издательство

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

УЧЕБНОЕ ПОСОБИЕ

П.В. НОВИКОВ

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Москва • 2019

Министерство науки и высшего образования Российской Федерации

Московский авиационный институт
(национальный исследовательский университет)

П.В. НОВИКОВ

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие к лабораторным работам

Под редакцией профессора, доктора техн. наук О.М. Брехова

Утверждено
на заседании редсовета
18 февраля 2019 г.

Москва
Издательство МАИ
2019

УДК 681.3.06

Новиков П.В. Объектно-ориентированное программирование: Учебное пособие к лабораторным работам / Под редакцией проф., д-ра техн. наук О.М. Брехова. – М.: Изд-во МАИ, 2019. – 124 с.

Описаны основные приёмы объектно-ориентированного программирования на примерах алгоритмических языков C++ и Pascal with Object, рассмотрены теоретические понятия и конструкции, общие для всех языков программирования, поддерживающих объектно-ориентированную парадигму. Приведены образцы и примеры программ, и представлены различные задания на создание классов и объектов, на применение механизма наследования и механизма композиции классов, на разработку переопределённых методов и перегруженных методов, а также виртуальных функций. Рассмотрены и изучены образцы создания и применения абстрактных классов, образцы разработки полиморфных объектов, динамических объектов и т.п. на примере различных графических объектов. В заключение исследованы различные аспекты использования объектов как входных и выходных параметров методов, а также самостоятельных процедур и функций.

Пособие предназначено для студентов и аспирантов вузов, обучающихся по направлению «Информатика».

Рецензенты:

кафедра прикладной информатики Института цифрового образования МГПУ (зав. кафедрой д-р техн. наук, профессор О.Н. Ромашкова);

д-р техн. наук, профессор *Г.Н. Лебедев*;

канд. техн. наук, ст. науч. сотр., нач-ник НИС НИИ “Аргон” *Л.А.Сальман*

ISBN 978-5-4316-0625-0

© Московский авиационный институт (национальный исследовательский университет), 2019

ПРЕДИСЛОВИЕ

Настоящее учебное пособие предназначено для студентов специальностей «Программное обеспечение вычислительной техники, автоматизированных систем, комплексов и сетей», «Вычислительные машины, системы и сети» и «Программная инженерия», выполняющих лабораторные работы по курсу «Объектно-ориентированное программирование». Материал этого курса опирается на весьма подробные сведения о языках Паскаль и Си, получаемые студентами на первом и втором курсах при изучении дисциплин «Информатика» и «Программирование на языках высокого уровня». Вследствие этого оказывается возможным сразу излагать основные принципы объектно-ориентированной парадигмы, общие для всех алгоритмических языков, не углубляясь в особенности синтаксиса конкретного языка программирования. Выполнение предлагаемых в пособии лабораторных работ, рассчитанное на 32 часа учебного времени, позволяет освоить основные, базовые приёмы разработки классов, создания объектов и работы с объектами. В цикле работ изучаются основные приёмы этой парадигмы: инкапсуляция, наследование классов, полиморфизмы методов (перегрузка и переопределение). Даются понятия о методах-конструкторах и методах-акцессорах, об открытых, закрытых и защищённых компонентах классов, рассматриваются преимущества виртуальных функций, абстрактных классов и полиморфизм объектов. Композиция классов и объектов, часто используемая на практике, но мало освещённая в литературе, специально рассматривается в отдельной работе. Изучаются особенности работы с динамическими объектами и использование объектов как входных и выходных параметров методов и функций.

С целью упрощения перехода учащихся к новой технологии программирования для выполнения работ выбраны гибридные языки программирования C++ и Object Pascal (версия фирмы Borland), совмещающие императивный и объектно-ориентированный подходы, в отличие от строго объектно-ориентированных языков, таких, как Java или C#. Освоив разработку классов как особых типов данных, а объектов как переменных этих типов, несложно перейти к программированию в интегрированных средах, таких, как Delphi, C++Builders, Java, Visual Studio C++, Visual Studio C# и т.п.

Общий порядок выполнения лабораторных работ

1. Внимательно и детально изучить общую часть методических указаний к лабораторной работе, предварительно ознакомившись с целью этой работы.

2. Изучить на основе приведённых примеров новые синтаксические конструкции объектно-ориентированных программ. Тексты примеров могут быть хорошей заготовкой к выполнению индивидуального задания.

3. В соответствии с вариантом задания разработать и отладить объектно-ориентированную программу на двух алгоритмических языках с использованием новых приёмов программирования и новых синтаксических конструкций.

4. Убедившись в правильности работы программы, продемонстрировать её успешное функционирование преподавателю.

5. После того как выполнение лабораторной работы зачтено преподавателем, следует распечатать на принтере текст программы с примерами её функционирования.

6. Файлы с текстами программ, присвоив им оригинальное имя, следует сохранить на личном носителе (например, флешке) для дальнейшей работы на домашнем компьютере, после чего удалить эти файлы с жёсткого диска.

7. Оформив отчёт, защитить лабораторную работу у преподавателя.

Все задания должны быть выполнены на алгоритмических языках C++ и Object Pascal.

Содержание отчётов по лабораторным работам

1. Тема работы.
2. Цель работы.
3. Индивидуальное задание на выполнение.
4. Проектирование классов (подробное описание имён и характеристик полей данных и методов разрабатываемых классов).
5. Текст программ, использующих спроектированные классы и иллюстрирующих работу с созданными объектами.
6. Результаты работы программ.
7. Выводы на основе выполненной работы.
8. Список использованных источников (литература).

РАБОТА №1. КЛАССЫ И ОБЪЕКТЫ. ИНКАПСУЛЯЦИЯ

Цель работы – изучение понятий объект и класс, а также понятий поля данных и методы класса. Освоение технологии разработки классов и создания объектов на примере языков программирования C++ и Object Pascal. Работа с открытыми и закрытыми полями данных и методами.

Общие сведения

Объект есть *самостоятельный программный компонент*, наделённый собственными индивидуальными характеристиками (**полями данных**) и имеющий своё оригинальное поведение (задаваемое **методами**). Соединение в объекте данных и методов обработки этих данных называется **инкапсуляцией.**

Для создания объектов практически во всех современных языках программирования необходимо создавать особые пользовательские типы данных **– объектные типы данных** (называемые обычно **классами**).

Переменные объектных типов данных, создаваемые в программе, есть **экземпляры** классов, т. е. вышеназванные объекты.

В отличие от традиционных процедур и функций, методы (или *функции, члены класса,* как их называют в C++) не вызываются сами по себе. Вызов каждого метода связан с конкретным объектом и является **обращением** к объекту (**посылкой сообщения** к объекту). Тем самым программирование в рамках объектно-ориентированной технологии есть разработка классов, создание объектов и обмен сообщениями между объектами (посылка сообщений объектам).

При таком подходе непосредственное обращение к полям данных объектов не считается рациональным приёмом. Напротив, поля данных рекомендуется закрывать, то есть делать доступными только для своих собственных методов (характерная особенность вышеназванной инкапсуляции). Закрытыми могут быть не только поля данных, но и методы. Такие методы нужны для вызова из других методов класса и не предназначены для посылки сообщений объектам. В языках C++ и Object Pascal закрытые поля данных и закрытые методы вместе составляют закрытые секции класса. Аналогично открытые поля данных и открытые методы составляют в этих языках открытые секции класса. Закрытые секции классов объявляют с модификатором **private**, а открытые – с модификатором **public**. При этом в теле методов класса всегда можно обращаться как к открытым, так и к закрытым методам и полям данных этого же класса.

В алгоритмических языках есть правила умолчания. Так, в C++ поля данных по умолчанию являются закрытыми, а функции, члены класса – открытыми. В то же время в Object Pascal и поля данных, и методы по умолчанию являются открытыми. Однако для удобства рекомендуется *явно* задавать доступ.

Для создания объектов следует применять особый метод, так называемый **конструктор**. Этот метод осуществляет присвоение начальных значений полям данных при создании объектов. Также, при работе с динамической памятью, конструкторы могут использоваться для выделения областей памяти под поля данных создаваемых объектов. В C++, как и в большинстве языков программирования, имя конструктора совпадает с именем его класса. Но в Object Pascal,

как и в некоторых других языках, конструктор имеет своё собственное оригинальное имя и обозначается зарезервированным словом **constructor** (вместо использующихся в Object Pascal обозначений procedure и function).

Если поля данных закрыты, то для осуществления доступа к ним создают специальные открытые методы, так называемые методы-***акцессоры.*** Одни из таких методов возвращают значения закрытых полей данных объектов, а другие – изменяют эти закрытые значения.

В силу того что объектно-ориентированная парадигма не является особенностью какого-либо одного языка программирования, а представляет собой набор общих принципов, стоящий выше конкретной реализации парадигмы в том или ином алгоритмическом языке, при разработке проекта класса (объектного типа данных) на этапе *объектно-ориентированного проектирования* следует описывать компоненты будущего класса в виде, не связанном с синтаксисом языка. Тогда один проект класса может быть реализован на разнх языках.

Ниже приведён пример (см. пример 1.1) проекта класса Complex (комплексное число). С целью максимального упрощения проекта в нём не предусмотрены методы для арифметических операций с комплексными числами.

В приведённых далее примерах 1.2 и 1.3 показаны программы, реализующие разработанный проект класса (объектного типа данных) Complex на языках C++ и Object Pascal. Объектами (экземплярами) класса Complex являются комплексные числа C1 и C2. Показаны механизм создания объектов и обращение к объектам с помощью методов. Также показаны обращение в теле методов класса к закрытым полям данных этого класса (например, в теле методов-акцессоров) и вызов закрытых методов в теле других методами этого же класса.

Из приведённых примеров видно, что программы на языках C++ и Object Pascal фактически состоят из трёх разделов: раздел объявления классов, раздел определения методов классов и раздел основной программы, где используются объекты созданных классов. Эти языки предоставляют программистам удобную возможность разделять объявление класса и определение методов класса, отсутствующую, например, в языках java и C#.

В разделе объявления класса каждому классу присваивается имя, объявляются открытые и закрытые секции, в которых присваиваются полям данных имена и типы, а методам – имена, типы входных параметров и типы возвращаемых значений. Фактически такая возможность позволяет программисту осуществлять вышеназванное объектно-ориентированное проектирование прямо в программе, так как объявление класса есть декларативная часть проекта класса, но уже с учётом синтаксиса языка. Таким образом, программист может определять характеристики полей данных, а также поведение будущих объектов, руководствуясь только особенностями решаемой задачи и не вдаваясь в детали программной реализации методов.

Каждый метод класса описывается в разделе определения методов и может быть отлажен независимо от других методов. А в языке C++ обектно-ориентированные программы успешно выполняются при объявленных, но неопределённых методах, если эти методы не вызываются в основной программе.

Пример 1.1. Проектирование класса Complex

Имя класса: Complex

Поля данных (закрытые, вещественные):
Re – действительная часть;
Im – мнимая часть.

Методы:

1) **Открытые** методы

а) Конструктор, входные параметры, вещественные InitRe и InitIm – начальные значения полей Re и Im.

б) Методы-акцессоры

GetRe — возвращает вещественное значение поля данных Re,
GetIm — возвращает вещественное значение поля данных Im,
PutRe (новое Re) – задаёт новое вещественное значение для поля Re,
PutIm (новое Im) – задаёт новое вещественное значение для поля Im.

в) Прочие открытые методы

PrintComp — печатает комплексное число в алгебраической форме (значение вещественной части Re и значение мнимой части Im) с переносом строки.
PrintTrig — печатает комплексное число в тригонометрической форме, то есть сначала модуль комплексного числа, вычисляемый методом Amp, а затем фазу, вычисляемую методом Fi, с последующим переносом строки.

2) **Закрытые** методы (вызываются в методе с открытым доступом PrintTrig)

Amp – возвращает вещественный модуль A комплексного числа,

$$A = \sqrt{\mathrm{Re}^2 + \mathrm{Im}^2}$$

Fi — возвращает вещественную фазу φ комплексного числа.

$$\varphi = \begin{cases} \operatorname{arctg}\left(\frac{\mathrm{Im}}{\mathrm{Re}}\right), \text{при } \mathrm{Re} > 0, \\ \operatorname{arctg}\left(\frac{\mathrm{Im}}{\mathrm{Re}}\right) - \pi, \text{при } \mathrm{Re} < 0 \text{ и } \mathrm{Im} < 0, \\ \operatorname{arctg}\left(\frac{\mathrm{Im}}{\mathrm{Re}}\right) + \pi, \text{при } \mathrm{Re} < 0 \text{ и } \mathrm{Im} \geq 0, \\ \frac{\pi}{2}, \text{при } \mathrm{Re} = 0 \text{ и } \mathrm{Im} > 0, \\ -\frac{\pi}{2}, \text{при } \mathrm{Re} = 0 \text{ и } \mathrm{Im} < 0, \\ 0, \text{при } \mathrm{Re} = 0 \text{ и } \mathrm{Im} = 0. \end{cases}$$

Последняя строка в формуле фазы нужна для устранения неопределённости.

Пример 1.2. Разработка класса и создание объектов на языке C++

```
#include <conio.h>
#include <iostream.h>
#include <math.h>

class Complex                    /* объявление класса */
 {
    double Re;                    // закрытые по умолчанию
    double Im;                    //                  поля данных

  public:                       // секция открытых методов
   Complex (double, double);      // конструктор класса
                          /*  объявление методов-акцессоров */
   double GetRe();              // функции, возвращающие
   double GetIm();              //                  значения закрытых полей
   void PutRe(double);          // процедуры, задающие
   void PutIm(double);          //                  значения закрытых полей
/*  объявления иных открытых методов */
   void PrintComp();            // процедуры печати чисел  в алгебраической
   void PrintTrig();            //                  и в тригонометрической форме

  private:                      // секция закрытых методов
    double Amp();
    double Fi();
 };
                          /* определение методов класса  */

Complex:: Complex (double InitRe, double InitIm)
 {
    Re=InitRe;                                  // присвоение закрытым полям
    Im=InitIm;                                  // данных начальных значений
 }

double Complex::GetRe() {return Re;} // возвращает значение закрытого поля Re

double Complex::GetIm() {return Im;} // возвращает значение закрытого поля Im

void Complex ::PutRe (double NewRe) { Re=NewRe; }
// задаёт новое значение NewRe закрытому полю данных Re

void Complex ::PutIm (double NewIm) { Im=NewIm; }
// задаёт новое значение NewIm закрытому полю данных Im
```

```
void Complex ::PrintComp() {cout<<"Re="<<Re<<" Im="<<Im<<endl;}
                    // бесформатный вывод закрытых полей с переносом строки

void Complex ::PrintTrig()  // вызов закрытых методов из других методов класса
     { cout<<"Am="<<Amp()<<" Fi="<<Fi()<<endl; }

double Complex ::Amp() {return  sqrt(Re*Re + Im*Im);}   // вызов закрытых полей

double Complex ::Fi() { return (((Im==0)&&(Re==0))? 0 : atan2(Im, Re)); }
   //использование тернарной операции вместо оператора IF для экономии кода

void main()     // основная программа
{
  clrscr();         // очистка экрана
  Complex C1(1, 0), C2(3, 4); // создание C1 и  C2 -  экземпляров класса Complex

        // обращение к  объекту -  комплексному  числу C1
  cout<<"C1:  Re="<<C1.GetRe()<<" Im="<<C1.GetIm()<<"\n";
                              // вывод на печать полей данных с помощью акцессоров
        getch();  // ожидание нажатия клавиши
  C1.PrintComp();  // вывод на печать полей данных методом PrintComp
        getch();
  C1.PrintTrig();  //вывод на печать полей данных с помощью метода PrintTrig
        getch();
  C1.PutRe(-12);
  C1.PutIm(5);
  C1.PrintComp();
        getch();
  C1.PrintTrig();
        getch();
                        // обращение к  объекту -  комплексному  числу C2
  cout<<"C2:  Re="<<C2.GetRe()<<" Im="<<C2.GetIm()<<"\n";
        getch();
  C2.PrintComp();
        getch();
  C2.PrintTrig();
        getch();
  C2.PutRe(21);
  C2.PutIm(-20);
  C2.PrintComp();
        getch();
  C2.PrintTrig();
        getch();
};
```

Пример 1.3. Разработка класса на языке Object Pascal

```
{$N+}
program Comp1;
uses Crt;

type

 pComplex=^Complex;                    { объявление указателя на класс }

 Complex = object      { объявление класса - объектного типа данных }
              { открытые  по умолчанию  методы класса: }
    constructor Init(InitRe, InitIm: double);          { конструктор класса }
    function     GetRe: double;      { объявления                          }
    function     GetIm: double;      {          методов-акцессоров         }
    procedure   PutRe(NewRe: double);
    procedure   PutIm(NewIm: double);

 private { закрытая секция }
              Re: double;                          {   закрытые          }
              Im: double;                          {          поля данных }
   function Amp: double;                           {   закрытые          }
   function   Fi  : double;                        {              методы }

  public
   procedure PrintComp;  { процедуры печати чисел  в алгебраической   }
   procedure PrintTrig;     {              и в тригонометрической форме}
end;

              { определение методов класса - объектного типа данных: }
constructor Complex.Init(InitRe, InitIm: double);
  begin
     Re:=InitRe;
     Im:=InitIm;
  end;

function Complex.GetRe: double; begin GetRe:=Re end;

function Complex.GetIm: double; begin GetIm:=Im end;

procedure Complex.PutRe(NewRe: double); begin Re:=NewRe end;

procedure Complex.PutIm(NewIm: double); begin Im:=NewIm end;
```

```
function Complex.Amp: double;  begin  Amp:=sqrt(Re*Re + Im*Im)  end;

function Complex.Fi: double;
  begin
    if (Re=0) and (Im=0) then Fi:=0;
    if (Re=0) and (Im<0) then Fi:= -0.5*Pi;
    if (Re=0) and (Im>0) then Fi:=0.5*Pi;
    if (Re>0) then Fi:=ArcTan(Im/Re);
    if (Re<0) then if (Im>=0) then Fi:=ArcTan(Im/Re)+Pi
                              else Fi:=ArcTan(Im/Re) -Pi
  end;

procedure Complex.PrintComp; begin  writeln('Re=', Re:7:3,' Im=', Im:7:3)  end;

procedure Complex.PrintTrig; begin  writeln('Amp=',Amp:7:3,' Fi=', Fi:7:3)  end;

var
  C1, C2: Complex;          {объявление объектов C1 и C2 - экземпляров класса}
begin
  C1.Init(1,0); {создание объектов – задание начальных значений полям данных}
  C2.Init(3,4);                          {с помощью метода-конструктора Init}
  clrscr;  {очистка экрана}

                  {обращение к  объекту -  комплексному  числу C1}
  writeln('C1:  Re=',C1.GetRe:7:2,' Im=',C1.GetIm:7:2);
        readln;
  C1.PrintComp;   readln;
  C1.PrintTrig;     readln;
  C1.PutRe(-12);
  C1.PutIm(5);
  C1.PrintComp;    readln;
  C1.PrintTrig;     readln;

                  {обращение к  объекту -  комплексному  числу C2}
  writeln('C2:   Re=',C2.GetRe:7:2,' Im=',C2.GetIm:7:2);
        readln;
  C2.PrintComp;    readln;
  C2.PrintTrig;     readln;
  C2.PutRe(21);                {посылка сообщений к объектам посредством}
  C2.PutIm(-20);                       {методов-акцессоров PutRe и PutIm}
  C2.PrintComp;    readln;
  C2.PrintTrig;     readln;
end.
```

Удобное наличие не объявленной ни в одном классе основной программы (**main** в C++ или **program** в Object Pascal) характеризует эти языки как гибридные языки объектно-ориентированного программирования, в отличие от чисто объектно-ориентированных языков, таких, как Java и C#.

Реализация отдельных методов на обоих языках, как видно, весьма схожа. Отличия проявляются, в основном, при использовании функций из встроенных библиотек (оригинальных в каждом языке программирования). Так, в методе Fi программа на C++ использует функцию atan2(Im,Re), возвращающую значения угла во всей комплексной плоскости, а в программа на Object Pascal использует функцию ArcTan(), возвращающую значение угла только в двух квадрантах. Поэтому методы Fi на этих языках весьма различны.

Типичной ошибкой при освоении объектно-ориентированного подхода является мнение о том, что один класс может иметь только один объект. В связи с этим в задании к этой работе предлагается создать не менее трёх различных объектов, имеющих различное поведение. Для наглядности создаваемые объекты представляют собой движущиеся геометрические фигуры.

В процессе создания этих геометрических объектов следует одну часть полей данных и методов назначать для управления положением геометрических центров объектов, а другую часть – для изменения геометрических размеров этих фигур относительно их центров. Тогда сложные движения легко создать с помощью комбинации простых движений. Оригинальное поведение каждого из трёх созданных объектов должно определяться отдельным методом, отличным от других. Два разных метода должны задавать два простых движения: одно — относительно геометрического центра, а другое – движение самого центра. Третий же из методов, задающий сложное движение, должен моделировать это движение путём комбинации простых, то есть путём согласованного вызова двух методов, задаюших простые движения.

В примере 1.4 показан проект класса «Колесо» (Wheel), в примерах 1.5 и 1.6 приведены программы на C++ и на Object Pascal, решающие следующую задачу: «*Разработав класс “колесо”, создать три объекта, изображающие:*

а) вращающееся “колесо”,

б) скользящее “колесо”,

в) катящееся “колесо”».

Важно заметить, что методы моделируют лишь элементарные составляюшие движений, а полные циклы движений вынесены в основную программу. Такой приём необходимо использовать при выполнении этой лабораторной работы.

В программе на языке C++ используется указатель на компоненты класса **this->** для того, чтобы упростить запись конструктора **Wheel** и методов-акцессоров **PutX** и **PutFi** и преодолеть неоднозначность работы компилятора типа <поле> X= <аргумент>X в ситуации, когда имя входного аргумента совпадает с именем поля данных. Например, вместо определения метода **PutX(int NewX) { X = NewX; }** будет **PutX(int X) { this -> X = X; }**.

К сожалению, аналогичная указателю **this** псевдопеременная **self** в языке Turbo Pascal 7.0 не может создать аналогичную конструкцию **self.X:=X.**

Пример 1.4. Проектирование класса Wheel – «Колесо»

Имя класса: Wheel

Поля данных (закрытые):

X, Y — целочисленные экранные координаты центра колеса;

R — целочисленный радиус колеса;

Fi — вещественный угол поворота колеса.

Методы (открытые):

а) Конструктор, входные параметры – целочисленные начальные значения полей данных X, Y и R, а также и вещественное начальное значение поля данных Fi.

б) Методы-акцессоры

GetX — возвращает целочисленное значение поля данных X;

PutX (новое X) — задаёт новое целочисленное значение для поля X;

PutFi (новое Fi) — задаёт новое вещественное значение для поля Fi.

в) Прочие методы:

Show – рисует колесо в виде обода с двумя пересекающимися осями с помощью окружности (радиус R, центр в точке с координатами X и Y) и с помощью двух пересекающихся отрезков, соединяющих точки 1 и 3, а также точки 2 и 4. Координаты концов отрезков вычисляются как

$$X_1 = X + R \cdot \sin(\varphi), \qquad Y_1 = Y - R \cdot \cos(\varphi),$$

$$X_2 = X + R \cdot \sin\left(\varphi + \frac{\pi}{2}\right), \qquad Y_2 = Y - R \cdot \cos\left(\varphi + \frac{\pi}{2}\right),$$

$$X_3 = X + R \cdot \sin(\varphi + \pi), \qquad Y_3 = Y - R \cdot \cos(\varphi + \pi),$$

$$X_4 = X + R \cdot \sin\left(\varphi + \frac{3\pi}{2}\right), \qquad Y_4 = Y - R \cdot \cos\left(\varphi + \frac{3\pi}{2}\right),$$

где переменные

$$\Delta X,\ \varphi,\ \Delta\varphi,\ X_1,\ Y_1,\ X_2,\ Y_2,\ X_3,\ Y_3,\ X_4,\ Y_4$$

соответствуют идентификаторам DX, Fi, Dfi, X1, Y1, X2, Y2, X3, Y3, X4, Y4.

Hide – стирает колесо с экрана, а именно прячет, рисуя цветом фона. При этом рабочая переменная сохраняет код цвета линий на экране, затем текуший цвет линий меняется на цвет фона, и колесо рисуется цветом фона. По окончании восстанавливается сохранённый прежний цвет линий.

Slide(целый DX) — сдвигает колесо вдоль оси X на DX пикселей, для чего прячет колесо методом Hide, потом увеличивает координату X центра колеса методом PutX на величину DX, затем Show рисует колесо на новом месте;

Turn(вещественный DFi) — поворачивает колесо по часовой стрелке на угол DFi, для чего сначала прячет колесо методом Hide, затем увеличивает координату Fi угла поворота колеса методом PutFi на DFi, после чего рисует колесо на новом месте методом Show;

Roll(вещественный DFi) — перекатывает колесо по экрану вдоль оси X, поворачивая колесо вокруг его центра на угол DFi методом Turn и сдвигая центр колеса на величину DX методом Slide. Сдвиг DX для создания эффекта перекатывания синхронизирован с углом DFi и рассчитывается по формуле:

$$\Delta X = \Delta\varphi \cdot R\ .$$

Пример 1.5. Создание графических объектов на языке C++

```
#include <graphics.h>
#include <iostream.h>
#include <conio.h>
#include <math.h>
#include <dos.h>

class Wheel                          //объявление класса «Колесо» – Wheel
 {
   private:
            int X, Y;                // координаты центра колеса
            int R;                   // радиус колеса
        double Fi;                   // угол поворота колеса

    public:
                Wheel(int, int, int, double); // конструктор
           void PutX(int);           // методы-
             int GetX();             //    акцессоры
           void PutFi(double);       //      класса
           void Show();                          // «показать» объект
           void Hide();                          // «спрятать» объект
           void Slide(int);                        // скольжение колеса
           void Turn(double);                      // вращение колеса
           void Roll(double);                      // перекатывание колеса
 };

 /*  определение методов объявленного класса */

         Wheel::Wheel(int X, int Y, int R, double Fi)
                          {
                            this ->X=X;
                            this ->Y=Y;
                            this -> R=R;
                            this ->Fi=Fi;
                          }

    int Wheel ::GetX() { return (X); }

  void Wheel ::PutX(int X) { this -> X=X; }

  void Wheel ::PutFi(double Fi) { this -> Fi=Fi; }
```

```
void Wheel ::Show()
                {
                 int  X1, Y1, X2, Y2;  // рабочие переменные – координаты концов
                 int  X3, Y3, X4, Y4;  // двух пересекающихся осей колеса
                 circle(X, Y, R);        // нарисовать окружность – обод колеса
                  X1=X+R*sin(Fi);                          //
                  Y1=Y -R*cos(Fi);                          // расчёт
                  X2=X+R*sin(Fi+M_PI_2);                    // координат
                  Y2=Y -R*cos(Fi+M_PI_2);                   // концов
                  X3=X+R*sin(Fi+M_PI);                      // пересекающихся
                  Y3=Y -R*cos(Fi+M_PI);                     // осей
                  X4=X+R*sin(Fi+3*M_PI_2);                  // колеса
                  Y4=Y -R*cos(Fi+3*M_PI_2);                 //
                 line(X1,Y1,X3,Y3);     //нарисовать 1-й  отрезок – ось колеса
                 line(X2,Y2,X4,Y4);     //нарисовать 2-й  отрезок – ось колеса
                }
  void Wheel::Hide()
                {                          // Чтобы спрятать объект, нужно:
                  unsigned TempColor;     // создать рабочую переменную,
                 TempColor=getcolor();   // запомнить текущий цвет линий,
                 setcolor(getbkcolor());  // изменить цвет линий на цвет фона,
                 Show();                  // нарисовать объект цветом фона,
                 setcolor(TempColor);     // вернуть прежний цвет линий.
                }
  void Wheel::Slide(int DX)
                        {                    // Чтобы сдвинуть колесо, нужно
                          Hide();           // спрятать колесо,
                          PutX(X+DX);       // изменить координату центра
                          Show();           // и показать колесо.
                        }
  void Wheel::Turn(double DFi)
                              {                    //Чтобы повернуть колесо, нужно
                                Hide();            // спрятать колесо,
                                PutFi(Fi+DFi);     // изменить угол поворота,
                                Show();            // и показать колесо.
                              }
void Wheel::Roll(double DFi)
                {                     // Чтобы перекатить колесо, нужно
                 Turn(DFi);           // повернуть его,
                  int DX=R*DFi;       // синхронизировать сдвиг
                  Slide(DX);          // и сдвинуть.
                }
```

```
/* создание и использование графических объектов в основной программе */

int main()
{
   int gdriver = DETECT, gmode, errorcode;          // блок
   initgraph(&gdriver, &gmode, "");                 // инициализации
   errorcode = graphresult();                       // графического
                                                    // режима
   if (errorcode != grOk)
     {
       cout<<"Graphics error: "<<grapherrormsg(errorcode)<<endl;
       cout<<"Press any key to halt:"<<endl;
       getch();
       return(1);
     }

  setcolor(15);                              // цвет линий – ярко-белый

  Wheel W1(320,100,50,0);                    // создание объектов класса Wheel
  Wheel W2(100,250,50,0);                    //
  Wheel W3(100,400,50,0);                    //

  W1.Show();                        // изображение
  W2.Show();                        // созданных
  W3.Show();                        // объектов
  getch();          // остановить программу и ждать нажатия клавиши

  while(!kbhit())                             // повторять до нажатия клавиши
    {
      W1.Turn(0.1);                           // вращение 1-го колеса
      W2.Slide(50*0.1);                       // скольжение 2-го колеса
      W3.Roll(0.1);                           // перекатывание 3-го колеса
      if(W2.GetX()>=700) W2.PutX(-50);        // защита от выхода объектов
      if(W3.GetX()>=700) W3.PutX(-50);        //                за пределы экрана
      delay(100);           // задержка изображения на экране
     };
  getch();

  closegraph();
  return(0);
};
```

Пример 1.6. Создание графических объектов на языке Pascal with Object

```
{$N+}
program figures;
uses Crt, Graph;

type

 WheelPtr = ^Wheel;          {WheelPtr указатель на объектный тип Wheel }
 Wheel = object    {объявление объектного типа данных "Колесо" — Wheel}
                        {объектный тип данных — другое назваение класса}
  private
    X, Y: integer;          {координаты центра колеса}
    R: integer;             {радиус колеса}
    Fi: double;             {угол поворота колеса}

  public
    constructor Init(InitX, InitY, InitR: integer; InitFi: double);
    function Get_X: integer;            {методы-      }
    procedure PutX(NewX: integer); {   акцессоры }
    procedure PutFi(NewFi: double); {      класса   }
    procedure Show;                     {«показать» колесо}
    procedure Hide;                     {«спрятать» колесо}
    procedure Slide(DX: integer);             { скольжение колеса}
    procedure Turn(DFi: double);              { вращение колеса}
    procedure Roll(DFi: double);              { перекатывание колеса}
end;

                        {определение объявленных методов}

constructor Wheel.Init(InitX, InitY, InitR: integer; InitFi: double);
  begin
  X:=InitX;
         Y:=InitY;
         R:=InitR;
         Fi:=InitFi
  end;

function Wheel.Get_X;  begin  Get_X:=X  end;

procedure Wheel.PutX(NewX: integer); begin  X:=NewX  end;

procedure Wheel.PutFi(NewFi: double); begin  Fi:=NewFi end;
```

```
procedure Wheel. Show;
  var
        X1, Y1: integer;              { вспомогательные }
        X2, Y2: integer;              { координаты концов }
        X3, Y3: integer;              { двух пересекающихся }
        X4, Y4: integer;              { осей колеса }
  begin
        X1:=X + round(R*sin(Fi));
        Y1:=Y  - round(R*cos(Fi));
        X2:=X + round(R*sin(Fi+0.5*Pi));
        Y2:=Y  - round(R*cos(Fi+0.5*Pi));
        X3:=X + round(R*sin(Fi +Pi));
        Y3:=Y  - round(R*cos(Fi +Pi));
        X4:=X + round(R*sin(Fi+1.5*Pi));
        Y4:=Y  - round(R*cos(Fi+1.5*Pi));
        Circle(X, Y, R);
        Line(X1,Y1,X3,Y3);
        Line(X2,Y2,X4,Y4)
  end;

 procedure Wheel. Hide;
  var
        TempColor: word;
  begin
        TempColor:=GetColor;
        SetColor(GetBkColor);
        Show;
        SetColor(TempColor);
  end;

 procedure Wheel. Slide(DX: integer);
  begin
      Hide;
      PutX(X+DX);
      Show
  end;

 procedure Wheel. Turn(DFi: double);
  begin
      Hide;
      PutFi(Fi +DFi);
      Show
  end;
```

```
procedure Wheel. Roll(DFi: double);
  var
       DX:integer;
  begin
       Turn(DFi);
       DX:=round(DFi*R);
       Slide(DX)
  end;

       {создание и использование объектов в основной части программы}
var
gdriver, gmode, errcode: integer;
       W1,W2,W3: Wheel;      {создание объектов  — экземпляров класса Wheel}
begin
 clrscr;
 gdriver:=detect;
 gmode:=detect;
 initgraph(gdriver, gmode,'');
 errcode:=GraphResult;
 if not (errcode = grOk) then
                              begin
                                  writeln('Ошибка графики');
                                  readln;
                                  halt(1)
                              end;
 setcolor(15);

 W1.Init(300, 100, 50, 0);          {инициализация объектов класса Wheel}
 W2.Init(100, 250, 50, 0);
 W3.Init(100, 400, 50, 0);
 W1.Show;          {изображение объектов класса Wheel}
 W2.Show;
 W3.Show;
 readln;

 repeat
  W1.Turn(0.02);                              {вращение 1-го колеса}
  W2.Slide(1);                                { скольжение 2-го колеса}
  W3.Roll(0.02);                              { перекатывание 3-го колеса}
  if W2.Get_X>=700 then W2.PutX(-50); { сдвиг объекта за пределами экрана }
  if W3.Get_X>=700 then W3.PutX(-50); { сдвиг объекта за пределами экрана }
  delay(100); {задержка изображения на экране}
 until  KeyPressed;
 closegraph;
end.
```

Задание для лабораторной работы № 1

1. Разработав класс "монета" (плоский кружок), создать на экране три объекта, изображающие:
   а) движение «монеты», вращающейся вокруг горизонтальной оси,
   б) движение невращающейся «монеты», подброшенной вверх,
   в) движение подброшенной вращающейся «монеты».
2. Разработав класс "монета" (плоский кружок), создать на экране три объекта, изображающие движение:
   а) вращающейся вокруг вертикальной оси «монеты»,
   б) скользящей по горизонтали невращающейся «монеты»,
   в) скользящей по горизонтали вращающейся «монеты».
3. Разработав класс "окружность", создать три объекта, изображающие:
   а) расширяющуюся (сужающуюся) окружность с неподвижным центром,
   б) окружность постоянного радиуса, скользящую вдоль прямой,
   в) окружность, осуществляющую оба вышеописанных движения и имитирующую приближение (удаление) объекта в перспективе.
4. Разработав класс "эллипс", создать три объекта, изображающие:
   а) расширение и сжатие эллипса по горизонтали до размеров окружности при неподвижном центре эллипса,
   б) скольжение эллипса постоянных размеров вдоль горизонтальной линии,
   в) "переползание" эллипса по экрану путём одновременного расширения (сжатия) и скольжения.
5. Разработав класс "треугольник", создать три объекта, изображающие:
   а) вращение треугольника вокруг его неподвижного центра,
   б) скольжение вертикально ориентированного треугольника вдоль горизонтальной прямой,
   в) "перекатывание" треугольника.
6. Разработав класс "упругий шарик" («упругий эллипс»), создать на экране три "шарика", изображающие:
   а) синусоидально изменяющий свои размеры по вертикали «шарик» с неподвижным геометрическим центром,
   б) «шарик» с постоянными размерами, совершающий синусоидальные колебания по вертикали,
   в) упруго пружинящий «шарик» с неподвижной нижней точкой (то есть синхронно подпрыгивающий и деформирующийся шарик).
7. Разработав класс "равнобедренный треугольник", создать на экране три объекта, изображающие:
   а) вращение треугольника вокруг его геометрического центра,
   б) вращение вертикально ориентированного треугольника вокруг центра вращения, расположенного вне этого треугольника,
   в) вращение треугольника, ориентированного своей вершиной на расположенный вне этого треугольника центр вращения.

8. Разработав класс "ромб", создать три объекта, изображающие:
   а) растяжение и сжатие ромба вдоль горизонтальной оси путём удлинения его горизонтальной диагонали при неподвижном центре ромба,
   б) равномерное движение ромба по горизонтали,
   в) "переползание" ромба вдоль горизонтальной линии путём движения центра ромба синхронно с растяжением и сжатием.
9. Разработав класс "прямоугольник", создать три объекта, изображающие:
   а) растяжение и сжатие прямоугольника вдоль горизонтальной оси при неподвижном геометрическом центре,
   б) равномерное скольжение прямоугольника по горизонтали,
   в) "переползание" прямоугольника при синхронном соединении движения геометрического центра с изменением размера по горизонтали,
10. Разработав класс "египетский овал" (две полуокружности, соединённые отрезками), создать три объекта, изображающие:
   а) растяжение и сжатие овала вдоль горизонтальной оси с фиксированным геометрическим центром,
   б) равномерное горизонтальное движение овала с фиксированными размерами,
   в) "переползание" овала вдоль горизонтальной прямой соединением движения геометрического центра овала с синхронным растяжением и сжатием.
11. Разработав класс "треугольник", создать три объекта, изображающие:
   а) растяжение и сжатие треугольника вдоль горизонтальной оси при неподвижном геометрическом центре,
   б) равномерное движение треугольника по горизонтали,
   в) "переползание" треугольника по горизонтали путём соединения движения геометрического центра треугольника синхронно с растяжением и сжатием.
12. Разработав класс "ромб", создать три объекта, изображающие:
   а) вращение ромба вокруг своего геометрического центра,
   б) вращение вертикально ориентированного ромба вокруг центра, расположенного вне ромба,
   в) вращение ромба, ориентированного своей вершиной на расположенный вне его центр вращения.
13. Разработав класс "квадрат", создать три объекта, изображающие:
   а) вращение квадрата вокруг своего центра,
   б) вращение вертикально ориентированного квадрата вокруг центра, расположенного вне этого квадрата,
   в) вращение квадрата, ориентированного своей вершиной на расположенный вне его центр вращения.
14. Разработав класс "упругий прямоугольник", создать на экране три прямоугольника, изображающие:
   а) синусоидально растягивающийся и сжимающийся по вертикали прямоугольник с неподвижным геометрическим центром,

б) прямоугольник с постоянными размерами и центром масс, совершающим синусоидальные колебания,
в) упруго пружинящий прямоугольник с неподвижной нижней стороной (то есть синхронно движущийся и деформирующийся прямоугольник).

15. Разработав класс "эллипс", создать три объекта, изображающие движение:
а) пропорционально расширяющегося-сужающегося по обеим осям эллипса,
б) эллипса с постоянными размерами, скользящего вдоль наклонной прямой,
в) эллипса, осуществляющего оба вышеописанных движения и имитирующего приближение (удаление) в перспективе.

16. Разработав класс "параллелограмм" (не прямоугольный), создать три объекта, изображающие движение:
а) растягивающегося (сужающегося) вдоль горизонтальной оси параллелограмма при неподвижном геометрическом центре,
б) параллелограмма с постоянными размерами, скользящего вдоль горизонтальной прямой,
в) "переползание" параллелограмма при синхронном соединении движения геометрического центра параллелограмма с синхронным изменением его размеров по горизонтали.

17. Разработав класс "равнобедренная трапеция", создать три объекта, изображающие движение:
а) растягивающейся (сужающейся) вдоль горизонтальной оси трапеции при неподвижном геометрическом центре,
б) трапеции с постоянными размерами, скользящей вдоль горизонтали,
в) "переползание" трапеции при синхронном соединении движения геометрического центра трапеции с изменением её размеров по горизонтали.

18. Разработав класс "прямоугольная трапеция", создать три объекта, изображающие движение:
а) растягивающейся (сужающейся) вдоль горизонтальной оси трапеции при неподвижном геометрическом центре,
б) трапеции с постоянными размерами, скользящей вдоль прямой X,
в) "переползание" трапеции при синхронном соединении движения геометрического центра трапеции с изменением её размеров по горизонтали.

Методические указания

При подготовке к выполнению задания следует ознакомиться с основными понятиями объект и класс, изложенными в [1–3].

Методы, задающие движения, не должны иметь более трёх входных аргументов.

Все особенности изображения объектов, определяющие их размеры и форму, а не местоположение, нужно сосредоточить в методе Show.

Р А Б О Т А № 2. **НАСЛЕДОВАНИЕ КЛАССОВ**

Цель работы – изучение методов разработки классов с использованием механизма наследования. Освоение понятия **защищённые компоненты класса**, изучение особенностей взаимодействия методов-конструкторов при наследовании. Работа с наследуемыми полями данных. Вызов наследуемых методов. Определение преимуществ наследования.

Общие сведения

В объектно-ориентированной программе может быть несколько классов. При этом весьма часто встречается ситуация, когда многие поля и методы в различных классах совпадают. Для ускорения разработки, экономии объёма программного кода и для установления продуктивной связи между классами имеется специальный механизм **наследования** классов.

Класс, объявляемый как **наследник** другого класса, автоматически получает все открытые (**public**) и защищённые (**protected)** поля данных и методы этого класса-**предка** (то есть все компоненты, кроме закрытых – **private**). При этом защищённые компоненты класса доступны только методам своего класса и классу-наследнику, но закрыты вне класса, как private.

Открытые конструкторы класса-предка не наследуются как конструкторы, то есть не являются конструкторами класса-потомка. Но они доступны и могут быть вызваны в конструкторах (и в других методах) класса-наследника. Вызов конструктора класса-предка из конструктора класса-наследника для инициализации наследуемых полей данных не является обязательным с точки зрения объектно-ориентированного подхода. Но некоторые компиляторы не допускают прямой инициализации наследуемых полей данных в конструкторе класса-наследника и требуют обязательного вызова для этого конструктора класса-предка (см. пример 2.2 на Borland C++).

За счёт того, что унаследованные методы и поля данных не нужно заново создавать в классе-потомке, возможно существенное ускорение разработки программы, а также существенная экономия программного кода. При этом наследуемым полям данных и методам в классе-наследнике также может быть придан другой содержательный смысл.

Важно заметить, что не все наследуемые от класса-предка компоненты нужны классу-потомку. Но, формально, каждый унаследованный метод и каждое поле данных предка доступны объектам класса-потомка. При компиляции для работы со всеми методами и полями данных объекта выделяется оперативная память. Тем самым наследование, приводя к экономии исходного и исполнительного кода программы, одновременно может приводить к увеличению используемой оперативной памяти [1, 16].

Запретить использование некоторых унаследованных, но ненужных в классе-потомке методов возможно с помощью механизма переопределения методов. Однако сэкономить используемую оперативную память при этом не удаётся [16]. Переопределение методов (override) рассматривается в следующей лабораторной работе.

Пример 2.1. Проекты классов Point и Circle, независимых друг от друга

Имя класса: Point (точка)
Поля данных (закрытые): целочисленные X и Y – экранные координаты точки;
целочисленный Color – код цвета точки.
Методы (открытые**):**
а) Конструктор: параметры, целые начальные значения полей X, Y и Color;
б) Методы-акцессоры
GetX — возвращает целочисленное значения поля данных X;
GetY — возвращает целочисленное значения поля данных Y;
GetC — возвращает целочисленное значения поля данных Color;
PutX (новое X) – задаёт новое целочисленное значение для поля X;
PutY (новое Y) – задаёт новое целочисленное значение для поля Y;
PutColor (новый Color) – задаёт новое целочисленное значение для поля Color.
в) Прочие методы
Show — процедура рисования точки на экране;
Hide — процедура, стирающая изображение точки на экране, состоящая в том, что точка рисуется цветом фона;
MoveTo(целочисленные NX, NY) – процедура сдвига точки в позицию с координатами NX и NY, когда точку сперва стирают, затем задаются её новые значения координат, после чего точка рисуется в новой позиции.

Имя класса: Circle (окружность)
Поля данных (закрытые): целые X, Y - координаты центра окружности;
целый Color – код цвета окружности;
целый Radius – величина радиуса окружности.
Методы (открытые**):**
а)Конструктор: параметры, целые начальные знач-я полей X, Y, Color, Radius.
б) Методы-акцессоры
GetX — возвращает целочисленное значение поля данных — координаты X;
GetY — возвращает целочисленное значение поля данных — координаты Y;
GetC — возвращает целочисленное значение поля данных – код цвета Color;
GetRadius — возвращает целочисленное значение поля данных Radius;
PutX (новое X) – задаёт новое целочисленное значение для поля X;
PutY (новое Y) – задаёт новое целочисленное значение для поля Y;
PutColor (новый Color) – задаёт новое целочисленное значение для поля Color;
PutRadius (новый Radius) — задаёт новое целое значение для поля Radius.
в) Прочие методы
Draw – процедура рисования окружности на экране;
Clean — процедура стирания окружности (рисующая окружность цветом фона);
Move (целые NX, NY) – процедура сдвига окружности в позицию с координатами NX и NY, состоящая в том, что сначала окружность стирается, затем задаются новые координаты её центра, после чего окружность снова рисуется;
Expand (целое DR) – процедура расширения окружности на величину DR, состоящая в том, что сначала окружность стирается, затем задаётся новое значения радиуса, после чего окружность рисуется с новым радиусом.

Пример 2.2. Проекты класса Point и класса Circle, наследника класса Point

Имя класса: Point (точка)
Поля данных (защищённые): целые X и Y – экранные координаты точки;
целый Color – код цвета точки.
Методы (открытые**):**
а) Конструктор: параметры, целые начальные значения полей X, Y и Color;
б) Методы-акцессоры
GetX – возвращает целочисленное значения поля данных X;
GetY – возвращает целочисленное значения поля данных Y;
GetC – возвращает целочисленное значения поля данных Color;
PutX (целое NX) – задаёт новое целочисленное значение для поля X;
PutY (целое NY) – задаёт новое целочисленное значение для поля Y;
PutColor (целое NC) – задаёт новое целочисленное значение для поля Color.
в) Прочие методы
Show – процедура рисования точки на экране;
Hide – процедура, стирающая изображение точки на экране, состоящая в том, что точка рисуется цветом фона;
MoveTo(целочисленные NX, NY) – процедура сдвига точки в позицию с координатами NX и NY, когда точку сперва стирают, затем задаются её новые значения координат, после чего точка рисуется в новой позиции.

Имя класса: Circle (окружность) – **наследник** класса **Point**
Поля данных (защищённые): целый Radius – величина радиуса окружности;
Поля X, Y и Color наследуются у класса Point.
Методы (открытые**):**
а)Конструктор: параметры, целые начальные значения полей X, Y, Color, Radius.
б) Методы – акцессоры
GetRadius – возвращает целочисленное значение поля данных Radius;
PutRadius (integer) – задаёт новое целочисленное значение для поля Radius.
Акцессоры GetX, GetY, GetC, PutX, PutY, PutColor унаследованы у классаPoint.
в) Прочие методы
Draw – процедура рисования окружности на экране;
Clean – процедура стирания окружности (рисующая окружность цветом фона),
Move (целые NX, NY) – процедура сдвига окружности в позицию с координатами NX и NY, когда окружность сначала стирается, затем задают новые координаты центра, после чего окружность рисуется в новой позиции;
Expand (целое DR) – процедура расширения окружности на величину DR, состоящая в том, что сначала окружность стирается, затем задаётся новое значения радиуса, после чего окружность рисуется с новым радиусом.
Методы Show, Hide и MoveTo наследуются у класса Point, хотя и не нужны.

В программных реализациях проектов на Object Pascal класс Circle назван CCircle, чтобы в условиях регистровой нечувствительности языка не было совпадений со встроенной функцией circle графического интерфейса BGI.

Пример 2.3. Классы без наследования на языке C++

```
#include <graphics.h>
#include <iostream.h>
#include <conio.h>

class Point                    //объявление класса Point
{ private:
                int X;
                int Y;
                int Color;
  public:
               Point(int, int, int);
          int GetX();
          int GetY();
          int GetC();          //возвращает значение поля  Color
        void PutX(int);
        void PutY(int);
        void PutColor(int);
        void Show();           // рисует точку
        void Hide();           // прячет точку
        void MoveTo(int, int);  // перемещает точку
 };

class Circle                   //объявление класса Circle
 {private:
                int X;
                int Y;
                int Color;
                int Radius;
  public:
               Circle(int, int, int, int);
          int GetX();
          int GetY();
          int GetC();           //возвращает значение поля  Color
          int GetRadius();
        void PutX(int);
        void PutY(int);
        void PutColor(int);
        void PutRadius(int);
        void Draw();           // рисует окружность
        void Clean();          // прячет окружность
        void Expand(int);      // расширяет окружность
        void Move(int, int);   // перемещает точку
 };
```

```
                    /* определение методов объявленных классов */
Point:: Point(int X, int Y, int Color)
        {
         this -> X=X;
         this -> Y=Y;
         this -> Color=Color;
        }

void Point ::PutX(int X) { this -> X=X; }

void Point ::PutY(int Y) { this -> Y=Y; }

void Point ::PutColor(int Color) { this -> Color=Color; }

int Point ::GetX()              { return (X); }

int Point ::GetY()              { return (Y); }

int Point ::GetC()             { return (Color); }

void Point:: Show()  { putpixel(X, Y, Color); }

void Point:: Hide()   { putpixel(X,Y, getbkcolor()); }

void Point ::MoveTo(int X, int Y)
{
        Hide();
        PutX(X);
        PutY(Y);
        Show();
}

Circle:: Circle(int X, int Y, int Color, int Radius)
{
        this -> X=X;
        this -> Y=Y;
        this -> Color=Color;
        this -> Radius = Radius;
}

void Circle ::PutX(int X)      { this -> X=X; }

void Circle ::PutY(int Y)      { this -> Y=Y; }
```

```
void Circle ::PutColor(int Color)    { this -> Color=Color; }

void Circle:: PutRadius(int Radius) { this -> Radius=Radius; }

int Circle:: GetX()                  { return (X); }

int Circle:: GetY()                  { return (Y); }

int Circle:: GetC()                 { return (Color); }

int Circle:: GetRadius()         { return (Radius); }

void Circle:: Draw()
{
          int TempColor;
              TempColor=getcolor();
              setcolor(Color);
              circle(X, Y, Radius);
              setcolor(TempColor);
}

void Circle:: Clean()
{
          int TempColor;
              TempColor=GetC();
              PutColor(getbkcolor());
              Draw();
              PutColor(TempColor);
        }

void Circle:: Expand(int DR)
{
          Clean();
          if (GetRadius()>= - DR) PutRadius(Radius+DR);
          Draw();
}

void Circle:: Move(int X, int Y)
{
           Clean();
           PutX(X);
           PutY(Y);
           Draw();
        }
```

```
    /* использование созданных классов в основной программе */

int main()
{
 int gdriver = DETECT, gmode, errorcode;
 initgraph(&gdriver, &gmode, "");
 errorcode = graphresult();
 if (errorcode != grOk)
  {cout<<"Ошибка графики: "<<grapherrormsg(errorcode)<<endl;
   cout<<"Нажмите любую клавишу для прерывания:"<<endl;
   getch();  return(1); }
 setcolor(getmaxcolor());
 Point P1(100,100, getcolor());                    // создание объекта класса Point
 P1.Show();                getch();
 P1.MoveTo(150,150);   getch();
 P1.Hide();
 Point P2 = Point(200,200,5);                      // создание объекта класса Point
 P2.Show();                getch();
 P2.Hide();
 P2.PutX(360);
 P2.Show();                getch();
 P2.Hide();
 P2.PutY(250);
 P2.Show();                getch();
 P2.Hide();

 Circle C1(295,100,12,70), C2(150,250,13,50);    // объекты C1 и C2 класса Circle
 C1.Draw();                getch();
 C1.Clean();               getch();
 C1.Draw();                getch();
 C1.Expand(10);            getch();
 C1.Expand(-10);           getch();
 C2.Draw();                getch();
 C2.Clean();               getch();
 C2.PutX(360);
 C2.Draw();                getch();
 C2.Move(400,300);     getch();
 C2.Clean();               getch();
 closegraph();
 return(0);
};
```

Пример 2.4. Классы без наследования на языке Object Pascal

```
program figures;
uses Crt, Graph;
type
 PointPtr= ^Point;  {указатель на класс Point, нужный в старых версиях TPascal}
 Point= object                                          {класс Point}
  private
         X: integer;
         Y: integer;
    Color: integer;
  public
    constructor Init(InitX, InitY, InitC: integer);
    function GetX: integer;
    function GetY: integer;
    function GetC: integer;      {возвращает Color}
    procedure PutX(NewX: integer);
    procedure PutY(NewY: integer);
    procedure PutColor(NewColor: integer);
    procedure Show;              {рисует точку}
    procedure Hide;              {прячет точку}
    procedure MoveTo(NewX, NewY: integer);  {передвигает точку}
 end;

 CircPoint =^CCircle;   {указатель на класс CCircle в старых версиях TPascal}
 CCircle = object                                       {класс CCircle}
   private
            X:   integer;
            Y:   integer;
        Color:   integer;
       Radius: integer;
   public
     constructor Init(InitX, InitY, InitC, InitR: integer);
     function GetX: integer;
     function GetY: integer;
     function GetC: integer;       {возвращает Color}
     function GetRadius: integer;
     procedure PutX(NewX: integer);
     procedure PutY(NewY: integer);
     procedure PutColor(NewColor: integer);
     procedure PutRadius(NewRadius: integer);
     procedure Draw;               {рисует окружность}
     procedure Clean;              {прячет окружность}
     procedure Expand(DR: integer);
     procedure Move(NewX, NewY: integer); {передвигает окружность }
 end;
```

```
{определение объявленных методов}
constructor Point. Init(InitX, InitY, InitC: integer);
  begin
        X:=InitX;
        Y:=InitY;
        Color:=InitC
  end;

function Point.GetX;    begin    GetX:=X    end;

function Point.GetY;    begin    GetY:=Y   end;

function Point.GetC;    begin    GetC:=Color   end;

procedure Point.PutX(NewX: integer); begin X:=NewX end;

procedure Point.PutY(NewY: integer); begin Y:=NewY end;

procedure Point.PutColor(NewColor: integer); begin Color:=NewColor end;

procedure Point.Show;  begin  PutPixel(X, Y, Color)  end;

procedure Point.Hide;  begin  PutPixel(X, Y, GetBkColor)  end;

procedure Point.MoveTo(NewX, NewY: integer);
  begin
    Hide;
    PutX(NewX);
    PutY(NewY);
    Show
  end;

constructor CCircle.Init(InitX, InitY, InitC, InitR: integer);
  begin
    X:=InitX;
    Y:=InitY;
    Color:=InitC;
    Radius:=InitR
  end;

function CCircle.GetX;    begin    GetX:=X    end;

function CCircle.GetY;    begin    GetY:=Y   end;
```

```
function CCircle.GetC;     begin     GetC:=Color  end;

function CCircle.GetRadius; begin   GetRadius:=Radius   end;

procedure CCircle.PutX(NewX: integer); begin X:=NewX end;

procedure CCircle.PutY(NewY: integer); begin Y:=NewY end;

procedure CCircle.PutColor(NewColor: integer); begin Color:=NewColor end;

procedure CCircle.PutRadius(NewRadius: integer); begin Radius:=NewRadius end;

procedure CCircle.Draw;
  var TempColor: integer;
  begin
    TempColor:=GetColor;
    SetColor(Color);
    Circle(X, Y, Radius);
    SetColor(TempColor)
  end;

procedure CCircle.Clean;
  var TempColor: integer;
  begin
    TempColor:=GetC;
    PutColor(GetBkColor);
    Draw;
    PutColor(TempColor)
  end;

procedure CCircle.Move(NewX, NewY: integer);
   begin
    Clean;
    PutX(NewX);
    PutY(NewY);
    Draw
   end;

procedure CCircle.Expand(DR: integer);
  begin
    Clean;
    if GetRadius >= -DR then PutRadius(Radius + DR);
    Draw
end;
```

```
        {создание и использование объектов в основной части программы}
var X,Y, GDriver, GMode, ErrCode: integer;
   P1,P2:Point; C1,C2:CCircle; {создание экземпляров классов Point и CCircle}
begin
 clrscr;
 GDriver:=DETECT;
 GMode:=DETECT;
 InitGraph(GDriver, GMode,'');
 ErrCode:=GraphResult;
 if not (ErrCode = grOk) then
   begin
     writeln('Ошибка графики', GraphErrorMsg(ErrCode));
     writeln('‘Нажмите любую клавишу для прерывания.‘');
     readln;
     halt(1)
   end;
 P1.Init(100,100,getcolor);
 P1.Show;                  readln;
 P1.MoveTo(150,150);  readln;
 P1.Hide;                  readln;
 P2.Init(200,200,5);
 P2.Show;                  readln;
 P2.Hide;
 P2.PutX(360);
 P2.Show;                  readln;
 P2.Hide;
 C1.Init(295,100,12,70);          {инициализация объекта C1 класса CCircle;}
 C1.Draw;                  readln;
 C1.Clean;                 readln;
 C1.Draw;                  readln;
 C1.Expand(10);            readln;
 C1.Expand(-10);           readln;
 with C2 do                    {использование оператора  with  языка ПАСКАЛЬ}
   begin
     Init(150,250,13,50);          {инициализация объекта C2 класса CCircle}
     Draw;                 readln;
     Clean;
     PutX(360);
     Draw;                 readln;
     Move(400,300);        readln;
     Clean;                readln
  end;
 closegraph
end.
```

Пример 2.5. Наследование классов на языке C++

```
#include <graphics.h>
#include <iostream.h>
#include <conio.h>

class Point                    // объявление класса Point
  {
   protected:
              int X;
              int Y;
              int Color;
   public:
             Point(int, int, int);
         int GetX();
         int GetY();
         int GetC();           // возвращает значение поля Color
       void PutX(int);
       void PutY(int);
       void PutColor(int);
       void Show();
       void Hide();
       void MoveTo(int, int);
  };

class Circle: public Point          // класс Circle – наследник класса Point
  {
   protected:
        int Radius;
   public:
              Circle(int, int, int, int);
         int GetRadius();
       void PutRadius(int);
       void Draw();
       void Clean();
       void Expand(int);
       void Move(int, int);
   };

                /*  определение методов объявленных классов */

Point:: Point(int X, int Y, int Color) { this -> X=X;
                                         this -> Y=Y;
                                         this -> Color=Color; }
```

```
int Point ::GetX()      { return (X); }

int Point ::GetY()      { return (Y); }

int Point ::GetC()                  { return (Color); }

void Point ::PutX(int X)            { this -> X=X; }

void Point ::PutY(int Y)            { this -> Y=Y; }

void Point ::PutColor(int Color) { this -> Color=Color; }

void Point:: Show()  { putpixel(X, Y, Color); }

void Point:: Hide()   { putpixel(X,Y, getbkcolor()); }

void Point ::MoveTo(int X, int Y)  { Hide();
                                     PutX(X);
                                     PutY(Y);
                                     Show(); }

Circle::Circle(int X, int Y, int Color, int Radius) : Point(X, Y, Color)
                                                  // вызов конструктора класса-предка
                    { this -> Radius = Radius; }  // инициализация нового поля данных

int Circle ::GetRadius()            { return (Radius); }

void Circle ::PutRadius(int Radius) { this -> Radius=Radius; }

void Circle:: Draw() { int TempColor;
                       TempColor=getcolor();
                       setcolor(Color);
                       circle(X, Y, Radius);
                       setcolor(TempColor); }

void Circle:: Clean() { int TempColor;
                        TempColor=GetC();
                        PutColor(getbkcolor());
                        Draw();
                        PutColor(TempColor); }

void Circle:: Expand (int DR)
        {Clean();
         if (GetRadius()>= - DR) PutRadius(Radius+DR);
         Draw();}
```

```
void Circle:: Move(int X, int Y) { Clean();
                                   PutX(X);
                                   PutY(Y);
                                   Draw(); }

        /* использование созданных классов в основной программе */
int main()
{int gdriver = DETECT, gmode, errorcode;
 initgraph(&gdriver, &gmode, "");
 errorcode = graphresult();
 if (errorcode != grOk)
  {cout<<"Ошибка графики: "<<grapherrormsg(errorcode)<<endl;
   cout<<"Нажмите любую клавишу для прерывания:"<<endl;
   getch();  return(1); }
 setcolor(getmaxcolor());
 Point P1(100,100, getcolor());                  // создание объекта P1 класса Point
 P1.Show();          getch();
 P1.MoveTo(150,150);  getch();
 P1.Hide();
 Point P2 = Point(200,200,5);                    // создание объекта P2  класса Point
 P2.Show();          getch();
 P2.Hide();
 P2.PutX(360);
 P2.Show();          getch();
 P2.Hide();
 P2.PutY(250);
 P2.Show();          getch();
 P2.Hide();
 Circle C1(295,100,12,70),C2(150,250,13,50);    // объекты C1 и C2 класса Circle
 C1.Draw();          getch();
 C1.Clean();         getch();
 C1.Draw();          getch();
 C1.Expand(10);      getch();
 C1.Expand(-10);     getch();
 C1.Clean();         getch();
 C2.Draw();          getch();
 C2.Clean();
 C2.PutX(360);
 C2.Draw();            getch();
 C2.Move(400,300); getch();
 C2.Clean();           getch();
 closegraph();
 return(0);
};
```

Пример 2.6. Наследование классов на языке Object Pascal

```
program figures;
uses Crt, Graph;
type
 PointPtr= ^Point;  {указатель на класс Point, нужный в старых версиях TPascal}
 Point= object                          {объявление класса Point}
  public                                {в более поздних версиях – protected}
         X: integer;
         Y: integer;
    Color: integer;
  public
    constructor Init(InitX, InitY, InitC: integer);
    function GetX: integer;
    function GetY: integer;
    function GetC: integer;
    procedure PutX(NewX: integer);
    procedure PutY(NewY: integer);
    procedure PutColor(NewColor: integer);
    procedure Show;             {рисует точку}
    procedure Hide;             {прячет точку}
    procedure MoveTo(NewX, NewY: integer); {перемещает точку}
 end;

 CircPoint =^CCircle;    {указатель на класс CCircle для старых версий TPascal}
 CCircle = object (Point)  {объявление класса CCircle – наследника класса Point}
   public     {в более поздних версиях – protected}
       Radius: integer;
   public
     constructor Init(InitX, InitY, InitC, InitR: integer);
     function GetRadius: integer;
     procedure PutRadius(NewRadius: integer);
     procedure Draw;             {рисует окружность}
     procedure Clean;            {прячет окружность}
     procedure Expand(DR: integer);
     procedure Move(NewX, NewY: integer); {перемещает окружность}
 end;

                              {определение объявленных методов}
constructor Point. Init(InitX, InitY, InitC: integer);
  begin
         X:=InitX;
         Y:=InitY;
         Color:=InitC
  end;
```

```
function Point.GetX;    begin    GetX:=X   end;

function Point.GetY;    begin    GetY:=Y   end;

function Point.GetC;    begin    GetC:=Color   end;

procedure Point.PutX(NewX: integer); begin X:=NewX end;

procedure Point.PutY(NewY: integer); begin Y:=NewY end;

procedure Point.PutColor(NewColor: integer); begin Color:=NewColor end;

procedure Point.Show;  begin  PutPixel(X, Y, Color)  end;

procedure Point.Hide;  begin  PutPixel(X, Y, GetBkColor)  end;

procedure Point.MoveTo(NewX, NewY: integer);
  begin
    Hide;
    PutX(NewX);
    PutY(NewY);
    Show
  end;

constructor CCircle.Init(InitX, InitY, InitC, InitR: integer);
  begin
    Point.Init(InitX, InitY, InitC);          {вызов конструктора класса-предка}
    Radius:=InitR
  end;

function CCircle.GetRadius;   begin  GetRadius:=Radius  end;

procedure CCircle.PutRadius(NewRadius: integer);
begin  Radius:=NewRadius end;

procedure CCircle.Draw;
 var TempColor: integer;
 begin
    TempColor:=GetColor;
    SetColor(Color);
    Circle(X, Y, Radius);
    SetColor(TempColor)
 end;
```

```
procedure CCircle.Clean;
  var TempColor: integer;
  begin
     TempColor:=GetC;
     PutColor(GetBkColor);
     Draw;
     PutColor(TempColor)
  end;

procedure CCircle.Move(NewX, NewY: integer);
   begin
     Clean;
     PutX(NewX);
     PutY(NewY);
     Draw
   end;

procedure CCircle.Expand(DR: integer);
  begin
     Clean;
     if GetRadius >= -DR then PutRadius(Radius + DR);
     Draw
  end;

            {создание и использование объектов в основной части программы}
var  X, Y, GDriver, GMode, ErrCode: integer;
       P1, P2: Point;                          {объявление экземпляров классов Point}
       C1, C2: CCircle;                        {объявление экземпляров классов CCircle}
begin
  clrscr;
  GDriver:=DETECT;
  GMode:=DETECT;
  InitGraph(GDriver, GMode,'');
  ErrCode:=GraphResult;
  if not (ErrCode = grOk) then
    begin
      writeln('Ошибка графики', GraphErrorMsg(ErrCode));
      writeln('‘Нажмите любую клавишу для прерывания.‘');
      readln;
      halt(1)
    end;
P1.Init(100,100,getcolor); {инициализация объекта P1 – экземпляра класса Point}
P1.Show;
         readln;
```

```
P1.MoveTo(150,150);
        readln;
P1.Hide;
        readln;
P2.Init(200,200,5);        {инициализация объекта P2 – экземпляра класса Point}
P2.Show;
        readln;
P2.Hide;
P2.PutX(360);
P2.Show;
        readln;
P2.Hide;
C1.Init(295,100,12,70);{инициализация объекта C1 –
                                                    экземпляра класса CCircle}
C1.Draw;
        readln;
C1.Clean;
        readln;
C1.Draw;
        readln;
C1.Expand(10);
        readln;
C1.Expand(-10);
        readln;
with C2 do              {использование оператора  with  языка ПАСКАЛЬ}
  begin
    Init(150,250,13,50); {инициализация объекта C2 – экземпляра класса CCircle}
     Draw;
         readln;
     Clean;
     PutX(360);
     Draw;
          readln;
     Move(400,300);
          readln;
     Clean;
          readln
  end;
 closegraph
end.
```

Задание для лабораторной работы №2

1. Разработать предлагаемые ниже независимые друг от друга классы, предусмотрев акцессоры для всех полей данных. В каждом классе предусмотреть методы, изменяющие размеры создаваемых объектов, а также методы, осуществляющие поступательные и вращательные движения этих объектов. В разных классах это должны быть разные методы (Show, Hide и MoveTo у предка, Draw, Clean и Move у потомка в примерах 2.1–2.6).

2. С помощью наследования разработать заданную иерархию классов.

3. Создать и изобразить объекты разработанных классов. Показать возможные изменения размеров и положения созданных объектов на экране.

4. Обратиться к объектам классов-потомков с помощью унаследованных от классов-предков методов рисования и движения (Show и Draw, MoveTo и Move в примерах 2.1–2.6). Убедиться в том, что такие вызовы методов не приводят к правильному результату.

5. Разработать методы, возвращающие значение основных геометрических параметров (*длин, площадей и объёмов*) созданных объектов.

6. Сравнить объём программного кода программ без наследования классов и программ с наследованием классов. Сделать это сравнение для исходных файлов, для объектных файлов после компиляции и для исполнительных файлов после присоединения библиотек, поместив результаты в таблицу, куда нужно добавить результаты вычисления экономии объёма кода при наследовании (в процентах). Сравнение выполнить и для разных языков программирования (сравнить для C++ и для Object Pascal).

7. Оформить результаты в соответствии с общим порядком выполнения лабораторных работ.

Варианты иерархии классов

1. "точка" -> "отрезок" -> "квадрат" -> "куб"

2. "точка" -> "отрезок" -> "квадрат" -> "четырёхугольная пирамида"

3. "точка" -> "отрезок" -> "квадрат" -> "антипризма с квадратными основаниями"
(Объём антипризмы можно вычислить приближенно как объём такой же призмы)

4. "точка" -> "отрезок" -> "параллелограмм" -> "параллелепипед"

5. "точка" -> "отрезок" -> "ромб" -> "ромбическая призма"

6. "точка" -> "отрезок" -> "треугольник" -> "тетраэдр"

7. "точка" -> "отрезок" -> "треугольник" -> "треугольная призма"

8. "точка" -> "отрезок" -> "окружность" -> "цилиндр"

9. "точка" -> "отрезок" -> "окружность" -> "сфера"

10. "точка" -> "отрезок" -> "прямоугольник" -> "пирамида с прямоугольным основанием"

11. "точка" -> "отрезок" -> "прямоугольник" -> "прямоугольный параллелепипед"

12. "точка" -> "отрезок" -> "эллипс" -> "эллипсоид вращения"

13. "точка" -> "отрезок" -> "эллипс" -> "эллипсоид трёхосный"

14. "точка" -> "отрезок" -> "окружность" -> "конус"

15. "точка" -> "отрезок" -> "ромб" -> "ромбическая пирамида"

16. "точка" -> "отрезок" -> "эллипс" -> "эллиптический цилиндр"

17. "точка" -> "отрезок" -> "окружность" -> "конус"

18. "точка" -> "отрезок" -> "параллелограмм" -> "пирамида с параллелограммом в основании"

Методические указания

При подготовке к выполнению задания следует ознакомиться с механизмом наследования классов и с соответствующими понятиями защищённых компонент класса и т.п., изложенными в [1, 4—11].

При проектировании методов, отвечающих за различные повороты и деформации, следует создавать только методы, наиболее просто реализуемые с точки зрения геометрии. Ввиду большого объёма задания на лабораторную работу №2, для демонстрации функционирования механизма наследования не обязательно разрабатывать методы поворота для трёхмерных фигур с использованием эйлеровых углов. Общий приём для всех таких методов должен быть следующим: «Чтобы повернуть геометрический объект, его нужно спрятать, затем изменить угол поворота, после чего снова нарисовать объект».

Наряду с достоинства наследования, рассмотренными в [1, 4—11], изучите недостатки наследования в [1, 16].

РАБОТА №3. **ПОЛИМОРФИЗМЫ МЕТОДОВ**

Цель работы – изучение двух разновидностей полиморфизма методов в объектно-ориентированном программировании – **переопределения** методов при наследовании классов и **перегрузки** методов внутри класса. Разбор программных ситуаций, приводящих к переопределению методов. Освоение технологии перегрузки методов и практическое использование перегруженных методов на примере полиморфизма конструкторов.

Общие сведения

У наследуемых от класса-предка методов (у наследуемых функций-членов класса) в классе-потомке можно изменить код, то есть наследуемые методы могут быть переопределены (**переопределение** – *override*). Этот приём объектно-ориентированного программирования распространён на практике и является одной из разновидностей **полиморфизма** (т.е. многоформенности), а именно **полиморфизма методов**. Чтобы переопределить унаследованный метод, в языках Object Pascal и C++ требуется объявить этот метод в классе-наследнике, после чего заново определить его. В этом случае к объектам класса-потомка будут посылать сообщения уже с помощью перегруженных методов.

В примерах 3.1-3.3 можно наблюдать переопределение в классе Circle методов Show, Hide и MoveTo, унаследованных от класса Point. В сравнении с проектом из примера 2.2 и в сравнении с программами в примерах 2.5-2.6 из лабораторной работы №2, в проекте 3.1 и в программах 3.2-3.3 не нужны имена методов Draw, Clean и Move, так как вместо них в программах используются имена вышеназванных переопределённых методов Show, Hide и MoveTo, а тела – ранее используемых методов Draw, Clean и Move. Это позволяет избавиться от ошибочных вызовов унаследованных и непереопределённых методов из объектов классов-потомков, делает более удобной работу с иерархией классов, а также позволяет в дальнейшем использовать дополнительные возможности для наследования методов, но уже с использованием виртуальных функций (см. работу №4). К сожалению, переопределение методов, так же как и наследование методов, не приводят к уменьшению используемой оперативной памяти компьютера, как это часто кажется при первом, поверхностном взгляде на этот вопрос. Напротив, результаты исследований показывают, что наследование классов приводит к увеличению используемой оперативной памяти [1, 16].

В рамках объектно-ориентированного подхода часто разрешается создавать несколько методов (функций — членов класса) с одинаковыми именами даже в рамках одного класса. Такие методы обязательно отличаются друг от друга количеством и/или типом входных параметров. Это другая разновидность полиморфизма методов, так называемая **перегрузка** — *overload*. Для перегрузки методов не нужна иерархия наследования, так как все такие методы находятся в одном классе. В примерах 3.1—3.3 наряду с переопределением методов рассмотрена перегрузка конструкторов (так называемый *полиморфизм конструкторов*). Эта перегрузка позволяет инициализировать в создаваемых объектах разное количество полей.

Пример 3.1. Проекты классов Point и Circle (наследник класса Point), с переопределёнными методами и с перегруженными методами

Имя класса: Point (точка)
Поля данных (защищённые): целые X и Y – экранные координаты точки, целый Color – код цвета точки;
Методы (открытые**):**
а) Конструктор1: параметры, целые начальные значения полей X, Y и Color;
Конструктор2: (перегружен) начальные значения полей данных задаются по умолчанию: X и Y в центре экрана, а цвет Color – самый яркий.
б) Методы-акцессоры
GetX – возвращает целочисленное значения поля данных X;
GetY – возвращает целочисленное значения поля данных Y;
GetC – возвращает целочисленное значения поля данных Color;
PutX (целое NX) – задаёт новое целочисленное значение для поля X;
PutY (целое NY) – задаёт новое целочисленное значение для поля Y;
PutColor (целое NC) – задаёт новое целочисленное значение NC для поля Color.
в) Прочие методы
Show – процедура рисования точки на экране;
Hide – процедура, стирающая изображение точки на экране, состоящая в том, что точка рисуется цветом фона;
MoveTo(целочисленные NX, NY) – процедура сдвига точки в позицию с координатами NX и NY, состоящая в том, что точка сначала стирается, затем задаются новые координаты, после чего эта точка рисуется в новой позиции.

Имя класса: Circle (окружность) – **наследник** класса **Point**
Поля данных (защищённые): целый Radius – величина радиуса окружности.
Поля X, Y и Color наследуются у класса Point.
Методы (открытые**):**
а) Конструктор1: параметры, целые начальные значения X, Y, Color, Radius;
Конструктор2: параметр, целое начальное значения поля Radius. начальные значения полей данных X, Y, Color задаются по умолчанию: X и Y в центре экрана, а цвет Color – самый яркий.
б) Методы-акцессоры *(GetX, GetY, GetC, PutX, PutY и PutColor наследуются)*
GetRadius ─ возвращает целочисленное значение поля данных Radius;
PutRadius (integer) ─ задаёт новое целочисленное значение для поля Radius.
в) Прочие методы
Методы Show, Hide и MoveTo наследуются у класса Point и переопределяются.
Show – процедура рисования окружности на экране;
Hide – процедура, стирающая окружность (рисующая её цветом фона);
MoveTo(целые NX, NY) – процедура сдвига окружности, состоящая в том, что окружность стирается, затем задаются новые координаты центра, после чего окружность рисуется в новой позиции;
Expand (целое DR) – процедура расширения окружности на величину DR, состоящая в том, что сначала окружность стирается, затем задаётся новое значения радиуса, после чего окружность рисуется с новым радиусом.

Пример 3.2. Полиморфизм методов на языке C++

```
#include <graphics.h>
#include <iostream.h>
#include <conio.h>
class Point                    //объявление класса Point
{ protected:  int X;
              int Y;
              int Color;
  public:
     Point(int, int, int);   // перегрузка (полиморфизм)
          Point();                 // конструкторов
        int GetX();
        int GetY();
        int GetC();            //возвращает значение поля  Color
      void PutX(int);
      void PutY(int);
      void PutColor(int);
      void Show();
      void Hide();
      void MoveTo(int, int);
 };
class Circle: public Point          //класс Circle - наследник Point
 {protected:
        int Radius;
  public:
           Circle(int, int, int, int);
           Circle(int);
         int GetRadius();
      void PutRadius(int);
      void Show();              // переопределение унаследованного метода Show
      void Hide();              // переопределение унаследованного метода Hide
      void Expand(int);
      void MoveTo(int, int);  //переопределение унаследованного метода MoveTo
 };
            /*  определение методов объявленной иерархии классов */
Point:: Point(int X, int Y, int Color) { this -> X=X;
                                         this -> Y=Y;
                                         this -> Color=Color;}

Point:: Point() {X=0.5*getmaxx();
                 Y=0.5*getmaxy();
                 Color=getmaxcolor();}
int Point ::GetX()          { return (X); }
```

```
int Point ::GetY()            { return (Y); }

int Point ::GetC()          { return (Color); }

void Point ::PutX(int X) { this -> X=X; }

void Point ::PutY(int Y) { this -> Y=Y; }

void Point ::PutColor(int Color) { this -> Color=Color; }

void Point:: Show()  { putpixel(X, Y, Color); }

void Point:: Hide()   { putpixel(X,Y, getbkcolor()); }

void Point ::MoveTo(int X, int Y) { Hide();
                                    PutX(X);
                                    PutY(Y);
                                    Show(); }

Circle:: Circle(int X, int Y, int Color, int Radius) :
                Point(X, Y, Color)     // вызов конструктора класса-предка
                                { this -> Radius = Radius; }

Circle:: Circle(int Radius):Point(320,240,15) { this -> Radius = Radius; }

int Circle ::GetRadius()            { return (Radius); }

void Circle ::PutRadius(int Radius) { this -> Radius=Radius; }

void Circle:: Show() { int TempColor;
                       TempColor=getcolor();
                       setcolor(Color);
                       circle(X, Y, Radius);
                       setcolor(TempColor); }

void Circle:: Hide() { int TempColor;
                       TempColor=GetC();
                       PutColor(getbkcolor());
                       Show();
                       PutColor(TempColor);}

void Circle:: Expand (int DR) {Hide();
                               if (GetRadius()>= -DR) PutRadius(Radius+DR);
                               Show();}

void Circle:: MoveTo(int X, int Y) { Hide();
                                     PutX(X);
                                     PutY(Y);
                                     Show(); }
```

```
/* использование созданных классов в основной программе */

int main()
{
 int gdriver = DETECT, gmode, errorcode;
 initgraph(&gdriver, &gmode, "");
 errorcode = graphresult();
 if (errorcode != grOk)
   {
     cout<<"Ошибка графики: "<<grapherrormsg(errorcode)<<endl;
     cout<<"Нажмите любую клавишу для прерывания:"<<endl;
     getch();
     return(1);
   }
 setcolor(getmaxcolor());
 Point P1(100,100, getcolor());    // создание объекта класса Point
 P1.Show();                  getch();
 P1.MoveTo(150,150);  getch();
 P1.Hide();
 Point P2 = Point();              // создание объекта класса Point
 P2.Show();                 getch();
 P2.Hide();
 P2.PutX(360);
 P2.Show();                 getch();
 P2.Hide();
 P2.PutY(250);
 P2.Show();                 getch();
 P2.Hide();
 Circle C1(295,100,12,70), C2(50); // объекты C1 и C2 класса Circle
 C1.Show();                getch();
 C1.Hide();                getch();
 C1.Show();                getch();
 C1.Expand(10);            getch();
 C1.Expand(-10);           getch();
 C2.Show();                getch();
 C2.Hide();                getch();
 C2.PutX(360);
 C2.Show();                getch();
 C2.MoveTo(400,300); getch();
 C2.Hide();                getch();
 closegraph();
 return(0);
};
```

Пример 3.3. Полиморфизм методов на языке Object Pascal

```
program figures;
uses Crt, Graph;

type

 PointPtr= ^Point;
 Point= object                                    {класс Point}
  public      {в более поздних версиях - protected}
          X: integer;
          Y: integer;
    Color: integer;
  public
    constructor Init(InitX, InitY, InitC: integer);  {полиморфные}
    constructor Init1;                               {конструкторы}
    function GetX: integer;
    function GetY: integer;
    function GetC: integer;
    procedure PutX(NewX: integer);
    procedure PutY(NewY: integer);
    procedure PutColor(NewColor: integer);
    procedure Show;
    procedure Hide;
    procedure MoveTo(NewX, NewY: integer);
 end;


 CircPoint =^CCircle;
 CCircle = object (Point)     {класс CCircle – наследник класса Point}
   public      {в более поздних версиях - protected}
       Radius: integer;
   public
     constructor Init(InitX, InitY, InitC, InitR: integer); {полиморфные}
     constructor Init1(InitR: integer);                     {конструкторы}
     function GetRadius: integer;
     procedure PutRadius(NewRadius: integer);
     procedure Show;
     procedure Hide;
     procedure Expand(DR: integer);
     procedure MoveTo(NewX, NewY: integer);
 end;
```

```
                    {определение объявленных методов}
constructor Point. Init(InitX, InitY, InitC: integer);
  begin
        X:=InitX;
        Y:=InitY;
        Color:=InitC
  end;

constructor Point. Init1;
  begin
        X:=getmaxx div 2;
        Y:=getmaxy div 2;
        Color:=getmaxcolor
  end;

function Point.GetX;    begin    GetX:=X    end;

function Point.GetY;    begin    GetY:=Y   end;

function Point.GetC;    begin    GetC:=Color   end;

procedure Point.PutX(NewX: integer); begin X:=NewX end;

procedure Point.PutY(NewY: integer); begin Y:=NewY end;

procedure Point.PutColor(NewColor: integer); begin Color:=NewColor end;

procedure Point.Show;  begin  PutPixel(X, Y, Color)  end;

procedure Point.Hide;  begin  PutPixel(X, Y, GetBkColor)  end;

procedure Point.MoveTo(NewX, NewY: integer);
  begin
    Hide;
    PutX(NewX);
    PutY(NewY);
    Show
  end;

constructor CCircle.Init(InitX, InitY, InitC, InitR: integer);
  begin
    Point.Init(InitX,InitY,InitC);
    Radius:=InitR
  end;
```

```
constructor CCircle.Init1(InitR: integer);
  begin
    Point.Init1;
    Radius:=InitR
  end;

function CCircle.GetRadius; begin  GetRadius:=Radius  end;

procedure CCircle.PutRadius(NewRadius: integer); begin Radius:=NewRadius end;

procedure CCircle.Show;
 var TempColor: integer;
 begin
    TempColor:=GetColor;
    SetColor(Color);
    Circle(X, Y, Radius);
    SetColor(TempColor)
 end;

procedure CCircle.Hide;
 var TempColor: integer;
 begin
    TempColor:=GetC;
    PutColor(GetBkColor);
    Show;
    PutColor(TempColor)
 end;

procedure CCircle.MoveTo(NewX, NewY: integer);
  begin
    Hide;
    PutX(NewX);
    PutY(NewY);
    Show
  end;

procedure CCircle.Expand(DR: integer);
 begin
    Hide;
    if GetRadius >= -DR then PutRadius(Radius + DR);
    Show
 end;
```

```
        {создание и использование объектов в основной части программы}
var X,Y, GDriver, GMode, ErrCode: integer;
     P1,P2: Point; C1,C2: CCircle; {создание экземпляров классов Point и CCircle}
begin
 clrscr;
 GDriver:=DETECT;
 GMode:=DETECT;
 InitGraph(GDriver, GMode,'');
 ErrCode:=GraphResult;
 if not (ErrCode = grOk) then
   begin
     writeln('Ошибка графики', GraphErrorMsg(ErrCode));
     writeln('‘Нажмите любую клавишу для прерывания.‘');
     readln;
     halt(1)
   end;
 P1.Init(100,100,getcolor);
 P1.Show;                  readln;
 P1.MoveTo(150,150);  readln;
 P1.Hide;                  readln;
 P2.Init1;
 P2.Show;                  readln;
 P2.Hide;
 P2.PutX(360);
 P2.Show;                  readln;
 P2.Hide;
 C1.Init(295,100,12,70);          {инициализация объекта C1 класса CCircle;}
 C1.Show;                  readln;
 C1.Hide;                  readln;
 C1.Show;                  readln;
 C1.Expand(10);            readln;
 C1.Expand(-10);           readln;
 with C2 do                  {использование оператора  with  языка ПАСКАЛЬ}
   begin
     Init1(50);              {инициализация объекта C2 класса CCircle}
     Show;                   readln;
     Hide;
     PutX(360);
     Show;                   readln;
     MoveTo(400,300);   readln;
     Hide;                   readln
  end;
 closegraph
end.
```

Задание для лабораторной работы №3

1. На основе программ, созданных в соответствии с вариантом задания при выполнении лабораторной работы № 2, разработать проекты классов и написать программы, где в классах-наследниках будут переопределены все методы, отвечающие за рисование и за стирание, а также за движение графических объектов. Тем самым следует устранить различия в именах переопределяемых методов, такие, как имена Show, Hide и MoveTo у предка, и имена Draw, Clean и Move у потомка в примерах 2.1—2.6.

2. Создать и изобразить объекты разработанных классов. Показать, что переопределение методов позволяет избавиться от неправильных вызовов наследуемых от предков методов из объектов классов-потомков, как это должно было быть показано при выполнении п. 4 задания к работе №2.

3. Добавить в разработанные программы все возможные новые конструкторы, где часть полей данных или все поля данных будут инициализированы по умолчанию. Так как в языке Turbo Pascal нельзя осуществлять непосредственную перегрузку методов (*в отличие от* Object Pascal *в* Delphi, *где можно объявлять перегрузку методов с помощью модификатора* overload), следует в одном классе создать несколько конструкторов с похожими именами, например Init1, Init2 и т.п. В этом случае будет достигнут полиморфизм конструкторов в одном классе и на языке Turbo Pascal, так же как в C++. Продемонстрировать использование полиморфных конструкторов.

4. Оформить результаты в соответствии с объявленным ранее общим порядком выполнения лабораторных работ.

Методические указания

При подготовке к выполнению задания следует подробно ознакомиться с механизмом переопределения (override) методов в классах-потомках из созданной иерархии, изложенным, например, в [1—11]. Также следует изучить перегрузку (overload) методов в классе, рассмотренную в [1, 9, 14]. Наряду с очевидными достоинствами наследования, такими, как скорость разработки программ и экономия исходного кода, рассмотренными в [1, 4—14], важно помнить о недостатках наследования, прежде всего о росте требуемой оперативной памяти, о которых говорится в [1, 16].

Р А Б О Т А № 4. **ВИРТУАЛЬНЫЕ МЕТОДЫ**

Цель работы – изучение виртуальных методов (виртуальных функций), создание и использование виртуальных методов с целью экономии объёма программного кода.

Общие сведения

Переопределяемые методы в *связанных иерархией наследования* классах могут быть объявлены как **виртуальные**. Виртуальные методы (или виртуальные функции, члены класса), в отличие от обычных, невиртуальных методов, подключаются к объектно-ориентированной программе не на этапе компиляции программы, а на этапе её выполнения. Использование виртуальных функций позволяет, работая с объектами классов-потомков, вызывать переопределённые методы классов-потомков из унаследованных методов классов-предков. Это, например, позволяет экономить программный код за счёт того, что открывается возможность наследовать те методы, которые до этого наследовать было нельзя.

Так, в примерах 3.2 и 3.3 из лабораторной работы № 3 методы MoveTo(.) имеют полностью совпадающий код в классе-предке Point и в классе-наследнике Circle. Однако метод MoveTo(…) не может наследоваться из класса Point в класс Circle, так как метод MoveTo(…) вызывает метод Show, переопределённый в классе Circle. Формально код метода MoveTo(…) в обоих классах одинаковый, а фактически – разный. То же самое относится и к методу Hide, представленному в обоих этих классах.

Чтобы преодолеть этот разрыв между формой и содержанием, следует объявить метод Show виртуальным. Тогда методы MoveTo() и Hide могут наследоваться классом Circle без переопределения (как это сделано в примерах 4.1–4.3). В этом случае при обращении к объектам класса Circle с сообщениями MoveTo() и Hide будет вызван метод Show, переопределённый в классе Circle. Отсутствие необходимости переопределять методы MoveTo() и Hide в классе Circle приводит к очевидной экономии объёма программного кода.

Правильность использования виртуальных функций (виртуальных методов) в программе легко проверить. Если сначала запустить на выполнение программу с виртуальными методами, а потом снять с этих методов виртуальность и снова запустить на выполнение программу, то в первом случае программа должна выдавать правильный результат, а во втором – должна работать неправильно. Иные итоги работы программы при включении и отключении виртуальности (когда в обоих случаях будет получен верный результат либо в обоих случаях будет получен неверный результат) покажут, что виртуальность использована неправильно с позиции экономии программного кода.

Пример 4.1. Проекты классов Point и Circle (наследник класса Point), с переопределёнными и виртуальными методами

Имя класса: Point (точка)

Поля данных (защищённые): целые X и Y – экранные координаты точки;
целый Color – код цвета точки.

Методы (открытые**):**

а) Конструктор: параметры, целые начальные значения полей X, Y и Color.

б) Методы-акцессоры

GetX – возвращает целочисленное значения поля данных X;
GetY – возвращает целочисленное значения поля данных Y;
GetC – возвращает целочисленное значения поля данных Color;
PutX (целое NX) – задаёт новое целочисленное значение для поля X;
PutY (целое NY) – задаёт новое целочисленное значение для поля Y;
PutColor (целое NC) – задаёт новое целочисленное значение NC для поля Color.

в) Прочие методы

Show – процедура рисования точки на экране;
Hide – процедура, стирающая изображение точки на экране, состоящая в том, что точка рисуется цветом фона;
MoveTo(целочисленные NX, NY) – процедура сдвига точки в позицию с координатами NX и NY, состоящая в том, что сначала точка стирается, затем задаются новые значения координат точки, после чего эта точка рисуется в новой позиции.

Имя класса: Circle (окружность) – **наследник** класса **Point**

Поля данных (защищённые): целый Radius – величина радиуса окружности;
Поля X, Y и Color наследуются у класса Point.

Методы (открытые**):**

а) Конструктор: параметры, начальные значения полей X, Y, Color, Radius.

б) Методы-акцессоры *(GetX, GetY, GetC, PutX, PutY и PutColor наследуются)*
GetRadius – возвращает целочисленное значение поля данных Radius;
PutRadius(integer) – задаёт новое целочисленное значение для поля Radius.

в) Прочие методы

Методы Hide и MoveTo наследуются у класса Point.
Метод Show наследуется у класса Point и переопределяется.
Show – процедура рисования окружности на экране;
Expand (целое DR) – процедура расширения окружности на величину DR, состоящая в том, что окружность стирается, затем задаётся новое значения радиуса, после чего окружность рисуется с новым радиусом.

Пример 4.2. Использование виртуальных функций на языке C++

```
#include <graphics.h>
#include <iostream.h>
#include <conio.h>

class Point                    //объявление класса Point
{
 protected:
            int X;
            int Y;
            int Color;
 public:
                 Point(int, int, int);
            int GetX();
            int GetY();
            int GetC();      //возвращает значение поля  Color
           void PutX(int);
           void PutY(int);
           void PutColor(int);
   virtual void Show();      // виртуальный метод
           void Hide();
           void MoveTo(int, int);
 };

class Circle: public Point          //класс Circle - наследник Point
 {
  protected:
         int Radius;
  public:
             Circle(int, int, int, int);
          int GetRadius();
        void PutRadius(int);
        void Show();          // виртуальный метод
        void Expand(int);
 };

                /*  определение методов объявленных классов */
Point:: Point(int X, int Y, int Color)
                               { this -> X=X;
                                 this -> Y=Y;
                                 this -> Color=Color; }


int Point ::GetX()              { return (X); }
```

```
int Point ::GetY()              { return (Y); }

int Point ::GetC()              { return (Color); }

void Point ::PutX(int X) { this -> X=X; }

void Point ::PutY(int Y) { this -> Y=Y; }

void Point ::PutColor(int Color) { this -> Color=Color; }

void Point:: Show() { putpixel(X, Y, Color); }

void Point:: Hide() { int TempColor;
                      TempColor=GetC();
                      PutColor(getbkcolor());
                      Show();
                      PutColor(TempColor);  }

void Point ::MoveTo(int X, int Y) { Hide();
                                    PutX(X);
                                    PutY(Y);
                                    Show(); }


Circle:: Circle(int X, int Y, int Color, int Radius) :
                                Point(X, Y, Color) // вызов конструктора класса-предка
                                        { this -> Radius = Radius; }

int Circle ::GetRadius()            { return (Radius); }

void Circle ::PutRadius(int Radius) { this -> Radius=Radius; }

void Circle:: Show() { int TempColor;
                       TempColor=getcolor();
                       setcolor(Color);
                       circle(X, Y, Radius);
                       setcolor(TempColor); }

void Circle:: Expand (int DR)
          { Hide();
            if (GetRadius()>= - DR) PutRadius(Radius+DR);
            Show();}
```

```
        /* использование созданных классов в основной программе */

int main()
{
 int gdriver = DETECT, gmode, errorcode;
 initgraph(&gdriver, &gmode, "");
 errorcode = graphresult();
 if (errorcode != grOk)
  {cout<<"Ошибка графики: "<<grapherrormsg(errorcode)<<endl;
   cout<<"Нажмите любую клавишу для прерывания:"<<endl;
   getch();  return(1); }
 setcolor(getmaxcolor());
 Point P1(100,100, getcolor());    // создание объекта класса Point
 P1.Show();          getch();
 P1.MoveTo(150,150);  getch();
 P1.Hide();
 Point P2 = Point(320,240,15);    // создание объекта класса Point
 P2.Show();          getch();
 P2.Hide();
 P2.PutX(360);
 P2.Show();          getch();
 P2.Hide();
 P2.PutY(250);
 P2.Show();          getch();
 P2.Hide();
 Circle C1(295,100,12,70), C2(320,240,15,50); // объекты C1 и C2 класса Circle
 C1.Show();          getch();
 C1.Hide();          getch();
 C1.Show();          getch();
 C1.Expand(10);   getch();
 C1.Expand(-10);  getch();
 C2.Show();          getch();
 C2.Hide();          getch();
 C2.PutX(360);
 C2.Show();          getch();
 C2.MoveTo(400,300);          //вызов унаследованного метода,
                                 // вызывающего виртуальный метод
                     getch();
 C2.Hide();          getch();
 closegraph();
 return(0);
};
```

Пример 4.3. Виртуальные методы на языке Object Pascal

```
program figures;
uses Crt, Graph;
type

 PointPtr= ^Point;
 Point= object                                    {объявление класса Point}

  public              {в более поздних версиях - protected}
        X: integer;
        Y: integer;
    Color: integer;

  public
    constructor Init(InitX, InitY, InitC: integer);
    function GetX: integer;
    function GetY: integer;
    function GetC: integer;
    procedure PutX(NewX: integer);
    procedure PutY(NewY: integer);
    procedure PutColor(NewColor: integer);
    procedure Show; virtual;     { виртуальные методы }
    procedure Hide;
    procedure MoveTo(NewX, NewY: integer);
end;

CircPoint =^CCircle;
CCircle = object (Point)     {объявление класса CCircle – наследника класса Point}

   public      {в более поздних версиях - protected}
       Radius: integer;

   public
     constructor Init(InitX, InitY, InitC, InitR: integer);
     function GetRadius: integer;
     procedure PutRadius(NewRadius: integer);
     procedure Show; virtual;      { виртуальные методы }
     procedure Expand(DR: integer);
end;
```

{определение объявленных методов}

```
constructor Point. Init(InitX, InitY, InitC: integer);
  begin
        X:=InitX;
        Y:=InitY;
        Color:=InitC
  end;

function Point.GetX;    begin    GetX:=X    end;

function Point.GetY;    begin    GetY:=Y   end;

function Point.GetC;    begin    GetC:=Color   end;

procedure Point.PutX(NewX: integer); begin X:=NewX end;

procedure Point.PutY(NewY: integer); begin Y:=NewY end;

procedure Point.PutColor(NewColor: integer);
                                    begin Color:=NewColor end;

procedure Point.Show;  begin  PutPixel(X, Y, Color)  end;

procedure Point.Hide;
 var TempColor: integer;
 begin
   TempColor:=GetC;
   PutColor(GetBkColor);
   Show;
   PutColor(TempColor)
 end;

procedure Point.MoveTo(NewX, NewY: integer);
  begin
   Hide;
   PutX(NewX);
   PutY(NewY);
   Show
  end;
```

```
constructor CCircle.Init(InitX, InitY, InitC, InitR: integer);
  begin
    Point.Init(InitX,InitY,InitC);
    Radius:=InitR
  end;

function CCircle.GetRadius;  begin  GetRadius:=Radius  end;

procedure CCircle.PutRadius(NewRadius: integer);  begin  Radius:=NewRadius  end;

procedure CCircle.Show;
 var TempColor: integer;
 begin
   TempColor:=GetColor;
   SetColor(Color);
   Circle(X, Y, Radius);
   SetColor(TempColor)
 end;

procedure CCircle.Expand(DR: integer);
 begin
   Hide;
   if GetRadius >= -DR then PutRadius(Radius + DR);
   Show
 end;

{создание и использование объектов в основной части программы}

var X,Y, GDriver, GMode, ErrCode: integer;
  P1,P2: Point; C1,C2: CCircle; {создание экземпляров классов Point и CCircle}
begin
 clrscr;
 GDriver:=DETECT;
 GMode:=DETECT;
 InitGraph(GDriver, GMode,'');
 ErrCode:=GraphResult;
 if not (ErrCode = grOk) then
   begin
     writeln('Ошибка графики', GraphErrorMsg(ErrCode));
     writeln('‘Нажмите любую клавишу для прерывания.‘');
     readln;
     halt(1)
   end;
```

```
 P1.Init(100,100,getcolor);
 P1.Show;                  readln;
 P1.MoveTo(150,150);  readln;
 P1.Hide;                 readln;
 P2.Init(320,240,15);
 P2.Show;                  readln;
 P2.Hide;
 P2.PutX(360);
 P2.Show;                  readln;
 P2.Hide;
 C1.Init(295,100,12,70);          {инициализация объекта C1 класса CCircle;}
 C1.Show;                  readln;
 C1.Hide;                  readln;
 C1.Show;                  readln;
 C1.Expand(10);            readln;
 C1.Expand(-10);           readln;
 with C2 do                    {использование оператора  with  языка ПАСКАЛЬ}
   begin
     Init(320,240,15,50);      {инициализация объекта C2 класса CCircle}
     Show;                 readln;
     Hide;
     PutX(360);
     Show;                 readln;
     MoveTo(400,300);      {вызов унаследованного метода,}
                                          { вызывающего виртуальный}
                           readln;
     Hide;                 readln
   end;
 closegraph
end.
```

Задание для лабораторной работы №4

На примере программ с переопределением методов из лабораторной работы №3 показать преимущества использования виртуальных методов с точки зрения экономии объёма программного кода. Для этого, кроме методов MoveTo(.) и Hide, нужно создать методы с общим для разных классов кодом, отвечающие за повороты и за изменения размеров объектов.

В отчёте (аналогично работе №3) сравнить объём исходного программного кода программ с наследованием классов:

а) без переопределения методов,

б) с переопределением методов без виртуальных функций,

в) с переопределением методов, использующим виртуальные функции.

Методическое указание

Особенность использования виртуальных методов рассмотрена в [8, 17].

РАБОТА №5. АБСТРАКТНЫЕ КЛАССЫ И ПОЛИМОРФНЫЕ ОБЪЕКТЫ

Цель работы – изучение идеи абстрактных пользовательских типов данных. Разработка и применение абстрактных методов и классов. Создание и использование полиморфных объектов. Экономия программного кода при использовании полиморфных объектов совместно с абстрактными классами.

Общие сведения

В технологии объектно-ориентированного программирования широко используется и активно развивается идея абстракции данных. Воплощение этой идеи – абстрактные классы. **Абстрактным классом** является класс, содержащий хотя бы один абстрактный метод. **Абстрактный метод** (называемый в C++ **чисто виртуальной** функцией) — это метод только *объявленный*, но *не определённый* в данном классе. Абстрактный метод необходимо определить в классе-наследнике. Таким образом, практическое использование абстрактных классов возможно только при наличии иерархии наследования, в вершине этой иерархии.

В приведённых ниже примерах 5.1—5.3 на языках C++ и Turbo Pascal 7.0, с целью развития иерархии наследования классов, рассмотренной в лабораторных работах № 2, 3 и 4, разработаны абстрактные классы, содержащие абстрактные методы (или *чисто виртуальные функции*).

Реализована экономия программного кода с использованием указателя на абстрактный тип данных. Этот приём экономии кода аналогичен приёму из лабораторной работы №4, использующему виртуальные функции. Однако вместо наследуемого метода MoveTo() предлагается создать одну для всей иерархии классов независимую процедуру Move() с дополнительным аргументом – указателем на абстрактный тип. В этом случае для перемещения геометрических фигур – объектов классов из созданной иерархии — достаточно вызывать одну и ту же процедуру Move() с аргументами – указателями на вышеназванные объекты. Этот приём можно видеть в примерах 5.2 и 5.3. Вместо подстановки в качестве аргументов процедуры Move() указателей на различные объекты из созданной иерархии можно подставлять указатель на полиморфный объект (или полиморфный указатель).

Полиморфный объект – объект, изменяющий свой тип во время работы объектно-ориентированной программы. Если в программе существует иерархия классов, то полиморфный объект может становиться экземпляром любого класса из этой иерархии, сохраняя при этом своё имя. В примерах 5.2 и 5.3 этот полиморфизм реализован именно с помощью указателя на полиморфный объект. Этому указателю в основной программе присваиваются значения указателей на объекты классов из созданной иерархии. Сначала в этих программах процедура Move() вызывается с аргументом – указателем на очередной объект (то есть с полиморфным аргументом), затем – с одним и тем же полиморфным указателем pL, указывающим на разные объекты, то есть с полиморфным объектом.

Пример 5.1. Проекты абстрактного класса Location, его наследника – класса Point и класса Circle – наследника класса Point

Имя абстрактного класса: Location («позиция»)
Поля данных (защищённые): целые X и Y – экранные координаты точки.
Методы (открытые**):**
а) Конструктор: параметры, целые начальные значения полей X и Y;
б) Методы-акцессоры
GetX – возвращает целочисленное значения поля данных X;
GetY – возвращает целочисленное значения поля данных Y;
PutX (целое NX) – задаёт новое целочисленное значение для поля X;
PutY (целое NY) – задаёт новое целочисленное значение для поля Y.
в) Прочие методы
Show – **абстрактная** процедура для рисования на экране;
Hide – **абстрактная** процедура для стирания изображения с экрана.

Имя класса: Point («точка»)
Поле данных (защищённое): целый Color – код цвета точки.
Методы (открытые**):**
а) Конструктор: параметры, целые начальные значения полей X, Y и Color.
б) Методы-акцессоры
GetC – возращает целочисленное значения поля данных Color;
PutColor (целое NC) – задаёт новое значение NC для поля Color.
в) Прочие методы
Show – процедура рисования точки, определяющая абстрактный метод;
Hide – процедура, определяющая абстрактный метод, стирающая изображение точки на экране (геометрическая фигура рисуется цветом фона).

Имя класса: Circle (окружность) – **наследник** класса **Point**
Поля данных (защищённые): целый Radius – величина радиуса окружности.
Поля X, Y и Color наследуются у класса Point.
Методы (открытые**):**
а) Конструктор: параметры, целые начальные значения X, Y, Color, Radius.
б) Методы-акцессоры *(GetX, GetY, GetC, PutX, PutY и PutColor наследуются)*
GetRadius – возвращает целочисленное значение поля данных Radius;
PutRadius(integer) – задаёт новое целочисленное значение для поля Radius.

в) Прочие методы
Метод Hide наследуется у класса Point.
Метод Show наследуется у класса Point и переопределяется.
Show – процедура рисования окружности на экране;
Expand (целое DR) – процедура расширения окружности на величину DR, состоящая в том, что окружность стирается, затем задаётся новое значения радиуса, после чего окружность рисуется с новым радиусом.

Пример 5.2. Абстрактные классы и полиморфные объекты на языке C++

```
#include <graphics.h>
#include <iostream.h>
#include <conio.h>

class Location              //объявление абстрактного класса Location
 {
  protected:
              int X;
              int Y;
  public:
                   Location(int, int);
               int GetX();
               int GetY();
              void PutX(int);
              void PutY(int);
    virtual void Show()=0;    // чисто виртуальная функция – абстрактный метод
   virtual void Hide()=0;      // чисто виртуальная функция – абстрактный метод
 };

class Point: public Location              // класс Point: наследник абстрактного
класса
 {
  protected:
              int Color;
  public:
                   Point(int, int, int);
               int GetC();                    // возвращает значение поля  Color
              void PutColor(int);
              void Show();
              void Hide();
 };

class Circle: public Point                  // класс Circle - наследник Point
 {
   protected:
            int Radius;
   public:
                Circle(int, int, int, int);
             int GetRadius();
          void PutRadius(int);
          void Show();
          void Expand(int);
 };
```

```
            /* определение методов объявленных классов */
Location:: Location(int X, int Y)
        {
         this -> X=X;
         this -> Y=Y;
        }

int Location ::GetX()          { return X; }

int Location ::GetY()          { return Y; }

void Location ::PutX(int X)      { this->X=X; }

void Location ::PutY(int Y)     { this->Y=Y; }

Point::Point(int X,int Y,int Color): Location(X,Y) { this->Color=Color; }

int Point ::GetC()             { return Color; }

void Point ::PutColor(int Color) { this->Color=Color; }

void Point:: Show()   { putpixel(X,Y,Color); }

void Point::Hide()
 {int TempColor;
      TempColor=GetC();
      PutColor(getbkcolor());
      Show();
      PutColor(TempColor);
 }

Circle ::Circle(int X,int Y,int Color,int Radius): Point(X,Y,Color)
                                                    {this->Radius=Radius;}

int Circle ::GetRadius()          { return Radius; }

void Circle ::PutRadius(int Radius) { this->Radius=Radius; }

void Circle:: Show()
{ int TempColor;
          TempColor=getcolor();
          setcolor(Color);
          circle(X, Y, Radius);
          setcolor(TempColor); }
```

```
void Circle:: Expand (int DR)
  {
   Hide();
   if (GetRadius()>= - DR) PutRadius(Radius+DR);
   Show();
  }
void Move(int X, int Y, Location *pL)
  {        // аргумент pL функции Move – указатель на абстрактный объект
   pL->Hide();
   pL->PutX(X);
   pL->PutY(Y);
   pL->Show();
  }
            /* использование созданных классов в основной программе */
int main()
{
 int gdriver = DETECT, gmode, errorcode;
 initgraph(&gdriver, &gmode, "");
 errorcode = graphresult();
 if (errorcode != grOk)
   {
    cout<<"Ошибка графики: "<<grapherrormsg(errorcode)<<endl;
    cout<<"Нажмите любую клавишу для прерывания:"<<endl;
    getch();  return(1);
   }
 setcolor(getmaxcolor());

 Point P1(100,100,getcolor());          // создание объекта класса Point
 P1.Show();              getch();
 Move(150,250,&P1);  getch();           // полиморфный аргумент у Move

 Point P2 = Point(320,240,13);          // создание объекта класса Point
 P2.Show();              getch();
 Move(200,200,&P2);  getch();           // полиморфный аргумент у Move

 Circle C1(295,100,12,70);              // создание объекта C1 класса Circle
 C1.Show();              getch();
 C1.Expand(10);          getch();
 C1.Expand(-10);         getch();
 Move(400,300,&C1);  getch();           // полиморфный аргумент у Move

 Circle C2=Circle(320,240,15,50);       // создание объекта C2 класса Circle
 C2.Show();              getch();
 Move(300,400,&C2);  getch();           // полиморфный аргумент у Move
```

```
 Location *pL;          // объявление указателя на абстрактный объект

 pL=&P1;                // полиморфный указатель
 Move(250,150,pL);   getch();

 pL=&P2;                // полиморфный указатель
 Move(300,300,pL);   getch();

 pL=&C1;                // полиморфный указатель
 Move(300,400,pL);   getch();

 pL=&C2;                // полиморфный указатель
 Move(400,300,pL);   getch();

 P1.Hide();          getch();
 P2.Hide();          getch();
 C1.Hide();          getch();
 C2.Hide();          getch();
 closegraph();
 return(0);
};
```

Пример 5.3. Абстрактные методы на языке Turbo Pascal 7.0

```
program figures;
uses Graph, Crt, Objects;   { библиотека Objects содержит процедуру Abstract }
type

 LocPtr= ^Location;                  {LocPtr - указатель на абстрактный класс}

 Location= object                    {абстрактный класс Location}
  public      {в более поздних версиях - protected}
         X: integer;
         Y: integer;
  public
    constructor Init(InitX, InitY: integer);  {полиморфные}
    function GetX: integer;
    function GetY: integer;
    procedure PutX(NewX: integer);
    procedure PutY(NewY: integer);
    procedure Show; virtual;  { метод определён как abstract }
    procedure Hide; virtual;   { метод определён как abstract }
 end;
```

```
PointPtr= ^Point;
Point= object (Location)                    {класс Point}
 public      {в более поздних версиях - protected}
   Color: integer;
 public
   constructor Init(InitX, InitY, InitC: integer);
   function GetC: integer;
   procedure PutColor(NewColor: integer);
   procedure Show; virtual;
   procedure Hide; virtual;
end;

CircPoint =^CCircle;
CCircle = object (Point)     {класс CCircle – наследник класса Point}
  public      {в более поздних версиях - protected}
      Radius: integer;
  public
    constructor Init(InitX, InitY, InitC, InitR: integer);
    function GetRadius: integer;
    procedure PutRadius(NewRadius: integer);
    procedure Show; virtual;
    procedure Expand(DR: integer);
end;

                    {определение объявленных методов}

constructor Location. Init(InitX, InitY: integer);
  begin
        X:=InitX;
        Y:=InitY
  end;

function Location.GetX;    begin    GetX:=X   end;

function Location.GetY;    begin    GetY:=Y   end;

procedure Location.PutX(NewX: integer); begin X:=NewX end;

procedure Location.PutY(NewY: integer); begin Y:=NewY end;

procedure Location.Show;  begin  Abstract  end;  { абстрактный метод }

procedure Location.Hide;  begin  Abstract  end;  { абстрактный метод }
```

```
constructor Point. Init(InitX, InitY, InitC: integer);
  begin
        Location.Init(InitX,InitY);
        Color:=InitC
  end;

function Point.GetC: integer;    begin    GetC:=Color   end;

procedure Point.PutColor(NewColor: integer);  begin Color:=NewColor end;

procedure Point.Show;  begin  PutPixel(X, Y, Color)  end;

procedure Point.Hide;
 var TempColor: integer;
 begin
   TempColor:=GetC;
   PutColor(GetBkColor);
   Show;
   PutColor(TempColor)
 end;

constructor CCircle.Init(InitX, InitY, InitC, InitR: integer);
  begin
   Point.Init(InitX,InitY,InitC);
   Radius:=InitR
  end;

function CCircle.GetRadius; begin   GetRadius:=Radius   end;

procedure CCircle.PutRadius(NewRadius: integer); begin Radius:=NewRadius end;

procedure CCircle.Show;
 var TempColor: integer;
 begin
   TempColor:=GetColor;
   SetColor(Color);
   Circle(X, Y, Radius);
   SetColor(TempColor)
 end;

procedure CCircle.Expand(DR: integer);
 begin
   Hide;
   if GetRadius >= -DR then PutRadius(Radius + DR);
   Show
 end;
```

```
procedure Move(NewX, NewY: integer; pL: LocPtr);
  begin                                  {pL – указатель на абстрактный тип LocPtr}
    pL^.Hide;
    pL^.PutX(NewX);
    pL^.PutY(NewY);
    pL^.Show
  end;

        {создание и использование объектов в основной части программы}

var X,Y, GDriver, GMode, ErrCode: integer;
  pL: LocPtr;
  P1,P2: Point; C1,C2: CCircle; {объявление экземпляров классов Point и CCircle}
begin
 clrscr;
 GDriver:=DETECT;
 GMode:=DETECT;
 InitGraph(GDriver, GMode,'');
 ErrCode:=GraphResult;
 if not (ErrCode = grOk) then
   begin
     writeln('Ошибка графики', GraphErrorMsg(ErrCode));
     writeln('‘Нажмите любую клавишу для прерывания.‘');
     readln;
     halt(1)
   end;

 P1.Init(100,100,getcolor);
 P1.Show;                  readln;
 Move(150,150,@P1);  { полиморфный аргумент у Move }
                   readln;
 P2.Init(320,240,13);
 P2.Show;                  readln;
 Move(250,250,@P2);  { полиморфный аргумент у Move }
                   readln;
 C1.Init(295,100,12,70);          {инициализация объекта C1 класса CCircle;}
 C1.Show;                  readln;
 C1.Expand(10);            readln;
 C1.Expand(-10);           readln;
 Move(400,300,@C1);  { полиморфный аргумент у Move }
                   readln;
  C2.Init(320,240,15,50);           {инициализация объекта C2 класса CCircle}
 C2.Show;                  readln;
 Move(300,400,@C2); { полиморфный аргумент у Move }
                   readln;
```

```
 pL:=@P1;              {полиморфный указатель}
 Move(200,200,pL);          readln;
 pL:=@P2;              {полиморфный указатель}
 Move(100,100,pL);          readln;
 pL:=@C1;              {полиморфный указатель}
 Move(300,400,pL);          readln;
 pL:=@C2;              {полиморфный указатель}
 Move(400,300,pL);          readln;

 P1.Hide;              readln;
 P2.Hide;              readln;
 C1.Hide;              readln;
 C2.Hide;              readln;
 closegraph
end.
```

Задание для лабораторной работы №5

На основе программ, созданных при выполнении соответствующих заданий к лабораторным работам № 2, 3 и 4, создать программы, в которых иерархия классов расширяется абстрактными классами аналогично приведённым выше примерам. При этом следует, используя абстрактные классы, вместо наследуемых методов MoveTo и других методов с общим для разных классов кодом (отвечающих за повороты и за изменения размеров геометрических объектов), создать не принадлежащие классам процедуры, имеющие дополнительные аргументы – указатели на абстрактные типы даны. С их помощью продемонстрировать использование полиморфных объектов. Показать иную, чем при использовании виртуальных методов, возможность экономии кода. Сравнить с экономией кода во 2-й и 4-й работах.

Методические указания

Абстрактные методы и классы рассмотрены, например, в [1, 6, 9, 14]. Использование полиморфных указателей на объекты классов из одной иерархии показано в [4]. При этом важно заметить, что прикладная, практическая сторона использования абстрактных классов не всегда подробно и детально рассматривается в литературе по объектно-ориентированному программированию. Путаницу вносит неудачная терминология, когда абстрактные методы называют в языке C++ чисто виртуальными функциями. Это создаёт ложное представление, что якобы абстрактные методы есть разновидность виртуальных функций. Такое представление неверно (см. общие сведения к лабораторным работам № 4 и 5). Однако программист-практик сам может изобрести разные полезные программные конструкции и разработать новые эффективные приёмы, получив достаточный опыт в работе как с абстрактными методами, так и с полиморфными объектами.

РАБОТА №6. КОМПОЗИЦИЯ КЛАССОВ И ОБЪЕКТОВ

Цель работы – изучение композиции как особого метода разработки классов и создания объектов в объектно-ориентированном программировании. Освоение некоторых синтаксических различий при использовании композиции в разных алгоритмических языках.

Общие сведения

В объектно-ориентированном программировании возможно использовать в качестве полей данных одного класса объекты любого другого класса. Такой приём, называемый в литературе **композицией** (см. [1, 17]), позволяет устанавливать между классами иную, нежели при наследовании, связь.

Композиция (также называемая в [11] ***агрегацией***) позволяет разрабатывать программу, по своей структуре приближенную более к реальной задаче, нежели к особенностям синтаксиса конкретного языка программирования.

При использовании композиции следует знать об особенностях проектирования классов в таком контексте. В этом случае существенно различаются не только типы полей данных, но и типы аргументов конструкторов. Также различаются тексты и типы аргументов многих других методов.

При проектировании классов на алгоритмическом языке C++ необходимо создавать наряду с традиционными конструкторами конструкторы «по умолчанию» (см. ниже примеры 6.1 и 6.2).

Это так же необходимо, как и создание указателей на классы в текстах любых программ на традиционных версиях языка Турбо-Паскаль (см. все представленные в настоящем учебном пособии Паскаль-программы).

В приведённых далее примерах на языках C++ и Object Pascal показана композиция классов, имеющих в качестве полей данных экземпляры других классов (***подобъекты***, как их называют в [9]). Особо следует обратить внимание на создание объектов классов, использующих композицию. Для таких объектов должны быть заранее созданы более примитивные объекты – входные аргументы конструкторов.

Задания к этой работе разделены на две части для более полного освоения композиции. В первой части предлагается использовать композицию по принципу «матрёшки», когда экземпляры каждого класса в композиционной иерархии становятся полями данных следующего за ним по порядку класса. Объект каждого нового уровня может быть изображён массивом объектов более низкого уровня. Такова иерархия композиции классов и объектов «точка – отрезок – решётчатый треугольник – решётчатый тетраэдр».

Во второй части объекты всех нижних уровней могут быть полями данных объекта высшего уровня. Такова иерархия композиции «точка – отрезок – треугольник – тетраэдр», в которой каждая фигура задаётся точками-вершинами и может иметь либо стороны, либо рёбра и грани. При таком объектно-ориентированном проектировании в одном объекте могут на равных правах присутствовать экземпляры разных классов.

Пример 6.1. Проекты класса Point и класса Line с полями – объектами класса Point

Имя класса: Point (точка)

Поля данных (защищённые):
целые X и Y – экранные координаты точки.

Методы (открытые**):**
а) Конструктор: параметры InitX, InitY – начальные значения полей X и Y.

б) Методы-акцессоры

GetX – возвращает целочисленное значения поля данных X;

GetY – возвращает целочисленное значения поля данных Y;

PutX (целое NX) – задаёт новое целочисленное значение для поля X;

PutY (целое NY) – задаёт новое целочисленное значение для поля Y.

в) Прочие методы

Show – процедура рисования точки на экране;

Hide – процедура, стирающая изображение точки на экране, состоящая в том, что точка рисуется цветом фона;

MoveTo(целочисленные NewX, NewY) – процедура сдвига точки в позицию с координатами NewX и NewY, состоящая в том, что сначала точка стирается, затем задаются новые значения координат точки, после чего эта точка рисуется в новой позиции.

Имя класса: Line (отрезок)

Поля данных (открытые):
P1, P2 – концы отрезка: объекты класса **Point**.

Методы (открытые**):**
а) Конструктор: параметры – начальные значения полей P1 и P2.

б) Прочие методы

Show – процедура рисования отрезка на экране;

Hide – процедура стирания отрезка с экрана;

MoveTo(целочисленные DX, DY) – процедура сдвига отрезка на величину DX вдоль оси X и на величину DY вдоль оси Y, состоящая в том, что сначала отрезок стирается, затем задаются новые значения координат точки, после чего этот отрезок рисуется в новой позиции.

Пример 6.2. Механизм композиции на алгоязыке C++

```
#include <graphics.h>
#include <iostream.h>
#include <conio.h>

class Point
 {
  private:
         int X;
         int Y;
  public:
           Point(int, int);
           Point(){};     // обязательный для C++ конструктор "по умолчанию"
    void PutX(int);
    void PutY(int);
      int GetX();
      int GetY();
    void Show();
    void Hide();
    void MoveTo(int, int);
 };

class Line
 {
  public:
     Point P1;                          // поля данных класса Line -
     Point P2;                          // экземпляры (объекты) класса Point
           Line(Point, Point);
      void Show();
      void Hide();
      void MoveTo(int);
 };

                        /* определение объявленных методов */

     Point::Point(int X, int Y)
      {
       this -> X=X;
       this -> Y=Y;
      }

void Point:: PutX(int X)      { this -> X=X; }

void Point:: PutY(int Y)      { this -> Y=Y; }
```

```
int Point:: GetX()       { return(X); }

int Point:: GetY()       { return(Y); }

void Point::Show()      { putpixel(X, Y, getcolor()); }

void Point::Hide()      { putpixel(X, Y, getbkcolor()); }

void Point:: MoveTo(int X, int Y)
      {
       Hide();
       PutX(X);
       PutY(Y);
       Show();
      }

   Line:: Line(Point P1, Point P2)        // аргументы метода - объекты
      {
       this -> P1=P1;                     // присвоение объектов объектам
       this -> P2=P2;
      }

void Line:: Show() { line(P1.GetX(), P1.GetY(), P2.GetX(), P2.GetY()); }

void Line:: Hide()
      {
       unsigned TempColor;
       TempColor=getcolor();
       setcolor(getbkcolor());
       line(P1.GetX(), P1.GetY(), P2.GetX(), P2.GetY());
       setcolor(TempColor);
      }

void Line:: MoveTo(int DX, int DY)
      {
       Hide();
        P1.PutX(P1.GetX()+DX);
        P1.PutY(P1.GetY()+DY);
        P2.PutX(P2.GetX()+DX);
        P2.PutY(P2.GetY()+DY);
       Show();
      }
```

```
        /* использование разработанных классов в основной программе */
int main()
{
  int gdriver = DETECT, gmode, errorcode;
  initgraph(&gdriver, &gmode, "");
  errorcode = graphresult();
  if (errorcode != grOk)
{
 cout<<"Graphics error: <grapherrormsg(errorcode)<<endl;
        cout<<"Press any key to halt:"<<endl;
        getch();
        return(1);
       };
  setcolor(getmaxcolor());
  Point P1(100,100);                          // создание объекта "точка"
  P1.Show();  getch();
  Point P2 = Point (200,200);                 // создание объекта "точка"
  P2.Show();  getch();
  Line L(P1, P2);              // создание объекта с использованием "точек"
  L.Show();
  getch();
  closegraph();
  getch();
  return(0);
};
```

Пример 6.3. Композиция на алгоритмическом языке Object Pascal

```
program figures;
uses Crt, Graph;
type
 PointPtr= ^Point;
 Point = object                                { объявление класса Point }
   constructor Init(InitX, InitY: integer);
   function GetX: integer;
   function GetY: integer;
   procedure PutX(NewX: integer);
   procedure PutY(NewY: integer);
   procedure Show;
   procedure Hide;
   procedure MoveTo(NewX, NewY: integer);
  private
   X: integer;
   Y: integer;
 end;
```

```
LinePtr =^LLine;
LLine  = object                                   { объявление класса LLine }
   P1:  Point ;                              { поля данных - объекты класса Point }
   P2:  Point ;
   constructor Init(InitP1, InitP2:  Point);
   procedure Show;
   procedure Hide;
end;

constructor Point.Init(InitX, InitY: integer);
 begin
  X:=InitX;
  Y:=InitY;
 end;

function Point. GetX;  begin   GetX:=X  end;

function Point. GetY;  begin   GetY:=Y  end;

procedure Point. PutX(NewX: integer);  begin   X:=NewX  end;

procedure Point. PutY(NewY: integer);  begin   Y:=NewY  end;

procedure Point. Show;  begin   PutPixel(X, Y, GetColor)  end;

procedure Point. Hide;  begin   PutPixel(X, Y, GetBkColor)  end;

procedure Point. MoveTo(NewX, NewY: integer);
 begin
  Hide;
  PutX(NewX);
  PutY(NewY);
  Show
 end;

constructor LLine. Init(InitP1, InitP2: Point);
 begin
   P1.X:=InitP1.GetX;
   P1.Y:=InitP1.GetY;
   P2.X:=InitP2.GetX;
   P2.Y:=InitP2.GetY
 end;
```

```
 procedure LLine. Show;
  begin
   SetColor(GetMaxColor);
   Line(P1.GetX, P1.GetY, P2.GetX, P2.GetY)
  end;

 procedure LLine. Hide;
  var TempColor: word;
  begin
   TempColor:=GetColor;
   SetColor(GetBkColor);
   Line(P1.GetX, P1.GetY, P2.GetX, P2.GetY);
   SetColor(TempColor)
  end;
```

{ Использование разработанных классов в основной программе }

```
var P1: Point; P2: Point; L: LLine; X,Y, gdriver, gmode, errcode: integer;
begin
 clrscr;
 gdriver:=detect;
 gmode:=detect;
 initgraph(gdriver, gmode,'');
 errcode:=GraphResult;
 if not (errcode = grOk) then
  begin
   writeln('Ошибка графики');
   readln;
   halt(1);
  end;
 SetColor(GetMaxColor);
 readln;
 P1.Init(200,200);  P2.Init(400,400);
 P1.Show;  readln;
 P2.Show;  readln;
 L.Init(P1, P2);
 L.Show;  readln;
 L.Hide;  readln;
 closegraph
end.
```

Задание для лабораторной работы № 6

С помощью механизма композиции создать предложенные классы, поля данных которых являются объектами других классов в соответствии с вариантами задания. Изобразить объекты этих классов. Показать возможность определения координаты точки в основании трёхмерной фигуры.

Часть 1

1. "точка",
   "отрезок", определяемый двумя “точками”,
   "решётчатый квадрат" – массив параллельных “отрезков”,
   "куб" – массив параллельных “решётчатых квадратов”.

2. "точка",
   "отрезок", определяемый двумя “точками”,
   "решётчатый квадрат" – массив параллельных “отрезков”,
   "четырёхугольная пирамида", задаваемая несколькими убывающими в размерах параллельно расположенными “решётчатыми квадратами”.

3. "точка",
   "отрезок", определяемый двумя “точками”,
   "окружность" с центром – объектом “точка”, и радиусами – “отрезками”,
   "цилиндр" – массив параллельных направляющих "отрезков",
с основаниями, заданными параллельными “окружностями”.

4. "точка",
   "отрезок", определяемый двумя “точками”,
   "решётчатый параллелограмм" – массив параллельных “отрезков”,
   "параллелепипед", задаваемый массивом параллельно расположенных “решётчатых параллелограммов”.

5. "точка",
   "отрезок", определяемый двумя “точками”,
   "решётчатый ромб" – массив параллельных “отрезков”,
   "ромбическая призма" – массивом параллельных “решётчатых ромбов”.

6. "точка",
   "отрезок", определяемый двумя “точками”,
   "треугольник" – массив параллельных отрезков, убывающих по длине,
   "тетраэдр (треугольная пирамида)", задаваемый массивом параллельных, убывающих в размерах “треугольников”.

7. "точка",
   "отрезок", определяемый двумя “точками”,
   "треугольник" – массив параллельных, убывающих по длине отрезков,
   "треугольная призма" – массив параллельных, убывающих в размерах “треугольников”.

8. "точка",
"отрезок", определяемый двумя “точками”,
"окружность", центр которой – объект "точка", а радиусы – “отрезки”,
"цилиндр", заданный массивом параллельных “окружностей”.

9. "точка",
"отрезок", определяемый двумя “точками”,
"окружность", центр которой – объект "точка", а диаметр – “отрезок”,
"сфера"– массив из несколькими повёрнутых вокруг общего диаметра “окружностей”.

10."точка",
"отрезок", определяемый двумя “точками”,
"решётчатый прямоугольник" – массив параллельных “отрезков”,
"пирамида с прямоугольным основанием" – массив убывающих в своих размерах параллельно расположенных “решётчатых прямоугольников”.

11."точка",
"отрезок", определяемый двумя “точками”,
"прямоугольник", заданный массивом параллельных “отрезков”,
"прямоугольный параллелепипед", заданный несколькими параллельно расположенными “прямоугольниками”.

12."точка",
"отрезок", определяемый двумя “точками”,
"эллипс", центр которого – объект "точка", а полуоси – “отрезки”,
"эллипсоидный цилиндр", изображаемый массивом параллельных одинаковых “эллипсов”, расположенных друг над другом.

13."точка",
"отрезок", определяемый двумя “точками”,
"эллипс", центр которого – объект "точка", а полуоси – “отрезки”,
"эллипсоид", изображаемый массивом по-разному повёрнутых “эллипсов” с общей осью.

14."точка",
"отрезок", определяемый двумя “точками”,
"окружность", центр которой – объект "точка", а радиусы – “отрезки”,
"конус" – массивом параллельных “окружностей”, убывающих в размерах.

15."точка",
"отрезок", определяемый двумя “точками”,
"ромб", заданный массивом параллельно расположенных "отрезков",
"ромбическая пирамида", изображаемая массивом параллельно расположенных уменьшающихся в размерах “ромбов”.

16."точка",
"отрезок", определяемый двумя “точками”,
"эллипс", с центром, заданным объектом "точка",
"эллипсоид", изображаемый массивом параллельно расположенных “эллипсов” с согласованными размерами.

17."точка",
"отрезок", определяемый двумя “точками”,
"окружность", центр которой – объект “точка”, а радиусы – “отрезки”,
"конус"– несколько “отрезков” с общей точкой, образующих конус.

18."точка",
"отрезок", определяемый двумя «точками»,
"параллелограмм", заданный массивом параллельных "отрезков",
"пирамида с параллелограммом в основании", изображаемая массивом параллельных, убывающих в размерах “параллелограммов”.

Часть 2

1. "точка" Point,
"отрезок" Line, определяемый двумя «точками» Point,
"квадрат" Square, заданный двумя вершинами Point или стороной типа Line,
"куб" Cub с двумя вершинами Point, или с ребром Line, или с гранью Square.

2. "точка" Point,
"отрезок" Line, определяемый двумя «точками» Point,
"квадрат" Square, заданный двумя вершинами Point или стороной типа Line,
"четырёхугольная пирамида" SqPyramida с квадратным основанием Square и вершиной Point.

3. "точка" Point,
"отрезок" Line, определяемый двумя «точками» Point,
"окружность" Circle с радиусом – «отрезком» Line, один из концов Line – центр окружности,
"цилиндр" Cylinder, с основанием – «окружностью» Circle и с высотой, задаваемой отрезком Line.

4. "точка" Point,
"отрезок" Line, определяемый двумя «точками» Point,
"параллелограмм" Parallelogram со стороной Line и с вершиной Point, не принадлежащей этой стороне,
"параллелепипед" Parallelelepiped, с нижней гранью типа Parallelogram и с вершиной типа Point, не принадлежащей этой гране.

5. "точка" Point,
"отрезок" Line, определяемый двумя «точками» Point,
"ромб" Rhomb, где вершины – «точки» Point, стороны – «отрезки» Line,
"призма" Prizma с вершинами Point, рёбрами Line, с основаниями – «ромбами» Rhomb и боковыми гранями – «параллелограммами» Parallelogram.

6. "точка" Point,
"отрезок" Line, определяемый двумя “точками” Point,
"треугольник" Triangle со стороной типа Line и с вершиной типа Point,
"тетраэдр” (треугольная пирамида) Tetraedr, задаваемый четырьмя вершинами — “точками” Point или двумя скрещивающимися рёбрами типа Line или треугольной гранью Triangle и вершиной типа Point.

7. "точка" Point,
"отрезок" Line, определяемый двумя «точками» Point,
"треугольник" Triangle, заданный тремя “точками” Point или стороной типа Line и вершиной типа Point,,
"треугольная призма" TrePrizma с треугольными основаниями Triangle и с высотой H.

8. "точка" Point,
"отрезок" Line, определяемый двумя “точками” Point,
"окружность" Circle с радиусом — “отрезком” Line, один из концов Line — центр окружности,
"цилиндр" Cylinder с основаниев “окружность” и с высотой H.

9. "точка" Point,
"отрезок" Line, определяемый двумя “точками” Point,
"окружность" Circle с радиусом — “отрезком” Line, один из концов Line — центр окружности,
"сфера" Sphera, изображаемая тремя окружностями с общим центром и с одинаковым радиусом, “отрезком” Line, один из концов Line — центр сферы.

10."точка" Point,
"отрезок" Line, определяемый двумя “точками” Point,
"прямоугольник" Rectangle, со стороной основания Line и с длиной боковой стороны b,
"пирамида с прямоугольным основанием" RectPyramida, задаваемая вершиной Point и прямоугольным основаниями Rectangle.

11."точка" Point,
"отрезок" Line, определяемый двумя “точками” Point,
"прямоугольник" Rectangle, со стороной основания Line и с длиной боковой стороны b,
"прямоугольный параллелепипед" RectParallelelepiped, задаваемый прямоугольным основаниями Rectangle и высотой H.

12."точка" Point,
"отрезок" Line, определяемый двумя “точками” Point,
"эллипс" Ellipse с центром Point и полуосями размерами a и b,
"эллиптический цилиндр", имеющий основание Ellipse и высоту H.

13."точка" Point,
"отрезок" Line, определяемый двумя “точками” Point,
"эллипс" Ellipse с центром Point и полуосями размерами a и b,
"эллипсоид", изображаемый тремя объектами типа Ellipse с общим центром и с попарно совпадающими полуосями a, b и c.

14."точка" Point,
"отрезок" Line, определяемый двумя “точками” Point,
"окружность" Circle с радиусом – “отрезком” Line, один из концов Line – центр окружности,
"конус" с основанием типа Circle и с вершиной типа Point.

15."точка" Point,
"отрезок" Line, определяемый двумя “точками” Point,
"ромб" Rhomb с диагональю Line и с величиной второй диагонали d,
"ромбическая пирамида" RhombPyramida, имеющая в основании “ромб” Rhomb, а также вершину – “точку” Point.

16."точка" Point,
"отрезок" Line, определяемый двумя “точками” Point,
"эллипс" Ellipse с центром Point и полуосями размерами a и b,
"эллиптический конус" ElConus с основанием Ellipse и с вершиной Point.

17."точка" Point,
"отрезок" Line, определяемый двумя “точками” Point,
"окружность" Circle с центром Point и с радиусом R,
"конус" Conus с основанием – “окружностью” Circle и с вершиной Point.

18."точка" Point,
"отрезок" Line, определяемый двумя “точками” Point,
"параллелограмм" Parallelogram со стороной Line и с вершиной Point, не принадлежащей этой стороне,
"пирамида с параллелограммом в основании" ParPyramida, имеющая основание – “параллелограмм” Parallelogram и вершину Point.

Методические указания

Термин композиция и особенности композиции рассмотрены в [1]. С точки зрения главного девиза объектно-ориентированного программирования «Всё есть объект» композиция естественна. Поэтому термин «композиция» не часто встречается в литературе. В [11] композиция называется термином агрегация, а в [9] объекты, являющиеся полями данных, названы подобъектами. «Пустые» конструкторы при композиции на C++ показаны в [17]. Хотя композиция естественна, при переходе на объектно-ориентированный подход от других способов программирования использование объектов в качестве полей данных непривычно. Поэтому тренировки в композиции весьма полезны для более полного освоения объектно-ориентированной парадигмы программирования.

РАБОТА № 7. ДИНАМИЧЕСКИЕ ОБЪЕКТЫ

Цель работы – изучение особых способов создания динамических объектов в объектно-ориентированном программировании. Изучение особенностей использования динамических объектов в разных алгоритмических языках.

Общие сведения

В объектно-ориентированных программах возможно использование динамических объектов [1,4-6]. Динамическое выделение памяти (так называемое выделение памяти «в куче») состоит в том, что оперативная память объектам выделяется не на этапе компиляции (в стеке), а на этапе выполнения программы, когда создаётся динамический объект. Когда динамический объект использован программой и уже не нужен, он может быть удалён, а занятая под объект оперативная память освобождена и использована во время работы программы для других целей.

Такой приём особенно эффективно может быть использован (и часто используется) в больших интерактивных объектно-ориентированных программах со множеством разновременно используемых объектов, когда предвидеть заранее общий объём потребной оперативной памяти сложно, а зарезервировать память сразу под все программные ситуации весьма затратно.

Для динамического выделения оперативной памяти в большинстве языков объектно-ориентированного программирования используется конструкция **new**. В языке C++ эта конструкция используется следующим образом:

Указатель_На_Объект = **new** *Конструктор_Класса* (*Аргументы_конструктора*);

Для создания динамического объекта в C++ необходимо создать указатель на этот объект. Также необходимо создать указатель на объект и в языке Object Pascal, где конструкция **new** является библиотечной процедурой:

new (*Указатель_На_Объект*);

Удаление объектов и динамическое освобождение памяти реализуется в разных алгоритмических языках по-разному. Так, в C++ для динамического удаления объектов используется операция **delete**:

delete *Указатель_На_Объект*;

В то же время на Object Pascal используется библиотечная процедура:

dispose (*Указатель_На_Объект*);

В приведённых далее примерах программ на языках C++ и Object Pascal, построенных на основе проекта класса из примера 1.1, использованы динамические объекты вместо традиционных, статических (см. примеры 7.1 и 7.2).

Пример 7.1. Особенности создания и удаления динамических объектов на C++

```
#include <conio.h>
#include <iostream.h>
#include <math.h>

class  Complex
 {
  private:
    double Re;
    double Im;
  public:
    Complex(double, double);
    double GetRe();
    double GetIm();
    void PutRe(double);
    void PutIm(double);
    void PrintComp();
    void PrintTrig();
  private:
    double Amp();
    double Fi();
 };

Complex ::Complex(double Re, double Im)
 {
     this->Re = Re;
     this->Im = Im;
 }

double Complex ::GetRe() { return Re; }

double Complex ::GetIm() { return Im; }

void Complex ::PutRe (double Re) { this->Re=Re;}

void Complex ::PutIm (double Im) { this->Im=Im;}

void Complex ::PrintComp() { cout<<"Re="<<Re<<" Im="<<Im<<endl; }

void Complex ::PrintTrig() { cout<<"Am="<<Amp()<<" Fi="<<Fi()<<endl; }

double Complex ::Amp() { return  sqrt(Re*Re + Im*Im); }

double Complex ::Fi() { return atan2(Im, Re); }
```

```
void main()
{
 clrscr();
 Complex * pC;              // объявление указателя на объект типа Complex
 cout<<"CPP: Dynamical Object"<<endl;
 pC= new Complex(1,-1);   // создание динамического объекта
 pC->PrintComp();        getch();
 cout<<"pC->Re="<<pC->GetRe()<<" pC->Im="<<pC->GetIm()<<"\n";   getch();
 pC->PutRe(3);
 pC->PutIm(4);
 cout<<"Re="<<pC->GetRe()<<" Im="<<pC->GetIm()<<"\n";
 pC->PrintTrig();    getch();
 delete pC;              // удаление динамического объекта
 cout<<"After Delete: Re="<<pC->GetRe()<<" Im="<<pC->GetIm()<<"\n"; getch();
};
```

Пример 7.2. Особенности создания и удаления динамических объектов на Object Pascal

```
{$N+}
program Comp1;
uses Crt;
type
 pComplex=^Complex;{объявление типа pComplex – указателя на тип Complex}
 Complex = object
    constructor Init(InitRe, InitIm: double);
    function    GetRe: double;
    function    GetIm: double;
    procedure   PutRe(NewRe: double);
    procedure   PutIm(NewIm: double);
  private
    Re: double;
    Im: double;
    function Amp: double;
    function Fi: double;
  public
    procedure PrintComp;
    procedure PrintTrig;
 end;

constructor Complex.Init(InitRe, InitIm: double);
   begin
     Re:=InitRe;
     Im:=InitIm;
   end;
```

```
function Complex.GetRe: double; begin GetRe:=Re end;

function Complex.GetIm: double; begin GetIm:=Im end;

procedure Complex.PutRe(NewRe: double); begin Re:=NewRe end;

procedure Complex.PutIm(NewIm: double); begin Im:=NewIm end;

function Complex.Amp: double; begin Amp:=sqrt(Re*Re+Im*Im) end;

function Complex.Fi: double;
 begin
  if (Re=0) and (Im=0) then Fi:=0;
  if (Re=0) and (Im<0) then Fi:=-0.5*Pi;
  if (Re=0) and (Im>0) then Fi:=0.5*Pi;
  if (Re>0) then Fi:=ArcTan(Im/Re);
  if (Re<0) then if (Im>=0) then Fi:=ArcTan(Im/Re)+Pi
                            else Fi:=ArcTan(Im/Re)-Pi
 end;

procedure Complex.PrintComp; begin writeln('Re=',Re:7:3,' Im=',Im:7:3) end;

procedure Complex.PrintTrig;
                    begin writeln('Amp=',Amp:7:3,' Fi=',(Fi*180/Pi):7:2) end;

var pC: pComplex; {объявление переменной-указателя pC типа pComplex}

begin
 clrscr;
 new(pC);        { динамическое выделение памяти}
 writeln('Object Pascal: Dynamical object');
 pC^.Init(1,-1);
 pC^.PrintComp; readln;
 writeln('pC^.Re=',pC^.GetRe:7:2,' pC^.Im=',pC^.GetIm:7:2); readln;
 pC^.PutRe(3);
 pC^.PutIm(4);
 pC^.PrintComp; readln;
 pC^.PrintTrig;   readln;
 dispose(pC);         { удаление динамического объекта }
 write('After Dispose:  ');
 pC^.PrintComp;   readln;
end.
```

Важно заметить, что удаление объекта – не столь очевидная операция, как создание объекта. Часто использование команд **delete** и **dispose** в C++ и Object Pascal (и аналогичных команд в других языках программирования) не освобождает оперативную память для дальнейшего использования, а только информирует операционную систему о том, что эта область памяти не используется и больше не нужна. Непосредственно «сборку мусора» осуществляет операционная система. В связи с этим во многих языках программирования есть команды динамического выделения памяти, но нет команд для удаления объектов и динамического освобождения памяти. В таких языках программирования, как, например, Common Lisp, Java и C#, нет команд для удаления объектов, а есть автоматический сборщик мусора – **garbage collector**.

В примерах программ 7.1 и 7.2 преднамеренно выводится информация о полях данных динамического объекта **до** и **после** «удаления» (фиктивного, на что указывают кавычкм) объекта. Из результатов работы программ, представленных в примерах 7.3 и 7.4, видно, что поля данных порой сохраняют своё значение и после «удаления» объекта. Так, на C++ значения обоих полей данных не изменились, а на Object Pascal изменилось значение только первого по счёту поля данных.

Пример 7.3. Результаты работы программы на C++ из примера 7.1.

```
CPP: Dynamical Object
Re=1 Im=-1
pC->Re=1 pC->Im=-1
Re=3 Im=4
Am=5 Fi=0.927295
After Delete:  Re=3 Im=4
```

Пример 7.4. Результаты работы программы на Object Pascal из примера 7.2.

```
Object Pascal: Dynamical Object
Re=  1.000 Im=  1.000

pC^.Re=  -1.00 pC^.Im=  -1.00

Re=  3.000 Im=  4.000

Amp=  5.000 Fi=  53.13

After Dispose:  Re=  0.000 Im=  4.000
```

Пример 7.5. Использование динамических полей данных у динамических объектов при создании и удалении динамических объектов на C++

```
#include <conio.h>
#include <iostream.h>
#include <math.h>

class  Complex
 {
  double *pRe;          //
  double *pIm;          //
 public:
  Complex(double, double);
  double GetRe();
  double GetIm();
  void PutRe(double);
  void PutIm(double);
  void PrintComp();
  void PrintTrig();
 private:
   double Amp();
   double Fi();
 };

Complex::Complex(double Re,double Im)
 {
    pRe = new double;
    pIm = new double;
    (*pRe) =Re;
    (*pIm) =Im;
 }

double Complex::GetRe() { return (*pRe); }

double Complex::GetIm() { return (*pIm); }

void Complex::PutRe(double Re) { (*pRe)=Re;}

void Complex::PutIm(double Im) { (*pIm)=Im;}

void Complex::PrintComp() { cout<<"Re="<<*pRe<<" Im="<<*pIm<<endl; }

void Complex::PrintTrig() { cout<<"Am="<<Amp()<<" Fi="<<Fi()<<endl; }

double Complex::Amp() { return  sqrt((*pRe)*(*pRe) + (*pIm)*(*pIm)); }

double Complex::Fi() { return atan2(*pIm, *pRe); }
```

```
void main()
{
 clrscr();
 Complex * pC;
 cout<<"CPP: Dynamical Fields"<<endl;
 pC= new Complex(1,-1);
 pC->PrintComp();  getch();
 cout<<"pC->Re="<<pC->GetRe()<<" pC->Im="<<pC->GetIm()<<"\n";  getch();
 pC->PutRe(3);
 pC->PutIm(4);
 pC->PrintComp();  getch();
 pC->PrintTrig();  getch();
 delete pC;
cout<<"After Delete:  Re="<<pC->GetRe()<<" Im="<<pC->GetIm()<<"\n"; getch();
};
```

Пример 7.6. Использование динамических полей данных на Object Pascal

```
{$N+}
program Comp1;
uses Crt;
type
 pComplex=^Complex;
 Complex = object
    constructor Init(Re, Im: double);
    function    GetRe: double;
    function    GetIm: double;
    procedure   PutRe(Re: double);
    procedure   PutIm(Im: double);
  private
    pRe: ^double;
    pIm: ^double;
    function Amp: double;
    function Fi: double;
  public
    procedure PrintComp;
    procedure PrintTrig;
 end;

constructor Complex.Init(Re, Im: double);
   begin
     new(pRe);
     new(pIm);
     pRe^:=Re;
     pIm^:=Im
   end;
```

```
function Complex.GetRe: double; begin GetRe:=pRe^ end;

function Complex.GetIm: double; begin GetIm:=pIm^ end;

procedure Complex.PutRe(Re: double); begin pRe^:=Re end;

procedure Complex.PutIm(Im: double); begin pIm^:=Im end;

function Complex.Amp: double; begin Amp:=sqrt(pRe^*pRe^+pIm^*pIm^) end;

function Complex.Fi: double;
 begin
  if (pRe^=0) and (pIm^=0) then Fi:=0;
  if (pRe^=0) and (pIm^<0) then Fi:=-0.5*Pi;
  if (pRe^=0) and (pIm^>0) then Fi:=0.5*Pi;
  if (pRe^>0) then Fi:=ArcTan(pIm^/pRe^);
  if (pRe^<0) then if (pIm^>=0) then Fi:=ArcTan(pIm^/pRe^)+Pi
                                else  Fi:=ArcTan(pIm^/pRe^)- Pi;
 end;

procedure Complex.PrintComp; begin writeln('Re=',pRe^:7:3,' Im=',pIm^:7:3) end;

procedure Complex.PrintTrig;
                          begin writeln('Amp=',Amp:7:3,' Fi=',(Fi*180/Pi):7:2) end;
var pC: pComplex;

begin
 clrscr;
 new(pC);
 writeln('Object Pascal: Dynamical Fields');
 pC^.Init(1,-1);
 pC^.PrintComp; readln;
 writeln('pC^.Re=',pC^.GetRe:7:2,' pC^.Im=',pC^.GetIm:7:2); readln;
 pC^.PutRe(3);
 pC^.PutIm(4);
 pC^.PrintComp; readln;
 pC^.PrintTrig;   readln;
 dispose(pC);
 write('After Dispose: ');  pC^.PrintComp;   readln;
end.
```

В примерах программ 7.5 и 7.6 вместе с динамическими объектами используются динамические поля данных, также создаваемые с помощью указателей. В проекте класса Complex из примера 1.1 для отражения этой ситуации нужно изменить позиции «Поля данных» и «Конструктор»:

Поля данных (закрытые, вещественные, динамические):
pRe – указатель на действительную часть комплексного числа;
pIm – указатель на мнимую часть комплексного числа.

Конструктор (открытый): входные параметры, вещественные начальные значения полей InitRe и InitIm, на которые указывают pRe и pIm. Перед присвоением начальных значений в теле конструктора должна быть динамически выделена оперативная память с помощью конструкции **new** и указателей pRe, pIm.

Как можно видеть, память для динамических полей данных выделяется при вызове конструктора. Из результатов работы программ, представленных в примерах 7.7 и 7.8, видно, что в этом случае поля данных теряют своё значение сразу после «удаления» объекта как на C++, так и на Object Pascal.

Пример 7.7. Результаты работы программы на C++ из примера 7.5

```
CPP: Dynamical Fields
Re=1 Im=-1
pC->Re=1 pC->Im=-1
Re=3 Im=4
Am=5 Fi=0.927295
After Delete:  Re=-7.848794e-54 Im=-1.737155e+193
```

Пример 7.8. Результаты работы программы на Object Pascal из примера 7.6

```
Object Pascal: Dynamical Fields
Re=  1.000 Im=  1.000

pC^.Re=  -1.00 pC^.Im=  -1.00

Re=  3.000 Im=  4.000

Amp=  5.000 Fi=  53.13

After Dispose:  Re=  0.000 Im=  0.000
```

В примерах программ 7.9 и 7.10 созданы и используются **деструкторы** классов для удаления объектов этих классов. В изменённом проекте класса Complex из примера 1.1 нужно дополнительно к динамическим полям данных и к позиции Конструктор добавить позицию:

Деструктор (открытый): удаляет динамические поля данных.

Деструкторы есть методы, предназначенные для удаления объектов. Фактически необходимость в деструкторах появляется лишь тогда, когда нужно удалять динамические поля данных, как это видно в примерах программ 7.9 и 7.10. В C++ имя деструктора образовывается с помощью имени класса, перед которым ставится знак «тильда», например у класса Complex — деструктор **~Complex()**. Тип возвращаемого значения у деструктора, как и у конструктора Complex(), отсутствует и не указывается. В Object Pascal деструктор объявляется с помощью синтаксической конструкции **destructor**, так же как конструкторы объявляются с помощью constructor. Например, у класса *Complex* деструктор объявляется как *destructor Del*. Из примеров видно, что деструкторы вызываются *явно* как на Object Pascal, так и на C++.

Деструкторы должны осуществлять удаление динамических полей данных, аналогично тому как конструкторы создают динамические поля данных и задают начальные значения полей данных. Однако, как видно из результатов работы программ, представленных в примерах 7.11 и 7.12, в этом случае поля данных теряют своё значение сразу после вызова деструкторов на Object Pascal. В то же время у полей данных объектов на C++ сразу после вызова деструктора ~Complex() и сразу после «удаления» командой delete изменяются только дробные части. Очевидно, что память, освободившаяся при «удалении» полей данных, будет использована операционной системой позже, что можно проверить экспериментально в другой, более крупной программе.

Пример 7.9. Использование на языке C++ деструкторов для удаления динамических объектов

```
#include <conio.h>
#include <iostream.h>
#include <math.h>

class  Complex
{ double *pRe;
  double *pIm;
 public:
  Complex(double, double);
  ~Complex();
  double GetRe();
  double GetIm();
  void PutRe(double);
  void PutIm(double);
  void PrintComp();
  void PrintTrig();
 private:
   double Amp();
   double Fi();
};

Complex:: Complex(double Re,double Im)
 {  pRe = new double;
    pIm = new double;
    (*pRe) =Re;
    (*pIm) =Im;
 }

Complex:: ~Complex()              // деструктор
 {  delete pRe;
    delete pIm;
 }
```

```
double Complex::GetRe() { return (*pRe); }
double Complex::GetIm() { return (*pIm); }
void Complex::PutRe(double Re) { (*pRe)=Re;}
void Complex::PutIm(double Im) { (*pIm)=Im;}
void Complex::PrintComp() { cout<<"Re="<<*pRe<<" Im="<<*pIm<<endl; }
void Complex::PrintTrig() { cout<<"Am="<<Amp()<<" Fi="<<Fi()<<endl; }
double Complex::Amp() { return  sqrt((*pRe)*(*pRe) + (*pIm)*(*pIm)); }
double Complex::Fi() { return atan2(*pIm, *pRe); }
void main()
{
 clrscr();
 Complex * pC;
 pC= new Complex(1,-1);
 pC->PrintComp();  getch();
 cout<<"pC->Re="<<pC->GetRe()<<" pC->Im="<<pC->GetIm()<<"\n";  getch();
 pC->PutRe(3);
 pC->PutIm(4);
 pC->PrintComp();  getch();
 pC->PrintTrig();     getch();
 pC->~Complex();
 cout<<"After Destructor:  ";  pC->PrintComp();
 delete pC;
 cout<<"After Delete:  "; pC->PrintComp();
};
```

Пример 7.10. Использование деструкторов на Object Pascal

```
{$N+}
program Comp1;
uses Crt;
type

 pComplex=^Complex;
 Complex = object
  constructor Init(Re, Im: double);
  destructor  Del;                    {деструктор}
  function    GetRe: double;
  function    GetIm: double;
  procedure   PutRe(Re: double);
  procedure   PutIm(Im: double);
```

```
  private
    pRe: ^double;
    pIm: ^double;
    function Amp: double;
    function Fi: double;

  public
    procedure PrintComp;
    procedure PrintTrig;
 end;

constructor Complex.Init(Re, Im: double);
   begin
     new(pRe);
     new(pIm);
     pRe^:=Re;
     pIm^:=Im
   end;

destructor Complex.Del;
   begin
     dispose(pRe);
     dispose(pIm)
   end;

function Complex.GetRe: double; begin GetRe:=pRe^ end;

function Complex.GetIm: double; begin GetIm:=pIm^ end;

procedure Complex.PutRe(Re: double); begin pRe^:=Re end;

procedure Complex.PutIm(Im: double); begin pIm^:=Im end;

function Complex.Amp: double; begin Amp:=sqrt(pRe^*pRe^+pIm^*pIm^) end;

function Complex.Fi: double;
 begin
  if (pRe^=0) and (pIm^=0) then Fi:=0;
  if (pRe^=0) and (pIm^<0) then Fi:=-0.5*Pi;
  if (pRe^=0) and (pIm^>0) then Fi:=0.5*Pi;
  if (pRe^>0) then Fi:=ArcTan(pIm^/pRe^);
  if (pRe^<0) then if (pIm^>=0) then Fi:=ArcTan(pIm^/pRe^)+Pi
                          else Fi:=ArcTan(pIm^/pRe^)-Pi
 end;

procedure Complex.PrintComp; begin writeln('Re=',pRe^:7:3,' Im=',pIm^:7:3) end;

procedure Complex.PrintTrig;
                              begin writeln('Amp=',Amp:7:3,' Fi=',(Fi*180/Pi):7:2) end;
```

```
var pC: pComplex;

begin
 clrscr;
 new(pC);
 writeln('Object Pascal: Destructor');
 pC^.Init(1,-1);
 pC^.PrintComp;   readln;
 writeln('pC^.Re=',pC^.GetRe:7:2,' pC^.Im=',pC^.GetIm:7:2);  readln;
 pC^.PutRe(3);
 pC^.PutIm(4);
 pC^.PrintComp;   readln;
 pC^.PrintTrig;      readln;
 pC^.Del;
 write('After Destructor:  ');  pC^.PrintComp;   readln;
 dispose(pC);
 write('Ater Dispose:  ');  pC^.PrintComp;   readln;
end.
```

Пример 7.11. Результаты работы программы на C++ из примера 7.9

```
CPP: Destructor
Re=1 Im=-1
pC->Re=1 pC->Im=-1
Re=3 Im=4
Am=5 Fi=0.927295
After Destructor: Re=3.070804 Im=4
After Delete:  Re=3.000001 Im=4.141609
```

Пример 7.12. Результаты работы программы на Object Pascal из примера 7.10

```
Object Pascal: Destructor
Re=  1.000 Im=  1.000

pC^.Re=  -1.00 pC^.Im=  -1.00

Re=  3.000 Im=  4.000

Amp=  5.000 Fi=  53.13

After Destructor:  Re=  0.000 Im=  0.000

After Dispose:  Re=  0.000 Im=  0.000
```

Задание для лабораторной работы № 7

Используя класс, созданный при выполнении лабораторной работы №1:
1. Создать динамические объекты этого класса в соответствии с вариантами задания. Продемонстрировав движение созданных динамических объектов, удалить их, после чего проверить факт удаления, выведя на экран значения полей данных объектов до и после удаления, как показано в примерах 7.1—7.4.
2. Видоизменить созданный класс, создав динамические поля данных, как показано в примерах 7.5 и 7.6. Показав движение таких динамических объектов, с динамически создаваемыми полями данных, удалить объекты этих классов, после чего проверить факт удаления, выведя на экран значения полей данных объектов до и после удаления, как показано в примерах 7.7 и 7.8.
3. Добавить к видоизменённому в п.2 классу деструктор, удаляющий динамические поля данных, аналогично примерам 7.9 и 7.10. Затем, после демонстрации движения объектов этого класса, удалить объекты этих классов, после чего проверить факт удаления, выведя на экран значения полей данных этих объектов до и после удаления, как показано в примерах 7.11 и 7.12.

Отразить результаты исследования функционирования динамических объектов, динамических полей данных и деструкторов в выводах к настоящей лабораторной работе. Высказать собственные соображения по поводу эффективности динамического освобождения динамически выделенной оперативной памяти непосредственно по команде пользователя в исходном тексте прикладной программы. Также рассмотреть и обсудить эффективность создания и использования деструкторов класса.

Высказать собственное мнение по поводу автоматической *сборки мусора*, а также по поводу отсутствия деструкторов во многих объектно-ориентированных языках программирования, поддерживающих динамическое выделение памяти, таких, например, как Common Lisp, Java и C#.

Методические указания

При подготовке к выполнению задания следует ознакомиться с понятием *динамическая память* и со способами её выделения и освобождения, а также с понятием *деструктор*, рассмотренными, например, в [1, 4 – 6].

Р А Б О Т А № 8. ИСПОЛЬЗОВАНИЕ ОБЪЕКТОВ В КАЧЕСТВЕ ВХОДНЫХ И ВЫХОДНЫХ АРГУМЕНТОВ ФУНКЦИЙ

Цель работы — изучение всех возможных способов передачи объектов внутрь функций (как методов классов, так и функций, не являющихся членами класса) при осуществлении объектно-ориентированного программирования на алгоритмических языках C++ и Object Pascal, а также изучение всех возможных способов возвращения объектов из функций в вызывающую программу при осуществлении программирования на этих языках.

Освоение общих приёмов и синтаксических различий использования объектов в качестве входных и выходных аргументов функций в этих алгоритмических языках.

Общие сведения

Рассмотренные в настоящей лабораторной работе приёмы программирования родственны приёмам, изучаемым в лабораторной работе №6. Если в лабораторной работе № 6, посвящённой композиции, объекты одних классов являются полями данных других классов, то в настоящей лабораторной работе объекты рассматриваются как входные и выходные аргументы функций. Это могут быть методы классов (функции, члены класса в C++), а также процедуры и функции, не принадлежащие никакому классу (в гибридных объектно-ориентированных языках).

Важно знать, что передать аргументы-объекты в метод (в процедуру или в функцию) можно несколькими разными способами. Также несколько разных способов имеется для возврата объектов из методов (или из самостоятельных процедур и из самостоятельных функций) в вызывающую программу.

Для демонстрации этих способов предлагается рассмотреть применение объектов класса Vector3D (*вектор в трёхмерном пространстве* — см. далее, пример 8.1) при реализации операции векторного умножения.

Операция векторного умножения двух векторов **A** и **B**, результатом которой является вектор **C**:

$$\mathbf{C=A*B,} \qquad (8.1)$$

осуществляется в соответствии с общеизвестными формулами:

$$\mathbf{C_x= A_yB_z - A_zB_y} \ ,$$

$$\mathbf{C_y= A_zB_x - A_xB_z} \ ,$$

$$\mathbf{C_z= A_xB_y - A_yB_x} \ .$$

где $\mathbf{A_x}$, $\mathbf{A_y}$, $\mathbf{A_z}$, $\mathbf{B_x}$, $\mathbf{B_y}$, $\mathbf{B_z}$, $\mathbf{C_x}$, $\mathbf{C_y}$, $\mathbf{C_z}$ – скалярные координаты векторов **A** , **B** и **C** соответственно.

Пример 8.1.Проект класса Vector3D (вектор с тремя координатами)

Имя класса: Vector3D

Поля данных (закрытые): вещественные X, Y и Z – координаты конца вектора с началом в центре координат.

Методы (открытые**):**

а) Конструктор: входные параметры InitX, InitY, InitZ — вещественные начальные значения полей X, Y.

б) Методы-акцессоры

GetX — возвращает вещественное значение поля данных X;
GetY — возвращает вещественное значение поля данных Y;
GetZ — возвращает вещественное значение поля данных Z;
PutX (вещественное NX) – задаёт новое значение NX для поля X;
PutY (вещественное NY) – задаёт новое значение NY для поля Y;
PutZ (вещественное NZ) – задаёт новое значение NZ для поля Z.

в) Прочие методы

Методы, осуществляющие различными способами векторное умножение двух входных векторов и возвращающие вектор, являющийся их векторным произведением.

PrintVector – метод, осуществляюший печать координат вектора.

Объекты класса Vector3D при реализации операции векторного умножения используются в качестве входных и выходных аргументов методов этого класса, а также в качестве входных и выходных аргументов самостоятельных функций и процедур, созданных на языках C++ и Object Pascal.

Передача объектов в методы и подпрограммы

Возможна передача аргументов внутрь процедур и функций (методов) «по значению», когда параметры-объекты копируются внутрь подпрограмм для дальнейшего их использования. В этом случае изменения в объектах внутри подпрограммы не влияют на передаваемые объекты в вызывающей программе. Однако оперативная память, выделяемая для этих аргументов-объектов, за счёт копирования увеличивается вдвое.

Ниже представлены объявления трёх функций, членов класса Vector3D на C++ и трёх методов класса Vector3D на Object Pascal.

```
void MultVect(Vector3D, Vector3D&);      // функция №1, член класса
void pMultVectV(Vector3D, Vector3D*); // функция №2, член класса
Vector3D* pMultVect(Vector3D);           // функция №3, член класса
```

```
procedure MultVect(B: Vector3D; var C: Vector3D);      {метод №1}
procedure pMultVectV(B: Vector3D;  pC: pVector3D);    {метод №2}
function pMultVect(B: Vector3D): pVector3D;               {метод №3}
```

Синтаксис использования метода **MultVect** (№1) для операции (8.1) и на C++, и на Object Pascal представляет собой конструкцию

A.MultVect(B, C); (8.2)

Синтаксис использования метода **pMultVect** (№2) для операции (8.1) представляет собой конструкции:

на C++ **A.pMultVectV(B, pC);** (8.3)

где **pC** – указатель на объект **C (Vector3D* pC)**;

на Object Pascal **A.pMultVectV(B, @C)**; (8.4)

где **@C** – указатель на объект **C**.

Синтаксис использования метода pMultVect (№3) для операции (8.1) представляет собой конструкции:

на C++ **pC= A.pMultVect(B)**; (8.5)

где pC – указатель на объект C **(Vector3D* pC);**

на Object Pascal **C:= A.pMultVect(B)^**; (8.6)

где **A.pMultVect(B)^ –** определение содержимого по указателю, возвращаемому конструкцией **A.pMultVect(B).**

Из конструкций (8.2)-(8.6) видно, что первый операнд операции (8.1) выступает как объект **A**, к которому посылаются сообщения с помощью методов **MultVect**, **pMultVectV** или **pMultVect**). Первый аргумент метода есть второй операнд **B** операции (8.1). Аргумент **B** в конструкциях (8.2)–(8.6) передаётся внутрь методов «по значению».

Так же точно передаются входные параметры-объекты в самостоятельные процедуры и функции, не являющихся методами (функциями, членами класса).

Далее представлены объявления функций и процедур на языках C++ и Object Pascal. Входные аргументы-объекты передаются внутрь этих функций и процедур «по значению».

```
void MultV(Vector3D, Vector3D, Vector3D&);        // функция №4 на C++
Vector3D* pMultVct(Vector3D, Vector3D);           // функция №5 на C++
void MultVec(Vector3D, Vector3D, Vector3D*);      // функция №6 на C++
procedure MultV(A, B: Vector3D; var C: Vector3D); {процедура №4 на Object
                                                    Pascal}
function pMultVct(A, B: Vector3D): pVector3D;     {функция №5 на Object Pascal}
procedure MultVec(A, B: Vector3D; pC: pVector3D); {процедура №6 на Object
                                                    Pascal}
```

Синтаксис использования процедуры **MultV** (№4) при реализации операции (8.1) как на C++, так и на Object Pascal выглядит как конструкция

MultV(A, B, C); (8.7)

Синтаксис использования функции **pMultVct** (№5) при реализации операции (8.1) на C++ выглядит как конструкция

pC= pMultVct(A, B); (8.8)

где **pC** – указатель на объект **C** (объявляется как **Vector3D * pC)**.

Синтаксис использования функции **pMultVct** (№5) при реализации операции (8.1) на Object Pascal выглядит как конструкция

pC := pMultVct(A, B); (8.9)

где **pC** – указатель на объект **C** (объявляется как **pC: ^Vector3D)**;

Синтаксис использования процедуры **MultVec** (№6) при реализации операции (8.1) как на C++, так и на Object Pascal выглядит как конструкция

MultVec(A, B, pC); (8.10)

где **pC** – указатель на объект **C** (см. выше комментарии к (8.8) и (8.9)).

Из конструкций (8.7)–(8.10) видно, что первый и второй операнды операции (8.1), объекты **A** и **B**, есть первые два аргумента процедур **MultV** и **MultVec.** Также первый и второй операнды операции (8.1), объекты **A** и **B**, есть два аргумента функции **pMultVct.** Аргументы-объекты **A** и **B** в конструкциях (8.7)–(8.10) передаются внутрь этих процедур и функций «по значению».

Передача объектов в методы и подпрограммы с помощью указателей на объекты

С другой стороны, кроме передачи параметров «по значению» также возможна другая передача аргументов внутрь процедур и функций (методов) – «по имени» или с помощью указателя, когда значения параметров-объектов в подпрограмме и в вызывающей программе находятся в одной и той же области оперативной памяти. В таком случае параметры-объекты в вызывающей программе и в подпрограмме различаются только именами (передача «по имени»), либо внутрь подпрограммы передаются указатели на параметры-объекты (адреса параметров-объектов). Тогда одна и та же область памяти, где хранятся значения объектов, имеет два имени (две параметрические ссылки) – в вызывающей программе и в подпрограмме. Так же точно указатели в вызывающей программе и в подпрограмме указывают на одну и ту же область памяти, где расположены значения объектов.

Эта передача параметров в подпрограммы не увеличивает расход оперативной памяти, в отличие от передачи параметров «по значению».

Ниже представлены объявления трёх функций, членов класса Vector3D на C++ и трёх методов класса Vector3D на Object Pascal, внутрь которых параметры-объекты передаются с помощью указателей.

```
void MltVectP(Vector3D*, Vector3D&);       // функция №7, член класса
void pMltVectP(Vector3D*, Vector3D*);      // функция №8, член класса
Vector3D* pMltVect(Vector3D*);             // функция №9, член класса

procedure MltVectP(pB: pVector3D; var C: Vector3D);     {метод №7}
procedure pMltVectP(pB, pC: pVector3D);                 {метод №8}
function pMltVect(pB: pVector3D): pVector3D;            {метод №9}
```

Синтаксис использования метода **MltVectP** (№7) для операции (8.1) представляет собой конструкции:

на C++ **A.MltVectP(&B, C);** (8.11)

где **&B** – указатель на объект **B**;

на Object Pascal **A.MltVectP(@B, C);** (8.12)

где **@B –** указатель на объект **B**.

Синтаксис использования метода **pMultVectP** (№8) для операции (8.1) представляет собой конструкции:

на C++ **A.pMltVectP(&B, &C);** (8.13)

где **&B** и **&C** – указатели на объекты **B** и **C** соответственно;

на Object Pascal **A.pMltVectP(@B, @C);** (8.14)

где **@B и @C–** указатели на объекты **B** и **C** соответственно.

Синтаксис использования метода (функции, члена класса) **pMltVect** (№9) для операции (8.1) на C++ представляет собой конструкцию:

pC= A.pMltVect(&B); (8.15)

где **&B** – указатель на объект **B**, а **pC** – указатель на объект **C** (объявляется как **Vector3D* pC)**;

Синтаксис использования метода **pMltVect** (№9) для операции (8.1) на Object Pascal в версии Turbo Pascal 7.0 представляет собой достаточно сложную конструкцию с использованием методов-акцессоров:

```
pC^.PutX(A.pMltVect(@B)^.GetX);
pC^.PutY(A.pMltVect(@B)^.GetY);
pC^.PutZ(A.pMltVect(@B)^.GetZ);
```
(8.16)

где **@B –** указатель на объект **B**, а **pC** – указатель на объект **C** (объявляется как **pC: ^Vector3D)**.

Из конструкций (8.11)–(8.16) видно, что первый операнд операции (8.1) выступает как объект **A**, к которому посылаются сообщения с помощью методов **MltVectP**, **pMltVectP** или **pMltVect**. Первый аргумент метода есть указатель на второй операнд **B** операции (8.1). Таким образом, аргумент **B** в конструкциях (8.11)–(8.16) передаётся внутрь методов «по указателю».

Далее представлены объявления функций и процедур на языках C++ и Object Pascal. Входные аргументы-объекты передаются внутрь этих функций и процедур «по имени» или с помощью указателей.

```
void MltVct(Vector3D*, Vector3D*, Vector3D&);        // функция №10 на C++
Vector3D* pMltVct(Vector3D*, Vector3D*);             // функция №11 на C++

procedure MltVct(pA, pB: pVector3D; var C: Vector3D);
                                          {процедура №10 на Object Pascal}
function pMltVct(pA, pB: pVector3D): pVector3D;
                                          {функция №11 на Object Pascal}
```

Синтаксис использования функции **MltVct** (№10) для операции (8.1) на C++ представляет собой конструкцию:

MltVct(&A, &B, C); (8.17)

где **&A** и **&B** – указатели на объекты **A** и **B** соответственно.

Синтаксис использования процедуры **MltVct** (№10) при реализации операции (8.1) на Object Pascal выглядит как конструкция

MltVct(@A, @B, C); (8.18)

где **@A** и **@B** – указатели на объекты **A** и **B** соответственно.

Синтаксис использования функции **pMltVct** (№11) для реализации операции (8.1) на C++ представляет собой конструкцию:

pC= pMltVct(&A, &B); (8.19)

где **&A** и **&B** – указатели на объекты **A** и **B** соответственно.

Синтаксис использования функции **pMltVct** (№11) для операции (8.1) на Object Pascal в версии Turbo Pascal 7.0 представляет собой достаточно сложную конструкцию с использованием методов-акцессоров:

```
pC^.PutX(pMltVct(@A, @B)^.GetX);
pC^.PutY(pMltVct(@A, @B)^.GetY);
pC^.PutZ(pMltVct(@A, @B)^.GetZ);
```
(8.20)

где **@A** и **@B –** указатели на объекты **A** и **B**, а **pC** – указатель на объект **C** (объявляется как **pC: ^Vector3D)**.

Возвращение объектов из методов и подпрограмм

Возврат результирующего аргумента-объекта методами класса **Vector3D,** а также возврат результирующего аргумента-объекта из самостоятельных процедур и функций также возможен различными способами, как и рассмотренная выше передача аргументов-объектов внутрь методов, процедур и функций.

Среди рассмотренных примеров методов (функций, членов класса) № 1—3 и № 7—9, а также среди самостоятельных процедур и функций № 4—6 и № 10 и 11, можно увидеть несколько разных способов возврата аргументов-объектов в вызывающую программу из подпрограмм.

Возвращение аргументов-объектов из методов и из самостоятельных процедур и функций с помощью параметрических ссылок и параметров-переменных

Методы **MultVect** (№1) и **MltVectP** (№7) имеют вторые аргументы, тип которых объявлен на C++ как параметрическая ссылка Vector3D&, а на Object Pascal этот тип объявлен как параметр-переменная var VR: Vector3D. Такой тип аргументов позволяет передавать аргументы-объекты из подпрограммы в

вызывающую программу «по ссылке» или «по имени». Таким образом, второму аргументу-объекту в методах-подпрограммах MultVect и MltVectP присваивается результирующее значение операции (8.1), которое при использовании таких синтаксических конструкций становится доступным в вызывающей программе (см. использование метода **MultVect** в (8.2) и метода **MltVectP** в (8.11) и (8.12), а также в примерах 8.2 и 8.3, приведённых далее).

Аналогично методу **MultVect** самостоятельная процедура **MultV** (№4) возвращает в вызывающую программу результат операции (8.1) как параметрическую ссылку типа Vector3D& на C++ и как параметр-переменную типа var C: Vector3D на Object Pascal (см. использование процедуры **MultV** в (8.7), а также в примерах 8.2 и 8.3).

Так же точно, как и **MultV**, самостоятельная процедура **MltVct** (№10) возвращает в вызывающую программу результат операции (8.1) (см. процедуры **MltVct** на C++ в (8.17) и на Object Pascal (8.18), а также в примерах 8.2 и 8.3.

Возвращение аргументов-объектов из методов (из функций, членов класса) и из самостоятельных процедур и функций с помощью указателей на объекты

Другой способ возврата аргументов-объектов в вызывающую программу из подпрограмм – с помощью указателей, как в методе **pMultVectV** (№2) (см. (8.3) и (8.4)) и в методе **pMltVectP** (№8) (см. (8.13) и (8.14)). В этом случае второй аргумент (указатель на объект) в методах **pMultVectV** и **pMltVectP** указывает на результирующее значение операции (8.1). При таком использовании указателей аргументы-объекты становятся доступными в вызывающей программе (см. использование **pMultVectV**и **pMltVectP** в примерах 8.2 и 8.3). На C++ указатель на объект **C,** обозначенный **pC,** объявляется как **Vector3D* pC**, а на Object Pascal указатель на объект **C** может быть взят с помощью конструкции **@C**. При этом переменная типа «указатель на Vector3D» может быть объявлена на Object Pascal как

pC: ^Vector3D.

Либо может быть создан тип-указатель **pVector3D=^Vector3D** (см. пример 8.3).

Аналогично, 3-й аргумент в самостоятельной процедуре **MultVec** (№6) (см. (8.10)) является указателем на объект **C** (результирующее значение операции (8.1)), который на языке C++ объявляется как **pC: pVector3D,** а на Object Pascal как **pC: ^Vector3D.**

Указатели на объекты как возвращаемые значения функций

Указатель на объект может быть также и возвращаемым значением метода. Так, методы (функции, члены класса) **pMultVect** (№3) и **pMltVect** (№9) возвращают указатель на объект типа Vector3D (результат операции (8.1)). Синтаксис применения методов **pMultVect** и **pMltVect** показан в (8.5) и в (8.6), в (8.15) и в (8.16), а также в примерах программ 8.2 и 8.3. Здесь, как и выше, pC – указатель на объект **C** (объявляется на C++ как Vector3D * pC, а конструкция

A.pMultVect(B)^ — определение на языке Object Pascal объекта по указателю, возвращаемому конструкцией A.pMultVect(B)).

Самостоятельная функция **pMultVct** (№5) также возвращает указатель на объект типа Vector3D (результат операции (8.1)). Синтаксис применения функции **pMultVct** показан в конструкции (8.9) и в примерах 8.2 и 8.3.

Как было сказано выше, использование объектов (экземпляров классов) в качестве входных и выходных аргументов функций относится к приёму программирования, который называется композицией (работа №6). Некоторые языки программирования допускают прямое использование объектов как переменных. Так, в C++ при этом нужно дополнительно создавать конструкторы по умолчанию в классах, объекты которых используются в композиции. В примере 8.2 показано использование таких «пустых» конструкторов.

Пример 8.2. Функции, члены класса и самостоятельные функции на C++, имеющие объекты (экземпляры класса Vector3D) как входные параметры и объект (экземпляр класса Vector3D) как возвращаемое значение

```
#include<conio.h>
#include<math.h>
#include<iostream.h>
class Vector3D
{
 private:
  double X,Y,Z;
 public:
      Vector3D(double, double, double);
      Vector3D() {};
  double  GetX();
  double  GetY();
  double  GetZ();
  void   PutX(double);
  void   PutY(double);
  void   PutZ(double);
  void   PrintVector();
  void MultVect(Vector3D, Vector3D&);                    // 1
  void pMultVectV(Vector3D, Vector3D*);                  // 2
  Vector3D* pMultVect(Vector3D);                         // 3
  void MltVectP(Vector3D*, Vector3D&);                   // 7
  void pMltVectP(Vector3D*, Vector3D*);                  // 8
  Vector3D* pMltVect(Vector3D*);                         // 9
  Vector3D MultVector(Vector3D);                         // 12
  Vector3D operator*(Vector3D);                          // 13
```

```
  friend void MultVect(Vector3D, Vector3D, Vector3D&);      // 14
  friend void MultVect2(Vector3D, Vector3D, Vector3D*);     // 15
};
void MultV(Vector3D, Vector3D, Vector3D&);                  //  4
Vector3D* pMultVct(Vector3D, Vector3D);                     //  5
void MultVec(Vector3D, Vector3D, Vector3D*);                //  6
void MltVct(Vector3D*, Vector3D*, Vector3D&);               // 10
Vector3D* pMltVct(Vector3D*, Vector3D*);                    // 11
Vector3D::Vector3D(double X, double Y, double Z): X(X), Y(Y), Z(Z) {}
double Vector3D::GetX()  {return X;}
double Vector3D::GetY()  {return Y;}
double Vector3D::GetZ()  {return Z;}
void Vector3D::PutX(double X)  {this->X=X;}
void Vector3D::PutY(double Y)  {this->Y=Y;}
void Vector3D::PutZ(double Z)  {this->Z=Z;}
void Vector3D::PrintVector() {cout<<"X="<<X<<" Y="<<Y<<" Z="<<Z<<endl;}
void Vector3D::MultVect(Vector3D B, Vector3D& CR)           //  1
 {
  CR.X = Y * B.Z – Z * B.Y;
  CR.Y = Z * B.X – X * B.Z;
  CR.Z = X * B.Y – Y * B.X;
 }

void Vector3D::pMultVectV(Vector3D B, Vector3D* pC)         //  2
 {
  pC->X = Y * B.Z – Z * B.Y;
  pC->Y = Z * B.X – X * B.Z;
  pC->Z = X * B.Y – Y * B.X;
 }

Vector3D* Vector3D::pMultVect(Vector3D B)                   //  3
 {
  Vector3D* pC;
  pC->X = Y * B.Z – Z * B.Y;
  pC->Y = Z * B.X – X * B.Z;
  pC->Z = X * B.Y – Y * B.X;
  return pC;
 }
```

```
void MultV(Vector3D A, Vector3D B, Vector3D &C)                    // 4
 {
 C.PutX( A.GetY() * B.GetZ() - A.GetZ() * B.GetY() );
 C.PutY( A.GetZ() * B.GetX() - A.GetX() * B.GetZ() );
 C.PutZ( A.GetX() * B.GetY() - A.GetY() * B.GetX() );
 }

Vector3D* pMultVct(Vector3D A, Vector3D B)                         // 5
 {
 Vector3D *pC;
 pC->PutX( A.GetY() * B.GetZ() - A.GetZ() * B.GetY());
 pC->PutY( A.GetZ() * B.GetX() - A.GetX() * B.GetZ());
 pC->PutZ( A.GetX() * B.GetY() - A.GetY() * B.GetX());
 return pC;
 }

void MultVec(Vector3D A, Vector3D B, Vector3D *pC)                 // 6
 {
 pC->PutX( A.GetY() * B.GetZ() – A.GetZ() * B.GetY());
 pC->PutY( A.GetZ() * B.GetX() – A.GetX() * B.GetZ());
 pC->PutZ( A.GetX() * B.GetY() – A.GetY() * B.GetX());
 }

void Vector3D::MltVectP(Vector3D* pB, Vector3D &C)                 // 7
 {
 C.PutX( Y * pB->GetZ() – Z * pB->GetY());
 C.PutY( Z * pB->GetX() – X * pB->GetZ());
 C.PutZ( X * pB->GetY() – Y * pB->GetX());
 }

void Vector3D::pMltVectP(Vector3D* pB, Vector3D* pC)               // 8
 {
 pC->PutX( Y * pB->GetZ() – Z * pB->GetY());
 pC->PutY( Z * pB->GetX() – X * pB->GetZ());
 pC->PutZ( X * pB->GetY() – Y * pB->GetX());
 }

Vector3D* Vector3D::pMltVect(Vector3D* pB)                         // 9
 {
 Vector3D* pR;
 pR->PutX( Y * pB->GetZ() – Z * pB->GetY());
 pR->PutY( Z * pB->GetX() – X * pB->GetZ());
 pR->PutZ( X * pB->GetY() – Y * pB->GetX());
 return pR;
 }
```

```
void MltVct(Vector3D* pA, Vector3D* pB, Vector3D& CR)          // 10
 {
  CR.PutX( pA->GetY() * pB->GetZ() – pA->GetZ() * pB->GetY());
  CR.PutY( pA->GetZ() * pB->GetX() – pA->GetX() * pB->GetZ());
  CR.PutZ( pA->GetX() * pB->GetY() – pA->GetY() * pB->GetX());
 }

Vector3D* pMltVct(Vector3D* pA, Vector3D* pB)                  // 11
 {
  Vector3D* pV;
  pV->PutX( pA->GetY() * pB->GetZ() - pA->GetZ() * pB->GetY());
  pV->PutY( pA->GetZ() * pB->GetX() - pA->GetX() * pB->GetZ());
  pV->PutZ( pA->GetX() * pB->GetY() - pA->GetY() * pB->GetX());
  return pV;
 }

Vector3D Vector3D::MultVector(Vector3D B)                      // 12
 {
  Vector3D R;
  R.X = Y * B.Z – Z * B.Y;
  R.Y = Z * B.X – X * B.Z;
  R.Z = X * B.Y – Y * B.X;
  return R;
 }

Vector3D Vector3D::operator* (Vector3D B)                      // 13
 {
  Vector3D R;
  R.X = Y * B.Z – Z * B.Y;
  R.Y = Z * B.X – X * B.Z;
  R.Z = X * B.Y – Y * B.X;
  return R;
 }

void MultVect(Vector3D A, Vector3D B, Vector3D &C)             // 14
 {
  C.X = A.Y * B.Z - A.Z * B.Y;
  C.Y = A.Z * B.X - A.X * B.Z;
  C.Z = A.X * B.Y - A.Y * B.X;
 }

void MultVect2(Vector3D A, Vector3D B, Vector3D *pC)           // 15
 {
  pC->X = A.Y * B.Z - A.Z * B.Y;
  pC->Y = A.Z * B.X - A.X * B.Z;
  pC->Z = A.X * B.Y - A.Y * B.X;
 }
```

```
void main()
{
 clrscr();
 Vector3D A(1,0,0), B(0,2,0),
          C1, C4, C7, C8, C10, C12, C13, C14, C15,
          *pC2, *pC3, *pC5, *pC6, *pC8, *pC9, *pC11;

 cout<<"A: ";
 A. PrintVector();  getch();

 cout<<"B: ";
 B. PrintVector();  getch();

 A.MultVect(B, C1);                                  // 1
 cout<<"C1: A.MultVect(B, C1): ";
 C1. PrintVector();              getch();

 A.pMultVectV(B, pC2);                               // 2
 cout<<"C2: A.pMultVectV(B, pC2): ";
 pC2-> PrintVector();            getch();

 pC3=A.pMultVect(B);                                 // 3
 cout<<"C3: pC3=A.pMultVect(B): ";
 pC3-> PrintVector();          getch();

 MultV(A, B, C4);
 cout<<"7 {4}: C4: MultV(A, B, C4): ";               // 4
 C4. PrintVector();              getch();

 pC5= pMultVct(A, B);                                // 5
 cout<<"C5: Pointer *pC5: pMultVct(A, B): ";
 pC5-> PrintVector();          getch();

 MultVec(A, B, pC6);                                 // 6
 cout<<"C6: MultVec(A, B, *pC6): ";
 pC6-> PrintVector();          getch();

 A.MltVectP(&B, C7);                                 // 7
 cout<<"C7: A.MltVectP(&B, C7): ";
 C7. PrintVector();              getch();

 A.pMltVectP(&B, pC8);                               // 8
 cout<<"C8: A.pMltVectP(&B, pC8): ";
 pC8-> PrintVector();            getch();

 A.pMltVectP(&B, &C8);                               // 8
 cout<<"C8: A.pMltVectP(&B, &C8): ";
 C8. PrintVector();              getch();
```

```
pC9= A.pMltVect(&B);                    //  9
cout<<"pC9: A.pMltVect(&B): ";
pC9-> PrintVector();
getch();

MltVct(&A, &B, C10);                    // 10
cout<<"C10: MltVect(&A, &B, C10): ";
C10. PrintVector();
getch();

pC11= pMltVct(&A, &B);                  // 11
cout<<"C11: A.pMltVect(&B): ";
pC11-> PrintVector();
getch();

C12= A.MultVector(B);                   // 12
cout<<"C12=A.MultVect(B): ";
C12. PrintVector();
getch();

cout<<"C12:  A. MultVector(B). PrintVector(): ";
A.MultVector(B).PrintVector();          // 12
getch();

C13= A*B;                               // 13
cout<<"C13: Overload C= A*B: ";
C13. PrintVector();
getch();

MultVect(A, B, C14);                    // 14
cout<<"C14:  MultVect(A, B, C14): ";
C14. PrintVector();
getch();

MultVect2(A, B, &C15);                  // 15
cout<<"C15: MultVect2(A, B, &C15): ";
C15. PrintVector();
getch();

}
```

```
#include<conio.h>
#include<math.h>
#include<iostream.h>

class Vector3D
{
 private:
  double X,Y,Z;

 public:
      Vector3D(double, double, double);
      Vector3D() {};

  double  GetX();
  double  GetY();
  double  GetZ();
  void   PutX(double);
  void   PutY(double);
  void   PutZ(double);

  void   PrintVector();

  friend Vector3D operator* (Vector3D, Vector3D);  //16
};

Vector3D::Vector3D(double X, double Y, double Z): X(X), Y(Y), Z(Z) {}

double Vector3D::GetX()  {return X;}

double Vector3D::GetY()  {return Y;}

double Vector3D::GetZ()  {return Z;}

void Vector3D::PutX(double X)  {this->X=X;}

void Vector3D::PutY(double Y)  {this->Y=Y;}

void Vector3D::PutZ(double Z)  {this->Z=Z;}

void Vector3D::PrintVector() {cout<<" X="<<X<<" Y="<<Y<<" Z="<<Z<<endl;}

Vector3D operator* (Vector3D A, Vector3D B)    // 16
{
 Vector3D C;
 C.X = A.Y * B.Z - A.Z * B.Y;
 C.Y = A.Z * B.X - A.X * B.Z;
 C.Z = A.X * B.Y - A.Y * B.X;
 return C;
}
```

```
void main()
{
 clrscr();
 Vector3D A(1,0,0), B(0,2,0), C16;

 cout<<"A: "; A. PrintVector(); getch();

 cout<<"B: "; B.PrintVector(); getch();

 C16= A*B;                                          // 16
 cout<<"Overload C16= A* B: ";
 C16. PrintVector();            getch();
}
```

Пример 8.3. Методы класса и самостоятельные процедуры и функции на Turbo Pascal, имеющие объекты (экземпляры класса Vector3D) как входные параметры и объект (экземпляр класса Vector3D) как возвращаемое значение

```
{$N+}
program v3d;
uses crt;
type
 pVector3D=^Vector3D;
 Vector3D=object
  private
    X: double;
    Y: double;
    Z: double;
  public
    constructor Init(InitX, InitY, InitZ: double);
    function GetX: double;
    function GetY: double;
    function GetZ: double;
    procedure PutX(NewX: double);
    procedure PutY(NewY: double);
    procedure PutZ(NewZ: double);
    procedure PrintVector;
    procedure MultVect(B: Vector3D; var C1: Vector3D);       {1}
    procedure pMultVectV(B: Vector3D;  pC2: pVector3D);      {2}
    function pMultVect(B: Vector3D): pVector3D;              {3}
    procedure MltVectP(pB: pVector3D; var C7: Vector3D);     {7}
    procedure pMltVectP(pB, pC8: pVector3D);                 {8}
    function pMltVect(pB: pVector3D): pVector3D;             {9}
end;
```

```
procedure MultV(A, B: Vector3D; var C4: Vector3D);          {4}
 begin
  C4.PutX(A.GetY * B.GetZ - A.GetZ * B.GetY);
  C4.PutY(A.GetZ * B.GetX - A.GetX * B.GetZ);
  C4.PutZ(A.GetX * B.GetY - A.GetY * B.GetX)
 end;

function pMultVct(A, B: Vector3D): pVector3D;               {5}
 var C: Vector3D;
 begin
  C.PutX(A.GetY * B.GetZ - A.GetZ * B.GetY);
  C.PutY(A.GetZ * B.GetX - A.GetX * B.GetZ);
  C.PutZ(A.GetX * B.GetY - A.GetY * B.GetX);
  pMultVct:=@C
 end;

procedure MultVec(A, B: Vector3D; pC6: pVector3D);          {6}
 begin
  pC6^.PutX(A.GetY * B.GetZ - A.GetZ * B.GetY);
  pC6^.PutY(A.GetZ * B.GetX - A.GetX * B.GetZ);
  Pc6^.PutZ(A.GetX * B.GetY - A.GetY * B.GetX)
 end;

procedure MltVct(pA, pB: pVector3D; var C10: Vector3D);    {10}
 begin
  C10.PutX(pA^.GetY * pB^.GetZ - pA^.GetZ * pB^.GetY);
  C10.PutY(pA^.GetZ * pB^.GetX - pA^.GetX * pB^.GetZ);
  C10.PutZ(pA^.GetX * pB^.GetY - pA^.GetY * pB^.GetX)
 end;

function pMltVct(pA, pB: pVector3D): pVector3D;             {11}
 var CR: Vector3D;
 begin
  CR.PutX(pA^.GetY * pB^.GetZ - pA^.GetZ * pB^.GetY);
  CR.PutY(pA^.GetZ * pB^.GetX - pA^.GetX * pB^.GetZ);
  CR.PutZ(pA^.GetX * pB^.GetY - pA^.GetY * pB^.GetX);
  pMltVct:=@CR
 end;

constructor Vector3D.Init(InitX, InitY, InitZ: double);
 begin
  X:=InitX;
  Y:=InitY;
  Z:=InitZ;
end;
```

```
function Vector3D.GetX: double; begin GetX:=X end;
function Vector3D.GetY: double; begin GetY:=Y end;
function Vector3D.GetZ: double; begin GetZ:=Z end;
procedure Vector3D.PutX(NewX: double); begin X:=NewX end;
procedure Vector3D.PutY(NewY: double); begin Y:=NewY end;
procedure Vector3D.PutZ(NewZ: double); begin Z:=NewZ end;
procedure Vector3D.PrintVector; begin write(' X=',X:7:2,' Y=',Y:7:2,' Z=',Z:7:2) end;
procedure Vector3D.MultVect(B: Vector3D; var C1: Vector3D);      {1}
 begin
  C1.PutX(Y * B.GetZ – Z * B.GetY);
  C1.PutY(Z * B.GetX – X * B.GetZ);
  C1.PutZ(X * B.GetY – Y * B.GetX)
 end;
procedure Vector3D.pMultVectV(B: Vector3D; pC2: pVector3D);   {2}
 begin
  pC2^.PutX(Y * B.GetZ – Z * B.GetY);
  pC2^.PutY(Z * B.GetX – X * B.GetZ);
  pC2^.PutZ(X * B.GetY – Y * B.GetX)
 end;
function Vector3D.pMultVect(B: Vector3D): pVector3D;              {3}
 var CR: Vector3D;
 begin
  CR.PutX(Y * B.GetZ – Z * B.GetY);
  CR.PutY(Z * B.GetX – X * B.GetZ);
  CR.PutZ(X * B.GetY – Y * B.GetX);
  pMultVect:=@CR
 end;
procedure Vector3D.MltVectP(pB: pVector3D; var C7: Vector3D); {7}
 begin
  C7.PutX(Y * pB^.GetZ – Z * pB^.GetY);
  C7.PutY(Z * pB^.GetX – X * pB^.GetZ);
  C7.PutZ(X * pB^.GetY – Y * pB^.GetX)
 end;
procedure Vector3D.pMltVectP(pB, pC8: pVector3D);                 {8}
 begin
  pC8^.PutX(Y * pB^.GetZ – Z * pB^.GetY);
  pC8^.PutY(Z * pB^.GetX – X * pB^.GetZ);
  pC8^.PutZ(X * pB^.GetY – Y * pB^.GetX)
 end;
```

```
function Vector3D.pMltVect(pB: pVector3D): pVector3D;   {9}
  var CR: Vector3D;
  begin
    CR.PutX(Y * pB^.GetZ – Z * pB^.GetY);
    CR.PutY(Z * pB^.GetX – X * pB^.GetZ);
    CR.PutZ(X * pB^.GetY – Y * pB^.GetX);
    pMltVect:=@CR;
  end;

var A, B, C1, C2, C4, C7, C8, C10: Vector3D;
      pC3, pC5, pC6, pC9, pC11: pVector3D;

begin
   clrscr;
   A.Init(1, 2, 3);
   B.Init(3, 2, 1);
   write('A: '); A.PrintVector; readln;
   write('B: '); B.PrintVector; readln;

  A.MultVect(B, C1);                                {1}
  write('C1: MultVect(B, C1): ');
  C1.PrintVector; readln;

  A.pMultVectV(B, @C2);                             {2}
  write('C2: pMultVectV(B, @C2):  ');
  C2.PrintVector; readln;

  write('C3: A.pMultVect(B)^.PrintVector: ');
  A.pMultVect(B)^.PrintVector; readln;              {3}

  pC3:=A.pMultVect(B);                              {3}
  write('C3: pC3:=A.pMultVect(B): ');
  pC3^.PrintVector; readln;

  MultV(A, B, C4);                                  {4}
  write('C4: MultVct(A, B, C4):  ');
  C4.PrintVector; readln;

  pC5:=pMultVct(A, B);                              {5}
  write('C5: pC5:=pMultVct(A,B): ');
  pC5^.PrintVector; readln;

  MultVec(A, B, pC6);                               {6}
  write('C6: pC6^: MultVec(A, B, pC6): ');
  pC6^.PrintVector; readln;
```

```
  A.MltVectP(@B, C7);                                {7}
  write('C7: MltVectP(@B, C7):  ');
  C7.PrintVector;
  readln;

  A.pMltVectP(@B, @C8);                              {8}
  write('C8: pMltVectP(@B, @C8): ');
  C8.PrintVector;
  readln;

  pC9^.PutX(A.pMltVect(@B)^.GetX);                   {9}
  pC9^.PutY(A.pMltVect(@B)^.GetY);
  pC9^.PutZ(A.pMltVect(@B)^.GetZ);
  write('C9: pC9^: A.pMltVect(@B): ');
  pC9^.PrintVector;
  readln;

  write('C9: A.pMltVect(@B)^.GetXYZ: ',              {9}
            A.pMltVect(@B)^.GetX:7:2,' ',
            A.pMltVect(@B)^.GetY:7:2,' ',
            A.pMltVect(@B)^.GetZ:7:2);
  readln;

  MltVct(@A, @B, C10);                               {10}
  write('C10: MltVctP(@A, @B, C10): ');
  C10.PrintVector;
  readln;

  write('C11: pMltVct(@A, @B)^.GetXYZ=A*B=', {11}
              pMltVct(@A, @B)^.GetX:7:2,' ',
              pMltVct(@A, @B)^.GetY:7:2,' ',
              pMltVct(@A, @B)^.GetZ:7:2);
  readln;

  pC11^.PutX(pMltVct(@A, @B)^.GetX);                 {11}
  pC11^.PutY(pMltVct(@A, @B)^.GetY);
  pC11^.PutZ(pMltVct(@A, @B)^.GetZ);
  write('11: pC11^: pMltVct(@A, @B)^: ');
  pC11^.PrintVector;
  readln;

end.
```

Программы из примеров 8.2 и 8.3 приведены одновременно на двух языках программирования, чтобы показать общность приёмов передачи аргументов-объектов *внутрь* методов и подпрограмм, а также чтобы показать общность способов возврата объектов *из* методов и подпрограмм. Для этого как на C++, так и на Object Pascal методы классов, а также самостоятельные подпрограммы, реализующие одни и те же (либо схожие) способы передачи объектов, одинаково пронумерованы на обоих языках. Это рассмотренные выше передача «по значению», передача «по имени» или «по ссылке» и передача с помощью указателя на объект, а также возврат «по имени» или «по ссылке» и возврат с помощью указателя на параметры-объекты. Особый способ возврата – указатель на объект как возвращаемое значение функции.

Комбинирование разных способов передачи и возврата объектов привело к созданию одиннадцати вариантов методов, функций и процедур.

В теле всех этих подпрограмм, общее количество которых равно 11, обращение к координатам входных аргументов-векторов возможно как напрямую через поля данных объектов-векторов (см. функции, члены класса №1–3 на C++), так и с помощью методов-акцессоров (см. все методы на Object Pascal и самостоятельные подпрограммы на обоих языках). Использование каждого из этих методов и подпрограмм имеет свои особенности:

В метод №1 входной аргумент-объект передаётся «по значению», а выходной аргумент-объект возвращается в вызывающую программу «по ссылке» (C++) или как параметр-переменная (Object Pascal).

В метод №2 входной аргумент-объект передаётся «по значению», а выходной аргумент-объект возвращается в вызывающую программу с помощью указателя на объект.

В метод №3 входной аргумент-объект также передаётся «по значению», а указатель на выходной аргумент-объект является возвращаемым значением соответствующей функции, члена класса (метода pMultVect).

В процедуру №4 входные аргументы-объекты передаются «по значению», а выходной аргумент-объект возвращается в вызывающую программу «по ссылке» (C++) или как параметр-переменная (Object Pascal).

В функцию №5 входные аргументы-объекты передаются «по значению», а указатель на выходной аргумент-объект является возвращаемым значением этой функции.

В процедуру №6 входные аргументы-объекты также передаются «по значению», а выходной аргумент-объект возвращается в вызывающую программу с помощью указателя на объект.

В метод №7 передаётся указатель на входной аргумент-объект, а выходной аргумент-объект возвращается в вызывающую программу «по ссылке» (C++) или как параметр-переменная (Object Pascal).

В метод №8 передаётся указатель на входной аргумент-объект, а выходной аргумент-объект возвращается в вызывающую программу с помощью указателя на объект.

В метод №9 также передаётся указатель на входной аргумент-объект, а указатель на выходной аргумент-объект является возвращаемым значение соответствующей функции, члена класса (метода pMltVect).

В процедуру №10 передаются указатели на входные аргументы-объекты, а выходной аргумент-объект возвращается в вызывающую программу «по ссылке» (C++) или как параметр-переменная (Object Pascal).

В функцию №11 передаются указатели на входные аргументы-объекты, а указатель на выходной аргумент-объект является возвращаемым значением этой функции.

В зависимости от программной ситуации и от языка программирования, использование методов-акцессоров порой следует осуществлять и в вызывающей программе, совмещая их с вызовами методов и подпрограмм, осуществляющих векторное произведение (см. pMltVect №9 и pMltVct №11).

Важной особенностью представленных программ на языке C++ являются конструкторы «по умолчанию» («пустые» конструкторы), обязательные при использовании композиции (см. лабораторную работу № 6).

Дополнительные возможности языка программирования C++ для использования объектов как входных и выходных аргументов функций (подпрограмм)

Однако индивидуальные особенности каждого языка программирования могут давать программистам дополнительные возможности для вышеназванных передач аргументов-объектов внутрь подпрограмм и обратно. Поэтому в примере 8.2 на языке C++ приведены дополнительно ещё пять способов реализации векторного умножения объектов (экземпляров класса Vector3D):

```
Vector3D MultVector (Vector3D);                          // 12
Vector3D  operator* (Vector3D);                          // 13
friend void MultVect (Vector3D, Vector3D, Vector3D&);    // 14
friend void MultVect2(Vector3D, Vector3D, Vector3D*);    // 15
friend Vector3D operator*(Vector3D, Vector3D);           //16
```

Так, замечательной особенностью языка C++ является возможность иметь в качестве возвращаемого значения функции объект (а не только указатель на объект, как в примере методов **№3, №9** и в примере функции **№5**). Метод **№12 MultVector** имеет тип Vector3D и как тип входного, и как тип возвращаемого значения. В результате операция (8.1) реализуется с помощью простой синтаксической конструкции:

```
C = A.MultVector(B);                                     (8.21)
```

Справедливо заметить, что входной аргумент-объект метода №12 может быть передан внутрь метода и иначе, с помощью указателя. Тем самым вместо одного метода №12 можно создать несколько разных методов. Но в этом случае

будут утрачены столь замечательные простота и единообразие в описании типов входного и выходного аргументов-объектов. По этой причине, а также из-за большого количества приведённых выше примеров типов входных аргументов методов, автор ограничился лишь этим методом.

Ещё одну возможность для работы с аргументами-объектами в языке C++ предоставляет **перегрузка операций**, состоящая в том, что символам арифметических (и ряда других) операций может быть придан новый смысл [4, 6].

В общем виде для перегрузки некоей бинарной операции, условно обозначенной как #, другой смысл может быть придан с помощью конструкции

<тип возвращаемого значения > **operator#** (<тип входного аргумента>) (8.22)

объявленной как метод соответствующего класса. Определив этот метод аналогично тому, как показано в методе №13 (**operator***), возможно использовать перегруженную операцию, входными и выходными операндами которой будут объекты соответствующего класса. Таким образом, операция (8.1) принимает вид, соответствующий математической форме:

C = A * B; (8.23)

Дополнительные возможности для работы с аргументами-объектами в языке C++ предоставляют так называемые *дружественные* (**friend**) *функции.* Дружественные функции в языке C++ не являются функциями, членами класса, но объявляются в классе, к которому дружественны, с модификатором friend. В этом случае в теле этих дружественных функций у объектов этого класса становятся доступными закрытые поля данных и методы. Примеры дружественных функций – это функции №14, №15 и №16, дружественные к классу Vector3D.

В дружественную (к классу Vector3D) функцию №14 (**MultVect**) параметры-объекты передаются «по значению» как первые два аргумента, а результат возвращается в вызывающую программу «по ссылке» посредством третьего аргумента этой функции. Тогда операция (8.1) реализуется с помощью такой синтаксической конструкции:

MultVect(A, B, C); (8.24)

В то же время в дружественную функцию №15 (**MultVect2**) параметры-объекты передаются «по значению» как первые два аргумента, аналогично функции MultVect. Результат же возвращается в вызывающую программу с помощью указателя, передаваемого через третий аргумент этой функции. В этом случае операция (8.1) реализуется следующим образом:

MultVect2(A, B, &C); (8.25)

где &C – указатель на объект C.

С помощью дружественной функции можно перегрузить бинарную операцию:

friend <возвращаемый тип> **operator#**(<список типов входных аргументов>) (8.26)

Например, дружественная функция №16 перегружает операцию умножения.

При использовании функции №16 операция (8.1) принимает вид, соответствующий математической форме:

$$C = A * B; \qquad (8.27)$$

Это полностью соответствует (8.23). Имя метода №13 **operator*** совпадает с именем дружественной функции №16 **operator***. Поэтому в одной программе нельзя перегрузить операцию методом №13 и дружественной функцией №16. Чтобы преодолеть это противоречие, дружественная функция **operator*** перегружается в отдельной программе (см. пример 8.2 на C++).

Изученные выше методы, процедуры и функции №1—16 демонстрируют различные возможности для использования объектов в качестве как входных, так и выходных аргументов. Хотя параметры-ссылки (C++), а также параметры-переменные и параметры-константы (Object Pascal) не использованы для передачи параметров-объектов внутрь подпрограмм (а только лишь для возврата параметров-объектов из подпрограмм), большинство приёмов использования объектов как входных, так и выходных аргументов подпрограмм рассмотрено в настоящей работе. Многие из этих приёмов являются общими для разных языков программирования, реализующих объектно-ориентированную парадигму, а значит, могут широко применяться на практике.

Следует заметить, что для решения рассматриваемых в настоящей лабораторной работе задач теоретически возможно использование глобальных переменных. Применение глобальных переменных для передачи объектов в подпрограммы и обратно возможно на многих алгоритмических языках, в том числе на C++ и на Object Pascal. Но автор намеренно не привёл примеров с глобальными переменными-объектами, так как глобальные переменные разрушают модульность программ и противоречат принципам объектно-ориентированного программирования (например, противоречат инкапсуляции).

Задание для лабораторной работы № 8

В соответствии с вариантами задания, создать предложенные классы и разработать методы (функции, члены класса), а также самостоятельные процедуры и функции, реализующие заданные операции с объектами заданных классов. Разрабатываемые подпрограммы (методы, процедуры и функции) должны использовать объекты этих классов в качестве входных и выходных аргументов. Постараться показать все возможные способы, все доступные (в каждом конкретном языке программирования) синтаксические конструкции (кроме глобальных переменных) для использования объектов как входных и возвращаемых аргументов (входных и возвращаемых значений) вышеназванных методов, функций и процедур.

1. Класс Complex – комплексное число в арифметической форме. Операция — умножение двух комплексных чисел.

2. Класс Complex – комплексное число в арифметической форме. Операция — деление двух комплексных чисел.

3. Класс Vector – вектор заданной размерности. Операции — сложение и вычитание двух векторов одинаковой размерности.

4. Класс Matrix – квадратная матрица. Операции — сложение и вычитание двух матриц одинаковой размерности.

5. Классы Vector3D (вектор размерности 3) и Matrix33 (квадратная матрица размерности 3**x**3). Операция умножения матрицы на вектор.

6. Классы Matrix23 (матрица размерности 2**x**3) и Matrix33 (квадратная матрица размерности 3**x**3). Операция умножения матрицы 2**x**3 на матрицу 3**x**3.

7. Класс Date – дата (поля данных – целочисленные год, месяц, число). Операция – разность между двумя датами в виде структуры типа Date, представляющая собой целую разность в годах, в месяцах (за вычетом лет) и в днях (за вычетом лет и месяцев).

8. Класс Time – время (целые поля данных — час, минута, секунда). Операция – разность между двумя моментами времени в виде структуры типа Time, представляющая собой целую разность в часах, минутах (за вычетом часов) и в секундах (за вычетом часов и минут).

9. Класс ComplexTrig – комплексное число в тригонометрической форме. Операции – умножение и деление двух комплексных чисел.

10. Класс Matrix – квадратная матрица. Операция — умножение двух матриц одинаковой размерности.

11. Классы Vector3D (вектор размерности 3) и Matrix3D (квадратная матрица размерности 3**x**3). Операция – умножение вектора-столбца типа Vector3D на вектор-строку типа Vector3D, результат которой — матрица типа Matrix3D.

12. Классы Matrix23 (матрица размерности 2**x**3), Matrix32 (матрица размерности 3**x**2) и Matrix22 (квадратная матрица размерности 2**x**2). Операция умножения матриц 2**x**3 на 3**x**2, результат которой — матрица размерности 2**x**2.

13. Классы Vector3D (вектор размерности 3), Matrix33 (квадратная матрица размерности 3**x**3) и Matrix333 (кубическая матрица размерности 3**x**3**x**3). Операция прямого (декартового) поэлементного умножения вектора Vector3D на матрицу Matrix33, результатом которой является матрица типа Matrix333.

14. Класс Matrix22 – квадратная матрица размерности 2**x**2. Операция – деление двух матриц, результат которого – матрица типа Matrix22.

Методические указания

При подготовке к выполнению задания следует изучить возможности использования объектов как входных и выходных аргументов или как возвращаемых значений методов, процедур и функций, изложенные, например, в [4—6, 10 и 15].

Библиографический список

1. *Бадд Т.* Объектно-ориентированное программирование в действии. – СПб.: Питер, 1997. — 464 с.

2. *Головачев А.Г., Максимов В.В.* Практикум по объектно-ориентированному программированию на ПЭВМ. — М.: Изд-во МАИ, 1994. — 47 с.

3. *Намиот Д.Е.* Язык программирования Turbo C++. — М.: Изд-во МГУ, 1991. – 121 с.

4. *Березин Б.И., Березин С.Б.* Начальный курс С и С++. — М.: ДИАЛОГ-МИФИ, 2005. — 288 с.

5. *Епанешников А.М., Епанешников В.А.* Программирование в среде Turbo Pascal 7.0. — М.: ДИАЛОГ-МИФИ, 1996. — 288 с.

6. *Шилдт Г.* Самоучитель С++: Пер. с англ. — 3-е изд. — СПб.: БХВ-Петербург, 2001. — 688 с.

7. *Культин Н.Б.* Delphi 6. Программирование на Object Pascal. — СПб.: БХВ-Петербург, 2002. — 528 с.

8. *Шамис В.А.* Borland C++Builder. Программирование на С++ без проблем. — М.: Нолидж, 1997. — 266 с.

9. *Фэйсон Т.* Объектно-ориентированное программирование на Borland C++ 4.5. – Киев: Диалектика, 1996. – 544 с.

10. *Павловская Т.А.* C/C++ Программирование на языке высокого уровня. – СПб.: Питер, 2001. – 464 с..

11. *Павловская Т.А., Щупак Ю.А.* C/C++ Структурное программирование. Практикум. – СПб.: Питер, 2007. – 239 с.

12. *Кетков Ю.Л., Кетков А.Ю.* Практика программирования: Visual Basic, C++Builder, Delphi. – СПб.: БХВ-Петербург, 2002. – 464 с.

13. *Марченко А.И., Марченко Л.А.* Программирование в среде Turbo Pascal 7.0. – СПб.: Корона, 2007. – 464 с.

14. *Фаронов В.В.* Delphi 2005. Язык, среда, разработка приложений. – СПб.: Питер, 2007. – 560 с.

15. *Фаронов В.В.* Turbo Pascal 7.0. Учебный курс. – М.: КноРус, 2011. – 727 с.

16. *Новиков П.В.* Увеличение объёма используемой оперативной памяти компьютера при наследовании классов в объектно-ориентированном программировании // Современная наука: актуальные проблемы теории и практики. Серия «Естественные и технические науки». 2018. №6, июнь. Раздел «Информатика, вычислительная техника и управление», С. 116-122.

17. *Новиков П.В.* Объектно-ориентированное программирование. – М.: Изд-во МАИ, 2007. — 49 с.

СОДЕРЖАНИЕ

Тем. план 2019, ч. 2, поз. 4

Новиков Павел Владимирович

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Редактор *Е.Л. Мочина*

Компьютернавя вёрстка *П.В. Новикова*

Подписано в печать 16.10.2019.
Бумага писчая. Формат 60x84 1/16. Печать офсетная.
Усл. печ. л. 7,21. Уч.-изд. л. 7,75. Тираж 100 экз.
Заказ 1039/733.

Издательство МАИ
(МАИ), Волоколамское ш., д. 4,
Москва, А-80, ГСП-3 125993

Отпечатано с готового оригинал-макета
Типография Издательства МАИ
(МАИ), Волоколамское ш., д. 4,
Москва, А-80, ГСП-3 125993