произвольный угол меньше 90) с реализацией методов базового класса.

7. Создать класс Solution (решение) с методами вычисления корней уравнения. На его основе реализовать классы Linear

(линейное уравнение) и Square (квадратное уравнение). Для создания массивов решений уравнений (не систем) с различными аргументами создать дополнительный класс Series.

8. Создать класс Function (функция) с методами вычисления значения функции y = f(x) в заданной точке x и вывода результата на консоль. На его основе определить классы Ellipse, Hiperbola и Parabola, в которых реализуются соответствующие математические зависимости. В дополнительном классе Series все три функции должны вызываться для заданного интервала изменения x с выводом результатов.

9. Создать класс Triad (тройка) и на его основе классы Date (дата) и Time (время) с методами увеличения и уменьшения на 1 каждого из значений тройки: год/час, месяц/минуты, день/секунды. В дополнительном классе Memories создать массив пар (дата-время) объектов этих классов: 01.09.2020 – 8.00.00, ….Количество пар задается конструктором класса Memories.

10. Описать класс Element (элемент логической схемы) с двумя входами и одним выходом и полем, хранящим название элемента. Определить функцию, которая преобразует входные двоичные значения в выходное. На его основе реализовать классы AND и OR - двоичные вентили, которые реализуют логическое умножение и сложение соответственно. В дополнительном классе Scheme создать массив элементов (до 10) и обеспечить подачу двоичных сигналов на их входы с выводом выходных значений. Входные сигналы хранятся в файле.

11. Создать класс Container для какого-либо типа данных с методами push, pop, isEmpty и front. На его основе реализовать классы Stack (стек) и Queue (очередь). Размер контейнера задается при его создании.

12. Создать класс Progression (прогрессия) с методом вычисления суммы прогрессии. На его основе реализовать классы Linear (арифметическая) и Exponential (геометрическая). В дополнительном классе организовать вычисление обеих прогрессий для аргумента, изменяющегося в заданном интервале.

13. Создать класс Pair (пара значений) с методами, реализующими арифметические операции сложения и вычитания. На его основе реализовать классы Fractional (дробное) и Complex (комплексное число). В классе Fractional вещественное число представляется в виде двух целых, в которых хранятся целая и дробная часть числа соответственно, в Complex – вещественная и мнимая часть комплексного числа.

14. Создать класс Integer (целое) с методами, реализующими арифметические операции и ввода-вывода на экран. На его основе реализовать классы Decimal (десятичное) и Binary (двоичное). Арифметические операции выполнять в соответствующей системе счисления, но в классе Binary дополнительно предусмотреть методы перевода 2->10.

15. Создать класс Sorting (сортировка), и на его основе классы Choice (сортировка выбором) и Quick (быстрая сортировка). Размер сортируемых массивов задается при их создании, а элементы считываются из файла.

16. Создать класс Pair (пара значений) с методами, реализующими арифметические операции сложения и вычитания. На его основе реализовать классы Money (деньги) и Complex (комплексное число). В классе Money денежная сумма представляется в виде двух целых, в которых хранятся рубли и копейки соответственно.

17. Создать базовый класс Supermarket со следующими полями: название, адрес, перечень товаров, включающий наименования и количество, менеджеры и кассира. Кассир реализует операцию покупки конкретного товара посетителем, в результате чего должно измениться количество товара в перечне. Менеджер регулярно следит за перечнем товара и при необходимости дополняет количество до нормы. Все методы в базовом классе должны быть абстрактными.

На основе этого класса создать 2 супермаркета с конкретными названиями, адресами, персоналом.

Контрольные вопросы

1. Опишите свою иерархию классов, используя конструкцию «… это есть .. ».

2. В чем заключается наследование одного класса другому? В чем разница в организации наследования полей и методов?

3. Определены 2 класса:

```
class Based{
protected: int x;
public:
void setX(int _x){x=_x;}
int getX() const {return x;}
};
class Derived:public Based
{ };
```

Какие значения выводятся на консоль?

```
Based b1;
b1.setX(3);
cout<<b1.getX();
Derived d1;
```

```
cout<<d1.getX();
```

4. Удачной ли является иерархия классов, при которой некоторый класс X является производным от большого количества классов с большим числом полей в каждом (A←B ← C ←…← X)? Какая существует альтернатива наследованию?
5. Есть ли ошибки в нижеследующих объявлениях, если класс Shape абстрактный:

```
Shape sh;
Shape *psh;
Shape *psh1=new Shape();
```

6. Что из себя представляет виртуальная функция и как она должна вызываться?

Лабораторная работа №3. Динамический выбор типа объектов

Цель работы.

Создание объекта класса во время выполнения программы с выбором его типа.

Основные сведения

Продолжим обсуждение использования виртуальных функций, начатое в предыдущей работе. Там тип создаваемых объектов записан в тексте (см., например, конструктор Garage в листинге 2.2); виртуальные функции определяют поведение объектов, подстраиваясь под их тип. Для изменения типа объекта надо вносить изменения в программу.

В данной работе механизм виртуальных функций используется для создания программ, в которых выбор типа создаваемого объекта производится *во время выполнения* программы по запросу пользователя. Это идея паттерна проектирования программ под названием «Фабричный метод» (Factory Method) [4].

Еще немного о виртуальных функциях. Когда говорят об их вызове, обычно имеют в виду т.н. позднее связывание объекта класса и функции. Дело в том, что наличие виртуальной функции в иерархии классов приводит к созданию для каждого класса таблицы, в которую заносятся адреса реализаций виртуальных функций этого класса. Таким образом, адреса разных реализаций одной виртуальной функции различны. Объект класса при создании получает адрес таблицы своего класса. Указателем, обращающимся ко всем таблицам иерархии, является указатель на базовый класс. Если при выполнении программы в какой-либо точке этот указатель получает адрес объекта одного их производных классов, то он через

поле this объекта обращается к таблице этого класса и вызывает соответствующую реализацию виртуальной функции. Поэтому конкретная реализация виртуальной функции определяется во время выполнения программы. Поэтому же вызов виртуальных функций обходится дороже, чем вызов обычной - выполняется дополнительная операция адресации.

Если в производном классе виртуальный метод не переопределен, то вызов будет передаваться вверх по иерархии классов вплоть до базового.

В лабораторной работе должна быть создана программа, создающая объекты двух классов (T1, T2), выбранных из таблицы 1 согласно номеру варианта. Эти классы должны быть производными от класса Shape.

Перечисленные в таблице классы создают следующие плоские объекты: квадрат, треугольник, прямоугольник, параллелограмм, трапеция, правильный шестиугольник, правильный восьмиугольник (square, triangle, rectangle, parallelogram, trapeze, hexagon, octagon). В базовом классе должны быть объявлены а в производных переопределены виртуальные методы вычисления площади, центра тяжести, а также методы Вращения (Rotate) и Перемещения (Move).

Ниже приводится пример программы, которая по запросу создает объект одной из трех фигур.

//Листинг 3.1

```
class Point {
public:int x=0, y=0;
};
class Shape { //  базовый класс
protected:
      Point* arc=0;
      public:
      string ID="";
      virtual ~Shape() { }
};

class Triangle: public Shape {
public:
```

```
      Triangle() { arc = new Point[3];
                   ID = "Triangle"; }

      ~Triangle() { delete[] arc;}
};
class Octagon : public Shape {
public:
      Octagon() { arc = new Point[8];
                  ID = "Octagon"; }
      ~Octagon() { delete[] arc;}
};
// Класс - фабрика производных от
//Shape //объектов
class FactoryShape {
public:
static Shape* createShape(char Ch)
      {
            switch (Ch) {
            case 'T':
                  return new Triangle();
            case 'O':
                  return new Octagon();
            default: return nullptr;
            }
      }
};

int main()
{

      setlocale(LC_ALL, "Russian");
      Shape* s1 = nullptr;
      bool cond = true;
      while (cond) {
            cout << "Тип фигуры?\n";
            char type;
            cin >> type;
            s1 = FactoryShape::createShape(type);
            if (s1)
            {
                  cout << s1->ID<<endl;
            }
```

```
            else
        cout << "Такой фигуры нет\n";
        cout << "Продолжим? (1/0)\n";
            cin >> cond;
            if (s1) delete s1;
        }
```

Особенностью этой программы является использование статического метода createShape, который можно вызывать без создания объекта соответствующего класса.

Этот метод (он может быть и нестатическим) называется фабричным из-за того, что он, не будучи конструктором, создает объект класса. При этом точка создания объекта отделена от соответствующего класса.

Необходимо создать операции над объектами 2-х классов:

- сравнить два объекта по площади - Compare;
- определить факт пересечения объектов – isIntersect;
- определить факт включения одного объекта в другой – isInclude.

Эти методы также можно делать статическими.

Если вас затрудняет точная реализация этих алгоритмов, то можно вместо координат объектов использовать области вращения фигур, которые представляют из себя окружность, описанную вокруг нее там, где это возможно, или проведенную из центра тяжести с радиусом, равным расстоянию от него до наиболее удаленной точки фигуры.

Эти операции надо инкапсулировать в класс Operations, создать объект этого класса и вызывать методы для объектов классов типа Shape.

```
class Operation {
       bool isInclude(Shape*
              first, Shape* second){
            //. . .
            return true;
       }
};
```

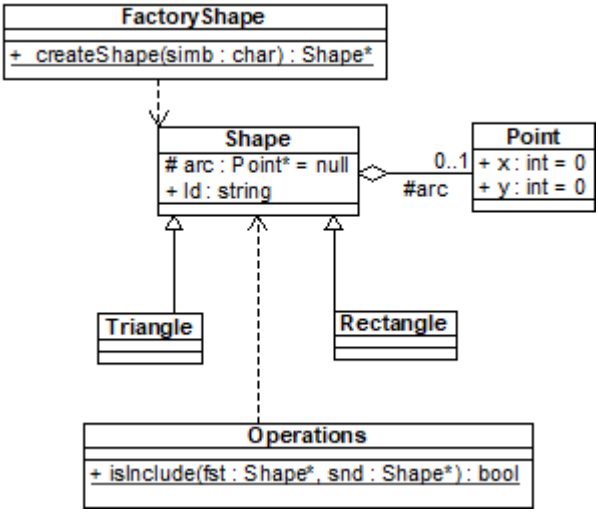

Рисунок 3.1

Замечание.

Можно для идентификации класса текущего объекта воспользоваться библиотекой typeinfo (#include <typeinfo>). Название класса выводится на консоль функцией typeid(*p).name(), где p-указатель на базовый класс.

Ответы на вопросы, связанные со свойствами геометрических фигур, можно получить в [3,4].

Варианты задания

Варианты заданий приведены в следующей таблице:

Таблица 1

№	T1	T2
1	Треугольник	Квадрат

2	Треугольник	Прямоугольник
3	Треугольник	Параллелограмм
4	Треугольник	Трапеция
5	Треугольник	Шестиугольник
6	Треугольник	Восьмиугольник
7	Квадрат	Прямоугольник
8	Квадрат	Параллелограмм
9	Квадрат	Трапеция
10	Квадрат	Шестиугольник
11	Квадрат	Восьмиугольник
12	Прямоугольник	Параллелограмм
13	Прямоугольник	Трапеция
14	Прямоугольник	Шестиугольник
15	Прямоугольник	Восьмиугольник
16	Параллелограмм	Трапеция
17	Параллелограмм	Шестиугольник

В программе надо предусмотреть реакцию на неправильно заданный тип фигуры.

Контрольные вопросы

1. Различают 4 элемента определения функции: тип, имя, список параметров, тело. Какими элементами могут отличаться экземпляры одной виртуальной функции, находящиеся в разных производных классах?

2. В каких случаях применяется фабричный метод?

3. Если вызывается виртуальная функция производного класса, в котором она не определена, то в какую точку переходит управление программой?

4. При описании виртуальных функций встречаются выражения “динамическое связывание”, “статическое связывание”. О связывании чего идет речь?

5. Укажите на недостаток использования виртуальной функции.

6. Можно ли использовать для вызова виртуальных функций не указатели, а ссылки, и имеет ли это какой-нибудь смысл?

7. Какие 2 особенности имеет статический метод?

8. Если вы собираетесь хранить полученные различные фигуры в одном массиве, то какой тип элементов массива вы выберете?

Лабораторная работа №4. Контейнеры STL

Цель работы.

Цель работы – ознакомление с классами-контейнерами библиотеки STL.

Основные сведения

Контейнеры

Контейнером в C++ называют объект одного из классов-контейнеров, входящих в библиотеку STL (Standard Template Library). Контейнер предназначен для хранения и выполнения некоторых операций над объектами классов. Классы-контейнеры подключаются к программе с помощью одноименных заголовочных файлов.

Контейнер может быть настроен на любой тип элементов и количество элементов в нем может увеличиваться во время выполнения программы.

Объект класса-контейнера создается конструктором класса в свободной памяти. Удаление этого объекта выполняется автоматически при выходе его из области видимости – никакие new и delete использовать не надо.

По способу организации памяти различают последовательные и ассоциативные классы-контейнеры.

Среди последовательных - **vector**(вектор), **list**(список)**, deque**(двусторонняя очередь). Ассоциативные: **set** (множество) и **map** (словарь).

Операции, общие для всех контейнеров:

- равенства (==) и неравенства (!=); возвращают значение true или false;
- присваивания (=); копирует один контейнер в другой;
- clear – удаляет все элементы;
- insert – добавляет один элемент или диапазон элементов;

- erase – удаляет один элемент или диапазон элементов;
- unsigned int size( ) const – возвращает число элементов;
- unsigned int max__size() const –возвращает максимально допустимый размер контейнера;
- bool empty( ) const – возвращает true, если контейнер пуст;
- iterator begin( ) – возвращает итератор на начало контейнера (итерации будут производиться в прямом направлении);
- iterator end ( ) – возвращает итератор на конец контейнера (итерации в прямом направлении будут закончены).

Итераторы

Итератор – аналог указателя в контейнере. Для всех контейнерных классов STL определен тип iterator, однако реализация его в разных классах разная. Поэтому при объявлении объектов типа итератор всегда указывается область видимости в форме конструктор_класса:: iterator имя, например:

vector<int> vecI;

vector<int>: :iterator iterl:

list <Man> vecM;

list<Man>::iterator iter2;

Замечание:

в стандарте С++11 итераторы можно задавать следующим образом (контейнер при этом не должен быть пустым):

auto iter1= vecI.begin();

auto iter2=vecM.begin();

К основным операциям, выполняемыми над любыми итераторами, относятся:

- разыменование итератора: если p — итератор, то *p — значение объекта, на который он ссылается;
- присваивание одного итератора другому;
- сравнение итераторов на равенство и неравенство (== и !=);
- перемещение его по всем элементам контейнера с помощью префиксного (++p) или постфиксного (p++) инкремента.

В контейнерах типа vector и deque над итераторами можно выполнять операции сложения и вычитания (итераторы произвольного доступа). Кроме того, в этих контейнерах определена операция индексации

Универсальный способ перебора элементов контейнеров в STL – перебор с помощью итераторов и с помощью цикла по диапазону.

Vector

Создание:

- пустого контейнера vector <int> vecInt;
- контейнера с начальным размером
  vector <string> vecStr(40);

Увеличение текущего размера контейнера выполняет функция resize(константа) – vecInt.resize(60).

Вставку и удаление элементов можно выполнять операциями insert и erase, аргументами которых являются итераторы, указывающих на позицию элемента в контейнере:

– erase(vecInt.begin()+2, vecInt.begin()+5) – удаление элементов с 3-го по 6-й;

– insert(vecInt.begin(), 5) - вставка значения 5 в начало контейнера.

К использованию insert надо относиться внимательно, так как происходит неконтролируемое увеличение контейнера.

Использование этих функций не очень удобно и не всегда безопасно. Поэтому, как правило, используются следующие функции, не использующие позиции элементов.

– push_back(), emplace_back();//вставка в конец конт.

– vecInt.push_back(7);
  vecStr.push_back("Привет");

– вставка объекта класса

```
class X {
 public: int _a;
 X(int a){_a=a;}
};
```

```
vector<X> vecX;
X x1(4);
vecX.push_back(x1);
или
vecX.emplace_back(4); //Аргументом функции является
                      //аргумент конструктора
```

– тип элементов – указатель на класс:

```
vector<X*> vecXp;
X *ptr=new X;
push_back(ptr);
или просто
push_back( new X);
```

Удаление элементов:

pop_back() – удаление последнего элемента. Элемент удаляется, не возвращая значения; посмотреть на последний элемент, не удаляя его, можно функцией back().

(Список и очередь, но не вектор, поддерживают операции вставки и удаления первого элемента контейнера push_front() и pop_front().)

В vector, как и в deque, работает операция индексации. Если контейнер не пустой, то, например,

```
vecInt[3]=6;
string str=vecStr[9];
```

Используя индексацию, можно просмотреть весь контейнер обычным циклом.

```
unsigned int length=vecInt.size();
for(i=0; i<length; i++)
 vecInt[i]….
```

Но удобнее это делать с помощью итератора.

```
vector<int> ::iterator it;
for (it=vecInt.begin(); it!= vecInt.end(); ++it)
*it …
```

В стандарте C++11 реализован т.н. range-based цикл. Для нашего контейнера

```
for (auto elem : vecX)
  { cout << elem.a; }
```

Алгоритмы

В STL существуют т.н. обобщенные алгоритмы, реализующие большое количество процедур, применимых к последовательным контейнерам, таких, например, как сортировка, поиск по условию, слияние, поиск максимального и минимального элементов. Для их использования к программе необходимо подключить файл algorithm.

Рассмотрим сортировку.

Если типом элементов контейнера является один из стандартных типов языка, для которых определена операция меньше, то все просто, например:

```
sort(vecInt.begin(), vecInt.end());
```

Если элементы – объекты пользовательского класса и сортировать надо по какому-либо полю класса, то в классе необходимо прибегнуть к одному из следующих приемов:

- переопределить операцию “меньше”;
- создать функциональные объекты (ФО);
- создать лямбда-функции.

ФО – класс, в котором перегружена операция вызова функции (). Вызов ОФ: «имя_класса()». Если возвращаемое значение ОФ имеет тип bool, то ОФ называется предикатом.

Предикат используется в качестве параметра алгоритма; алгоритм в качестве аргумента выбирает из контейнера объект, удовлетворяющий ФО.

Ниже следует программа с примерами использования некоторых алгоритмов.

Листинг 4.1

```
class Person {
      string name;
      int age;
public:
bool operator <(Person b) // перегрузка операции
      { return name < b.getName();
```

```
      }
string getName() const { return name; }
int getAge() const { return age; }

Person(string n, int a) : name(n),age(a){}
Person() {}
};

string value;

//Функциональный аргумент для find_if
struct foName {
      bool operator()(Person a) {
       return (a.getName()==value); }
};


int main()
{

      vector<Person> vecP;
      vector<Person>::iterator it;
      Person p[] = { Person("Иванов", 25),
          Person("Сидоров", 20),Person("Петров", 38)
};

      int sz = sizeof(p) / sizeof(Person);
      vecP.resize(sz);
      //Загружаем в контейнер объекты
      for (int i = 0; i < sz; i++)
            vecP[i] = p[i];
      //Используя перегрузку операции <, сортируем по
//name
      sort(vecP.begin(), vecP.end());

            value = "Петров";
//find_if находит первое вхождение объекта согласно
//ФО
it = find_if(vecP.begin(), vecP.end(), foName());

//Вывод резудьтата поиска
if (it != vecP.end()) {
```

```
cout << endl<<"Возраст "<< value<<" "<<it->getAge();
cout <<"  номер " << it - vecP.begin()+1 <<endl;
}
else cout<<endl << value<< " в списке нет" <<endl;
```

//Если несколько элементов удовлетворяют критерию поиска

```
it = vecP.begin();

while (it != vecP.end()) {
  it = find_if(it, vecP.end(),FO());
  if (it!= vecP.end()) {
      //вывод значений
      it ++;
  }
else {      …  break; }
}

// Задание типа контейнера указателем
vector <Person*> vecPP;
auto itPP = vecPP.begin();
vecPP.push_back(new Person("Иванов", 25));
vecPP.push_back(new Person("Сидоров", 20));
vecPP.push_back(new Person("Петров", 38));
itPP = find_if(vecPP.begin(), vecPP.end(), foName());

if (itPP != vecPP.end()) {
 cout << endl << "Возраст " << value << " " <<
      (*itPP)->getAge();
cout << "  Номер " << itPP - vecPP.begin() + 1;
}
 else cout << endl << value << " в списке нет";

//Не забывайте освобождать память от объектов класса
for (itPP = vecPP.begin(); itPP != vecPP.end();
++itPP)
            delete *itPP;
    return 0;
}
```

Для одного контейнера функциональных объектов может быть несколько.

Лямбда-функция представляет собой блок кода, который рассматривается как определение безымянной (анонимной) функции. Формат для C++ (упрощенный):

[] (список параметров) -> тип_возвращаемого_значения { инструкции}

Этот блок можно использовать в качестве параметра алгоритма. Пример сортировки (vecW контейнер для объектов класса Writers, в котором определены персоны, среди атрибутов которых есть дата рождения).

```
sort(vecW.begin(), vecW.end(),
                [](const Writers& one, const Writers&
next) -> bool
{
   /По возрастанию года рождения
       return one.birth < next.birth;
                        }
    )
```

Map

Организован как отсортированное бинарное сбалансированное дерево, поэтому время поиска элементов по сравнению с последовательными контейнерами существенно меньше (до NlogN).

В элементе контейнера хранится не одно значение, а пара (pair): первый(first) является ключом, второй (second) соответствующим ему значением. Типы элементов пары назначаются программистом. Для двух объектов типа pair определены шаблонные операции над парами ==, !=, <, >, <=, >=.

Создание контейнеров:

map<string, string> mapString; //Пустой словарь

Здесь и ключ и значение имеют тип string.

Добавление посредством пар:

контейнер содержит пару ключ-название дня, значение-номер дня недели.

map<string, int> mapW;

map<string, int> :: iterator itmW;

mapW.insert(pair<string, int>("Monday", 1));

Кроме пар, с контейнером можно работать, используя индекс:

контейнер[ключ]=значение;

mapW["Sunday"]=7;

Для поиска элемента используется не один из алгоритмов, а метод класса find.

В листинге 4.2 показаны некоторые приемы работы с контейнером.

Листинг 4.2

```
map<string, int> mapW;
map<string, int> ::iterator itmW;
mapW.insert(pair<string, int> ("Понедельник",
      1));
//Проще
typedef pair<string, int> MyP;
MyP p2("Вторник", 2);
mapW.insert(p2);
//Использование индексации
//Добавляем
mapW["Среда"] = 3;
mapW["Среда"] = 3; //Этот элемент не будет до
бавлен - такой ключ уже есть
//Просмотр
for (itmW = mapW.begin(); itmW != mapW.end();
      itmW++)
{cout << (*itmW).first << (*itmW).second;
}
//Еще вариант просмотра
      for (auto elem : mapW) {
      cout << elem.first << " "<< elem.second;
}

//Удаление
mapW.erase("Вторник");

//Поиск по ключу – find не обобщенный алгоритм,
//а метод класса map
```

```
      itmW = mapW.find("Понедельник");
      if (itmW != mapW.end()){ cout << (*itmW).sec-
ond;
      }

//Для классов
      //Person и массив p[] определен в листинге 4.1
      map<string, Person> mapP;
      map<string, Person> ::iterator itmP;
      //Вставка объекта
      mapP.insert(pair<string,Person>("1",p[0]));
      //Вывод имен
      for (itmP = mapP.begin(); itmP != mapP.end();
itmP++)     {
            cout << (*itmP).second.getName()<<endl;
      }
      //Контейнер содержит указатели на класс
      map<string, Person*> mapPP;
      Person *strp = new Person( "Жуков", 45);
      mapPP.insert(pair<string, Person*>("Жуков",
strp));
      //Индексация
      mapPP[strp->getName()] = strp;
      //Просмотр
      auto itmPP=mapPP.begin();
      for (itmPP; itmPP != mapPP.end();  itmPP++) {
      cout << (*itmPP).second->getName();
      }
      //Поиск
      itmPP = mapPP.find("Жуков");
      if (itmPP != mapPP.end()) {
            cout << (*itmPP).second->getAge();
      }
//Чистка памяти
for (itmPP = mapPP.begin(); itmPP != mapPP.end();
itmPP++)
      delete (*itmPP).second;
```

В map ключи уникальны. Если есть необходимость хранения нескольких значений под одним ключом, то надо воспользоваться контейнером multimap.

Пример работы с ним.

```
multimap < string, Person> mulmap;

//Два объекта под одним ключом
      Person p1[] = { Person("Иванов", 25), Per-
son("Сидоров", 20),Person("Иванов", 38) };
      mulmap.insert(pair<string,Person>("Иванов",
p1[0]));
      mulmap.insert(pair<string, Person> ("Сидоров",
p1[1]));
      mulmap.insert(pair<string, Person> ("Иванов",
p1[2]));
auto itm = mulmap.begin();
for (itm;itm != mulmap.end(); ++itm)
cout << (*itm).first << " " << (*itm).second.get-
Age();
```

//метод count возвращает количество объектов с одним //и тем же ключом; эти объекты расположены в памяти //друг за другом. Выбираем Ивановых старше //20,например, так:

```
      int cou = mulmap.count("Иванов");
      int i=0;
      itm = mulmap.find("Иванов");
      while (i < cou)
      {
            if  (itm->second->getAge() > 20)
      cout << (*itm).first << " " << itm->second-
>getAge();
            ++itm;
            ++i;
      }
```

Эффективным для multimap является метод поиска **equal_range (ключ)**, который возвращает два итератора, первый из которых указывает на начало диапазона элементов с заданным ключом, второй – на конец этого диапазона.

Необходимо отметить, что при использовании в качестве типа элемента контейнера пользовательского класса

последний должен иметь пустой конструктор и конструктор копирования или по умолчанию или созданный согласно «правилу трех».

При использовании контейнеров надо помнить, что за увеличение возможностей, которые они предоставляют программисту, надо платить памятью и увеличением времени выполнения программы.

Варианты задания

Схема выполнения работы следующая.

Имеется перечень из 17 классов. Выбрав один из них, необходимо для работы с ним создать 2 контейнера, один последовательный (vector/ deque), другой типа map. Для этих контейнеров надо создать классы, в которых эти контейнеры объявлены и в них же определены операции сортировки (если нужно), поиска, ввода и вывода.

Исходным материалом для создаваемых контейнеров является текстовый файл, каждая запись которого содержит значения для создания объекта соответствующего класса. Вывод данных предполагается на консоль. Объектов в контейнере должно быть не менее 10, причем тексты должны быть вразумительными, а не qwert. Сочетание слов в задании "Задаваемый пункт (время, ФИО и т.п.)" предполагает ввод этих значений с клавиатуры во время работы программы.

Ключ для словаря и критерии поиска для vector надо создавать, используя информацию о выводимых сведениях. Критерий может быть реализован или в виде функционального объекта или в виде лямбда-функции. Если ключ словаря нельзя сделать уникальным, то воспользуйтесь multimap. Альтернативой последнему может служить map с парой, вторым значением которой является последовательный контейнер, объект которого содержит объекты исходного класса с одинаковыми ключами.

Контейнерные классы следует объединить классом-интерфейсом, который позволяет «на ходу» выбрать тот или иной контейнер. Таким образом, в программе должно быть не менее 4-х классов.

Не забывайте о try-throw-catch.

1. Класс “Студент (Student)” с полями: ФИО студента, номер группы, средний балл, наличие стипендии. Вывести сведения по задаваемой группе:

- фамилии студентов, средний балл у которых больше задаваемой цифры;
- фамилии студентов, не имеющих стипендию.

2. Класс “Расписание_полетов (FlightShedule)” с полями: дата вылета, время вылета, пункт назначения, время прибытия, номер рейса. Вывести номера рейсов, время вылета и время прибытия на задаваемую дату и задаваемый пункт назначения.

3. Класс «Поезд (Train)» с полями: пункт назначения, номер поезда, дата и время отправления, время прибытия. Вывести номера поездов в задаваемый пункт, отправляющихся после задаваемого времени.

4. Класс “ Записная книжка (Notebook)” с полями: ФИО, номер телефона, дата рождения. Вывести по задаваемой дате ФИО и номер телефона человека, день рождения которого отстоит от запрашиваемой даты на 3 дня вперед.

5. Класс “Знаки зодиака (ZodiacSigns)” с полями: ФИО, знак зодиака, день рождения, номер телефона, краткая характеристика знака. Вывести по задаваемой дате рождения телефон, знак зодиака и его характеристику.

6. Класс “Цена(PriceList)” с полями: название товара, название магазина и его адрес, цена товара. Вывести название магазина, его адрес и цену товара по запросу, в котором есть название товара и максимальная сумма, которую готов платить покупатель.

7. Класс “Счет(Invoice)” с полями: расчетный счет(6 знаков) плательщика, расчетный счет покупателя, перечисляемая сумма, дата. Вывести информацию о суммах, снятых с задаваемого расчетного счета плательщика до задаваемой даты.

8. Класс “Университет(University)” с полями: название, город, перечень специальностей (до 5), с пропускным баллом по ЕГЭ каждая. Вывести информацию о баллах ЕГЭ по задаваемой специальности в разных университетах. (Перечень специальностей объедините в одну строку; название конкретной специальности извлекайте как подстроку.)

9. Класс “Общежитие (Hostel)” с полями: ФИО студента, номер группы, номер комнаты, срок окончания проживания. Вывести информацию о студентах задаваемой группы, срок проживания которых заканчивается в задаваемом году.

10. Класс “Вычислительный центр(ComputingCenter)” с полями: номер лаборатории, тип процессора, величину ОЗУ, емкость диска, тип монитора, год выпуска. Вывести информацию о компьютерах во всех лабораториях, которые выпущены ранее вводимого года.

11. Класс “Работник(Employee)” с полями: ФИО, должность, пол, год поступления на работу, год рождения. Вывести информацию о работниках, которые уходят на пенсию в вводимом году (60 или 55 лет).

12. Класс «Библиотека(Library)» с полями: ФИО автора книги, название, год издания, количество экземпляров данной книги в библиотеке. Вывести сведения обо всех книгах заданного автора, начиная с заданного года издания,

13. Класс «Квартира(Flat)» с полями, описывающими квартиру, предназначенную к продаже: площадь, количество комнат, этаж, район. Вывести сведения о квартирах по заявке на покупку, в которой указана только площадь. В выводимых сведениях площадь квартиры может отличаться от заявленной, но не более, чем на 10%.

14. Класс «Выборы(Elections)» с полями: ФИО кандидата, дата рождения, место работы, рейтинг предварительных опросов; Вывести сведения о кандидатах, рейтинг которых превышает вводимое число.

15. Класс «Каталог(Catalogue)» с полями: имя файла, дата создания, объем, тип, дата последнего изменения. Вывести 5 имен файлов заданного типа, отсортированные по дате последнего изменения в сторону убывания.

16. Класс «Город(Town)» с полями: название, средний доход жителя, средняя цена кв.м. жилья, наличие зеленых зон (в процентах к площади). Вывести города с запрашиваемыми ценой квартиры и степенью озеленения.

17. Класс «Кинотеатр(Cinema)» с полями: название фильма, количество мест, количество зрителей, стоимость билета, начало сеанса. Вывести процент заполнения зала для запрашиваемого фильма.

Контрольные вопросы

1. Чем отличается контейнер STL от обыкновенного массива? Покажите контейнер(ы) в своей программе.
2. Что представляет из себя итератор? Что возвращают begin() и end()? У каких классов-контейнеров самые сильные итераторы?
3. Почему ассоциативные контейнеры не нужно сортировать?
4. Чем различаются map и multimap?
5. Почему функции в <algorithm> называют обобщенными?
6. Что представляет из себя функциональный объект?
7. Что представляет из себя λ-функция?

Лабораторная работа №5. Классы, объекты, наследование в C#

Цель работы

Ознакомление с основами объектно-ориентированного программирования на C#.

Основные сведения

Известной монографией по программированию на C# является [1]. Приведем программу, реализующую класс комплексных чисел из листинга 1.1.

Листинг 5.1

```
using System
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Complex
    {    //Поля
        public double real;
        public double image;
        //Конструкторы
           //Пустой
        public Complex() { }
          //С параметрами
        public Complex(double _r, double _im) { real
= _r; image = _im; }

        // Методы
        //Ввод числа
        public void SetComplex()
        {
            string re, im;
   Console.WriteLine("Действительная часть ? ");
```

```
            re = Console.ReadLine();
            this.real=Convert.ToDouble(re);
     Console.WriteLine("Мнимая часть ? ");
            im = Console.ReadLine();
            this.image = Convert.ToDouble(im);

        } // Вывод  числа
        public void ShowComplex()
        {
         Console.WriteLine(real + " +i" + image);
        }
        //Перегрузка операций
        //Сложения
        public static Complex operator +(Complex t,
Complex s)
        {
            Complex tmp = new Complex();
            tmp.real = s.real + t.real;
            tmp.image = s.image + t.image;
            return tmp;
        }
        //Умножения
        public static Complex operator *(Complex t,
Complex s)
        {
            Complex tmp = new Complex();
  tmp.real = s.real * t.real - s.image * t.image;
 tmp.image = s.image * t.real + s.real * t.image;
            return tmp;
        }
    } //Complex

    class Program
    {
        static void Main(string[] args)
        {
          Complex cm1=new Complex();
          Complex cm2=new Complex();
          Complex cm3 =new Complex();
      Console.WriteLine ("1-е число? ");
       cm1.SetComplex();
       Console.WriteLine ("2-е число? ");
```

```
      cm2.SetComplex();
      Console.WriteLine ("Сумма:");
      cm3=cm1+cm2;
      cm3.ShowComplex();
      Console.WriteLine ("Произведение:");
      cm3=cm1*cm2;
      cm3.ShowComplex();
    //Использование конструктора с параметрами
      Complex cm4=new Complex (34.7, 21.8);
      Console.WriteLine ("Массив из 2-х чисел");
       Complex[] cm5=new Complex[2];
     for (int i = 0; i < 2; i++)
     {
         cm5[i] = new Complex();
         cm5[i].SetComplex();
     }
   }
   }
}
```

Не касаясь принципиальных различий между выполнением программ на стандартном C++ и CLR Visual Studio .Net, посмотрим на различия между этой программой и программой 1.1.

Определение класса Complex мало чем отличается от определения такого же класса на языке C++: нет деструкторов, модификаторы доступа public, private предшествуют каждому члену класса и не имеют символа ‘:’ ; после определения класса нет символа ‘;’ . Далее. Пространство имен System используется вместо подключаемых библиотек. Отличается ввод и вывод информации на консоль. При создании объекта класса в C++ оператор new возвращает адрес созданного объекта указателю, а в C# указатель, как правило, не употребляется; вместо него – ссылка.

Существенное различие существует в концепциях типов в C# и в C++. Прежде всего, типы данных в C# подразделяются на типы-значения (value types) и ссылочные типы (reference types). Последними являются системные или пользовательские классы. В частности, массивы, объявляемые в

программе, являются ссылками на системный класс Array. Все классы являются наследниками класса System.Object (object), однако упоминание его в программе как базового необязательно.

Величины ссылочного типа хранятся в динамической памяти, а переменные типов-значений в памяти, управляемой компилятором. Объекты классов создаются только в динамической памяти с помощью оператора new. Освобождение памяти от объектов ссылочного типа производится автоматически.

Главная функция является компонентом класса, а не глобальной функцией, как в C++: глобальных компонент в C# нет. Главная функция является статическим членом, что позволяет исполнительной системе вызывать ее без создания объекта соответствующего класса.

При перегрузке операций количество операндов должно совпадать с количеством операндов стандартной операции. Перегруженная операция является публичной и статической.

В C# широко используемыми являются т.н. свойства. В определении свойства используются ключевые слова get, set, value. Например,

```
class Person
    {
    public string Name { private set; get; }
    public int Age { private set; get; }
    public Person(string name, int age)
           { Name = name; Age = age; }
    }
class Program
    {
        static void Main(string[] args)
        {
          Person pr = new Person(" Antony", 25);
         Console.WriteLine($"Имя: {pr.Name}
                       Возраст: {pr.Age}");
```

```
        }
    }
```

Механизм наследования в C# работает так же, как и в C++. Определения виртуальных функций в производных классах сопровождается словом override, как это показано в листинге 5.2, который является аналогом соответствующего листинга на C++. Вызов виртуальной функции производится через ссылку на базовый класс.

Листинг 5.2

```
abstract class Vehicle
      {
            protected   string brand;//Марка
            protected int price;//цена
            protected int year ; //год выпуска

      public Vehicle(string aBrand, int aPrice,
                                        int aYear)
            {

                  brand = aBrand;
                  price = aPrice;
                  year = aYear;
            }
            //Абстрактная функция
      abstract public void displayVehicle();

}
      //Производный класс
      class Lorry :Vehicle {
      int carrying ; // грузоподъемность

            //Конструктор
      public  Lorry(string aBrand, int aPrice,
                        int aYear, int aCarring)
      :base(aBrand, aPrice, aYear)//Вызов базо
                                  //        вого
      {
            carrying = aCarring;
      }

public override void displayVehicle()  {
```

```
Console.WriteLine("  Марка {0} Цена {1}  Год выпуска
{2}  Грузоподъемность {3}",brand, price,  year,  car-
rying);
      }
};

// Производный класс
class Car : Vehicle
{
      int speed;
public      Car(string aBrand, int aPrice, int
                              aYear, int aSpeed)
            :base(aBrand, aPrice, aYear) {
      speed = aSpeed;
}
//Переопределение виртуальной функции
public override void displayVehicle()
{
Console.WriteLine("  Марка {0}Цена {1}
    Год выпуска {2} Скорость {3}", brand, price,
                                    year, speed);
            }
}

      class Garage
      {
            int nL, nC;
            Vehicle[] veh;

      public      Garage(int n, int m)
      {     nL = n; nC = m;
            veh = new Vehicle[nL + nC];
            string b; int p, ye, c,s;
            Console.WriteLine("Грузовики");
   for (int i = 0; i < nL; i++) {
      b = Convert.ToString(Console.ReadLine());
      p = Convert.ToInt32(Console.ReadLine());
      ye = Convert.ToInt32(Console.ReadLine());
      c = Convert.ToInt32(Console.ReadLine());

      veh[i] = new Lorry(b, p, ye, c);
                  }
```

```
         Console.WriteLine("Автомобили");
   for (int i = 0; i < nC; i++){
      b = Convert.ToString(Console.ReadLine());
      p = Convert.ToInt32(Console.ReadLine());
      ye = Convert.ToInt32(Console.ReadLine());
      s = Convert.ToInt32(Console.ReadLine());

      veh[i+nL] = new Car(b, p, ye, s);
                  }
            }

            //Вывод
      public      void printGarage()
            {
      if (nL + nC == 0) {
Console.WriteLine("Машин нет");  return; }

for (int i = 0; i < nL + nC; i++)
                  {//Вызов виртуальной функции
                        veh[i].displayVehicle();
                  }
            }

      }

      class Program
    {
            static void Main(string[] args)
            {
                  Garage g = new Garage(1, 1);
                  g.printGarage();

         }
    }
```

Схема классов листинга 5.2, изготовленная средствами Visual Studio, показана на рис. 5.1

Аналогом контейнера в C# являются коллекции. Работу коллекций определяют не итераторы, а стандартные интерфейсы. Распространенные классы параметризованных коллекций:

List, LinkedtList – список:

Dictionary, SortedDictionary – набор пар «ключ -значение»;

Queue – очередь;

Stack – стек;

Для работы с коллекциями используется пространство имен System.Collections.Generic.

Варианты задания

В качестве задания на выполнение работы выбирается соответствующий вариант лабораторной работы 3/4. При написании кода необходимо максимально использовать средства C#: свойства, абстрактные классы, интерфейсы и т.п.

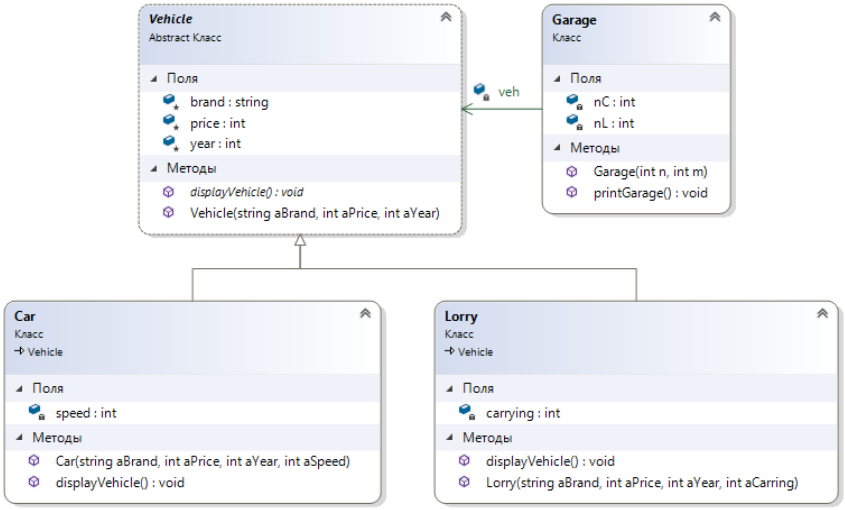

Рисунок 5.1

Контрольные вопросы

1. Как построена система типов в C#?
2. В чем заключается разница между массивом в C++ и в C#(кроме объявления)?
3. В какой строке нижеследующей программы на C# компилятор обнаружит ошибку?

```
class  A {public int x;}
/* ….....*/
A  a1;
a1.x=10;
```

4. Почему функция Main в C# должна быть статической?
5. Чем отличаются определения перегруженных операций в C++ и C#?

Лабораторная работа №6. Создание графического интерфейса

Поведение программ с графическим интерфейсом в отличие от консольных определяется последовательностью событий, которые инициирует пользователь.

Графический интерфейс пользователя (GUI) представляет собой формы (окна), на которых располагаются элементы управления – кнопки, списки, иконки, используемые для обмена информацией пользователя с программой и управления ходом ее выполнения.

Использование элементов управления – нажатие кнопок мышки или клавиатуры, генерация сигналов таймера приводит к наступлению соответствующего события, например, «Нажатие Кнопки» (Button Click). Каждой форме (окну) в программе соответствует свой класс; события от элементов управления, расположенных на форме, обрабатываются методами класса этой формы(окна). После обработки события программа ожидает наступления следующего.

В этом разделе приводится последовательность построения интерфейса в VS для программ на C# и C++, реализующих класс комплексных чисел (листинги 2.1 и 5.3). Интерфейс включает в себя две формы с названием "Начало" и «Продолжение», причем вторая появляется при нажатии кнопки «Продолжить» на первой. На второй форме в первые два окна пользователь вводит действительную и мнимую части 1-го комплексного числа, затем второго и, нажав кнопку `Сложить` при условии имеющегося разрешения, получает их сумму (рис.6.1, 6.2).

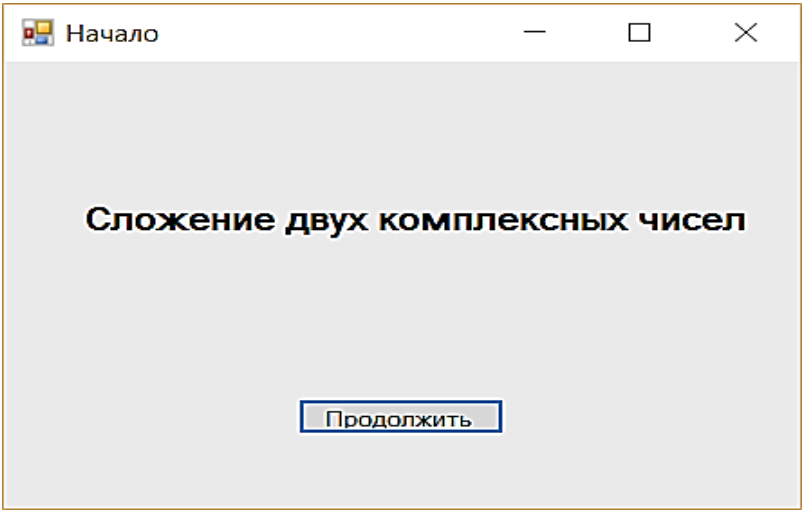

Рисунок 6.1

6.1. Интерфейс для программ на C#

Выполняем `Проект > Visual C#>Классическое приложение Windows>Приложение Windows Form (.Net Framework)+ ComplexGui` (имя проекта).

На экране появляется дизайнер автоматически созданной формы `Form1`. Одновременно создается соответствующий класс. Через правую кнопку мышки выходим на `Свойства формы` (рис.6.3) и занимаемся ее дизайном.

В разделе свойств `Внешний вид` изменяем название (`Text`) в левом верхнем углу появляющейся формы с `Form1` на `Начало` (имя Form1 как объекта в программе остается); здесь же можно установить другой цвет и т.п.

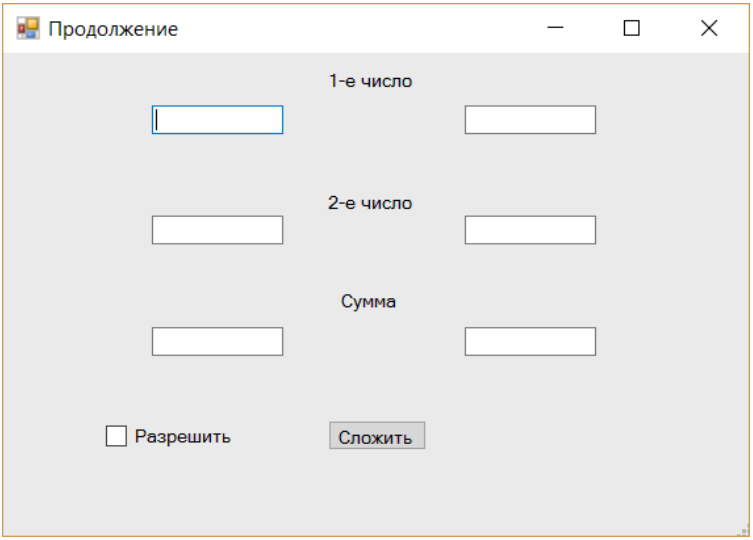

Рисунок 6.2

Далее открываем панель инструментов `Вид>Панель элементов>Стандартные элементы управления` (рис. 6.4)

Выберем элемент `Button`, перетащим его на нашу форму и по щелчку перейдем в ее свойства. (У каждого графического элемента интерфейса имеются собственные свойства.) Изменим на вкладке `Свойства` (по прежнему в разделе `Внешний вид`) название кнопки с `button1` на `Продолжить`.

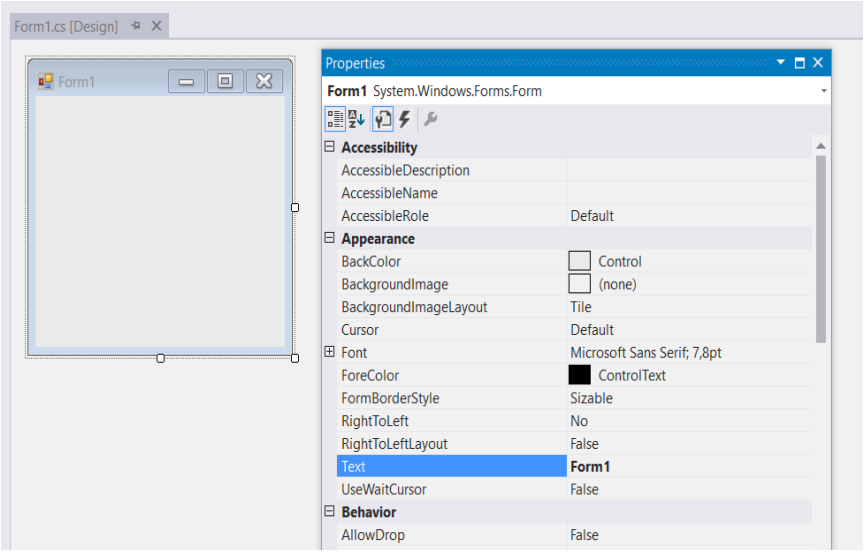

Рисунок 6.3

Выберем элемент `Button`, перетащим его на нашу форму и по щелчку перейдем в ее свойства. (У каждого графического элемента интерфейса имеются собственные свойства.) Изменим на вкладке `Свойства` (по прежнему в разделе `Внешний вид`) название кнопки с `button1` на `Продолжить`.

Перетащим на нашу форму элемент `Label` и напишем текст "Сложение комплексных чисел". Наша форма приобрела вид рис. 6.1.

При помощи `Свойств` меняем название с `Form2` на `Продолжение`; выбрав на `Панели элементов` элемент `TextBox`, создадим на форме 6 таких элементов. Затем перетаскиваем кнопку `Button` и элемент типа `CheckBox`. Изменив названия последних на `Разрешить` и `Сложить` и использовав три элемента `Label` для текстов `1-е число`, `2-е число`, `Сумма`, получим форму рис.6.2. (В программе эти элементы по умолчанию получают имена `textbox1`, `textbox2`, …`textbox6`, `checkbox1`, `button1`, `label1`, `label2`, `label3`. Эти

имена можно изменить с помощью категории `Разработка Свойств`, но делать это надо осторожно).

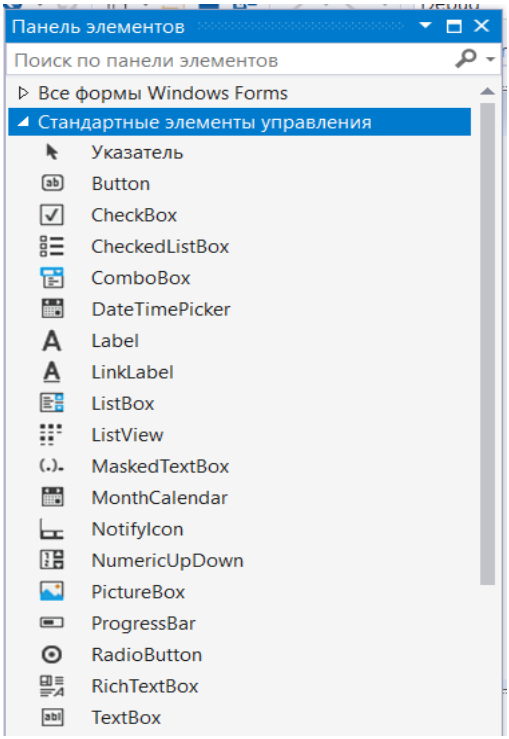

Рисунок 6.4

Предполагается, что по нажатию кнопки `Продолжить` на первой форме открывается вторая. Нажатие кнопки – событие в Windows и, как любое событие, требует обработки. Для обработки надо написать метод, шаблон которого создается автоматически. Перечень событий, которые может генерировать элемент управления, приведен на вкладке События панели Свойств.

Переходим в дизайнер Form1. Два раза щелкаем по кнопке, в результате чего появляется следующий код – шаблон обработчика:

```
       private void button1_Click(object sender,
EventArgs e)
        {    }
```

В параметрах метода указывается источник события (в данном случае кнопка на форме Form1) и его тип (щелчок левой кнопкой мыши).

(Одновременно с созданием шаблона событие button1.Click регистрируется в классе в виде

this.button1.Click += new System.EventHandler(this.button1_Click);

Заполняем его тело двумя инструкциями:

a) создание объекта класса Form2

```
Form2 f = new Form2();
```

b) вывод этого объекта на экран в форме окна диалога.

```
f.ShowDialog();
```

На второй форме нажатие кнопки Сложить – событие, заключающееся в сложении двух комплексных чисел, которое мы собираемся выполнить как перегруженную операцию в классе Complex.

С помощью Проект>Добавить класс добавляем в пространство решений класс ComplexCs из листинга 5.3, выбросив из него методы SetComplex и ShowComplex. Заполняем шаблон обработчика, не обращая пока внимания на кнопку Разрешить.

```
double t1 = System.Convert.ToDouble(text-
Box1.Text);
double t2 = System.Convert.ToDouble(text-
Box2.Text);
    Complex c1 = new Complex(t1, t2);
```

```
double t3 = System.Convert.ToDouble(text-
Box3.Text);
double t4 = System.Convert.ToDouble(text-
Box4.Text);
    Complex c2 = new Complex(t3, t4);
    Complex c3 = c1 + c2;
textBox5.Text = System.Con-
vert.ToString(c3.real);
textBox6.Text = System.Convert.ToString(c3.im-
age);
            }
```

`textBox1.Text` – свойство элемента `TextBox`, которое содержит информацию, внесенную пользователем в первое окно; тип данных в окнах по умолчанию – `string`; так как в нашей операции участвуют числа типа `double`, то к этому типу необходимо привести данные из окна.

Класс ComplexCs и классы форм находятся в одном и том же пространстве имен, поэтому проблем с видимостью не возникает.

Теперь изменяем код для задействования кнопки `Разрешить`.

```
if (checkBox1.Checked)//Выводим сумму по условию
  { textBox5.Text = System.Con-
vert.ToString(c3.real);
   textBox6.Text = System.Con-
vert.ToString(c3.image);
}
```

В программе есть недостаток. При отсутствии значений в окнах нажатие кнопки `Сложить` приводит к исключительной ситуации. Устраняем этот недостаток следующим образом. В свойствах элементов `textbox1, …, textbox4` в строке `Text` категории `Внешний вид` заносим 0.

Подробно с технологией создания сложных графических интерфейсов можно ознакомиться в [18].

6.2. Интерфейс на основе MFC

Библиотека MFC (Microcoft Foundation Classes) создана корпорацией Microsoft специально для работы с C++ еще до создания Visual Studio .Net. В настоящее время для создания программ на C++ с GDI используются IDE Qt Creator или GTK+. Тем не менее, если вы работаете на VS, то можно воспользоваться MFC.

Проект создается по команде `Файл>Создать>Проект> Visual C++> MFC/ATL> Приложение MFC+ComplexMFC (имя проекта)`. Тип приложения выбираем на основе диалоговых окон и использования MFC в общей библиотеке (рис.6.5).

Выбранный вариант означает, что созданное приложение может работать только при условии загруженной библиотеки. Вариант `Использовать MFC в статической библиотеке` позволяет приложению работать без MFC. При этом надо иметь в виду, что во втором случае в приложение загружаются необходимые для его работы классы, функции и т.п., так что приложение становится очень большим. (Программа, созданная в CLR, не может работать самостоятельно, а только вместе со студией или, по крайней мере, с Net Framework.)

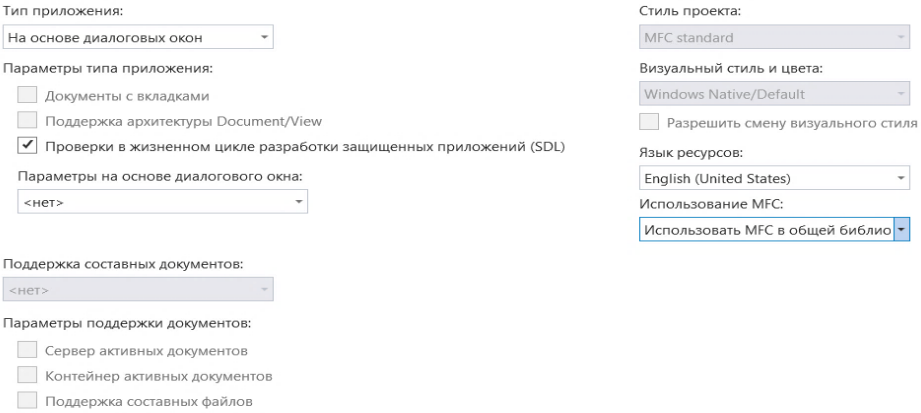

Рисунок 6.5

На следующей странице выбираем `Функции пользовательского интерфейса. Дополнительные функции и Пользовательские классы` оставляем без изменения. В результате получаем программу с главным окном (рис. 6.6), в котором определены два класса: `CComplexMFCApp` и `CComplexMFCDlg`

Первый называется классом главного приложения, второй – классом главного окна. Классы представлены в виде заголовочных файлов и файлов реализации. Эти классы присутствуют в любом приложении MFC. В нашем случае тип главного окна Dialog.

Термина Конструктор формы в MFC нет, изображение окна выполнено визуальным редактором ресурсов. (В файлах ресурсов приложения сосредоточены, кроме окон диалога, `Accelerators` – горячие клавиши, `Bitmaps` – растровые изображения в формате bmp, `Cursors` – курсоры, `Icons` – иконки, `Menus` – меню, `String tables` – таблицы текстовых

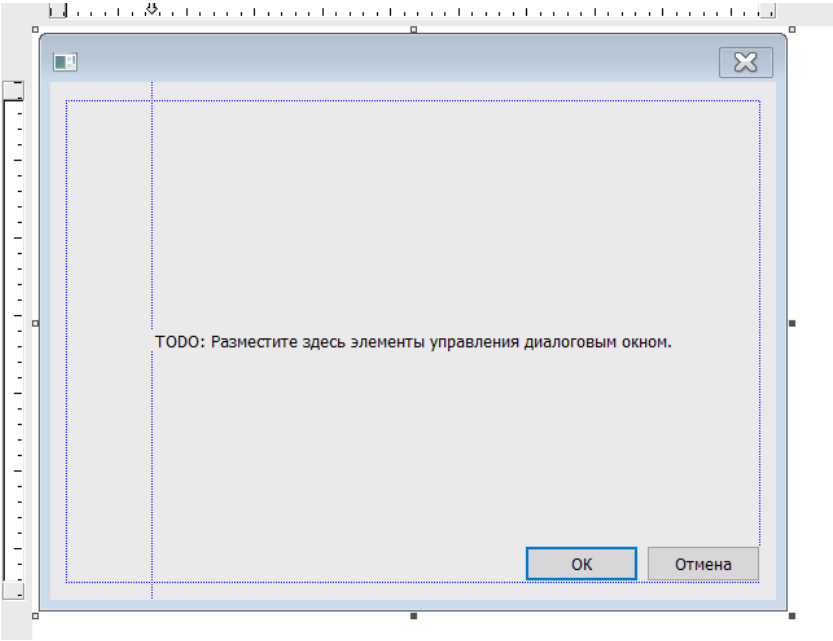

Рисунок 6.6

строк, `Toolbars` – панели управления. Тексты этих файлов можно изменять; внесенные изменения отображаются на изображении окна.)

Дизайн окна выполняется практически так же, как и для C# – с помощью `Свойств` и `Панели инструментов`: изменяем название (`Надпись`) окна в разделе `Свойства>Внешний вид`, создаем кнопку и меняем ее название; для текста на окне предназначен элемент `Static Text`, а не `Label`.

Далее создаем второе окно: `Проект>Добавить ресурс>Dialog` или `Ресурсы>Добавить ресурс`. Новое диалоговое окно по умолчанию получает имя IDD_Dialog1.

Для интерфейса с C# класс для новой формы создается автоматически; в MFC автоматически создается класс только

для главного окна. Для второго и следующих окон класс надо создавать самостоятельно.

Щелкаем по визуальному изображению окна и на появившейся панели (рис. 6.7) записываем имя класса – Dialog1.

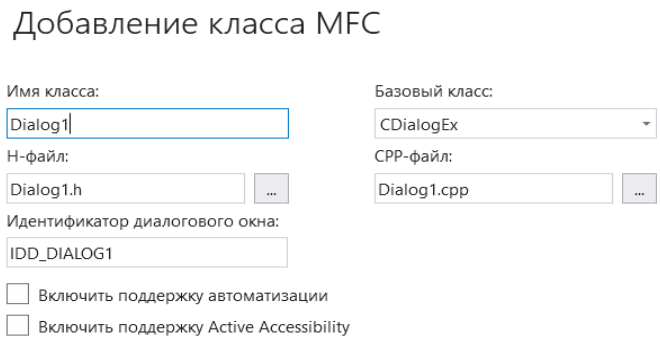

Рисунок 6.7

Тип элементов управления, используемых для чисел и текста, называется `Edit Control`. Перетаскиваем из `Панели элементов` соответствующие элементы; в файле ресурсов они получают имена `IDC_EDIT1, … ,IDC_EDIT6`. Затем формируем элемент `Check Box` (кнопка `Разрешить`) с именем `IDC_CHECK1` и кнопку (Сложить) – `IDC_BUTTON1`.

Элементы типа `Edit Control` в отличие от элементов .Net не имеют свойства `Text`, поэтому для взаимодействия их с программой необходимо внести в нее соответствующие переменные. Последние имеют одну из двух категорий: `Значение` или `Элемент управления`. Щелкаем правой кнопкой мыши на изображении поля и выбираем `Добавить переменную` (рис. 6.8).

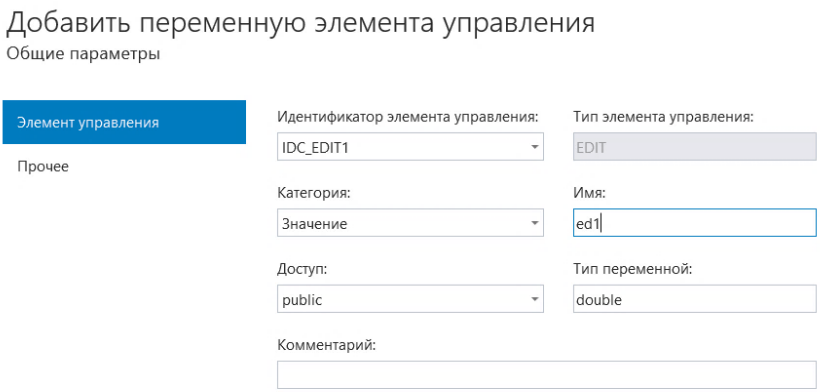

Рисунок 6.8

Выбираем: Категория: `Значение`; Тип переменной: `double`; Имя:`ed1` (произвольное). Аналогичные действия выполняем с всеми элементами `Edit Control`. В результате получаем в классе Dialog1 шесть переменных типа `double`: `ed1,.., ed6`, инициализированных 0.

Директивой `#include "Dialog1.h"` подключаем класс второго окна к классу главного окна.

На главном окне дважды щелкаем по кнопке `Продолжить` и получаем шаблон обработчика.

```
void CComplexMFCDlg::OnBnClickedButton1()
{
// TODO: Add your control notification handler
code //here
}
```

Шаблон обработчика не имеет параметров. Это объясняется тем, что в MFC связь между событием и его обработчиком устанавливается в специальной таблице, называемой Message Map, создаваемой мастером для каждого окна.

Код для создания и вывода объекта окна на экран следующий:

```
Dialog1 *d = new Dialog1;
d->DoModal();
```

Теперь добавляем к приложению наш класс Complex. Для этого создаем новый класс – Проект>Добавить класс, называем его Complex и заполняем .h и .cpp файлы ранее разработанным кодом, естественно без консольного ввода-вывода. Подключаем complex.h к Dialog1.cpp и дважды щелкаем по кнопке Сложить для получения шаблона обработчика. Заполняем его следующим кодом:

```
void Dialog1::OnBnClickedButton1()
{
    UpdateData(true);
    Complex t1(ed1, ed2);
    Complex t2(ed3, ed4);
    Complex t3 = t1 + t2;
    ed5 = t3.real;
    ed6 = t3.image;
    UpdateData(false);
}
```

Вызов функции `UpdateData(true)` активизирует обмен информацией между элементами типа `Edit Control` и соответствующими им переменными: переменные с именем `ed...` принимают значения, введенные пользователем в соответствующие элементы управления окна. Вызов `UpdateData(false)` обеспечивает копирование значений переменных в элементы управления. (Функция `UpdateData` делает обмен не сама, а вызывает другую функцию `DoDataExchange`, сформированную в классе Dialog1 после добавления переменных `ed..`)

Приведенный способ взаимодействия управляющих элементов окна и программы имеет недостатки и не единственный в MFC, но, наверное, самый простой.

Для элемента IDC_CHECK1 определим переменную категории Элемент управления с именем check1 и добавим к коду обработчика, управляющего сложением чисел код

```
if(check1.GetCheck()==1)// Флажок активный
{
UpdateData(true);
…
}
```

Подробно с технологией создания сложных графических интерфейсов можно ознакомиться в [5,6].

Варианты задания

Разработайте графический интерфейс для проекта из предыдущих работ. Если вы хотите использовать WindowsForm, то можно соединить 5-ю и 6-ю работы, если выбираете MFC, то можно присоединить GDI к любой из первых 4-х работ.

Контрольные вопросы

1. Что такое событие в программе с графическим интерфейсом?
2. Чем обрабатывается событие?
3. Как правильно удалять обработчики события?
4. В каком окне(вкладке) проекта можно посмотреть перечень возможных событий для конкретного элемента управления?

Приложения

1.Диаграммы классов UML

Unified Modeling Language (UML) является языком графического описания для объектно-ориентированного моделирования в области программного обеспечения, моделирования бизнес -процессов, системного проектирования [3].

UML является языком широкого профиля, использующий графические обозначения для создания абстрактной модели системы, называемой UML-моделью.

Результатом графического описания являются UML-диаграммы, такие как диаграмма вариантов использования или диаграмма прецедентов (use-case diagram), диаграмма объектов (object diagram), диаграммы взаимодействия, диаграмма последовательностей (sequence diagram), диаграмма классов (class diagram) и др.

Диаграмма классов.

Класс на диаграмме отображается в виде прямоугольника, разделенного на три части. Обязательно заполнение первой из них, т.е. задание уникального имени для класса. Остальные служат для указания атрибутов (полей) объектов класса и операций (методов)(рис.П1).

Имя класса выравнивается по центру и пишется полужирным шрифтом. Имена классов начинаются с заглавной буквы. Если класс абстрактный, то его имя пишется полужирным курсивом.

Посередине располагаются поля (атрибуты) класса. Они выровнены по левому краю и начинаются с маленькой буквы.

Нижняя часть содержит методы(операции) класса. Они также выровнены по левому краю и пишутся с маленькой буквы.

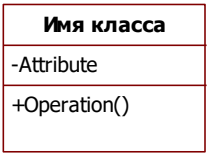

Рисунок П1. Изображение класса

Атрибутам и операциям должны предшествовать знаки:+, – , # определяющие доступ к членам класса.

Открытый доступ (**public)** обозначается знаком "+", закрытый (**private**) знаком " –", защищенный (**protected)** – знаком "#".

Обычно в соответствии с правилами инкапсуляции атрибуты класса объявляются закрытыми. Видимость операций может быть различной.

Справа от атрибута/операции записываются их типы. На рис. П 2 приведен пример.

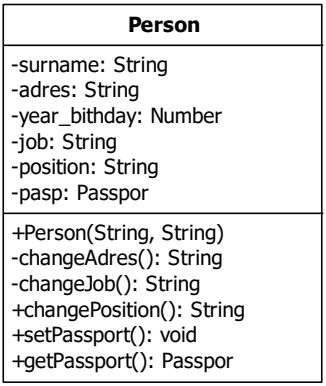

Рисунок П 2

На рис. П2 наименования типов данных не относятся ни к какому конкретному языку. Это позволяет составлять диаграммы для любого языка программирования. Но если известно, что программа разрабатывается, например, для C++, то можно сразу писать string и int.

В диаграмму можно включать объекты классов, в виде прямоугольника с именем объекта и через двоеточие имя класса.

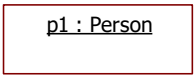

Рисунок П3

Все классы на диаграмме должны быть связаны отношениями. В UML-диаграммах классов имеются четыре разновидности отношений:

1. зависимость;

2. ассоциация;
3. обобщение;
4. реализация.

1.*Зависимость* (dependency) – слабая форма отношения использования (use)такое, при котором изменение в спецификации одного влечёт за собой изменение другого, причём обратное не обязательно. Изображается в виде пунктирной линии, направленной на независимый класс и иногда имеющей метку. Возникает, когда объект выступает, например, в форме параметра или локальной переменной.

Пример на C++

```
class A {
      void f1(B b)   { b.foo(); }
      void f2(B &b) { b.foo(); }
      void f3(B *b) { b->foo();}
      void f4()          { B b; b.foo();}
      };
```

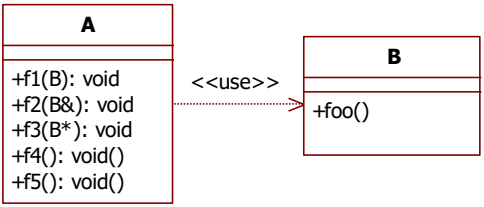

Рисунок П4

2. *Ассоциация (association)* – отношение, показывающее связь между объектами классов объектами такую, что можно перемещаться от объектов одного класса к другому. Является общим случаем композиции и агрегации. Ассоциация

изображается в виде сплошной линии, возможно направленной, часто включающей кратность.

Кратность (multiplicity) определяет, сколько объектов одного класса может быть ассоциировано с одним объектом другого класса. Нуль или больше: 0..*; “много”: *. Один или больше:1..* От одного до сорока:1..40.

Значение по умолчанию соответствует состоянию 1 ("ровно один"). Значение 0..1 ("нуль или один") указывает, что объектов может не быть вовсе, а если есть, то только 1.

Примеры ассоциаций приведены на рис. :

a)Взрослый человек имеет только один паспорт (двойное гражданство не учитываем).

b) В школе учатся дети.

Связь мужа и жены двусторонняя (можно нарисовать стрелки)

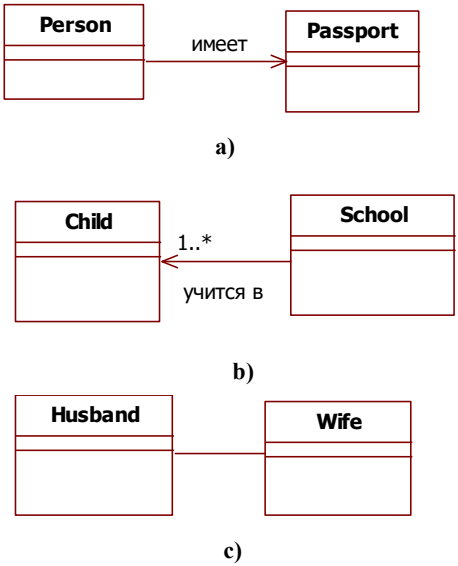

Рисунок П5

Если структура класса известна, то вид ассоциации можно уточнить, воспользовавшись ее разновидностями – агрегацией (aggregation) и композицией (composition).

Они изображаются в виде линий с ромбом на одном конце и, возможно, стрелкой на другом. Когда используют агрегацию и композицию, обычно употребляют термин "это часть целого" (is part of) или " включено" (include). Класс, играющий роль целого, называют контейнером. Ромб при этом изображается на стороне контейнера. Закрашенный ромб отражает композицию, пустой – агрегацию.

Агрегация и композиция различаются по поведению в случае разрушения объекта контейнера. Если вместе с ним разрушаются и объекты включенного класса, то это композиция; если их нет, а есть только ссылки, то агрегация.

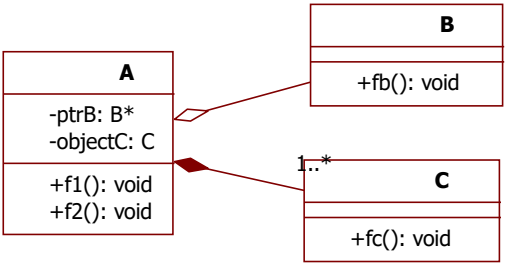

Рисунок П6

Например, в класс A включен указатель на класс B и объект класса C.

```
class B{
void fb(){/*…*/}
};

class C{
```

```
void fc(){/*…*/
};

class A{
  B*  ptrB;
 C   objectC;
public:
 void f1(){ptrB->fb();}
 void f2(){objectC.fc();}
};
```

Отношение классов A и B – агрегация, A и C – композиция.

3. *Обобщение(generalization))* – отношение между базовым и производными классами.

Обобщение изображается в виде сплошной стрелки с полым наконечником, указывающим на базовый класс (рис. П6).

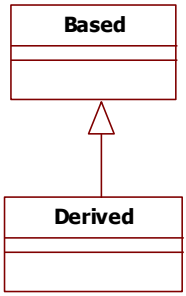

Рисунок П6

4. Реализация – отношение между двумя элементами модели, в котором один элемент (*клиент*) реализует поведение, заданное другим (*поставщиком*), которое клиент обязуется выполнять. Графически реализация представляется так же, как и наследование, но с пунктирной линией. Поставщик, как

правило, является абстрактным классом или классом-интерфейсом.

В C++ отдельная синтаксическая конструкция для интерфейсов не нужна, т.к. интерфейсы определяются при помощи абстрактных классов, а реализация интерфейса производится посредством одиночного и множественного наследования этих классов.

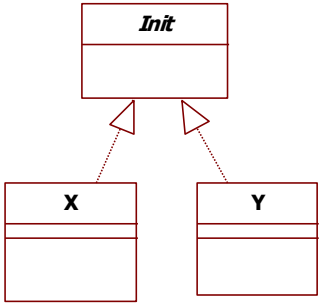

Рисунок П7

2. Обработка исключений в C++

Под исключением понимается некоторая ситуация, возникающая при выполнении программы, вследствие появления которой нормальное продолжение программы невозможно. Дальнейшее выполнение программы определяется программистом на этапе проектирования и носит название обработки исключения. (Прерывание программы пользователем или от аппаратуры (нажатие клавиш Ctrl-Break, сигнал таймера) как исключения не рассматриваются.)

В C++ введен специальный механизм по предотвращению появления исключительных ситуаций. Он организуется при помощи трех ключевых слов: `try, catch` и `throw`.

В блок `try` помещается или вся программа, или фрагмент ее, в котором может возникнуть исключительная ситуация. При ее возникновении возбуждается (`throw`) исключение, которое перехватывается и обрабатывается в блоке `catch`, тип аргумента которого совпадает с типом выражения в операторе throw.

Схема размещения такова:

```
try { …
throw выражение1;
….
throw выражение2;
…..
}//try
catch (тип арг1)
{ ….}
catch (тип  арг2)
{ ….}
```

Выражение может быть простой переменной, ссылкой, указателем или вызовом конструктора класса. Аргументами catch должны быть соответственно типы простых переменных, ссылок, указателей или имя класса.

Механизм обработки исключений, вообще говоря, обеспечивает корректное завершение программы: при возникновении исключительной ситуации в блоке try с помощью

деструктора уничтожаются созданные в начале блока объекты и сворачиваются соответствующие открытые стеки; при сбоях в работе с файлами последние закрываются и т.д.

Операторы throw могут быть включены в функцию, а вызов ее помещен в блок try

```
void MyFunc(){
……..
if(…)throw 1;

……..}//MyFunc

void main()
{try
  { MyFunc(); ……
  }//try
catch(int){……}
}//main
```

Или блоки try и catch могут быть включены в функцию.

Рекомендуется создавать пользовательские классы исключений. Например

```
class  ReadError {
 public: ReadError () {cout << "Error Read";}
};
class  WritedError {
 public: WriteError () {cout << "Error Write";}
};
void main ()
{ try{
 ifstream in("имя");
  if(!in) throw ReadError ();
….
ofsteam out ("имя");
if(!out) throw WriteError();
….
}//try
catch(ReadError){…}
```

```
catch(WriteError){….}
}//main
```

Классы исключений можно объединять в иерархию. Например, класс арифметических ошибок, класс ошибок ввода данных и др. могут быть производными классами некоторого базового класса исключений.

Если при возникновении исключения в блоке не найден соответствующий catch, то система обращается к охватывающему блоку. При этом с помощью возбуждаемого деструктора класса уничтожаются объекты, созданные в блоке; если исключение возникло в конструкторе в момент работы оператора new, то разрушаются только что построенные элементы; при сбоях при работе с файлами последние закрываются. Т.е. функция, в которой возникло необработанное исключение, корректно завершается, если в охватывающем блоке есть обработчик.

Что будет, если исключение не найдет обработчика и в main? В этом случае вызывается стандартная библиотечная функция terminate(). Последняя по умолчанию вызывает abort().

```
int main()
{throw 5;
 return 0;
}
```

3.Классы с полями-указателями

Ниже приводится определение класса – одномерного целочисленного массива с возможностью задания количества элементов конструктором. В определении класса выполняется «правило трех» и для любознательных «правило пяти».

Эти правила применяются в тех случаях, когда одним из полей класса является указатель и, соответственно, в конструкторе класса используется оператор new.

Тогда в классе кроме конструктора следует определять:

a) Деструктор.
b) Конструктор копирования.
c) Операцию копирования с присваиванием.

```
#include <iostream>
//#include <vld.h>
using std::cout;
using std::cin;
using std::move;
class Array1D {
      int* ptr = nullptr;
      int size = 0;//Количество элементов

public:
      //Конструкторы
      Array1D(int);   //1
      Array1D(int*, int);        //2
      //Копирование
      Array1D(const Array1D&);   //3
      //Копирование с перемещением
      Array1D(Array1D&&);//4

 //Деструктор
      ~Array1D();
      //Методы
      void viewArray();
      void setArray();
      //Сложение
       Array1D operator+(Array1D&);
       //Присваивание с копированием
       Array1D& operator=(const Array1D&);
      //Присваивание с перемещением
      Array1D& operator=(Array1D&&);

private:
      void destroy() { if (ptr) delete[] ptr; }
};

Array1D::Array1D(int* pt, int n) {
```

```
      size = n;
      ptr = new int[size];
      for (int i = 0; i < size; i++)
            ptr[i] = pt[i];
}
Array1D::Array1D(const Array1D& s) {
      destroy();
      size = s.size;
      ptr = new int[size];
      memcpy(ptr, s.ptr, size);

}

Array1D::Array1D(int p) {
      size = p;
      ptr = new int[p];
      for (int i = 0; i < size; i++)
            ptr[i] = 0;
}

Array1D& Array1D:: operator=(const Array1D& s) {
      if (this == &s) //предотвращение самокопи-
рования
            return* this;
      destroy();
      size = s.size;
      ptr = new int[size];
      memcpy(ptr, s.ptr, size);
      return *this;
}

Array1D::~Array1D() {
      destroy();
}

void Array1D::setArray() {
      cout << "\nВведите " << size << " элемен-
тов\n";
```

```
      for (int i = 0; i < size; i++)
            cin >> ptr[i];
}
void Array1D::viewArray() {
      for (int i = 0; i < size; i++)
            cout << ptr[i] << " ";
}

Array1D Array1D::operator+(Array1D& s) {
      size = s.size;
      int* tmp = new int[size];
      for (int i = 0; i < size; i++)
            tmp[i] = s.ptr[i] + ptr[i];
      Array1D ret(tmp, size);
      delete[] tmp;
      return ret;
}
////////////////////////////////////////////////
/
//Конструктор и операция копирования с перемеще-
нием
Array1D::Array1D(Array1D&& ar) {
      size = ar.size;
      ptr = ar.ptr;
      ar.ptr = nullptr;

}

Array1D& Array1D:: operator=(Array1D&& s) {
      cout << "= &&\n";
      destroy();
      size = s.size;
      ptr = s.ptr;
      s.ptr = nullptr;
      return *this;
}
////////////////////////////////////////////
```

```
int main()
{
      Array1D ar1(2);
      ar1.setArray();
      Array1D ar6 = ar1;
      Array1D ar7(ar6);
      Array1D ar2(2);
      ar2.setArray();
      Array1D ar3(2);
      ar3 = ar1 + ar2;
      ar3.viewArray();
      //Использование перемещения
      Array1D ar5 = move(ar3);
      ar1 = move(ar2);


}
```

В классе определены три конструктора с параметрами. Первый создает объект с количеством элементов, передаваемым в качестве параметра. Второй – конструктор копирования. Обратите внимание, что копирование значения объекта-аргумента выполняется в объект с новым указателем. Это гарантирует независимость созданного конструктором объекта от имеющегося. Конструктор копирования, созданный по умолчанию, создаст указатель, не используя функцию new, в результате чего в программе будут два имени одного объекта.

По этой же причине перегружена операция присваивания.

Третий конструктор в качестве параметров принимает указатель на целочисленный массив и количество элементов в нем. Этот конструктор используется при перегрузке операции сложения.

Так как в конструкторах применяется функция `new`, создающая массив, то в деструкторе класса необходимо предусмотреть разрушение этого массива функцией `delete[]`.

В стандарте C++11 используется тип T&&. Конструктор с аргументом такого типа называется конструктором перемещения, так как переносит значения полей в новый объект, после чего аннулирует аргумент (в обычном конструкторе копирования аргумент является константой). Такой же процесс происходит и в операторе копирования с перемещением. Для работы с этими конструктором и оператором удобно использовать функцию move.

Создание конструктора перемещения и соответствующего оператора превращает правило трех в правило пяти.

Если правило трех обязательно (во всяком случае переопределение деструктора) к выполнению в классах, конструкторы которого работают с каким-либо ресурсом (в нашем случае со свободной памятью), то наличие в классе конструкторов и операторов перемещения желательно для оптимизации вычислительного процесса, в частности, при работе с контейнерами STL.

Литература

1. Джепикс Ф., Троелсен Э. Язык программирования C# 7 и платформы .NET и .NET Core. - Вильямс, 2018
2. Лафоре Р. Объектно-ориентированное программирование в C++ - Питер, 2018
3. Леоненков А. Нотация и семантика языка UML – М.: ИНТУИТ, 2016.
4. Гамма Э., Хелм Р., Джонсон Р., Влассидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. –СПб.: Питер, 2012
5. Введение в Windows Forms [Электронный ресурс]. – Режим доступа: https://metanit.com/sharp/ windowsforms/ 1.1.php (дата обращения 28.11.2018).

6. Основные понятия MFC [Электронный ресурс]. – Режим доступа:https://msdn.microsoft.com/ ru/library/kkcb3t0w.aspx (дата обращения 28.11.2018).
7. http://www.dpva.info/Guide/GuideMathematics/ http://www.dpva.info/Guide/GuideMathematics/ Периметры, площади …геометрических фигур/ Свойства и площади плоских фигур/Вычисление элементов плоских фигур