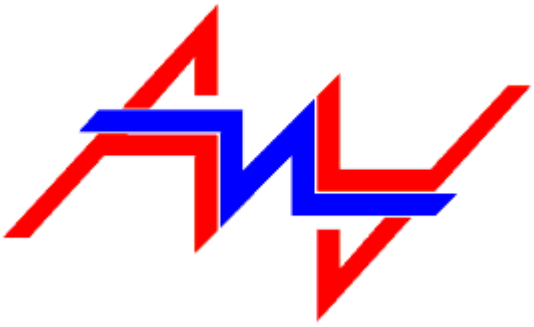
Министерство образования и науки Российской Федерации

Государственное образовательное учреждение
КАЗАНСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им. А.Н.ТУПОЛЕВА



А.Г. Аузяк, Ю.А. Богомолов, А.И. Маликов,
Б.А. Старостин

ЛАБОРАТОРНЫЙ ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ

Казань 2013

УДК 681.3

Лабораторный практикум по программированию. Для инженерных специальностей технических университетов и вузов. /А.Г. Аузяк, Ю.А. Богомолов, А.И. Маликов, Б.А. Старостин. Казань: Изд-во Казан. гос. техн. ун-та, 2013. 132 с.
ISBN

Лабораторный практикум включает 7 лабораторных работ по дисциплине “Программирование и основы алгоритмизации”, охватывающих вопросы программирования типовых алгоритмов обработки информации, реализации вычислительных алгоритмов решения инженерных задач, технологии структурного, модульного и объектно-ориентированного программирования в среде Borland C++ Builder. Он способствует углублению знаний и выработке навыков по программированию и использованию ПЭВМ для обработки информации и решения инженерных задач. Предназначен для студентов, обучающихся по направлению 220400 «Управление в технических системах» различных форм обучения.

Табл.: 3. Ил.: 19. Библиогр.: 10 назв.

Рецензенты:
докт. ф.-м. наук, проф. Павликов С.В.;
докт. ф.-м. наук, проф. Пакшин П.В.

ISBN

ВВЕДЕНИЕ

Практикум предназначен для студентов инженерных специальностей университета. Цель практикума – выработка навыков по программированию типовых алгоритмов и реализации численных методов решения инженерных задач на ПЭВМ с использованием интегрированной среды Borland C++ Builder.

Практикум охватывает вопросы, предусмотренные программой дисциплины “Программирование и основы алгоритмизации”. В практикуме даны лабораторные работы по применению структурного и объектно-ориентированного программирования. Рассмотрены типовые операции над массивами и записями (1-2), итерационные алгоритмы (3), алгоритмы сортировки массивов (4), численные методы решения систем дифференциальных уравнений (5), обработка динамических структур данных (6, 7). В каждой лабораторной работе приводится пример разработки программы, даются рекомендации по выбору визуальных объектов и их использованию для организации интерфейса пользователя, ввода и вывода данных, управления работой программы.

В конце каждой лабораторной работы приведены контрольные вопросы, которые позволяют проверить знания или расширить и уточнить отдельные важные понятия.

При выполнение лабораторной работы студенту может быть предложено индивидуальное задание из числа имеющихся в практикуме или по усмотрению преподавателя. Завершается лабораторная работа оформлением отчета и проверкой уровня усвоения материала по разработке и программированию алгоритмов, по использованию объектов C++ Builder, по элементам программы и операторам алгоритмического языка. По каждой работе в практикуме содержатся сведения, необходимые для ее выполнения.

ЛАБОРАТОРНАЯ РАБОТА № 1. ФУНКЦИИ

Цель занятия - ознакомление и приобретение навыков использования функций для реализации алгоритмов на языке C++.

1. Общие сведения

В процессе решения задачи на разных этапах может потребоваться выполнение одних и тех же вычислений с разными данными. Программу решения такой задачи целесообразно составлять так, чтобы операторы повторяемых вычислений записывались один раз, а результат можно было получить, обращаясь к выполнению этих операторов из разных мест программы. Части вычислительного процесса, повторяющиеся при решении данной задачи, можно реализовать в виде подпрограмм или функций.

Для вычисления часто встречающихся математических функций в библиотеке C++ имеется набор стандартных функций, с которыми можно познакомиться с помощью справочной системы Borland C++ Builder.

Кроме стандартных функций в программе можно использовать заранее определенные функции пользователя. При обращении из программы к функциям, которые содержатся в других файлах или библиотеках, в программе необходимо поместить описания этих функций (прототипы функций). В таком описании достаточно указать заголовок функции. Для облегчения работы такие описания формируются в виде отдельных файлов, которые называют *заголовочными файлами*.

2. Описания функций

Описание функции содержит заголовок, за которым следует программный блок (*тело функции*). В заголовке функции определяется тип результата функции, идентификатор функции и формальные параметры (если они имеются). Формальные параметры заключаются в круглые скобки.

Функция активизируется при ее вызове. При вызове функции указывается идентификатор функции и фактические параметры. Функция может возвращать или не возвращать значение в вызвавшую ее программу. Если функция возвращает значение, то его тип указывается в описании функции непосредственно пред ее именем, а возвращаемое значение указывается после оператора *return*. Такие функции могут вызываться внутри операторов (например, в операторе присвоения "=") или как аргументы других функций. Если функция не возвращает значения, то в качестве возвращаемого типа указывается *void*. Вызывается такая функция как обычный оператор.

Вызов функции может включаться в выражения в качестве операнда. Когда выражение вычисляется, функция выполняется и значением операнда становится значение, возвращаемое функцией.

В операторной части блока функции задаются операторы, которые должны выполняться при активизации функции. Программный блок выделяется фигурными скобками "{" и "}".

Если идентификатор функции используется для вызова функции внутри модуля-функции, то функция выполняется рекурсивно.

Приведем пример описания функции:

```
double Max(double a[], int n)
/* функция вычисления максимального элемента вектора a
   входные параметры:
   a - массив размера n для размещения исходного вектора
   n - переменная целого типа - размерность вектора
   возвращает значение максимального элемента вектора a */
{
 double x;
 x=a[0];
 for(int i=1 i<n; i++)
     if(x<a[i]) x=a[i];
 return x;
}
```

Пример подпрограммы умножения матрицы *a* на вектор *b*.

```
void umnmv(int n, double a[100][100], double b[], double c[])
/* входные параметры:
   n - переменная целого типа - размерность
   a - массив размера n*n для размещения исходной матрицы
   b - массив размера n для размещения исходного вектора
   выходные параметры:
   c - массив размера n для размещения результата */
{
 double s;
 for(int i=0; i<n; i++) {
    s=0.0;
    for(int j=0; j<n; j++) s=s+a[i][j]*b[j];
    c[i]=s;
 }
}
```

3. Параметры

Для передачи входных данных из вызывающей программы в функцию и выходных данных из функции в вызывающую программу используются формальные и фактические параметры.

В описании функции задается список формальных параметров. Каждый параметр, указанный в списке формальных параметров, является локальным по отношению к данной функции и в программном блоке функции на него можно ссылаться по его идентификатору (имени).

Существует два способа передачи параметров внутрь функции: *по значению* и *по ссылке*. При передаче параметра по значению формальному параметру функции присваивается копия значения фактического параметра. При этом все изменения формального параметра внутри функции никак не отражаются на фактическом параметре.

При передаче параметра по ссылке в качестве формального параметра передается адрес фактического параметра. Внутри функции этот адрес открывает доступ к фактическому параметру. Это значит, что все изменения формального параметра будут отражаться на значении фактического параметра.

3.1. Передача параметров по значению

Формальный параметр-значение обрабатывается, как локальная по отношению функции переменная, за исключением того, что он получает свое начальное значение из соответствующего фактического параметра при активизации функции. Изменения, которые претерпевает формальный параметр-значение, не влияют на значение фактического параметра.

Фактический параметр должен иметь тип, совместимый по присваиванию с типом формального параметра-значения.

По умолчанию в C++ используется передача по значению. В этом случае операторы, образующие тело функции, не могут изменять фактические параметры, указанные при ее вызове.

Пример:

```
#include <stdio.h>
int sqr(int x);
void main(void)
{
 int t=10;
 printf("%d %d", sqr(t), t);
}
int sqr(int x)
{
 x=x*x;
 return x;
}
```

В этом примере после выполнения присваивания `x=x*x` изменится только значение переменной *x* внутри функции *sqr*. Значение переменной *t*, указанной при вызове функции *sqr*(*t*), по-прежнему останется равным 10.

3.2. . Передача параметров по ссылке

Для передачи параметра по ссылке необходимо передать внутрь функции указатель на этот параметр. В этом случае изменение значения формального параметра внутри функции приведет к изменению значения фактического параметра в вызывающей программе.

В этом случае параметры должны быть объявлены как указатели либо как переменные типа ссылка.

Пример:

```
void swap(int *x, int *y)
{
 int temp;
 temp=*x;   /* сохраняем значение, записанное по адресу x */
 *x=*y;     /* записываем значение, записанное  по адресу y,
               в ячейку по адресу x */
 *y=temp;   /* записываем содержимое temp по адресу y */
}
```

Функция *swap* может менять местами значения двух переменных, на которые ссылаются указатели *x* и *y*. К содержимому этих переменных можно обращаться, используя обычные операции над указателями **x* и **y*.

При вызове функции *swap* в качестве фактических параметров должны быть переданы указатели на переменные.

Пример:

```
void swap(int *x, int *y);
void main(void)
{
 int a=1, b=2;
 swap(&a, &b);      /* передаются адреса переменных a и b */
}
```

В данном примере используется оператор получения адреса &.

В C++ существует еще один способ передачи параметров по ссылке. Для этого используется специальный тип данных - *ссылка*. Тип ссылка определяется с помощью значка &, поставленного перед именем переменной. Предыдущий пример при использовании типа ссылка приобретет следующий вид:

```
void swap(int &x, int &y);
void main(void)
{
 int a=1, b=2;
 swap(a, b);        /* передаются адреса переменных a и b */
}
void swap(int &x, int &y)
{
 int temp;
 temp=x;
 x=y;
 y=temp;
}
```

4. Пример разработки в Borland C++ Builder программы для выполнения матричных операций

4.1. Постановка задачи

Пусть заданы числовые матрицы $A=\{a_{ij}\}_{i,j=1}^{n}$, $B=\{b_{ij}\}_{i,j=1}^{n}$ размерности $n\times n$.

Требуется составить программу для вычисления матрицы $C=\{c_{ij}\}_{i,j=1}^{n}$ по выражению $C=A*B$.

4.2. Представление данных в программе

Для хранения размерности массивов в программе следует определить переменную n целого типа. Для представления и хранения исходных числовых матриц

A, B и матрицы результата умножения *C* в программе определяются двумерные массивы, соответственно, *a, b* и *c*. Для отображения в интерфейсном окне программы (окне формы) данных (размерности, исходных матриц и матрицы результата) следует использовать стандартные объекты библиотеки визуальных компонент C++ Builder: элемент редактирования *Edit1* – для размерности; объекты (таблицы) *StringGrid1 – StringGrid3* – для матриц *A, B, C*, соответственно. Для отображения в окне формы пояснений к выбранным визуальным компонентам следует использовать метки *Label*. Текст пояснений вводится в поле *Caption* объекта *Label* инспектора объектов (вкладка *Properties*).

4.3. Описание алгоритма

- Задание размерности *n* матриц *A, B, C*;
- Ввод исходных матриц *A, B*.
- Определение матрицы *C*, элементы которой в соотвествии с правилами линейной алгебры для операции умножения матриц вычисляются по формуле

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj},\ i, j = 1,\ldots,n.$$

- Вывод матрицы *C* в виде таблицы.

4.4. Реализация алгоритма в C++ Builder

Согласно условиям задачи требуется ввести рамерность матриц *n*, две исходные числовые матрицы *A* и *B*. Для задания размерности *n* используем объект *Edit1*, в поле *Text* которого в инспекторе объектов введем значение 2. Для задания исходных матриц *A* и *B* и вывода матрицы результата *C* в программе будем использовать объекты *StringGrid1–StringGrid3*, соответственно. Для обеспечения возможности ввода данных в таблицы *StringGrid1* и *StringGrid2* в инспекторе объектов установим значение *true* для свойства *goEditig* раздела *Options*. Для управления работой программы выберем в качестве управляющих элементов кнопки *Button1–Button4*, с которыми свяжем соответствующие шаги алгоритма. В резуль-

тате окно формы с выбранными объектами может быть представлено как на рис. 1.1.

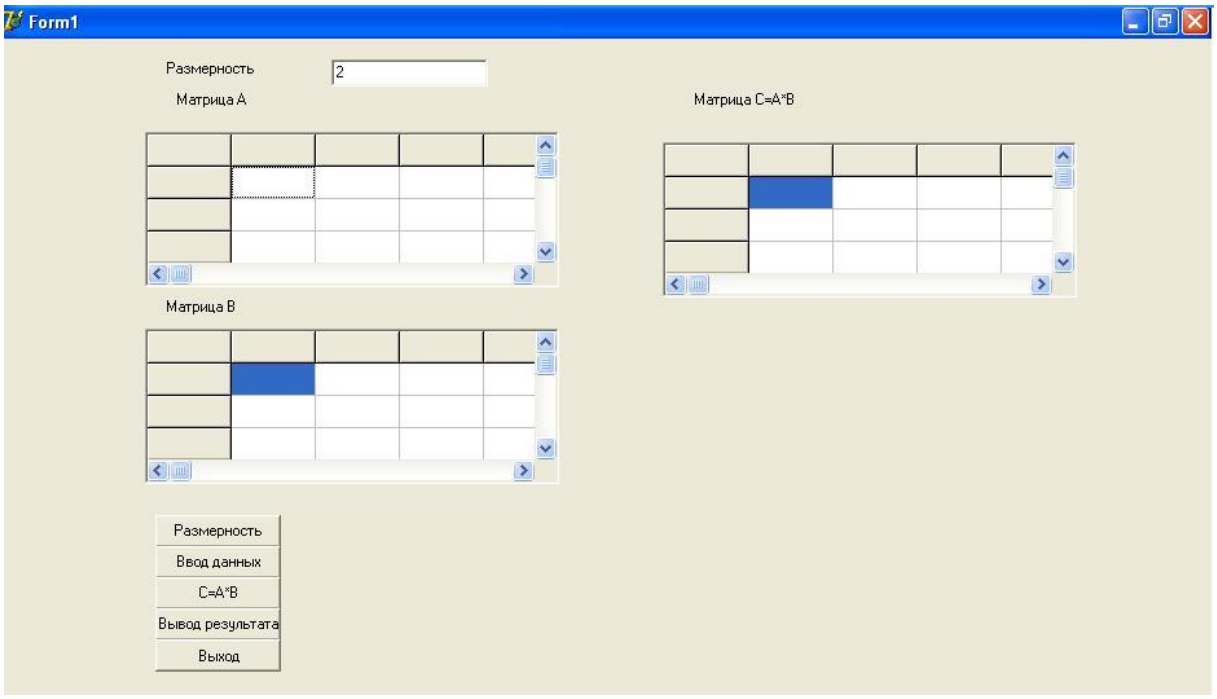


Рис.1.1. Окно формы программы с выбранными компонентами

Отметим, что в форме, представленной на рис. 1.1, размещена дополнительно кнопка *Button5* для закрытия программы.

С помощью инспектора объектов вкладки *Events* с событием *Click* для каждой кнопки свяжем определенное действие, которое должна выполнить программа в ответ на нажатие конкретной кнопки после запуска программы на выполнение: *Button1* – запрос и ввод размерности матриц; *Button2* – запрос и ввод исходных матриц (перед этим значения элементов исходных матриц должны быть введены в таблицы *StringGrid1* и *StringGrid2*; *Button3* – вычисление произведения матриц и размещения результата в массиве *C*; *Button4* – вывод матрицы результата из массива *C* в таблицу *StringGrid3* для отображения в окне формы; *Button5* – закрытие окна формы и выход из программы. В результате выбора, размещения на форме указанных объектов и связывания события *Click* с каждой кнопкой в C++ Builder будет автоматически сгенерирован следующий код программы.

Файл *Main.h*:

```
#ifndef MainH
```

```
#define MainH

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Grids.hpp>

class TForm1 : public TForm
{
__published:   // IDE-managed Components
        TStringGrid *StringGrid1;
        TStringGrid *StringGrid2;
        TStringGrid *StringGrid3;
        TButton *Button1;
        TButton *Button2;
        TButton *Button3;
        TButton *Button4;
        TButton *Button5;
        TEdit *Edit1;
        TLabel *Label1;
        TLabel *Label2;
        TLabel *Label3;
        TLabel *Label4;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall Button3Click(TObject *Sender);
        void __fastcall Button4Click(TObject *Sender);
        void __fastcall Button5Click(TObject *Sender);
private:  // User declarations
public:        // User declarations
        __fastcall TForm1(TComponent* Owner);
};
```

```
extern PACKAGE TForm1 *Form1;
#endif
```

Файл *Main.cpp*:

```
#include <vcl.h>
#pragma hdrstop

#include "Main.h"

#pragma resource "*.dfm"
TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{
}

void __fastcall TForm1::Button2Click(TObject *Sender)
{
}

void __fastcall TForm1::Button3Click(TObject *Sender)
{
}

void __fastcall TForm1::Button4Click(TObject *Sender)
{
}

void __fastcall TForm1::Button5Click(TObject *Sender)
```

```
{
}
```

Добавим в коде программы операторы для определения константы *m*, задающей максимальную размерность для массивов, переменной *n* для текущей размерности матриц. Определим также функции: *RazmernStrG* – для задания размерности таблиц *StringGrid*; *inpm* – для ввода матрицы из таблицы; *outpm* – вывода матрицы в таблицу; *umnm* – умножения матриц.

Для того чтобы программа в ответ на конкретное событие выполняло именно то действие, которое требуется, в окне кода вносим соответствующий программный код для обработчика события *Click* для каждой кнопки. При этом для выполнения требуемых операций над матрицами вызываются соответствующие функции, определенные в программе. Добавим в код программы пояснения в виде комментариев. В результате получаем следующий исходный код программы для вычисления произведения матриц.

Файл *Main.h*:

```
#ifndef MainH
#define MainH

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Grids.hpp>

class TForm1 : public TForm
{
__published:   // IDE-managed Components
        TStringGrid *StringGrid1;
        TStringGrid *StringGrid3;
        TStringGrid *StringGrid2;
```

```
        TButton *Button1;
        TButton *Button2;
        TButton *Button3;
        TButton *Button4;
        TButton *Button5;
        TEdit *Edit1;
        TLabel *Label1;
        TLabel *Label2;
        TLabel *Label3;
        TLabel *Label4;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall Button3Click(TObject *Sender);
        void __fastcall Button4Click(TObject *Sender);
        void __fastcall Button5Click(TObject *Sender);
private:  // User declarations
public:         // User declarations
        __fastcall TForm1(TComponent* Owner);
};

extern PACKAGE TForm1 *Form1;

const int m=5;

void RazmernStrG(int n, TStringGrid *StrG, AnsiString st);
void inpm(int n, double a[m][m], TStringGrid *StrG);
void outpm(int n, double a[m][m], TStringGrid *StrG);
void umnm(int n, double a[m][m], double b[m][m], double c[m][m]);
#endif
```

Файл *Main.cpp*:

```
#include <vcl.h>
#pragma hdrstop
```

```
#include "Main.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

int n;
double a[m][m], b[m][m], c[m][m];

void RazmernStrG(int n, TStringGrid *StrG, AnsiString st)
{ /* Формирование размерности n+1 таблицы StrG */
 StrG->ColCount = n+1;
 StrG->RowCount = n+1;
 StrG->Cells[0][0] = st;

 for(int i=1; i<=n; i++) {
    StrG->Cells[i][0] = IntToStr(i);
    StrG->Cells[0][i]= IntToStr(i);
 }
}

void inpm(int n, double a[m][m], TStringGrid *StrG)
{ /* Ввод матрицы a размера n*n из таблицы StrG */
 int i,j;

 for(i=0; i<n; i++)
    for(j=0; j<n; j++)
       a[i][j] = StrToFloat(StrG->Cells[j+1][i+1]);
}

void outpm(int n, double a[m][m], TStringGrid *StrG)
{ /* Вывод матрицы a размера n*n в таблицу StrG */
 int i,j;
```

```
 for(i=0; i<n; i++)
    for(j=0; j<n; j++) StrG->Cells[j+1][i+1] = FloatToStr(a[i][j]);
}
void umnm(int n, double a[m][m], double b[m][m], double c[m][m])
{ /* умножение матриц размера n*n A*B->C */
 int  i,j,k,s;

 for(i=0; i<n; i++)
    for(j=0; j<n; j++) {
       s=0;
       for(k=0; k<n; k++)  s = s+a[i][k]*b[k][j];
       c[i][j] = s;
    }
}

__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{ /* Метод для задания рамерности матриц и таблиц */
 n = StrToInt(Edit1->Text);
 RazmernStrG(n, StringGrid1, "A");
 RazmernStrG(n, StringGrid2, "B");
 RazmernStrG(n, StringGrid3, "C=A*B");
}

void __fastcall TForm1::Button2Click(TObject *Sender)
{ /* метод для ввода исходных матриц из таблиц */
 inpm(n, a, StringGrid1);
 inpm(n, b, StringGrid2);
}
```

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{ /* метод для умножения матриц A*B->C */
 umnm(n, a, b, c);
}

void __fastcall TForm1::Button4Click(TObject *Sender)
{ /* метод для вывода матрицы в таблицу */
 outpm(n, c, StringGrid3);
}

void __fastcall TForm1::Button5Click(TObject *Sender)
{
 Close();
}
```

4.5. Контрольный пример

Для проверки правильности работы программы составим контрольный пример. Зададим исходные матрицы *A, B* в виде

$$A=\begin{pmatrix}1 & 2\\ 3 & 0\end{pmatrix}, B=\begin{pmatrix}2 & 3\\ 1 & 4\end{pmatrix}.$$

Вычислим вручную произведение матриц

$$C=\begin{pmatrix}4 & 11\\ 6 & 9\end{pmatrix}.$$

На рис.1.2 представлено окно формы программы с результатом, полученным для данного контрольного примера. Отметим, что результат, полученный с помощью разработанной программы, совпадает с результатом, полученным вручную.

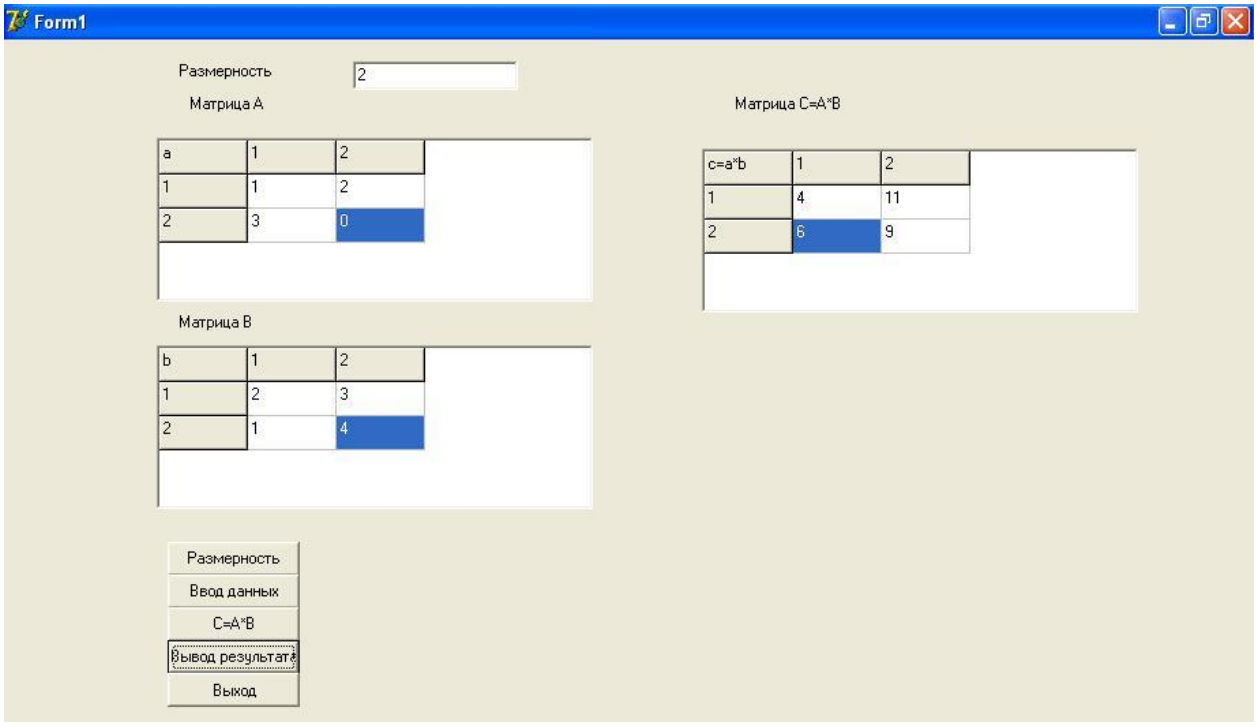


Рис.1.2. Окно программы с результатом вычисления произведения матриц для контрольного примера

6. Варианты заданий

Составить и отладить программу вычисления матрицы *F* по заданным матрицам *A*, *B* и *C* размера *n***n*. Матричные операции реализовать в виде отдельных функций. Варианты заданий приведены в табл. 1.1

Таблица 1.1

Номер варианта	Матричное выражение	Номер варианта	Матричное выражение
1	A+B*C	16	A*(B–C*A)
2	A*B–C	17	A*C–C*B
3	(A-B)*C	18	AT*B*A–C
4	A*(B–C)	19	AT*A–B*BT
5	A*C–B–C	10	C*A+A*B
6	(C+A)*(A–B)	21	C*A+A*C*B

7	(A–C)*(B-C)	22	A*C*(B–C)
8	(A+C)*(C+B)	23	(A–C)*C*B
9	(A–C)*B+C	24	(A+C)*B–C
10	C*A–B+A	25	C*A*B–A
11	A*B–A*C	26	A*CT+C*B
12	2*A–3*B*C	27	A*(C+CT)B
13	(A+B)/3+C*B	28	(C–A)*(C–A)T
14	C*A/5-B*A*2	29	(B+C)*(B–A)T
15	(B+C)*2+(B+A)/2	30	(B–A)T*A/4

7. Порядок выполнения работы

1) Для указанного варианта задания составить программу в Borland C++ Builder. При составлении программы реализовать матричные операции в виде функций.
2) Выполнить расчеты для контрольного примера, сравнив результат, полученный на ПЭВМ, с результатом, полученным вручную.
3) Оформить в виде файла отчет о проделанной работе. В отчете привести вариант задания, текст программы с комментариями, краткое описание программы, контрольный пример и результаты счета.
4) При сдаче работы студент показывает файл отчета и отвечает на контрольные вопросы.

8. Контрольные вопросы

1) В чем различие между стандартными функциями и функциями пользователя?
2) Для каких целей используются функции?
3) Для чего используются параметры функций?

4) В каких случаях и почему используются параметры переменные и параметры значения?
5) В чем отличие механизма передачи значений между вызывающей программой и функцией при использовании параметров переменных и параметров значений?
6) Какие объекты используются в качестве фактических параметров при вызове функций?
7) Как определить свойства объектов библиотеки визуальных компонент Borland C++ Builder? Для чего и как использовать эти объекты?

ЛАБОРАТОРНАЯ РАБОТА №2. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ. ОБРАБОТКА ЗАПИСЕЙ

Цель занятия – ознакомление с технологией структурного подхода к разработке алгоритмов и программ.

1. Структурный подход к разработке алгоритмов и программ

Одна из целей структурного подхода – избавиться от плохой структуры программ. Другая цель – создать программы, которые можно было бы понимать, использовать и модифицировать без участия авторов. Сделать программу более понятной, легкой для отладки и сопровождения означает увеличить эффективность процесса обработки данных.

Структурный подход можно представить состоящим из 3-х частей [4]:

- нисходящая разработка,
- структурное программирование,
- сквозной структурный контроль.

1.1. Нисходящая разработка

Традиционно проектирование прикладной системы делается сверху вниз, а программирование – снизу вверх. Когда программы самого нижнего уровня кодируются перед программами более высокого уровня, могут понадобиться отладочные программы для проверки этих модулей. Эти отладочные программы передают модулю некоторые данные и получают результаты его работы. Затем они могут проверить и (или) напечатать полученные данные, чтобы программист мог проверить промежуточные результаты. Хотя отладочные программы относительно несложные, на их написание уходит время. Обычно это чисто технологические программы.

Основная трудность при восходящем подходе состоит в том, что каждый модуль может правильно работать со своей отладочной программой, но когда подходит время выполнить все модули вместе, ничего не работает. Объясняется

это тем, что некоторые отладочные программы могли быть написаны несколько раньше и спецификации для остальных частей программы могли тем временем измениться или спецификации могли интерпретироваться по-разному разными людьми, или некоторые особенности полной программы оказалось трудно или даже невозможно учесть в отладочных программах.

Другая проблема при восходящем подходе и стремлении поскорее перейти к кодированию состоит в том, что трудности имеют тенденцию концентрироваться вокруг заключительной фазы разработки. Выявленные ошибки в проекте приводят к необходимости менять спецификации к программам к перепроектированию, перекодированию и перетестированию.

При нисходящей разработке и проектирование, и программирование ведутся сверху вниз. Соответствие функциональных спецификаций системы и составляющих ее программ проверяется до перехода на более низкий уровень спецификаций. Интеграция и тестирование ведутся непрерывно в течение всей разработки. Вместо программ нижнего уровня должны использоваться заглушки. Их назначение – позволить программе верхнего уровня выполниться и быть проверенной.

Заглушка может не содержать ничего кроме оператора вывода сообщения.

1.2. Пошаговая детализация

Каждый шаг этого процесса включает в себя разложение функции алгоритма на подфункции. В конечном итоге эти подфункции превращаются в шаги нужной программы. Пошаговая детализация применяется для декомпозиции функции каждого модуля в соответствии с внутренней логикой, необходимой для выполнения модулем этой функции.

Первый шаг связан с очень общим предложением. Разложение первого общего шага в последовательность шагов второго или более низкого уровня заставляет более точно определить логику модуля – это и есть детализация предыдущей формулировки задачи. Таким образом, модуль расширяется по мере добавления и уточнения деталей. Детализация проводится до уровня базовых алгоритмических

структур (линейной, разветвляющейся и циклической), готовых алгоритмов и программных модулей (функций).

При пошаговой детализации алгоритм представляется в виде блок-схем.

1.3. Структурное программирование

Основано на использовании соответствующих трем базовым алгоритмическим структурам следующих базовых структур программирования: линейной, разветвляющейся и циклической. При этом исходят из того положения, что любой алгоритм можно представить комбинацией трех указанных базовых структур. Их важной особенностью является то, что они имеют один вход и один выход и могут соединяться друг с другом в любой последовательности, образовывая снова одну из базовых структур. Это дает наглядную и простую структуру алгоритма, по которой далее легко составить программу.

Обычно при составлении блок-схемы процесс вычислений идет сверху вниз, возвращаясь назад только в циклах, что позволяет анализировать алгоритм как обычный текст, т.е. сверху вниз.

Если технологию разработки алгоритмов "сверху-вниз" совместить с использованием только структурных схем базовых алгоритмических структур, то получается технология, которая называется структурным программированием сверху-вниз. Разработанные в соответствии с ней алгоритмы обладают в некотором смысле свойством *правильности*. В таких алгоритмах, как правило, меньше ошибок, хотя в целом эта технология, как и другие, не гарантирует от неточностей, связанных с невнимательностью.

Конечно, для разработки структурированного сверху-вниз алгоритма (и программы) требуется затратить больше усилий, чем для получения неструктурированного алгоритма. Однако дополнительные затраты окупаются, так как в структурированных алгоритмах и программах легче разобраться, их легче изменять.

2. Пример применения структурного подхода для разработки алгоритма и программы

Пусть задан массив записей с оценками успеваемости группы студентов $Q(i)$, $i = 1, 2, ..., N$. Требуется определить количество и список студентов, оценки которых выше среднего балла группы.

При использовании метода разработки алгоритма "сверху вниз с пошаговой детализацией" первоначально продумывается и фиксируется множество данных и результатов алгоритма без детальной проработки отдельных его частей. Тогда блок-схема любого алгоритма выглядит так, как показана на рис. 2.1.

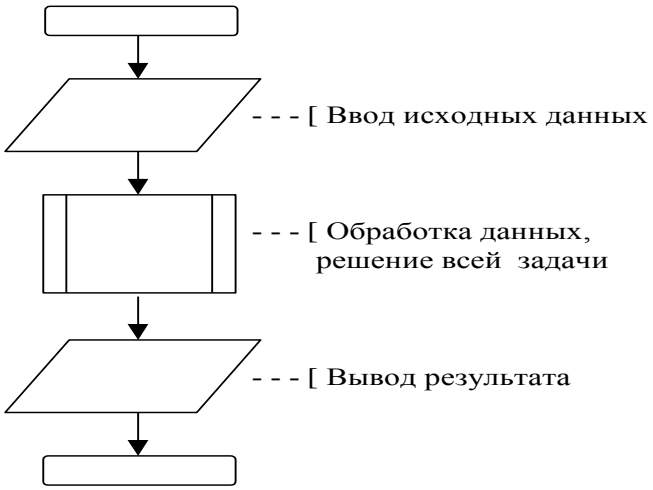


Рис 2.1. Начальное состояние блок-схемы алгоритма при разработке методом "сверху-вниз"

Исходную задачу разбивают на автономные части, каждая из которых существенно проще, чем исходная. Может оказаться необходимым повторять процесс детализации многократно, но это определяется только сложностью решаемых задач. Участки блок-схем, требующие дальнейшей детализации, обозначаются блоками "детализируемая программа", целиком заимствуемые части алгоритма обозначаются блоками "предопределенный процесс".

Конечным уровнем детализации алгоритма можно считать такой, при котором в алгоритме нет действий более крупных, чем: обращение к библиотечному

(вспомогательному) алгоритму; вычисление арифметического выражения и присваивание значение переменной; сравнение арифметических выражений (или переменных); ввод (вывод) данных или других действий, которые могут быть представлены базовыми структурами следования, разветвления и циклической.

Блок-схема алгоритма решения задачи (первого этапа детализации) приведена на рис. 2.2.

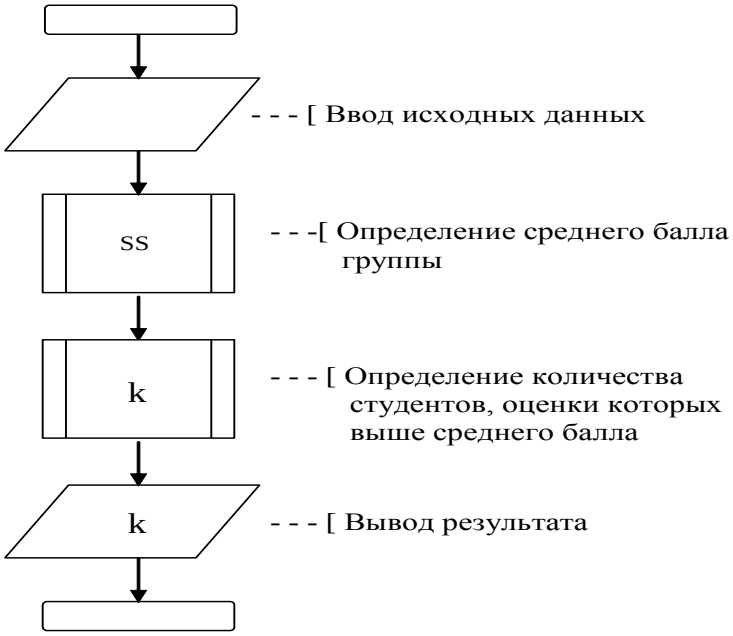


Рис. 2.2. Блок-схема алгоритма определения количества студентов, оценки которых выше среднего балла группы (I этап детализации)

После первого этапа составления алгоритма получается перечень относительно самостоятельных задач, каждая из которых может быть представлена в виде отдельного модуля. Итак, на первом этапе детализации алгоритма определился состав модулей и их функции.

Процесс детализации можно продолжить. Детализировать можно сразу все блоки предыдущей детализации или по частям (это определяется размером схемы). Желательно, чтобы на один лист входила или вся блок-схема или каждая ее отдельная детализация.

В данном примере оказывается достаточным двух этапов детализации алгоритма. Однако часто встречаются задачи, которые требуют трёх, четырёх и более

этапов. Блок-схема таких задач значительно превышает размеры листа. В этом случае каждую новую детализацию можно выполнять на отдельном листе, оформляя ее как самостоятельный алгоритм с именем. В дальнешем отдельные части алгоритма можно реализовать в виде модулей.

3. Пример разработки программы

После первого этапа детализации алгоритма определяется структура программы (состав модулей и их взаимосвязь – последовательность вызова, передача данных), а также функциональное назначение каждого модуля, входные и выходные параметры модулей (функций). Для организации интерфейса пользователя (ввода – вывода данных) в окне формы размещаются следующие объекты: *Edit1* – для задания количества студентов в группе, в поле *Text* которого в инспекторе объектов вводится значение 3, *StringGrid1* – для ввода полей записи списка студентов группы, *StringGrid2* – для вывода (отображения) полей записи списка студентов, чей средний балл успеваемости выше среднего балла в группе. Для обеспечения возможности ввода данных в поля таблицы *StringGrid1* в инспекторе объектов следует установить значение *true* для свойства *goEditing* раздела *Options*. Для управления работой программы в качестве управляющих элементов выбираются кнопки *Button*1 – *Button*4, с которыми связываются соответствующие шаги алгоритма. В результате окно формы с выбранными объектами может быть представлено так, как показано на рис. 2.3.

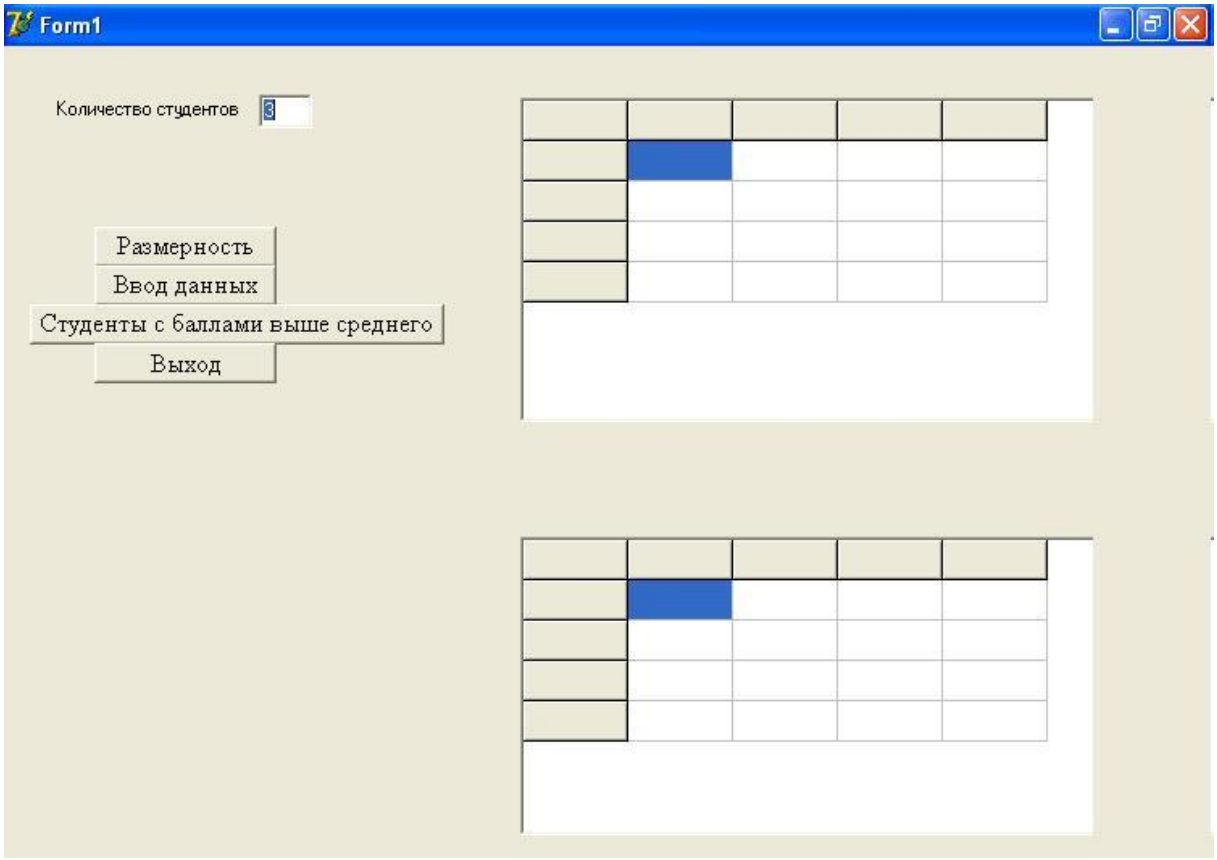


Рис. 2.3. Окно формы с выбранными объектами

На данном этапе составляется программа в виде головного модуля. Для этого с помощью инспектора объектов вкладки *Events* с событием *Click* для каждой кнопки свяжем определенное действие, которое должна выполнить программа в ответ на нажатие конкретной кнопки после запуска программы на выполнение: *Button1* – запрос и ввод количества студентов (размерности списка); *Button2* – запрос и ввод исходных полей списка студентов (перед этим значения элементов полей списка студентов группы должны быть введены в соответствующие клетки таблицы *StringGrid1*); *Button3* – вычисление среднего балла успеваемости студентов группы и формирования списка студентов, чей средний балл успеваемости не ниже среднего балла всей группы; *Button4* – закрытие окна формы и выход из программы. В результате выбора и размещения на форме указанных объектов, связывания события *Click* с каждой кнопкой в C++ Builder автоматически будет сгенерирован следующий код программы.

Файл *Unit1.h*:

```
#ifndef Unit1H
#define Unit1H

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Grids.hpp>

class TForm1 : public TForm
{
__published:   // IDE-managed Components
        TStringGrid *StringGrid1;
        TStringGrid *StringGrid2;
        TButton *Button1;
        TButton *Button2;
        TButton *Button3;
        TButton *Button4;
        TEdit *Edit1;
        TLabel *Label1;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall Button3Click(TObject *Sender);
        void __fastcall Button4Click(TObject *Sender);
private:  // User declarations
public:         // User declarations
        __fastcall TForm1(TComponent* Owner);
};

extern PACKAGE TForm1 *Form1;
```

Файл *Unit1.cpp*:

```
#include <vcl.h>
```

```
#pragma hdrstop
#include "Unit1.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{
}
void __fastcall TForm1::Button2Click(TObject *Sender)
{
}
void __fastcall TForm1::Button3Click(TObject *Sender)
{
}
void __fastcall TForm1::Button4Click(TObject *Sender)
{
}
```

Введем структуру *Student*, в полях которой разместим необходимую информацию о каждом из студентов:

```
struct Student {
   AnsiString fio; /* Фамилия, имя и отчество студента */
   int godr;       /* Год рождения студента*/
   int godp;       /* Год поступления */
   struct { int lina,analis,inf,fizika,fiz_ra; } ocenk;
/* Структура для хранения оценок по линейной алгебре, матем. анализу, информатике, физике и физкультуре */
};
```

Описание структуры *Student* находится в файле *Unit2.h*.

Для представления и хранения данных в программе определим в *Unit2* переменные *gruppa* – для исходного массива списка записей студентов группы, *n* – число элементов в списке студентов группы, *num* – массив для хранения номеров в списке студентов, чей балл выше среднего. Повторное описание этих переменных в файле *Unit2.h* с ключевым словом *extern* делает данные переменные внешними, т.е. доступными из других модулей.

В методах *Button1Click – Button3Click* в ответ на событие *Click* вызываются функции, которые реализуют требуемые операции: *RazmernStrG* – для задания размерности и именования столбцов таблиц списка студентов; *inpzap* – для ввода записей студентов из таблицы *stingGrid* в массив; *midlocenk* – для вычисления среднего балла успеваемости студентов группы; *kolstud* – для определения количества и номеров записей студентов, чей балл успеваемости выше среднего; *outpz* – для вывода записей списка студентов из массива в таблицу *stingGrid*.

Все требуемые функции определим сначала в виде заглушек в модуле *Unit2*. В результате на I этапе детализации получаем следующие модули.

Файл *Unit1.h*:

```
#ifndef Unit1H
#define Unit1H

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>

class TForm1 : public TForm
{
__published:   // IDE-managed Components
        TStringGrid *StringGrid1;
        TStringGrid *StringGrid2;
        TButton *Button1;
```

```
        TButton *Button2;
        TButton *Button3;
        TButton *Button4;
        TEdit *Edit1;
        TLabel *Label1;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall Button3Click(TObject *Sender);
        void __fastcall Button4Click(TObject *Sender);
private:  // User declarations
public:   // User declarations
        __fastcall TForm1(TComponent* Owner);
};

extern PACKAGE TForm1 *Form1;
#endif
```

Файл *Unit1.cpp*:

```
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{
```

```
 n = StrToInt(Edit1->Text);
 RazmernStrG(n, StringGrid1, "Группа");
}

void __fastcall TForm1::Button2Click(TObject *Sender)
{
 inpzap(n, gruppa, StringGrid1);
}

void __fastcall TForm1::Button3Click(TObject *Sender)
{
 double ss; /* средний балл успеваемости студентов группы */
 int k;     /* количество студентов, чей балл выше среднего */
 ss = midlocenk(n, gruppa);
 k = kolstud(n, gruppa, ss, num);
 outpz(n, k, ss, gruppa, num, StringGrid2);
}

void __fastcall TForm1::Button4Click(TObject *Sender)
{
 Close();
}
```

Файл *Unit2.h*:

```
/* Модуль с функциями для определения студентов, чей
   средний балл успеваемости выше среднего в группе */
#ifndef Unit2H
#define Unit2H

const int Nmax=20;

struct Student {
   AnsiString fio;
   int godr;
   int godp;
```

```
   struct { int lina,analis,inf,fizika,fiz_ra; } ocenk;
};

extern int  n;
extern int  num[Nmax];
extern Student  gruppa[Nmax], s1;

void RazmernStrG(int n, TStringGrid *StrG, AnsiString st);
void inpzap(int n, Student a[], TStringGrid *StrG);
void outpz(int n, int k, double ss, Student a[],
           int num[], TStringGrid *Strg);
double midlocenk(int n, Student Gruppa[]);
int kolstud(int n, Student gruppa[], double ss,
            int num[]);
#endif
```

Файл *Unit2.cpp*:

```
#include <vcl.h>
#pragma hdrstop

#include <Grids.hpp>
#include "Unit2.h"

#pragma package(smart_init)

int  n;
int  num[Nmax];
Student  gruppa[Nmax], s1;

void RazmernStrG(int n, TStringGrid *StrG, AnsiString st)
{ /*Процедура для задания размерности таблицы с данными о студентах
*/
 ShowMessage("заглушка-процедура задания размерности таблицы с дан-
ными о студентах ");
```

```
}

void inpzap(int n, Student a[], TStringGrid *StrG)
{ /*процедура ввода исходных записей из таблицы StrG */
 ShowMessage("заглушка-процедура ввода исходных записей ");
}

void outpz(int n, int k, double ss, Student a[],
           int num[], TStringGrid *Strg)
{  /*процедура вывода записей студентов,чей балл выше среднего в
группе */
 ShowMessage("заглушка-процедура вывода записей студентов, чей балл
выше среднего");
}

double midlocenk(int n, Student Gruppa[])
{ /* Функция  для пределение среднего балла студентов в группе */
 ShowMessage("заглушка-функция определение среднего балла студентов
в группе");
 return 0;
}

int kolstud(int n, Student gruppa[], double ss, int num[])
{/* Определение количества студентов в группе чей балл выше среднего
    с сохранением порядковых номеров этих студентов в массиве num */
 ShowMessage("заглушка-функция определения количества студентов в
группе с баллом выше среднего");
 return 0;
}
```

Для того чтобы определения из модуля *Unit2* стали доступны в модуле *Unit1* в начале файла *Unit1.cpp* необходимо добавить строку *#include "Unit2.h"*:

```
#include "Unit1.h"
#include "Unit2.h"
```

При трансляции и отладке данной программы с "заглушками" проверяется организация взаимодействия модулей в соответствии с алгоритмом, производится уточнение функционального назначения процедур и функций, их входных и выходных параметров.

На втором этапе производится дальнейшая детализация блоков алгоритма и реализация процедур и функций для задания размерности таблицы с данными о студентах, ввода записей с данными о студентах из таблицы в массив, определения среднего балла успеваемости студентов всей группы, определения количества и номеров записей о студентах, чей балл выше среднего, вывода списка студентов, чей балл выше среднего из массива в таблицу для его отображения. Затем проводится отладка программы с функциями, которые заменяют соответствующие заглушки. В результате получаем готовую программу, модуль *Unit*2 которой представлен ниже.

Файл *Unit2.h*:

```
/* Модуль с процедурами и функциями для определения студентов,
чей средний балл успеваемости выше среднего в группе */
#ifndef Unit2H
#define Unit2H

const int Nmax=20;

struct Student {
   AnsiString fio;
   int godr;
   int godp;
   struct { int lina,analis,inf,fizika,fiz_ra; } ocenk;
};

extern int  n;
extern int  num[Nmax];
extern Student  gruppa[Nmax], s1;
```

```
void RazmernStrG(int n, TStringGrid *StrG, AnsiString st);
void inpzap(int n, Student a[], TStringGrid *StrG);
void outpz(int n, int k, double ss, Student a[], int num[],
TStringGrid *Strg);
double midlocenk(int n, Student Gruppa[]);
int kolstud(int n, Student gruppa[], double ss, int num[]);
#endif
```

Файл *Unit2.cpp*:

```
#include <vcl.h>
#pragma hdrstop

#include <Grids.hpp>
#include "Unit2.h"

#pragma package(smart_init)

int  n;
int  num[Nmax];
Student  gruppa[Nmax], s1;

void RazmernStrG(int n, TStringGrid *StrG, AnsiString st)
{ /* Процедура для задания размерности таблицы с данными о студен-
тах */
 StrG->ColCount = 9;
 StrG->RowCount = n+1;
 StrG->Cells[0][0] = st;

 for(int i=0; i<=n; i++) {
    StrG->Cells[0][i] = IntToStr(i);
    StrG->Cells[1][0] = "ФИО";
    StrG->Cells[2][0] = "ГР";
    StrG->Cells[3][0] = "ГП";
```

```
    StrG->Cells[4][0] = "Л_А";
    StrG->Cells[5][0] = "М_А";
    StrG->Cells[6][0] = "ИНФ";
    StrG->Cells[7][0] = "ФИЗИКА";
    StrG->Cells[8][0] = "Ф-РА";
 }
}

void inpzap(int n, Student a[], TStringGrid *StrG)
{ /* Процедура ввода исходных записей из таблицы StrG */
 for(int i=0; i<n; i++) {
    a[i].fio = StrG->Cells[1][i+1];
    a[i].godr = StrToInt(StrG->Cells[2][i+1]);
    a[i].godp = StrToInt(StrG->Cells[3][i+1]);
    a[i].ocenk.lina = StrToInt(StrG->Cells[4][i+1]);
    a[i].ocenk.analis = StrToInt(StrG->Cells[5][i+1]);
    a[i].ocenk.inf = StrToInt(StrG->Cells[6][i+1]);
    a[i].ocenk.fizika = StrToInt(StrG->Cells[7][i+1]);
    a[i].ocenk.fiz_ra = StrToInt(StrG->Cells[8][i+1]);
 }
}

void outpz(int n, int k, double ss, Student a[], int num[],
TStringGrid *StrG)
{ /* Процедура вывода записей студентов, чей балл выше среднего в
группе */
 RazmernStrG(k, StrG, "Студ.>средн.");
 int j;
 for(int i=0; i<k; i++) {
    j=num[i];
    StrG->Cells[1][i] = a[j].fio;
    StrG->Cells[2][i] = IntToStr(a[j].godr);
    StrG->Cells[3][i] = IntToStr(a[j].godp);
    StrG->Cells[4][i] = IntToStr(a[j].ocenk.lina);
```

```
    StrG->Cells[5][i] = IntToStr(a[j].ocenk.analis);
    StrG->Cells[6][i] = IntToStr(a[j].ocenk.inf);
    StrG->Cells[7][i] = IntToStr(a[j].ocenk.fizika);
    StrG->Cells[8][i] = IntToStr(a[j].ocenk.fiz_ra);
 }
}

double midlocenk(int n, Student Gruppa[])
{ /* Функция  для пределение среднего балла студентов в группе */
 int count=0, sum=0;

 for(int i=0; i<n; i++) {
    sum = sum +gruppa[i].ocenk.lina +
          gruppa[i].ocenk.analis+ gruppa[i].ocenk.inf +
          gruppa[i].ocenk.fizika+ gruppa[i].ocenk.fiz_ra;
    count = count+5;
 }
 return  double(sum)/count;
}

int kolstud(int n, Student gruppa[],
            double ss, int num[])
{/* Определение количества студентов в группе, чей балл выше сред-
него с сохранением порядковых номеров этих студентов в массиве num
*/
 double r;
 int j=0;

 for(int i=0; i<n; i++) {
    r = double(gruppa[i].ocenk.lina+
       gruppa[i].ocenk.analis+ gruppa[i].ocenk.inf +
       gruppa[i].ocenk.fizika+ gruppa[i].ocenk.fiz_ra)/5;
    if(r>ss) {
       num[j]=i;
```

```
        j=j+1;
    }
 }
 return j;
}
```

На рис. 2.4 показано окно программы с исходными данными и результатами, полученными для списка из 3-х студентов с заданными баллами успеваемости по дисциплинам линейная алгебра (Л_А), математическому анализу (М_А), информатике (ИНФ), физике и физкультуре (ФИЗ_РА).

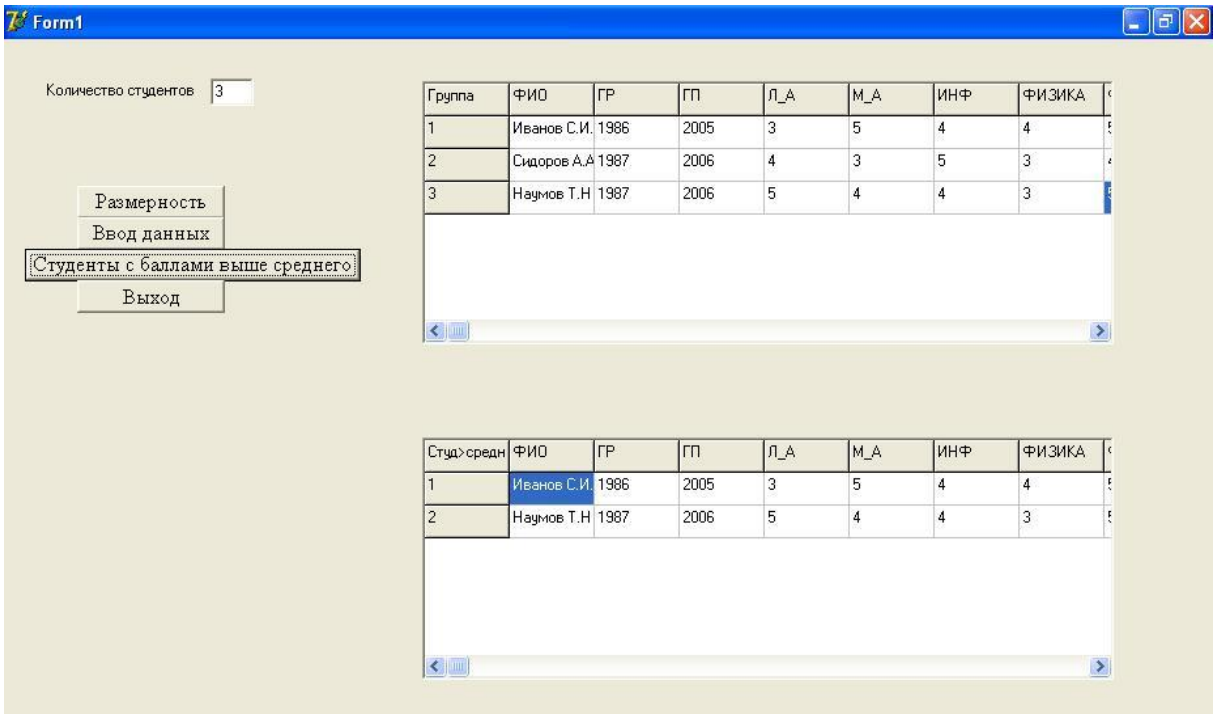


Рис. 2.4. Окно программы с исходными данными и полученными результатами обработки списка записей студентов группы из 3-х человек

4. Варианты заданий

Составить список учебной группы, включающей 7-10 человек. Для каждого студента указать дату рождения, год поступления в университет, курс, группу, оценки каждого года обучения. Информацию о каждом студенте оформить в программе в виде записи. Совокупность записей объединить в массив. Составить

программу, которая обеспечивает ввод полученной информации, отображение ее в виде таблицы, а также вывод информации о студентах согласно конкретному варианту задания.

1) Распечатать анкетные данные студентов-отличников.
2) Распечатать анкетные данные студентов, успевающих на 4 и 5.
3) Распечатать анкетные данные студентов, получивших одну оценку 3 за все время обучения.
4) Распечатать анкетные данные студентов, получивших в последнюю сессию оценки 2.
5) Распечатать анкетные данные студентов, получивших в первую сессию все оценки 5.
6) Распечатать анкетные данные студентов, получивших за все время обучения одну оценку 4, а все остальные оценки 5.
7) Распечатать фамилии и даты рождения студентов, не получивших ни одной оценки 3 за все время обучения.

5. Порядок выполнения работы

1) Для указанного варианта задания составить программу в Borland C++ Builder, отладить ее и проверить правильность работы на самостоятельно выбранном контрольном примере.
2) Оформить в виде файла отчет о проделанной работе. В отчете привести вариант задания, исходную информацию, текст программы с комментариями, краткое описание и результаты работы программы.
3) При сдаче работы студент показывает файл отчета, демонстрирует работу программы и отвечает на контрольные вопросы.

6. Контрольные вопросы

1) Какие типы данных языка C++ используется в программе для представления списка студентов группы?
2) Для чего используется структурный подход в программировании?

3) Что из себя представляет процесс пошаговой детализации алгоритма?
4) До какого уровня ведется детализация алгоритма?
5) Каким образом можно обратиться к полям записи?

ЛАБОРАТОРНАЯ РАБОТА № 3. РЕАЛИЗАЦИЯ ИТЕРАЦИОННЫХ АЛГОРИТМОВ С МАТРИЧНЫМИ ОПЕРАЦИЯМИ

Цель занятия - применение технологии структурного программирования для разработки итерационных алгоритмов с матричными операциями.

1. Постановка задачи

Пусть задано рекуррентное соотношение

$$B_{k+1} = B_k - (A^T B_k + B_k A + C), k = 0,1,2,3\ldots ,$$

где A, C - заданные вещественные $(n \times n)$-матрицы, причем матрица C - положительно определенная (все главные диагональные минор положительны) симметрическая ($a_{ij}=a_{ji}$) матрица. Требуется вычислить $(n \times n)$-матрицу B_k, при k=1,2,… которая бы удовлетворяла условию $\|B_{k+1} - B_k\| \le \varepsilon$, где ε - заданное положительное число, характеризующее точность вычисления.

Итерационные алгоритмы, называемые также разностными схемами, широко используются для решения алгебраических векторных и матричных уравнений. При этом задается начальное приближение. В качестве него обычно берется единичная матрица $B_0=I$ (I – матрица, у которой элементы главной диагонали 1 а остальные - 0).

Итак, для решения исходной задачи нам требуется реализовать итерационную формулу с матричными вычислениями на каждой итерации. К аналогичным итерационным процедурам сводятся задачи моделирования дискретных (цифровых) систем автоматического управления по математическим моделям, представленным в виде разностных уравнений.

Особенностью итерационных алгоритмов является то, что необходимо задавать условие для окончания вычислений, так как заранее бывает неизвестно сколько итераций потребуется. Кроме того, рекомендуется ограничивать количество итераций некоторым максимальным значением и в процессе счета осуществлять проверку, не превышено ли это максимальное значение, так как часто бывает

заранее трудно определить выполниться ли вообще когда-нибудь условие окончания вычислений.

В качестве условий окончания вычислений в таких задачах, как правило, принимается близость по некоторой норме получаемых на текущей итерации двух соседних приближений B_{k+1} и B_k, например, в виде $\|B_{k+1} - B_k\| \le \varepsilon$, где ε - заданная точность, а норма разности матриц

B_{k+1}–B_k=B определяется формулой $\|B\| = \sum_{ij=1}^{n} b_{ij}^2$.

2. Применение структурного подхода для разработки итерационных алгоритмов

Применим метод разработки алгоритма "сверху вниз". Сначала составляем головную программу, определяя, тем самым, состав модулей, их функции, входные и выходные параметры процедур и функций, реализующих блоки алгоритма.

На рис. 3.1 представлен внешний вид программы.

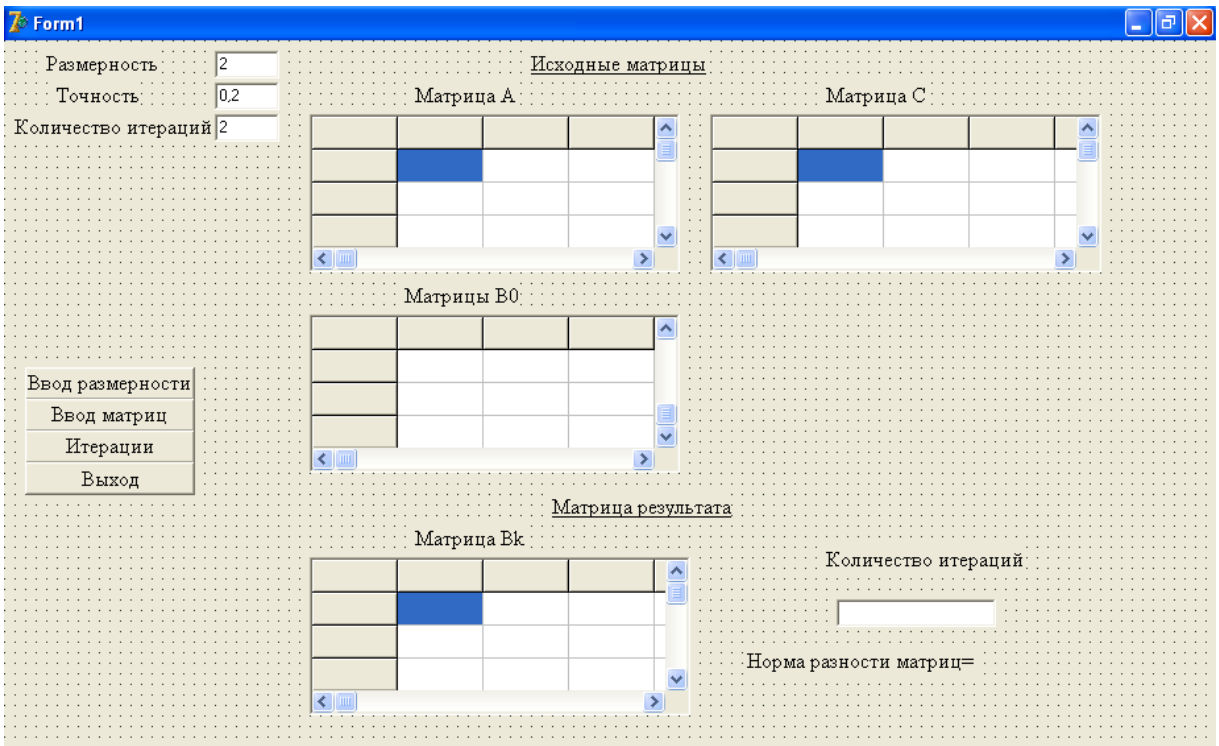


Рис. 3.1. Окно формы с выбранными объектами

Файл *Unit1.h*:

```
/* Модуль с головной программой для выполнения итераций с матричными операциями.
Итерационная формула: B(k+1)=B(k)-(AT*B(k)+B(K)*A+C), k=0,1,2...
Условие выхода. Норма(B(k+1)-B(k)<eps, где eps - заданная точность.
Норма матрицы B вычисляется по формуле корень квадратный из суммы квадратов ее элементов */
#ifndef Unit1H
#define Unit1H

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Grids.hpp>

const int M = 10;

class TForm1 : public TForm
{
__published:   // IDE-managed Components
        TEdit *Edit1;
        TEdit *Edit2;
        TEdit *Edit3;
        TLabel *Label1;
        TLabel *Label2;
        TLabel *Label3;
        TStringGrid *StringGrid1;
        TStringGrid *StringGrid2;
        TStringGrid *StringGrid3;
        TStringGrid *StringGrid4;
        TButton *Button1;
        TButton *Button2;
        TButton *Button3;
```

```
        TButton *Button4;
        TEdit *Edit4;
        TLabel *Label4;
        TLabel *Label5;
        TLabel *Label6;
        TLabel *Label7;
        TLabel *Label8;
        TLabel *Label9;
        TLabel *Label10;
        TLabel *Label12;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall Button3Click(TObject *Sender);
        void __fastcall Button4Click(TObject *Sender);
private:  // User declarations
public:   // User declarations
        __fastcall TForm1(TComponent* Owner);
};

extern PACKAGE TForm1 *Form1;
#endif
```

Файл *Unit1.cpp*:

```
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
#include "Matrop.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

double a[M][M],c[M][M],b[M][M],at[M][M],b1[M][M],
```

```
       z1[M][M],z2[M][M];
double  eps;
int  n, k, km;

__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{ /* Метод для задания размерности матриц и таблиц */
 n = StrToInt(Edit1->Text);
 RazmernStrG(n, StringGrid1, "A");
 RazmernStrG(n, StringGrid2, "C");
 RazmernStrG(n, StringGrid3, "B0");
 RazmernStrG(n, StringGrid4, "Bk");
}

void __fastcall TForm1::Button2Click(TObject *Sender)
{ /* Метод для ввода исходных данных: точности, числа итераций, мат-
риц A,C,B */
// ввод точности и количества итераций
 eps = StrToFloat(Edit2->Text);
 km = StrToInt(Edit3->Text);
// ввод исходных матриц
 inpm(n, a, StringGrid1);
 inpm(n, c, StringGrid2);
 inpm(n, b, StringGrid3);
}

void __fastcall TForm1::Button3Click(TObject *Sender)
{
 k=1;
 umnmm(n,b,a,z1);
```

```
 trnspm(n,a,at);
 umnmm(n,at,b,b1);
 summ(n,z1,b1,z1);
 summ(n,z1,c,z1);
 raznm(n,b,z1,b1);
 outpm(n,b1,StringGrid4);
 Label12->Caption = FloatToStr(norm(n,z1));

 while((norm(n,z1)>=eps)&&(k<km)) {
    ShowMessage("Итерация № "+IntToStr(k));
    outpm(n, b1, StringGrid4);
    Label12->Caption = FloatToStr(norm(n,z1));
    prisv(n, b1, b);
    umnmm(n, b, a, z1);
    umnmm(n, at, b, b1);
    summ(n, z1, b1, z1);
    summ(n, z1, c, z1);
    raznm(n, b, z1, b1);
    k = k+1;
 }
 outpm(n, b1, StringGrid4);
 Label12->Caption = FloatToStr(norm(n,z1));
 ShowMessage("Итерация № "+IntToStr(k));
 if(k==km) {
    Edit4->Text = IntToStr(k);
    ShowMessage ("Точность не достигнута за "+IntToStr(k)+"
итераций");
 }
 else {
    outpm(n, b1, StringGrid4);
    Edit4->Text = IntToStr(k);
    ShowMessage("Количество итераций = "+IntToStr(k));
 }
}
```

```
void __fastcall TForm1::Button4Click(TObject *Sender)
{
 Close();
}
```

Файл *Matrop.h*:

```
/* Модуль с процедурами, реализующими стандартные матричные операции
*/
#ifndef MatropH
#define MatropH

void RazmernStrG(int n, TStringGrid *StrG, AnsiString st);
void inpm(int n, double a[M][M], TStringGrid *StrG);
void outpm(int n, double a[M][M], TStringGrid *StrG);
void umnmm(int n, double a[M][M],
           double b[M][M], double c[M][M]);
double norm(int n, double a[M][M]);
void trnspm(int n, double a[M][M], double b[M][M]);
void summ(int n, double a[M][M],
          double b[M][M], double c[M][M]);
void prisv(int n, double a[M][M], double b[M][M]);
void raznm(int n, double a[M][M],
           double b[M][M], double c[M][M]);
#endif
```

Файл *Matrop.cpp*:

```
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
#include "Matrop.h"

void RazmernStrG(int n, TStringGrid *StrG, AnsiString st)
{ /* Формирование размерности n+1 таблицы StrG */
```

```
 StrG->ColCount = n+1;
 StrG->RowCount = n+1;
 StrG->Cells[0][0] = st;

 for(int i=1; i<=n; i++) {
    StrG->Cells[i][0] = IntToStr(i);
    StrG->Cells[0][i] = IntToStr(i);
 }
}
void inpm(int n, double a[M][M], TStringGrid *StrG)
{ /* Ввод матрицы a размера n*n из таблицы StrG */
 ShowMessage("Процедура ввода массива по строкам");
}
void outpm(int n, double a[M][M], TStringGrid *StrG)
{ /* Вывод матрицы A размера n*n в таблицу StrG */
 ShowMessage ("Процедура вывода массива по строкам");
}
void umnmm(int n, double a[M][M],
           double b[M][M], double c[M][M])
{ /* Умножение матриц размера n*n:  A*B -> C */
 ShowMessage("Процедура умножения массиов A*B=C ");
}
double norm(int n, double a[M][M])
{ /* Функция вычисления нормы матрицы A */
 ShowMessage("Функция вычисления нормы матрицы A ");
 return 0.1;
}
void trnspm(int n, double a[M][M], double b[M][M])
{ /* Транспонирование матрицы размера n*n At->B */
 ShowMessage("Процедура транспонирования массива A ");
}
void summ(int n, double a[M][M], double b[M][M], double c[M][M])
{ /* Суммирование матриц размера n*n A+B->C */
 ShowMessage("Процедура суммирования массивов A+B=C ");
```

```
}
void prisv(int n, double a[M][M], double b[M][M])
{ /* Пересылка матрицы размера n*n A->B */
 ShowMessage("Пересылка массива A->B");
}
void raznm(int n, double a[M][M],
           double b[M][M], double c[M][M])
{ /* Разность матриц размера n*n A-B->C */
 ShowMessage("Нахождение разности массивов A-B=C ");
}
```

После отладки головной программы с модулем *Matrop*, в котором все матричные операции представлены функциями-заглушками, переходим к детализации алгоритма в плане составления функций для каждой используемой матричной операции.

В результате реализации всех матричных операций получается следующий файл *Matrop.cpp*, реализующий требуемые матричные операции:

```
#include <vcl.h>
#pragma hdrstop

#include <Math.h>    /* для поддержки функции sqrt */
#include "Unit1.h"
#include "Matrop.h"

void RazmernStrG(int n, TStringGrid *StrG, AnsiString st)
{ /* Формирование размерности n+1 таблицы StrG */
 StrG->ColCount = n+1;
 StrG->RowCount = n+1;
 StrG->Cells[0][0] = st;

 for(int i=1; i<=n; i++) {
    StrG->Cells[i][0] = IntToStr(i);
    StrG->Cells[0][i] = IntToStr(i);
 }
```

```
}
void inpm(int n, double a[M][M], TStringGrid *StrG)
{ /* Ввод матрицы a размера n*n из таблицы StrG */
 int i, j;
 for(i=0; i<n; i++)
    for(j=0; j<n; j++)  a[i][j] = StrToFloat(StrG->Cells[j+1][i+1]);
}
void outpm(int n, double a[M][M], TStringGrid *StrG)
{ /* Вывод матрицы a размера n*n в таблицу StrG */
 int i,j;
 for(i=0; i<n; i++)
    for(j=0; j<n; j++)  StrG->Cells[j+1][i+1] = FloatToStr(a[i][j]);
}
void umnmm(int n, double a[M][M], double b[M][M], double c[M][M])
{ /* Умножение матриц размера n*n:  A*B -> C */
 int i,j,k;
 double s;

 for(i=0; i<n; i++)
    for(j=0; j<n; j++) {
       s = 0;
       for(k=0; k<n; k++) s=s+a[i][k]*b[k][j];
       c[i][j] = s;
 }
}
double norm(int n, double a[M][M])
{ /* Функция вычисления нормы матрицы A */
 int i,j;
 double nr = 0.0;

 for(i=0; i<n; i++)
    for(j=0; j<n; j++)  nr = nr+a[i][j]*a[i][j];

 return sqrt(nr);
```

```
}
void trnspm(int n, double a[M][M], double b[M][M])
{ /* Транспонирование матрицы размера n*n At->B */
 int i,j;
 for(i=0; i<n; i++)
    for(j=0; j<n; j++) b[i][j]=a[j][i];
}
void summ(int n, double a[M][M], double b[M][M], double c[M][M])
{ /* Суммирование матриц размера n*n A+B->C */
 int i,j;
 for(i=0; i<n; i++)
    for(j=0; j<n; j++)  c[i][j]=a[i][j]+b[i][j];
}
void prisv(int n, double a[M][M], double b[M][M])
{ /* Пересылка матрицы размера n*n A->B */
 int i,j;
 for(i=0; i<n; i++)
    for(j=0; j<n; j++) b[i][j]=a[i][j];
}
void raznm(int n, double a[M][M],
           double b[M][M], double c[M][M])
{ /* Разность матриц размера n*n A-B->C */
 int i,j;
 for(i=0; i<n; i++)
    for(j=0; j<n; j++) c[i][j]=a[i][j]-b[i][j];
}
```

Остальные файлы проекта остаются без изменений.

3. Контрольный пример

Для проверки правильности работы программы составим контрольный пример. Зададим исходные матрицы A, C, B_0 в виде

$$A=\begin{pmatrix} 1 & 2 \\ -3 & 2 \end{pmatrix},\ C=\begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix},\ B_0=\begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}.$$

Вычислим вручную произведение матриц. После 1-й итерации

$$B_1 = \begin{pmatrix} 2 & 2 \\ 3 & -16 \end{pmatrix},\ \|B_1 - B_0\| = 19.1311265.$$

После 2-й итерации

$$B_2 = \begin{pmatrix} 11 & 57 \\ -58 & -35 \end{pmatrix},\ \|B_2 - B_1\| = 99.4183082.$$

При запуске программы на выполнение и ввода исходных данных после нажатия кнопки "Итерации" в интерфейсном окне программы будет выводиться сообщение "Итерация №" с указанием номера текущей итерации, и результат вычисления матрицы B_k и нормы разности матриц B_k–B_{k-1}. На рис.3.2 Представлено окно формы программы с результатом, полученным для данного контрольного примера.

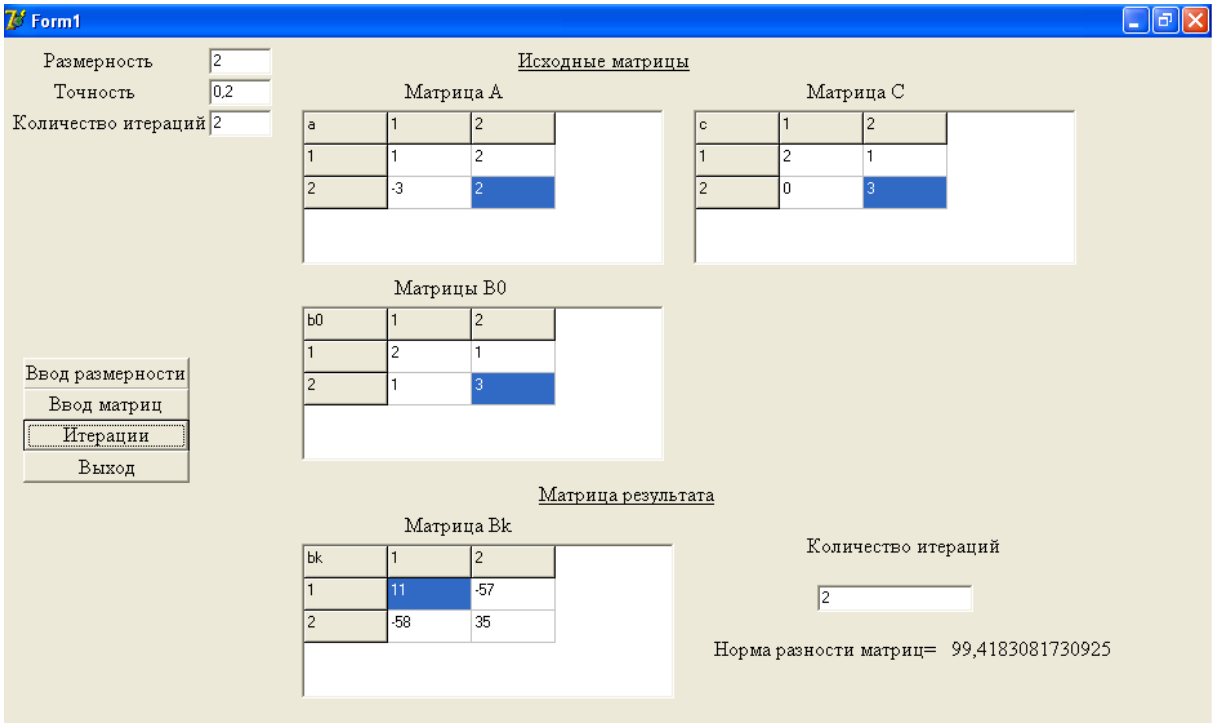


Рис. 3.2. Окно программы с результатами счета для контрольного примера

4. Варианты заданий

1) $D_{k+1} = A^T D_k A + B$, где A, B, D_1 - заданные ($n \times n$)-матрицы; k=1,2,... Условие выхода $||D_{k+1} - D_k|| < e$, где e - заданная точность, $\|D\| = \max_{ij} |d_{ij}|$ - норма матрицы $D = D_{k+1} - D_k$.

2) $Z_{k+1} = A^T Z_k + Z_k A + Z_k Q Z_k$, где A, Q, Z_0 - заданные ($n x n$)-матрицы. Условие выхода $||Z_{k+1} - Z_k|| < e$, где e - заданная точность, где $\|Z\| = \sum_{ij=1}^{n} |z_{ij}|$ - норма матрицы $Z = Z_{k+1} - Z_k$, k=1,2,3... .

3) $Q_{k+1} = (A + hA^2) Q_k$, где A, Q_0 - заданные ($n \times n$)-матрицы; h - const; $k=1,2,3...$.Условие выхода $||Q_k|| > M$, где M - заданное достаточно большое число, $\|Q_k\| = \sum_{i=1}^{n} \max_j |q_{ij}|$ - норма матрицы Q_k.

4) $X_{k+1} = [I + Ah + (Ah)^2 / 2] X_k$, где A - заданная ($n \times n$)-матрица; X_0 - заданный n - вектор; h - const; k=1,2,3... . Условие выхода $||X_{k+1} - X_k|| < e$, где e - заданная точность; $\|X\| = \max_i |x_i|$ - норма вектора X.

5) $Y_{k+1} = AY_k + B$, где A - заданная ($n \times n$)-матрица, b - заданный n-вектор; условие выхода $|| Y_{k+1} - Y_k || < e$, где e - заданная точность, $\|Y\| = \sum_{i=1}^{n} |y_i|$ - норма вектора $Y = Y_{k+1} - Y_k$, $k=1,2,3....$

6) $Z_{k+1} = -Q + Z_k^T D Z_k$, где D, Q, Z_0 - заданные ($n \times n$)-матрицы; Условие выхода$|| Z_{k+1} - Z_k || < e$, где e - заданная точность, $\|Z_k\| = \max_{ij} |z_{ij}|$ - норма матрицы $Z = Z_{k+1} - Z_k$, $k=1,2,3....$

7) $X_{k+1} = A^T X_k A + X_k^T G X_k$, где A, G, X_1 - заданные ($n \times n$)-матрицы; условие выхода $||X_{k+1} - X_k|| < e$, где e - заданная точность, $\|X\| = \sum_{ij=1}^{n} |x_{ij}|$ - норма матрицы $X = X_{k+1} - X_k$.

8) $X_{k+1} = AX_k$, где A, X_1, - заданные ($n \times n$)-матрицы; условие выхода $|| X_{k+1} - X_k || < e$, где e - заданная точность, $\|X\| = \sum_{i=1}^{n} |x_i|$ - норма матрицы $X = X_{k+1} - X_k$, $k=1,2,3....$

9) Логарифм матрицы $\ln(I + X) = X - \frac{X^2}{2} + \frac{X^3}{3} - \frac{X^4}{4} \pm ...$, где X - заданная ($n \times n$)-матрица; Условие выхода $||X^m/m|| < e$, e - заданная точность, $\|Z\| = \sum_{i,j=1}^{n} |z_{ij}|$ - норма матрицы $Z = X^m/m$.

10) Логарифм матрицы $\ln(I - X) = -X - \frac{X^2}{2} - \frac{X^3}{3} - \frac{X^4}{4} - ...$, где X - заданная ($n \times n$)-матрица; Условие выхода $||X^m/m|| < e$, e - заданная точность, $\|Z\| = \max_j \sum_{i=1}^{n} |z_{ij}|$ - норма матрицы $Z = X^m/m$.

11) Экспоненциал матрицы $Exp(A) = I + A + A^2/2! + A^3/3! + A^4/4! + ...$, где A - заданная ($n \times n$)-матрица. Условие выхода $\left\| \frac{A^m}{m!} \right\| < e$, e - заданная точность, $\|Z\| = \sum_{i,j=1}^{n} |z_{ij}|$ - норма матрицы $Z = \frac{A^m}{m!}$.

12) Экспоненциал матрицы $e^{-X^2} = I - \frac{X^2}{1!} + \frac{X^4}{2!} - \frac{X^6}{3!} \mp ... + (-1)^m \frac{X^{2m}}{m!}$, где X - заданная ($n \times n$)-матрица. Условие выхода $\left\| \frac{X^{2m}}{m!} \right\| < e$, e - заданная точность, $\|Z\| = \sqrt{\sum_{i,j=1}^{n} z_{ij}^2}$ - норма матрицы $Z = \frac{X^{2m}}{m!}$.

13) Экспоненциал матрицы $e^{-X} = I - \frac{X}{1!} + \frac{X^2}{2!} - \ldots$, где X - заданная ($n \times n$)-матрица. Условие выхода $\left\| \frac{x^m}{m!} \right\| < e$, e - заданная точность, $\|Z\| = \sum_{j=1}^{n} \max_i |z_{ij}|$ - норма матрицы $Z = \frac{x^m}{m!}$.

14) Матричный арктангенс $arctg(X) = X - \frac{X^3}{3} + \frac{X^5}{5} - \frac{X^7}{7} \pm \ldots$, где X - заданная ($n \times n$)-матрица; Условие выхода $||x^m/m|| < e$, e - заданная точность, $\|Z\| = \max_i \sum_{j=1}^{n} |z_{ij}|$ - норма матрицы $Z = x^m/m$.

15) Матричный арксинус

$\arcsin(X) = X + \frac{1}{2}\frac{X^3}{3} + \frac{1 \cdot 3}{2 \cdot 4}\frac{X^5}{5} + \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6}\frac{X^7}{7} + \ldots$, где X - заданная ($n \times n$)-матрица; Условие выхода $||X^m/m|| < e$, e - заданная точность, $\|Z\| = \sum_{j=1}^{n} \max_i |z_{ij}|$ - норма матрицы $Z = X^m/m$.

16) Матричный косинус $CosX = I - X^2/2! + X^4/4! - X^6/6! + - \ldots$, где X – заданная ($n \times n$)-матрица; Условие выхода $||X^m/m!|| < e$, e – заданная точность, $\|Z\| = \max_i \sum_{j=1}^{n} |z_{ij}|$ - норма матрицы $Z = X^m/m!$.

17) Матричный синус $X^{-1}Sin\ X = I - X^2/3! + X^4/5! - X^6/7! + - \ldots$, где X - заданная ($n \times n$)-матрица; условие выхода $||X^m/m!|| < e$, e - заданная точность, $\|Z\| = \sum_{i,j=1}^{n} |z_{ij}|$ - норма матрицы $Z = \frac{X^m}{m!}$.

18) Матричный синус $SinX=X - X^3/3! + X^5/5! - X^7/7! +-...$, где X - заданная $(n\times n)$-матрица; условие выхода $||X^m/m!||<e$, e - заданная точность, $\|Z\| = \sum_{j=1}^{n} \max_i |z_{ij}|$ – норма матрицы $Z = \frac{X^m}{m!}$.

19) Гиперболический матричный синус $\mathrm{sh}(X) = X + \frac{X^3}{3!} + \frac{X^5}{5!} + \frac{X^7}{7!} + \ldots$, где X – заданная $(n\times n)$-матрица. Условие выхода $||X^m/m!||<e$, e – заданная точность, $\|Z\| = \sum_{j=1}^{n} \max_i |z_{ij}|$ - норма матрицы $Z= X^m/m!$.

20) Гиперболический матричный косинус $ch(X) = I + \frac{X^2}{2!} + \frac{X^4}{4!} + \frac{X^6}{6!} + \ldots$, где X - заданная $(n\times n)$-матрица; Условие выхода $||X^m/m!||<e$, e - заданная точность, $\|Z\| = \max_{i,j} |z_{ij}|$ - норма матрицы $Z=X^m/m!$.

5. Порядок выполнения работы

1) Для указанного варианта задания составить программу в Borland C++ Builder. Процедуры, реализующие матричные операции оформить в виде отдельного модуля C++ Builder**.**
2) Отладить программу и проверить правильность ее работы на контрольном примере сравнив результаты работы программы с результатами, полученными вручную для 2 итераций. Контрольный пример выбрать самостоятельно, задав исходные матрицы порядка 2x2.
3) Выполнить несколько вариантов расчета (2-3) для различных значений точности и количества итераций.
4) Оформить в виде файла отчет о проделанной работе. В отчете привести вариант задания, текст программы с комментариями, краткое описание программы, контрольный пример для отладки, результаты счета.

5) При сдаче работы студент показывает файл отчета, демонстрирует работу программы и отвечает на контрольные вопросы.

6. Контрольные вопросы

1) Какие особенности необходимо учитывать при реализации итерационных алгоритмов?
2) Как ведется разработка и отладка сложной программы при использовании структурного подхода?
3) Как создать и использовать модули в Borland C++ Builder?

ЛАБОРАТОРНАЯ РАБОТА № 4. СОРТИРОВКА МАССИВОВ

Цель занятия – ознакомление и приобретение навыков реализации алгоритмов сортировки массивов на языке C++.

1. Общие сведения

Сортировку следует понимать как процесс перегруппировки заданного множества объектов в некотором определенном порядке. Цель сортировки – облегчить последующий поиск элементов в таком отсортированном множестве. Мы встречаемся с отсортированными объектами в телефонных книгах, в различных списках, в оглавлениях книг, в библиотеках, в словарях, на складах - почти везде, где нужно искать хранимые объекты. Даже малышей приучают держать свои вещи "в порядке".

Разговор о сортировке вполне уместен и важен, если речь идет об обработке данных. Кроме того, сортировка – это идеальный объект для демонстрации огромного разнообразия алгоритмов, все они изобретены для одной и той же задачи, многие в некотором смысле оптимальны, большинство имеет свои достоинства. На примерах сортировок можно показать, как путем усложнения алгоритма можно добиться значительного выигрыша в эффективности. Изложение алгоритмов сортировки с анализом их эффективности дано в [1,2]. В [3] приводится классификация известных методов сортировки.

При рассмотрении задачи сортировки, следуя [2] будем считать, что группа данных, которую нужно упорядочить, фиксирована, т.е. множество из n элементов задано в виде массива $a[n]$. Если у нас есть элементы $a_1,a_2,...,a_n$, то сортировка есть перестановка этих элементов в массив $a_{k_1}, a_{k_2},...,a_{k_{n1}}$, где при некоторой упорядочивающей функции f выполняются отношения $f(a_{k_1}) \le f(a_{k_2}) \le ... f(a_{k_n})$. Обычно упорядочивающая функция не вычисляется по какому-либо правилу, а хранится как явная компонента (поле) каждого элемента. Ее значение называется ключом (*key*) элемента. Поэтому для представления

элементов выбирается тип данных – структура, которую можно определить следующим образом:

```
struct Item {
   int key;
        /* здесь описаны другие компоненты */
};
```

Под "другими компонентами" подразумеваются данные, по существу относящиеся к сортируемым элементам, а ключ просто идентифицирует каждый элемент. Говоря об алгоритмах сортировки, мы будем обращать внимание лишь на компоненту-ключ, другие же компоненты можно даже и не определять. Поэтому в дальнейшем считаем, что тип элемента определен как *int*.

Метод сортировки называется устойчивым, если в процессе сортировки относительное расположение элементов с равными ключами не изменяется. Устойчивость сортировки часто бывает желательной, если речь идет об элементах, уже упорядоченных (отсортированных) по некоторым вторичным ключам, т.е. свойствам, не влияющим на основной ключ.

2. Методы сортировки

Основное условие: выбранный метод сортировки массивов должен экономно использовать доступную память. Это предполагает, что перестановки, приводящие элементы в порядок, должны выполняться на том же месте, т.е. методы, в которых элементы из массива *a* передаются в результирующий массив *b*, представляют существенно меньший интерес. Хорошей мерой эффективности может быть C - число необходимых сравнений ключей и M - число пересылок (перестановок) элементов.

Методы сортировки "на том же месте" можно разбить в соответствии с определяющими их принципами на три основные категории [2]:

- сортировки с помощью вставки;
- сортировки с помощью выбора;

– сортировки с помощью обмена.

Рассмотрим эти методы. Все программы оперируют переменной *a*, именно здесь хранятся переставляемые элементы в виде массива структур *Item*:

```
Item a[n];
```

2.1. Сортировка с помощью вставок

Элементы мысленно делятся на уже "готовую" последовательность a_1, ..., a_{i-1} и исходную последовательность. При каждом шаге, начиная с i=2 и увеличивая i каждый раз на единицу, из исходной последовательности извлекается i-й элемент и переносится в готовую последовательность, при этом он вставляется на нужное место. В табл. 4.1 показан в качестве примера процесс сортировки с помощью вставки восьми случайно выбранных чисел.

Таблица 4.1.

Начальные ключи	44	55	12	42	94	18	06	67
i=2	44	***55***	12	42	94	18	06	67
i=3	12	44	***55***	42	94	18	06	67
i=4	12	42	44	***55***	94	18	06	67
i=5	12	42	44	55	***94***	18	06	67
i=6	12	18	42	44	55	***94***	06	67
i=7	06	12	18	42	44	55	***94***	67
i=8	06	12	18	42	44	55	67	***94***

Алгоритм этой сортировки представим в виде:

```
for(i=2; i<n+1; i++) {
   x = a[i];
   включение x на соответствующее место среди a[1],...,a[i]
}
```

В реальном процессе поиска подходящего места удобно, чередуя сравнения и движения по последовательности, как бы просеивать *x*, т.е. *x* сравнивается с очередным элементом a_j, а затем либо *x* вставляется на свободное место, либо a_j

сдвигается (передается) вправо. Обратите внимание, что процесс просеивания может закончиться при выполнении одного из двух следующих различных условий:

1. Найден элемент a_j с ключом, меньшим чем ключ у x.

2. Достигнут левый конец готовой последовательности.

На языке C++ функция сортировки с помощью вставок выглядит следующим образом.

```
void SortInsert(int n, int a[])
{ /* Сортировка с помощью вставки */
 int i,j,x;
/* сдвигаем все элементы массива a на 1 позицию вправо */
 for(i=n; i>0; i--) a[i]=a[i-1];
 for(i=2; i<=n; i++) {
    x=a[i];
    a[0]=x;  j=i-1;
    while(x<a[j]) {
       a[j+1]=a[j];  j=j-1;
    }
    a[j+1]=x;
 }
/* сдвигаем все элементы массива a на 1 позицию влево */
 for(i=0; i<n; i++) a[i]=a[i+1];
}
```

Такой типичный случай повторяющегося процесса с двумя условиями окончания позволяет нам воспользоваться хорошо известным приемом "барьера". Здесь его легко применить, поставив барьер a_0 со значением x. Заметим, что для этого необходимо расширить величину массива a до n+1. В этом случае элемент a[0] служит "барьером", а сортируемые значения последовательно располагаются в элементах a[1], a[2], …, a[n].

Данный алгоритм описывает процесс устойчивой сортировки: порядок элементов с равными ключами при нем остается неизменным.

Алгоритм со вставками можно легко улучшить, если обратить внимание на то, что готовая последовательность, в которую надо вставить новый элемент, сама уже упорядочена. Естественно остановиться на двоичном поиске, при котором делается попытка сравнения с серединой готовой последовательности, а затем процесс деления пополам идет до тех пор, пока не будет найдена точка вставки. Такой модифицированный алгоритм называется сортировкой бинарными вставками. Он реализован в следующей функции.

```
void BinarInsert(int n, int a[])
{ /* Сортировка вставками с методом деления пополам */
 int i,j,m,L,r,x;
 for(i=1; i<n; i++) {
    x=a[i];  L=0;  r=i-1;
    while(L<=r) {
       m=(L+r)/2;
       if(a[m]>x) r=m-1; else L=m+1;
    }
    for(j=i-1; j>=L; j--) a[j+1]=a[j];
    a[L]=x;
 }
}
```

2.2. Сортировка с помощью выбора

Этот прием состоит в следующем:

1. Выбирается элемент с наименьшим ключом.
2. Он меняется местами с первым элементом a_1.
3. Затем этот процесс повторяется с оставшимися n-1 элементами, n-2 элементами и т.д. до тех пор, пока не останется один, самый большой элемент.

Алгоритм представляется так:

```
for(i=0; i<n; i++) {
   присвоить k индекс наименьшего из a[i],...,a[n] элементов;
   поменять местами a[i] и a[k];
```

```
}
```

Такой метод – его называют простым выбором – в некотором смысле противоположен методу вставки. При вставке на каждом шаге рассматриваются только один очередной элемент исходной последовательности и все элементы готовой последовательности, среди которых отыскивается точка вставки; при простом выборе для поиска одного элемента с наименьшим ключом просматриваются все элементы исходной последовательности и найденный помещается как очередной элемент в готовую последовательность. Полностью алгоритм простого выбора приводится в следующей функции.

```
void Selection(int n, int a[])
{ /* Сортировка с помощью простого выбора */
 int  i, j, k, x;
 for(i=0; i<n-1; i++) {
    k=i;  x=a[i];
    for(j=i; j<n; j++)
       if(a[j]<x) { k=j;  x=a[j]; }
    a[k]=a[i];  a[i]=x;
 }
}
```

2.3. Сортировка с помощью обмена

Алгоритм простого обмена основывается на сравнении и смене мест для пары соседних элементов и продолжении этого процесса до тех пор, пока не будут упорядочены все элементы. Как и в упоминавшемся методе простого выбора, мы повторяем проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к левому концу массива. Если мы будем рассматривать массивы как вертикальные, а не горизонтальные построения, то элементы можно интерпретировать как пузырьки в чане с водой, причем вес каждого соответствует его ключу. В этом случае при каждом проходе один пузырек как бы поднимается до уровня, соответствующего его весу - значению. Такой метод ши-

роко известен под именем "пузырьковая сортировка". В своем простейшем виде он представлен в следующей программе.

```
void BubbleSort(int n, int a[])
{ /* Пузырьковая сортировка */
 int  i,j,x;
 for(i=1; i<n; i++)
    for(j=n-1; j>=i; j--)
       if(a[j-1]>a[j]) {
          x=a[j-1];  a[j-1]=a[j];  a[j]=x;
       }
}
```

Очевидный прием улучшения этого алгоритма – запоминать, были или не были перестановки в процессе некоторого прохода. Если в последнем проходе перестановок не было, то сортировку можно заканчивать. Это улучшение, однако, можно опять же улучшить, если запоминать не только сам факт, что обмен имел место, но и положение (индекс) последнего обмена. Ясно, что все пары соседних элементов выше этого индекса *k* уже находятся в желаемом порядке. Поэтому просмотры можно заканчивать на этом индексе, а не идти до заранее определенного нижнего предела для *i*.

Анализ показывает, что "обменная" сортировка и ее небольшие усовершенствования представляют собой нечто среднее между сортировками с помощью вставок и с помощью выбора.

3. Улучшенные методы сортировки

3.1. Сортировка вставками с уменьшающимися расстояниями

В 1959 г. Д.Шеллом было предложено усовершенствование сортировки с помощью вставки. Сам метод объясняется и демонстрируется на примере (см. табл. 4.2). Сначала отдельно группируются и сортируются

Таблица 4.2.

Исходный массив
44 55 12 42 94 18 06 67
четверная сортировка дает
44 18 06 42 94 55 12 67
двойная сортировка дает
06 18 12 42 44 55 94 67
одинарная сортировка дает
06 12 18 42 44 55 67 94

элементы, отстоящие друг от друга на расстоянии 4. Такой процесс называется четверной сортировкой. В нашем примере восемь элементов и каждая группа состоит точно из двух элементов. После первого прохода элементы перегруппировываются - теперь каждый элемент группы отстоит от другого на две позиции - и вновь сортируются. Это называется двойной сортировкой. И наконец, на третьем проходе идет обычная или одинарная сортировка.

Ясно, что такой метод в результате дает упорядоченный массив, и конечно же, сразу видно, что каждый проход от предыдущих только выигрывает (так как каждая *i*-сортировка объединяет две группы, уже отсортированные 2*i*-сорировкой). Каждая *i*-сортировка программируется как сортировка с помощью вставки. Причем простота условия окончания поиска места для вставки обеспечивается методом барьеров. При этом приходится расширять массив *Item a*[*n*+*t*+1]. Сам алгоритм для *t*=4 реализован процедурой *ShellSort* в следующей программе на языке C++.

```
/* Сортировка Шелла */
const int n=8;
int b[n+5]={ 0,0,0,0,0,44,55,12,42,94,18,6,67 };
int a[n+5];
int i;

void ShellSort(int n, int a[])
{
```

```
 const int t=4;
 int i,j,k,s,x, m;
 int h[t];

 h[1]=9; h[2]=5; h[3]=3; h[4]=1;
 for(m=0; m<t; m++) {
    k=h[m];  s:=-k;   /* место барьера */
    for(i=k; i<=n; i++) {
       x=a[i];  j=i-k;
       if(s==0) s=-k;
       s=s+1;  a[s]=x;
       while(x<a[j]) {
          a[j+k]=a[j];  j=j-k;
       }
       a[j+k]=x;
    }
 }
}
void main()
{
 a=b;
 printf("массив a до сортировки");
 writev(n,n,a, " a");
 shellsort(n,a);
 printf("массив a после сортировки");
 writev(n,n,a," a");
}
```

3.3. Сортировка с помощью разделения

При улучшении пузырьковой сортировки, являющейся самой неэффективной из всех трех алгоритмов простой сортировки, был получен один из самых лучших метод сортировки для массивов. Этот метод называется быстрой сортировкой (*Quicksort*). В *Quicksort* исходят из того соображения, что для достижения наилучшей эффективности сначала лучше производить перестановки на большие

расстояния. Предположим, у нас есть *n* элементов, расположенных на ключах в обратном порядке. Их можно отсортировать за *n*/2 обменов, сначала поменять местами самый левый с самым правым, а затем последовательно двигаться с двух сторон. Конечно, это возможно только в том случае, когда мы знаем, что порядок действительно обратный. Но из этого примера можно извлечь и нечто действительно поучительное.

Воспользуемся таким алгоритмом: выберем наугад какой-либо элемент (назовем его *x*) и будем просматривать слева наш массив до тех пор, пока не обнаружим элемент $a_i>x$, затем будем просматривать массив справа, пока не встретим $a_j<x$. Теперь поменяем местами эти два элемента и продолжим наш процесс просмотра и обмена, пока оба просмотра не встретятся где-то в середине массива. В результате массив окажется разбитым на левую часть, с ключами меньше (или равными) *x*, и правую – с ключами больше (или равными) *x*.

Теперь напомним, что наша цель - не только провести разделение на части исходного массива элементов, но и отсортировать его. Сортировку от разделения отделяет лишь небольшой шаг: нужно применить этот процесс к получившимся двум частям затем к частям частей, и так до тех пор, пока каждая из частей не будет состоять из одного-единственного элемента. Эти действия описываются следующей программой, в которой процедура *Sort* рекурсивно обращается сама к себе.

```
const int n=8;
     /* Сортировка с помощью разделения*/
void sort(int L, int R, int a[])
{
 int i,j,w,x;
 i = L; j = R;
 x = a[(L+R)/2];
 do {
    while(a[i]<x) i++;
    while(x<a[j]) j--;
```

```
    if(i<=j) {
       w=a[i]; a[i]=a[j]; a[j]=w;   /* меняем местами a[i] и a[j] */
       i++; j--;
    }
 } while(i<=j);
 /* рекурсивный вызов */
 if(L<j) QSort(L, j, a);
 if(i<R) QSort(i, R, a);
}

void QuickSort(int n, int a[]);
{
 sort(0, n-1, a)
}
     /* Головная программа */
void main()
{
 int a[n]={44,55,12,42,94,18,6,67};
 printf("Массив a до сортировки ");
 writev(n, n, a, " a");
 printf("\n");
 QuickSort(n, a);
 writeln("Массив a после сортировки ");
 writev(n, n, a, " a");
}
```

4. Пример разработки программы сортировки массива в Borland C++ Builder

4.1. Постановка задачи

Пусть задан массив чисел $A = \{a_i\}_{i,=1}^{n}$ размерности *n*.

Требуется составить программу для формирования массива *B* из массива *A* с упорядочением элементов по возрастанию.

4.2. Описание алгоритма

- Задание размерности *n* массивов *A, B*;
- Ввод исходного массива *A*;
- Формирование массива *B* из элементов массива *A*.
- Упорядочение элементов массива *B* по возрастанию
- Вывод массива *C* в виде таблицы.

4.3. Реализация алгоритма в C++ Builder

Согласно условиям задачи требуется задать размерность матриц *n*, исходный одномерный числовой массив *A*. Для размещения размерности *n* определим в программе переменную *n*. Для хранения исходного массива *A*, а также для размещения массива результата *B* определим в программе тип данных массив и переменные *a, b* этого типа. Для отображения в интерфейсном окне программы (окне формы) данных (размерности, исходных матриц и матрицы результата) следует использовать стандартные визуальные объекты: элемент редактирования *Edit1* – для размерности; таблицы *StringGrid1 - StringGrid2* – для массивов *A* и *B*, соответственно. Для обеспечения возможности ввода данных в поля таблицы *StringGrid1* в инспекторе объектов следует установить значение *true* для свойства *goEditing* раздела *Options*. В поле *FixedCols* инспектора объектов для *StringGrid1* и *StringGrid2* ввести значение 0

Для отображения в окне формы пояснений к выбранным визуальным компонентам следует использовать метки *Label*. Для построения и отображения графика в виде столбиковой диаграммы используется объект *Chart*. В объект *Chart* с помощью кнопки *Add…* закладки *Chart* встроенного редактора свойств необходимо вставить два объекта *Series1* и *Series2* типа *Bar*. Встроенный редактор *Chart* вызывается с помощью пункта *Edit Chart…* локального меню этого объекта (для вызова нажать правую клавишу мыши на объекте *Chart*)

Для управления работой программы выберем в качестве управляющих элементов кнопки *Button1 - Button4,* с которыми свяжем соответствующие шаги ал-

горитма. В результате окно формы с выбранными объектами может быть представлено как показано на рис. 4.1.

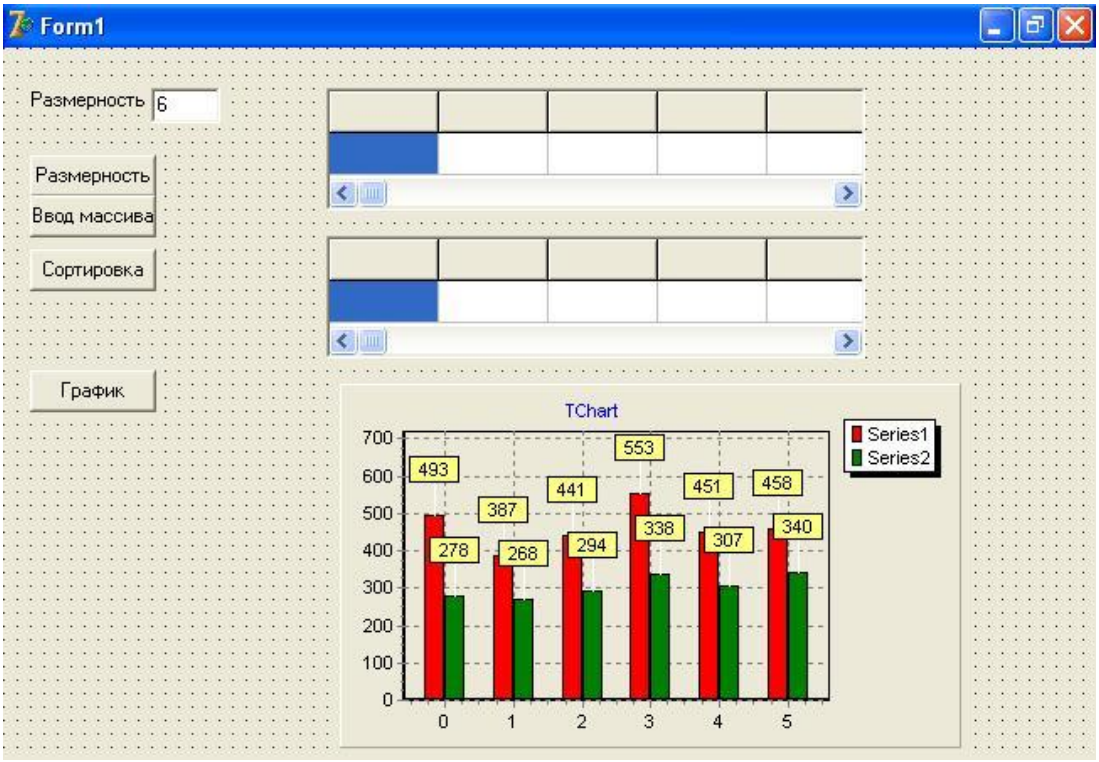


Рис. 4.1. Окно формы с выбранными объектами для сортировки одномерного массива

С событием *Click* каждой кнопки свяжем определенное действие, которое должна выполнить программа в ответ на нажатие конкретной кнопки при выполнении программы: *Button1* – запрос и ввод размерности массивов и таблиц; *Button2* – запрос и ввод исходного массива *A* (перед этим значения элементов исходного массива должны быть введены в таблицу *StringGrid1*; *Button3* – сортировка массива методом простого выбора и вывод упорядоченного массива в таблицу *StringGrid2* для отображения в окне формы; *Button4* – построения графика (столбиковой диаграммы) по значениям элементов массива *A* до и после сортировки. Для того чтобы программа в ответ на конкретное событие выполняла именно то действие, которое требуется, в окне кода вносим соответствующий программный код для обработчика каждого события *Click*. В результате получаем следующий исходный код программы для сортировки одномерного массива.

Файл Sort.h:

```
#ifndef SortH
#define SortH

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Chart.hpp>
#include <ExtCtrls.hpp>
#include <Grids.hpp>
#include <TeEngine.hpp>
#include <TeeProcs.hpp>
#include <Series.hpp>

const int M = 15;

class TForm1 : public TForm
{
__published:   // IDE-managed Components
        TEdit *Edit1;
        TButton *Button1;
        TButton *Button2;
        TButton *Button3;
        TButton *Button4;
        TLabel *Label1;
        TStringGrid *StringGrid1;
        TStringGrid *StringGrid2;
        TChart *Chart1;
        TBarSeries *Series1;
        TBarSeries *Series2;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall Button3Click(TObject *Sender);
```

```
        void __fastcall Button4Click(TObject *Sender);
private:  // User declarations
public:   // User declarations
        __fastcall TForm1(TComponent* Owner);
};

extern PACKAGE TForm1 *Form1;

void SortSelection(int n, int a[]);
#endif
```

Файл Sort.cpp:

```
#include <vcl.h>
#pragma hdrstop

#include "Sort.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

int a[M], b[M];
int n;

void SortSelection(int n, int a[])
{ /* Сортировка одномерного массива a размерности n методом простого
выбора */
 int i,j,k,x;

 for(i=0; i<n; i++) {
    k=i;   x=a[i];
    for(j=i+1; j<n; j++)
       if(a[j]<x) { k=j;  x=a[j]; }
    a[k]=a[i];   a[i]=x;
 }
```

```
}

__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{
 n = StrToInt(Edit1->Text);
 StringGrid1->ColCount = n;
 StringGrid2->ColCount = n;

 for(int i=0; i<n; i++) {
    StringGrid1->Cells[i][0]=IntToStr(i+1);
    StringGrid2->Cells[i][0]=IntToStr(i+1);
 }
}
void __fastcall TForm1::Button2Click(TObject *Sender)
{
 for(int i=0; i<n; i++)
    a[i]=StrToInt(StringGrid1->Cells[i][1]);
}
void __fastcall TForm1::Button3Click(TObject *Sender)
{
 int  i,j,k,x;
 for(i=0; i<n; i++) b[i]=a[i];
 SortSelection(n, b);
 for(i=0; i<n; i++)
    StringGrid2->Cells[i][1]=IntToStr(b[i]);
}
void __fastcall TForm1::Button4Click(TObject *Sender)
{
 Series1->Clear();   Series2->Clear();
```

```
 /* Рисуем график */
 for(int i=0; i<n; i++) {
    Series1->AddXY(i, a[i], "", clRed);
    Series2->AddXY(i, b[i], "", clGreen);
 }
}
```

На рис.4.2. представлено окно программы с исходным и упорядоченным массивом.

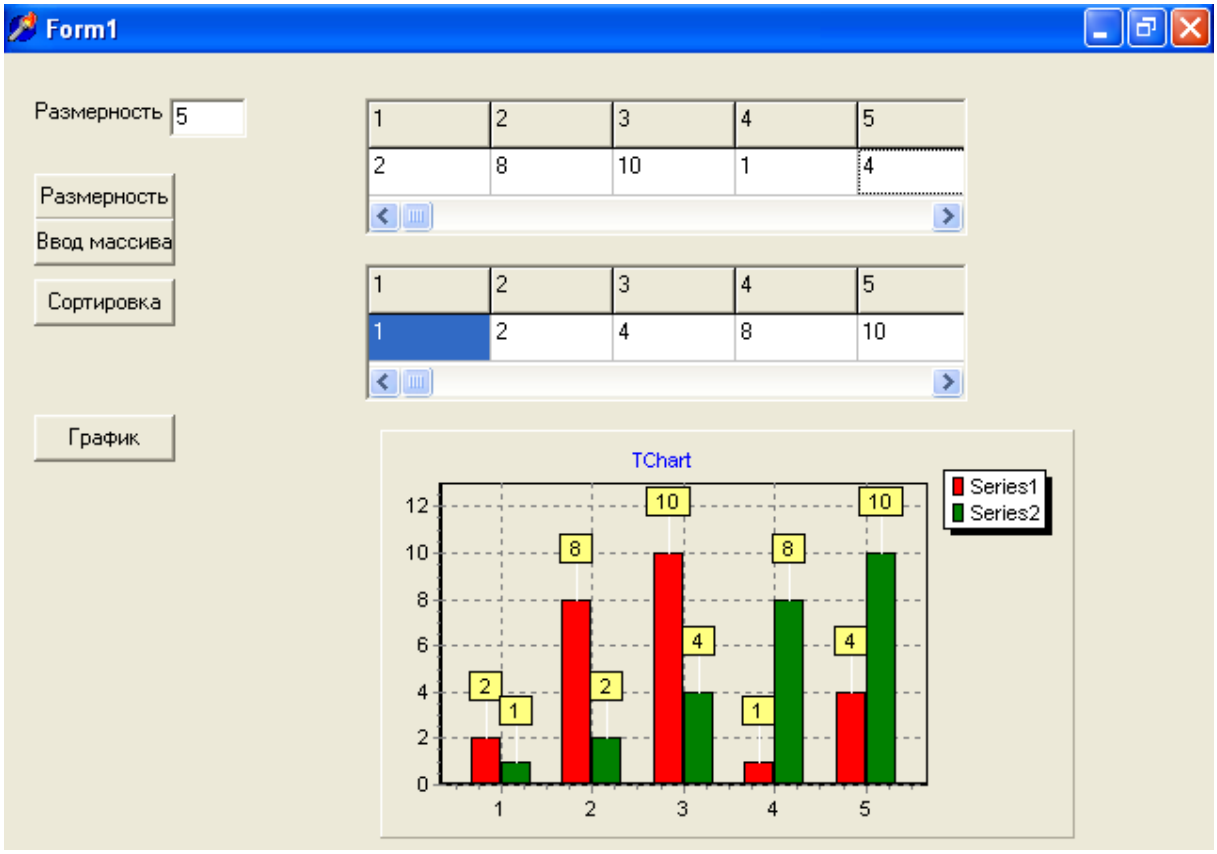


Рис. 4.2. Окно программы с исходным и упорядоченным массивом, которые отображены в виде таблиц и столбиковых диаграмм на графике

5. Варианты заданий

1) Упорядочить одномерный массив a[n] по убыванию его элементов, используя метод вставок.
2) Упорядочить одномерный массив a[n] по убыванию его элементов, используя метод вставок с бинарным поиском.

3) Упорядочить одномерный массив a[n] по убыванию его элементов, используя метод выбора.
4) Упорядочить одномерный массив a[n] по убыванию его элементов, используя метод обмена.
5) Упорядочить одномерный массив a[n] по убыванию его элементов, используя метод Шелла.
6) Упорядочить одномерный массив a[n] по убыванию его элементов, используя метод разделения.
7) Переставить строки двумерного массива a[n,n] в порядке возрастания элементов 1 столбца. Использовать метод вставок.
8) Переставить строки двумерного массива a[n,n] в порядке возрастания элементов 2 столбца. Использовать метод вставок с бинарным поиском.
9) Переставить строки двумерного массива a[n,n] в порядке возрастания элементов последнего столбца. Использовать метод выбора.
10) Переставить столбцы двумерного массива a[n,n] в порядке возрастания элементов 1 строки. Использовать метод обмена.
11) Переставить столбцы двумерного массива a[n,n] в порядке возрастания элементов 2 строки. Использовать метод Шелла.
12) Переставить столбцы двумерного массива a[n,n] в порядке возрастания элементов последней строки. Использовать метод разделения.
13) Упорядочить последовательно все столбцы двумерного массива a[n,n] по убыванию элементов. Использовать метод вставки с бинарным поиском.
14) Упорядочить последовательно все столбцы двумерного массива a[n,n] по возрастанию элементов. Использовать метод Шелла.
15) Упорядочить последовательно все строки двумерного массива a[n,n] по убыванию элементов. Использовать метод разделения.
16) Упорядочить последовательно главную и побочную диагонали двумерного массива a[n,n] по убыванию элементов. Использовать метод разделения.

17) Переставить столбцы двумерного массива a[n,n] в порядке возрастания элементов главной диагонали. Использовать метод обмена.
18) Переставить строки двумерного массива a[n,n] в порядке убывания элементов побочной диагонали. Использовать метод вставок.
19) Переставить элементы каждой строки в соответствии с убыванием их модулей. Использовать метод выбора.
20) Переставить элементы каждого столбца в соответствии с возрастанием их модулей. Использовать метод обмена.

6. Порядок выполнения работы

1) Для указанного варианта задания разработать программу, отладить и проверить правильность ее работы на самостоятельно выбранном контрольном примере. При разработке программы для варианта заданий, в котором кроме сортировки требуется также перестановка других элементов двумерного массива, сначала добейтесь правильности сортировки нужной группы элементов массива, а затем реализуйте перестановку других элементов массива. При этом старайтесь использовать готовую процедуру сортировки, минимизируя возможные ее изменения.
2) Проанализировать эффективность используемого алгоритма сортировки на 5 примерах, для чего предусмотреть задание исходного массива с помощью датчика случайных чисел (используя стандартную функцию random).
3) Оформить в виде файла отчет о проделанной работе. В отчете привести вариант задания, текст программы с комментариями, краткое описание программы, результаты работы программы и анализа ее эффективности.
4) При сдаче работы студент показывает файл отчета, демонстрирует работу программы и отвечает на контрольные вопросы.

7. Контрольные вопросы

1) В чем заключается задача сортировки?
2) Как оценивается эффективность алгоритмов сортировки?

3) Какие методы сортировки известны?
4) В чем суть простых методов сортировки: вставки, выбора, обмена?
5) Какие приемы используются для улучшения простых методов сортировки?

ЛАБОРАТОРНАЯ РАБОТА № 5. ЧИСЛЕННОЕ РЕШЕНИЕ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ

Цель занятия - приобретение навыков применения численных методов для решения дифференциальных уравнений.

1. Общие сведения

Любой реальный объект (система объектов) характеризуется свойствами и поведением. Для анализа поведения объекта применяются математические модели в виде дифференциальных уравнений. В сущности, любая задача проектирования, связанная с расчетом потоков энергии или движения тел, в конечном счёте, сводится к решению дифференциальных уравнений. Интегрирование же в квадратурах может быть выполнено лишь для немногих из них. Поэтому численные методы решения дифференциальных уравнений играют важную роль в практике инженерных расчетов. С их помощью моделируются и исследуются процессы, происходящие в реальных системах, что позволяет судить об их качестве.

Общеизвестным примером колебательной механической системы является "математический" маятник (рис. 5.1). Это идеализированная система, состоящая из груза массой m, прикрепленный к нижнему концу жесткого стержня длиной L с пренебрежимо малой массой, верхний конец которого вращается без трения в точке подвеса. Если груз вывести из положения равновесия и отпустить, то маятник будет совершать колебания в вертикальной плоскости.

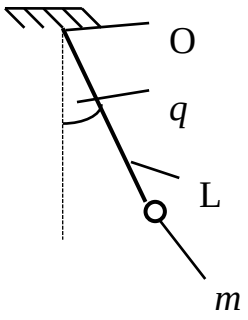


Рис. 5.1. "Математический" маятник

Поскольку движение груза происходит по дуге окружности радиуса L с центром в точке O, то положение груза характеризуется длиной дуги и углом q. Уравнение движения запишется в виде $mLd^2q/dt^2 = -mg\sin q$ или в виде

$$d^2q/dt^2 = -(g/L)\sin q.$$

Данное уравнение также как и большинство нелинейных уравнений не имеет аналитических решений в элементарных функциях. Один из способов получить представление о движении маятника в случае больших амплитуд колебаний - численно решить это уравнение.

Известно, что в природе большинство колебаний постепенно уменьшается до тех пор, пока смещение не становится нулевым; такие колебания называются затухающими. В качестве примера гармонического осциллятора с затуханием рассмотрим движение маятника, показанного на рис. 5.1, с учетом трения и сопротивления среды. Для движения с малыми скоростями в качестве приближения разумно принять, что сила сопротивления пропорциональна скорости. В этом случае уравнение движения можно записать в виде

$$d^2q/dt^2 = -cdq/dt - d\sin q,$$

где коэффициент затухания c представляет меру силы сопротивления. Заметим, что сила сопротивления в уравнении направлена в сторону, противоположную движению. Для того чтобы узнать, как ведет себя $q(t)$, необходимо найти решение указанного дифференциального уравнения. Для этого уравнение приводится к форме Коши – системе уравнений первого порядка, разрешенных относительно производных. Вводя замену переменных $x_1 = q, x_2 = dq/dt$, получим систему уравнений в форме Коши.

$$\begin{cases} dx_1/dt = x_2, \\ dx_2/dt = -cx_2 - d\sin x_1 \end{cases} \qquad (5.1)$$

2. Постановка задачи

Под дифференциальным уравнением понимается уравнение, содержащее как неизвестные функции, так и их производные.

Пусть дана система дифференциальных уравнений в форме Коши

$$dx/dt = f(t,x), \qquad (5.2)$$

где $t \in [a,b], x = (x_1,...,x_n)^T \in R^n$ – вектор неизвестных функций, и пусть задано начальное условие $x(t=a) = x_0$, где x_0 - задано. Требуется найти приближенное решение уравнения (5.2), т.е. найти такую вектор-функцию $y(t)$, которая при $t=a$ $y(a)=x_0$ а при $t \in [a,b], | y(t) - x(t) | \leq \varepsilon,$ где $x(t)$ - точное решение системы (5.2), ε - заданная точность.

3. Методы решения дифференциальных уравнений

3.1. Метод Эйлера

Наиболее простым способом построения решения в точке x_{n+1}, если оно известно в точке x_n, является способ, основанный на разложении в ряд Тейлора (в предположении надлежащей дифференцируемости решения) [1]

$$x(t_{n+1}) = x(t_n) + hdx(t_n)/dt + h^2/2!d^2x(t_n)/dt^2 + ..., \qquad (5.3)$$

где $h=t_{n+1}-t_n$. Если теперь этот ряд оборвать до членов первого порядка малости по h и заменить $x(t_n)$ приближенным значением x_n, то с учетом (5.3) получим формулу

$$x(t_{n+1})=x(t_n)+hf(t_n,x_n),\ n=0,1,2,..., \qquad (5.4)$$

которая описывает метод Эйлера, один из самых старых и широко известных методов численного интегрирования дифференциальных уравнений.

Геометрически этот метод иллюстрируется на рис. 5.2, где кривая представляет собой точное, но конечно, неизвестное решение уравнения, а прямая линия L проведена с тангенсом угла наклона $f(n_n, x_n)$.

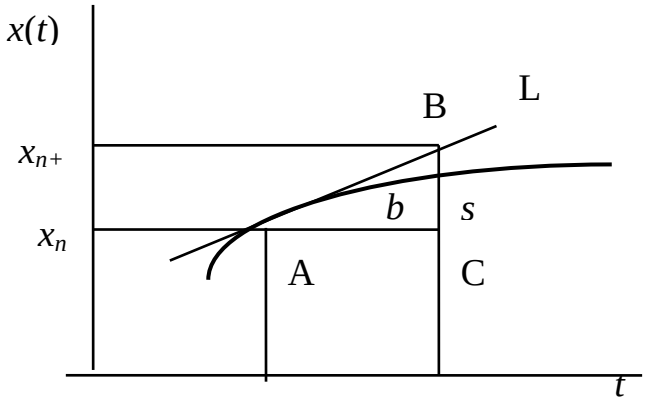


Рис. 5.2. Метод Эйлера

Из треугольника ABC находим $s=BC=AC\ tg\ b=hf(t_{n+1},x_n)$. Откуда получаем формулу метода Эйлера (5.4).

При достаточно малой величине шага h метод Эйлера даст решение с небольшой точностью. Погрешность решения близка к h^2 ($h<<1$) на каждом шаге интегрирования.

Недостаток метода Эйлера – замедление вычислений при выборе малой величины шага h, обеспечивающей высокую точность решения.

3.2. Исправленный метод Эйлера

В исправленном методе Эйлера находится средний тангенс угла наклона касательной для двух точек: t_n,x_n и t_n+h, $x_n+hf(t_n,x_n)$. Последняя точка есть та самая, которая в методе Эйлера обозначалась t_{n+1}, x_{n+1}. Геометрически процесс нахождения точки t_{n+1}, x_{n+1} можно проследить по рис. 5.3.

С помощью метода Эйлера находится точка t_n+h, $x_n+hf(t_n,x_n)$, лежащая на прямой L_1. В этой точке снова вычисляется тангенс угла наклона касательной. На рисунке этому значению соответствует прямая L_2. Усреднение двух тангенсов дает прямую L'. Наконец, через точку t_n,x_n проводим прямую L, параллельную L'. Точка, в которой прямая L пересечется с ординатой, восстановленной из $t_{n+1}=t_n+h$, и будет искомой точкой t_n+h, x_{n+1}.

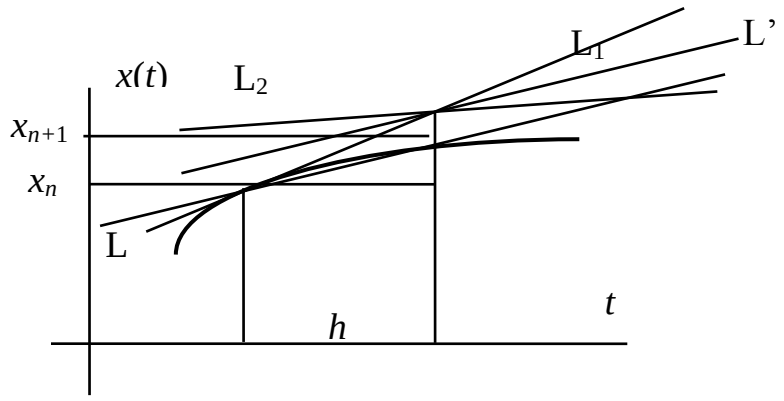


Рис. 5.3. Исправленный метод Эйлера

Тангенс угла наклона прямой L' и прямой L равен $F(t_n,x_n,h)= [f(t_n,x_n)+ +f(t_n+h,x_n+hf(t_n,x_n))]/2$. Уравнение прямой L при этом записывается в виде $x=x_n+(x-x_n)F(t_n,x_n,h)$, так что

$$x_{n+1}=x_n+h[f(t_n,x_n)+f(t_n+h,x_n+hf(t_n,x_n))]/2. \qquad (5.5)$$

Соотношение (5.5) описывают исправленный метод Эйлера. Ошибка исправленного метода Эйлера на каждом шаге имеет порядок h^3. За повышение точности приходится расплачиваться дополнительными затратами машинного времени, так как функцию $f(t,x)$ приходится вычислять дважды в точке t_n, x_n и в точке $t_n+h,x_n+hf(t_n,x_n)$.

3.3. Модифицированный метод Эйлера

Улучшить аппроксимацию производной можно также используя её среднее значение в начале и в конце интервала. Рассмотрим рис. 5.4, где первоначальное построение сделано точно так же, как и на рис. 5.2.

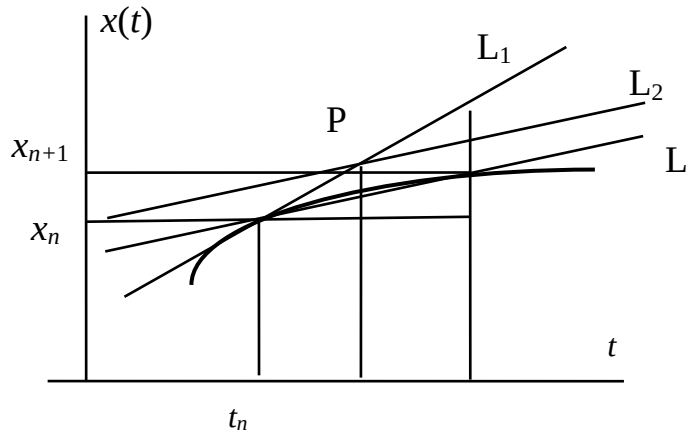


Рис.5.4. Модифицированный метод Эйлера

Через точку t_n,x_n проведена прямая L_1 с тангенсом угла наклона, равным $f(t_n,x_n)$. Но на этот раз мы берем точку, лежащую на пересечении прямой и ординаты $t=t_n+h/2$. На рисунке эта точка обозначена через P. Вычислим тангенс угла наклона касательной в этой точке:

$$F(t_n,x_n,h)= f(t_n+(h/2),x_n+(h/2)f(t_n,x_n)), \qquad (5.6)$$

Прямая с таким наклоном, проходящая через P, обозначена через L_2. Затем проводится через точку t_n,x_n прямая, параллельная L_2, которая обозначена через L. Пересечение этой прямой с ординатой $t=t_n+h$ и даст искомую точку t_{n+1}, x_{n+1}.

Уравнение прямой L можно записать в виде: $x=x_n+(t-t_n)F(t_n,x_n,h)$, где F задается формулой (5.5). Поэтому

$$x_n+1=x_n+h\ f(t_n+(h/2),\ x_n+(h/2)f(t_n,\ x_n)). \qquad (5.7)$$

Формула (5.7) описывает модифицированный метод Эйлера или исправленный метод ломаных. Ошибка на каждом шаге при использовании этого метода имеет порядок h^3.

Более высокая точность может быть достигнута, если пользователь готов потратить дополнительное машинное время на лучшую аппроксимацию производной посредством сохранения большего числа членов ряда Тейлора. Эта же идея лежит в основе методов Рунге-Кутта.

3.4. Методы Рунге-Кутта

Наибольшее распространение получил классический метод Рунге-Кутта четвертого порядка, который выражается формулой (5.8)

$$y_n+1=y_n+(k_0+2k_1+2k_2+k_3)/6, \qquad (5.8)$$

где $k_0=hf(x_n, y_n)$,

$k_1=hf(x_n+h/2, y_n+k_0/2)$,

$k_2=hf(x_n+h/2, y_n+k_1/2)$,

$k_3=hf(x_n+h, y_n+k_2)$

Этот метод обеспечивает более высокую точность. Ошибка на каждом шаге имеет порядок h^5. Более высокая точность метода Рунге-Кутта часто позволяет увеличивать шаг интегрирования h. Допустимая погрешность на шаге определяет его максимальную величину. Чтобы обеспечить высокую эффективность вычислительного процесса, величину h следует выбрать именно из соображения допустимой ошибки на шаге. Такой выбор часто осуществляется автоматически и включается как составная часть в алгоритм, построенный по методу Рунге-Кутта.

Во всех рассмотренных выше методах для получения информации в новой точке необходимо иметь данные лишь в одной предыдущей точке. Это свойство называется "самостартованием".

4. Пример программы для численного решения дифференциальных уравнений движения маятника (моделирование колебаний маятника)

На рис. 5.5 показано окно формы с выбранными объектами в программе для численного решения системы дифференциальных уравнений движения маятника. Кроме ранее рассмотренных объектов здесь также используется объект *Chat* для построения и отображения графиков искомых решений системы дифференциальных уравнений.

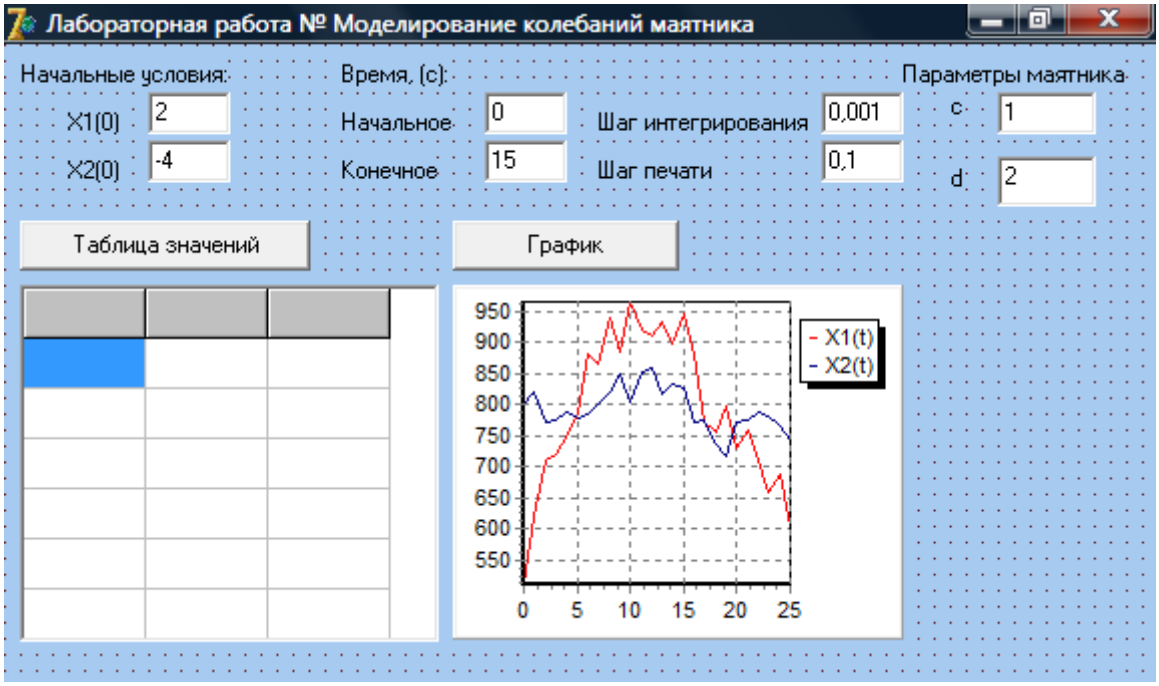


Рис. 5.5. Внешний вид программы

Ниже приведен исходный код программы. Головной модуль *Unit1* использует модуль *Eyler*, в котором реализованы процедуры для вычисления правых частей системы уравнений колебаний маятника и численного интегрирования по модифицированному методу Эйлера.

Файл *Unit1.h*:

```
/* Программа для численного решения систем
   обыкновенных дифференциальных уравнений модифицированным
   методом Эйлера (модель маятника).
 Задание: решить систему ОДУ:
```

```
 dx1/dt=x2,
 dx2/dt=-c*x2-d*sin(x1);
 Начальные условия x1(0)=2, x2(0)=-4 */

#ifndef Unit1H
#define Unit1H

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Chart.hpp>
#include <ExtCtrls.hpp>
#include <Grids.hpp>
#include <TeEngine.hpp>
#include <TeeProcs.hpp>
#include <Series.hpp>

class TForm1 : public TForm
{
__published:   // IDE-managed Components
        TLabel *Label1;
        TLabel *Label2;
        TLabel *Label3;
        TLabel *Label4;
        TLabel *Label5;
        TLabel *Label6;
        TLabel *Label7;
        TLabel *Label8;
        TLabel *Label9;
        TLabel *Label10;
        TLabel *Label11;
        TButton *Button1;
        TButton *Button2;
```

```
        TStringGrid *StringGrid1;
        TChart *Chart1;
        TEdit *Edit1;
        TEdit *Edit2;
        TEdit *Edit3;
        TEdit *Edit4;
        TEdit *Edit5;
        TEdit *Edit6;
        TEdit *Edit7;
        TEdit *Edit8;
        TLineSeries *Series1;
        TLineSeries *Series2;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
private:  // User declarations
public:   // User declarations
        __fastcall TForm1(TComponent* Owner);
};

extern PACKAGE TForm1 *Form1;
#endif
```

Файл *Unit1.cpp*:

```
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
#include "Eyler.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

double x[N], z[N];
double a,b,h,t,tp,tmax,hp,c,d;
```

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{ /* Таблица значений
     c, d - параметры маятника */
 StringGrid1->Visible = true;
 StringGrid1->Cells[0][0] = "t";
 StringGrid1->Cells[1][0] = "X1(t)";
 StringGrid1->Cells[2][0] = "X2(t)";
/* Начальное и конечное время */
 a = StrToFloat(Edit3->Text);
 b = StrToFloat(Edit4->Text);
/* Шаг интегрирования и шаг печати */
 h = StrToFloat(Edit5->Text);
 hp = StrToFloat(Edit6->Text);
/* Начальные значения */
 x[0] = StrToFloat(Edit1->Text);
 x[1] = StrToFloat(Edit2->Text);
/* Параметры маятника */
 c = StrToFloat(Edit7->Text);
 d = StrToFloat(Edit8->Text);

 f(N, a, c, d, x, z);

 StringGrid1->Cells[0][1] = FloatToStr(a);
 int k=1;
 for(int i=0; i<N; i++)
    StringGrid1->Cells[i+1][k] = FloatToStr(x[i]);
 t=a;
 tp = t+hp;
```

```
 tmax = b+(h/2);

 while(tp<=tmax) {
    k=k+1;
    Eyler(t, tp, h, c, d, N, x);
    StringGrid1->Cells[0][k] = FloatToStr(tp);

    for(int i=0; i<N; i++)
       StringGrid1->Cells[i+1][k] = FloatToStr(x[i]);
    t = tp;
    tp = tp+hp;
 }

 StringGrid1->RowCount = k+1;
}

void __fastcall TForm1::Button2Click(TObject *Sender)
{ /* Метод для построение графика найденных функций */
 int i,n;
 double t, x1, x2;

 Chart1->Visible=true;
 Series1->Clear();
 Series2->Clear();
 n = StringGrid1->RowCount-1;

 for(i=0; i<n; i++) {
    t = StrToFloat(StringGrid1->Cells[0][i+1]);
    x1 = StrToFloat(StringGrid1->Cells[1][i+1]);
    x2 = StrToFloat(StringGrid1->Cells[2][i+1]);
    Series1->AddXY(t, x1, "", clRed);
    Series2->AddXY(t, x2, "", clNavy);
 }
}
```

Файл *Eyler.h*:

```
/* Модуль с процедурами вычисления правых частей системы
 уравнений колебаний маятника и численного интегрирования
 по модифицированному методу Эйлера */
#ifndef EylerH
#define EylerH

const int N=2;

void f(int n, double t, double c, double d, double x[], double z[]);
void Eyler(double a, double b, double h, double c, double d, int n,
double x[]);
#endif
```

Файл *Eyler.cpp*:

```
#pragma hdrstop

#include <Math.h>
#include "Eyler.h"

void f(int n, double t, double c, double d, double x[], double z[])
{ /* правые части уравнения маятника
     x - массив значений искомых функций x[1]=q,x[2]=dq/dt
     z - массив значений производных */
 z[0]=x[1];
 z[1]=-c*x[1]-d*sin(x[0]);
}

void Eyler(double a, double b, double h, double c, double d, int n,
double x[])
{ /* Метод Эйлера модифицированный */
 double t, t1, z[N], x2[N], fi1[N];
 t=a;
 while(t+h<=b) {
    f(n, t, c, d, x, z);
```

```
    t1 = t+h/2;
    for(int i=0; i<n; i++) x2[i]=x[i]+h/2*z[i];
    f(n, t1, c, d, x2, fi1);
    t = t1+h/2;
    for(int i=0; i<n; i++)  x[i]=x[i]+fi1[i]*h;
  }
}
```

5. Контрольный пример

Рассмотрим движение маятника с параметрами c=0.6, d=2 при начальных отклонениях (условиях) $x_1(0)=2$, $x_2(0)=-4$. Границы интервала по времени интегрирования заданы как a=0, b=15. Шаг интегрирования задан как h=0.001 и шаг печати (отображения результата в таблице) задан как h_p=0.1. На рис. 5.6 представлено окно программы с результатами, полученными с помощью программы (таблица значений найденных функций $x_1(t)$, $x_2(t)$ и их графики).

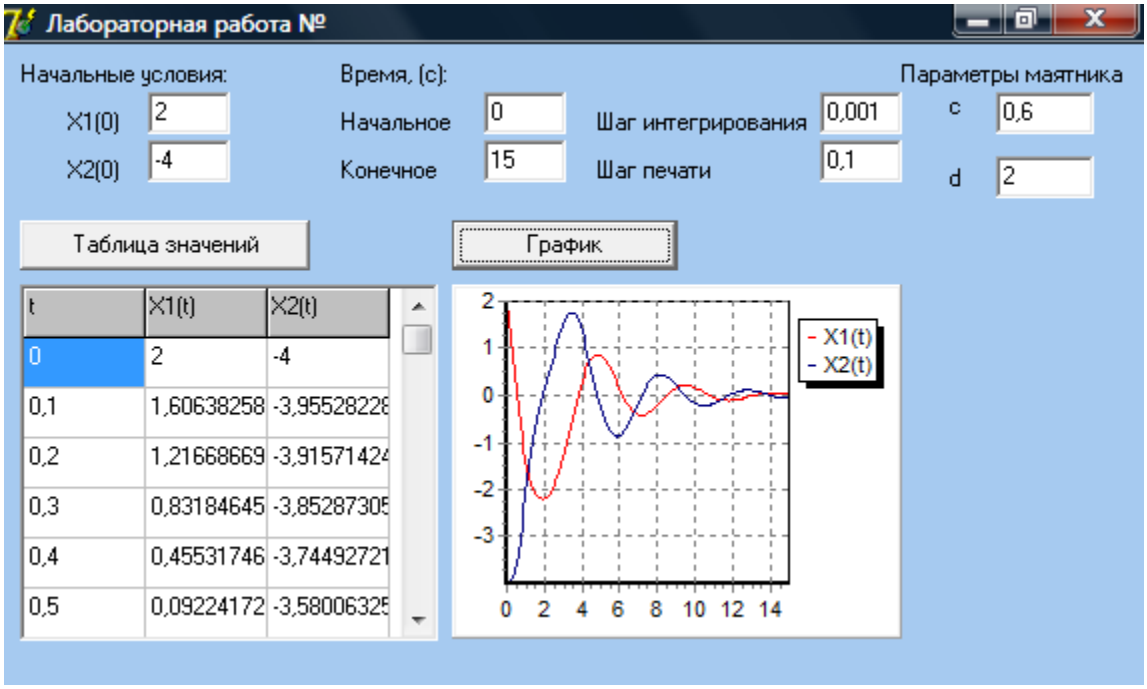


Рис. 5.6. Окно программы с результатами интегрирования уравнений маятника

6. Порядок выполнения работы

1) Для указанного варианта задания из приложения составить программу для численного решения системы дифференциальных уравнений, реализующую вычисление определенного интеграла методом Симпсона.
2) Найти несколько решений дифференциальных уравнений при различных h указанным методом.
3) Оформить отчет о проделанной работе в виде файла. В отчете привести вариант задания, текст программы с комментариями, краткое описание программы и результаты счета с оценкой погрешности.
4) При сдаче работы студент показывает файл отчета и отвечает на контрольные вопросы.

7. Контрольные вопросы

1) Что является решением дифференциального уравнения?
2) Для чего и почему требуется нахождение численного решения дифференциального уравнения?
3) Какие методы численного решения дифференциальных уравнений называются самостартующимися?
4) Как оценивается точность численного решения дифференциального уравнения?
5) В чем суть метода Эйлера и его модификаций?
6) Какова структура программы, реализующей алгоритм метода Эйлера?
7) Какие операторы языка C++ используются в программе?
8) В чем различие между формальными и фактическими параметрами процедур? Между параметрами-переменными и параметрами-значениями?

ЛАБОРАТОРНАЯ РАБОТА № 6. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Цель занятия – ознакомление со сложными структурами данных и их реализацией на языке C++.

1. Общие сведения

В C++ есть возможность по ходу выполнения программы выделять и освобождать необходимую память для размещения в ней различных данных. Таким образом, можно организовывать динамические, т.е. изменяющиеся размеры, структуры данных. Оперативная память при этом используется наиболее эффективным образом. Такая возможность связана с наличием в языке особых типов данных – указателей. Область оперативной памяти, где можно выделять отдельные участки для размещения данных и освобождать их, будем называть динамической областью памяти или динамической памятью.

2. Указатели

Указатель в C++ дает адрес объекта определенного типа, называемого базовым типом.

При определении типа-указателя используется этот базовый тип, после которого ставится признак указателя *.

Пример:

```
double *dpoint;         /* указатель на вещественный тип */
int *ipoint;                    /* указатель на целый тип */
```

Переменная типа-указателя, которая в дальнейшем будет называться просто указателем, является физическим носителем адреса величины базового типа. Значение этой переменной можно задать использованием адресного оператора &, а также присваиванием ей значения другой переменной этого же типа. Переменной типа-указателя можно также присвоить значение *NULL*, означающее

отсутствие ссылки на какой-либо объект (фактически в этом случае переменной присваивается значение *NULL*).

Пример.

```
struct Complex {                                  /* базовый тип */
   double Re, Im;
};
Complex *P1,*P2,*P3,*P4;                  /* указатели на Complex */
int *Adl;                                   /* указатель на целое */
Complex X;
. . . . . .
P1 = new(Complex); /* выделение нового элемента типа Complex */
P2 = &X;                      /* определение адреса переменной X */
P3 = P1;                /* присвоение значения другого указателя */
P4 = NULL;                             /* присвоение значения NULL */
Adl = new(int);           /* выделение нового элемента типа int */
```

Следует иметь в виду, что указатели, ссылающиеся на объекты разных типов, сами являются объектами разных типов и для них недопустима операция присваивания значений друг другу. Указатели одного и того же типа можно сравнивать с помощью операций == и !=.

Чтобы получить значение элемента, с которым связан указатель, следует взять имя указателя и перед ним поставить знак *.

Например,

```
X = *P1;   /* переменной X присваивается значение
              элемента, на который указывает P1 */
*P3 = X;   /* элементу, на который указывает P3
              присваивается значение переменной X */
```

3. Работа с динамической памятью

Использование указателей совместно с операторами *new* и *delete* позволяет осуществлять динамическое распределение памяти.

Оператор *P = new*(*базовый тип*), где *P* – указатель, позволяет выделить область памяти такого размера, в котором можно разместить величину базового типа. Указатель принимает значение адреса выделенной области.

Оператор *delete P*, где *P* – указатель, позволяет освободить область памяти, на которую указывает указатель *P*, для последующего использования. После выполнения оператора значение указателя *P* становится неопределенным.

Существует и другая возможность работы с динамической памятью – использовать функции *malloc* и *free*.

Вызов функции *P = malloc*(*size*), где *P* – переменная типа указатель, а *size* - выделяемая область памяти в байтах, позволяет выделить в динамической памяти область необходимого размера, при этом адрес выделенной области присваивается переменной *P*.

Функция *free*(*P*) – освобождает занятую область памяти с адресом, задаваемым переменной *P*. Эта область становится свободной для повторного использования, а указатель *P* становится неопределенным.

4. Обработка сложных структур данных

Возможность выделения и освобождения области памяти позволяет создавать структуры данных с переменным числом элементов – динамические структуры. Чаще всего используют т.н. связанные структуры, когда среди элементов устанавливается некоторая иерархия, например, наподобие генеалогического дерева. Среди таких структур наибольшее распространение в различных практических приложениях получили линейные структуры (списки) и структуры-деревья. В связных структурах обычно используют однотипные элементы. Каждый элемент имеет две части:

- информационную часть, которая содержит всю информацию о том или ином объекте (например, если это структура целых чисел, то значение конкретного числа);

– ссылку на соседний элемент (соседние элементы) в конкретной иерархии элементов (например, если структура является списком, то ссылка на элемент, стоящий в списке непосредственно за данным элементом, а может и на предыдущий элемент, если это двусвязный список).

Наиболее удобно для фиксации такой информации использовать структуру.

Пример. Тип элемента структуры, содержащего символьную информацию в виде строки.

```
struct Elem {
   AnsiString Info;
   Elem *Point;            /* указатель на элемент */
};
```

При работе с динамическими структурами данных выполняются следующие основные операции:

– добавление элемента структуры;
– исключение элемента структуры;
– поиск элемента структуры по какому-то признаку.

В разных структурах эти операции выполняются по-своему. В качестве примера рассмотрим добавление и исключение элементов из линейного списка, называемого очередью, – структуры, в которой добавление элементов осуществляется с одной стороны, а исключение – с другой стороны списка.

Пусть элемент структуры имеет информационную часть, представляющую собой одно целое число. Тогда тип такого элемента может выглядеть следующим образом:

```
struct Elem {
   int Info;
   Elem *Point;
};
```

Для работы с очередью введем следующие переменные:

```
Elem *Po, *PoB, *PoE;
int X;
```

Предположим, что сначала в очереди нет элементов – очередь пуста. Этот факт можно зафиксировать следующим образом (рис. 6.1а):

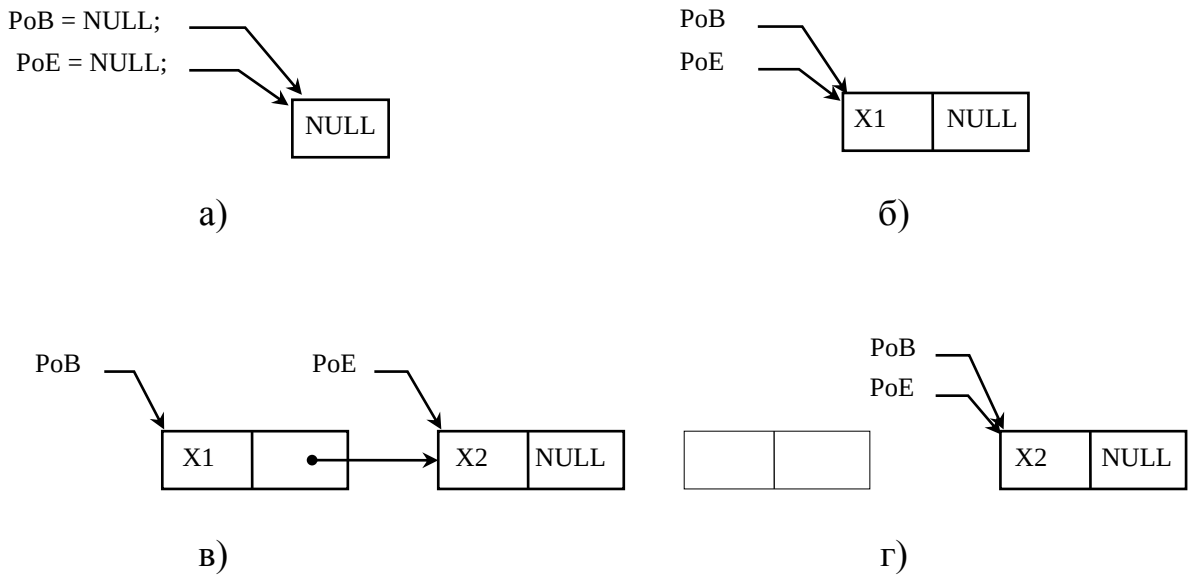


Рис. 6.1. Последовательность действий при работе с очередью:
а) нет элемента; б) добавление первого элемента; в) добавление второго элемента; г) удаление первого элемента

Введем в очередь первый элемент. Для этого можно выполнить следующие действия:

```
Read(X);                /* чтение значения первого элемента */
Po = new(Elem);         /* выделение области под элемент */
Po->Info = X;           /* информационная часть */
Po->Point = NULL;       /* нет следующего элемента */
PoB = Po;               /* указатель начала очереди */
PoE = Po;               /* указатель конца очереди */
```

После выполнения этих операторов возникает ситуация, изображенная на рис. 6.1б). На этом рисунке прямоугольником указан элемент очереди. Левая

часть прямоугольника представляет собой информационную часть, а правая – ссылку на следующий элемент. Стрелками изображены указатели.

Для добавления следующего элемента можно записать:

```
Read(X);
Po = New(Elem);
Po->Info = X;
Po->Point = NULL;
PoB->Point = Po;            /* установление связей} */
PoE = Po;
```

Полученная ситуация изображена на рис. 6.1в).

Рассмотренные выше две последовательности операторов по добавлению элементов похожи друг на друга и могут выполняться в цикле (с небольшой модификацией).

Посмотрим теперь, как удалить первый элемент из очереди. Для этого нужно выполнить следующие действия:

```
if(PoB!=NULL) {
        Po = PoB->Point;   /* указатель на второй элемент */
        delete PoB;        /* освобождение области */
        PoB = Po
}
```

Полученная ситуация изображена на рис. 6.1г).

При освобождении элементов структуры следует особое внимание уделять сохранению необходимых связей, т.к. в противном случае можно потерять часть или даже все оставшиеся элементы структуры. В рассмотренном примере для этой цели использовалась переменная *Po*.

5. Пример программы для обработки списка записей

Пусть требуется разработать программу для обработки списка записей, представляющих собой структуру данных, в которой добавление элементов производится в конец списка (как в очереди), а удаление элементов – по

заданному ключу (значению информационного поля). Для ввода и отображения значений информационных полей списка в форме разместим следующие визуальные объекты Borland C++ Builder: *Edit1* - *Edit3* – для задания значений информационному полю соответственно первого, добавляемого и заданного элемента; *Memo1* – для вывода и отображения списка (значений информационных полей элементов списка), *Label1* - *Label5* – для ввода и отображения пояснений назначения выбранных объектов. Для управления работой программы в форме разместим объекты *Button1* - *Button3* – для задания действий, соответствующих операциям по обработке списка: создание (инициализация) списка из одного элемента, добавление нового элемента в конец списка, удаление заданного элемента списка; *Button4* – для выхода из программы.

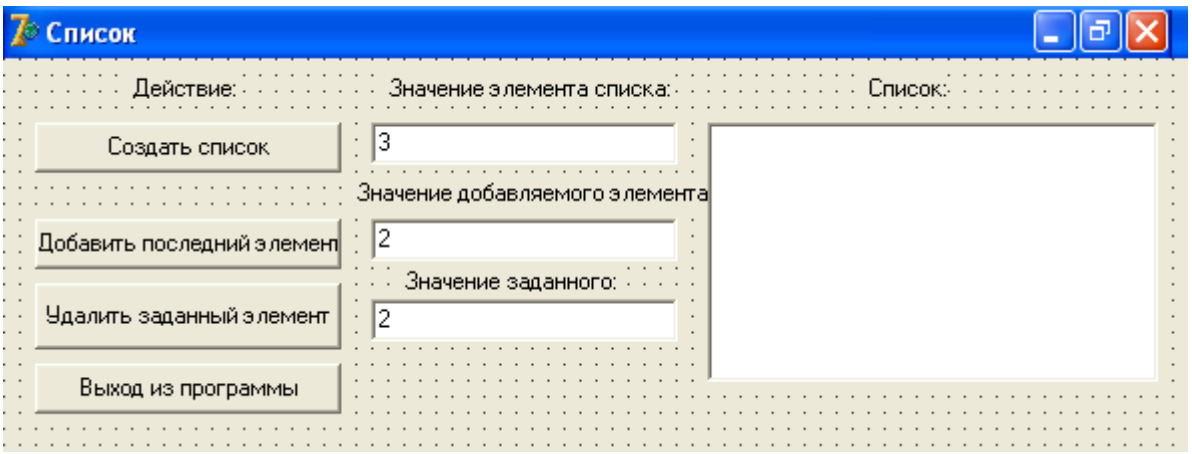


Рис. 6.2. Окно формы с выбранными объектами

Для того чтобы программа в ответ на нажатие кнопки выполняла именно то действие, которое требуется, в окне кода вносим соответствующий программный код для обработчика события *Click* для каждой кнопки.

В результате получаем следующий исходный код программы для обработки списка записей.

Файл *Unit.h*:

```
/* Программа обработки списка записей */
#ifndef Unit1H
#define Unit1H
```

```
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>

class TForm1 : public TForm
{
__published:   // IDE-managed Components
        TButton *Button1;
        TButton *Button2;
        TButton *Button3;
        TButton *Button4;
        TLabel *Label1;
        TLabel *Label2;
        TEdit *Edit1;
        TEdit *Edit2;
        TEdit *Edit3;
        TLabel *Label3;
        TLabel *Label4;
        TMemo *Memo1;
        TLabel *Label5;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall Button3Click(TObject *Sender);
        void __fastcall Button4Click(TObject *Sender);
private:  // User declarations
public:        // User declarations
        __fastcall TForm1(TComponent* Owner);
};

extern PACKAGE TForm1 *Form1;
#endif
```

Файл *Unit.cpp*:

```
#include <vcl.h>
#pragma hdrstop

#include "SpisokOper.h"
#include "Unit1.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

bool made;
Node *Ph = NULL;

__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{ /* Метод для инициализации списка из одного элемента */
 Memo1->Lines->Clear();
 Ph = init(Ph, &made, Edit1);
 outp(Ph, 1, Memo1);
}

void __fastcall TForm1::Button2Click(TObject *Sender)
{ /* Метод для идобавления элемента в конец списка */
 if(made) {
    Memo1->Lines->Clear();
    addLast(Ph,Edit2);
    Memo1->Lines->Clear();
    outp(Ph, 1, Memo1);
 }
```

```
 else {
    ShowMessage("Сначала создайте список!");
    return;
 }
}

void __fastcall TForm1::Button3Click(TObject *Sender)
{ /* Метод для удаления заданного элемента из списка */
 if(Edit3->Text.IsEmpty()) {
    ShowMessage("Неверный ввод! Повторите.");
    return;
 }
 Ph = delet(Ph, Edit3->Text);
 Memo1->Lines->Clear();
 outp(Ph, 1, Memo1);
}

void __fastcall TForm1::Button4Click(TObject *Sender)
{ /* Метод для удаления списка (освобождение паямяти) */
 disp(Ph);
 Close();
}
```

Файл *SpisokOper.h*:

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * *
 Модуль с процедурами обработки списка:
 Инициализации списка
 Добавление элемента в конец списка
 Удаление заданного элемента
 Отображение (печать) списка в окне
 Удаление списка (освобождение памяти)
 * * * * * * * * * * * * * * * * * * * * * * * * * */
#ifndef SpisokOperH
#define SpisokOperH
```

```
struct Node {
      AnsiString inf;
      Node *next;
};

Node *init(Node *p, bool *pmade, TEdit *Edt);
void outp(Node *p, int t, TMemo *mem);
void addLast(Node *p, TEdit *Edt);
Node *delet(Node *p, AnsiString a);
void disp(Node *p);
#endif
```

Файл *SpisokOper.cpp*:

```
#include <vcl.h>
#pragma hdrstop

#include "SpisokOper.h"

void outp(Node *p, int t, TMemo *mem)
{ /* * * * * * * * * * * * * * * * * * * * * * * * *
  Вывод списка в окно Mem
  p - указатель на первый элемент списка
  * * * * * * * * * * * * * * * * * * * * * * * * */
 Node *q;
 q = p;
 if(q!=NULL) {
    mem->Lines->Add(q->inf);
    outp(q->next, t+1, mem);
 }
}

Node *init(Node *p, bool *pmade, TEdit *Edt)
{ /* * * * * * * * * * * * * * * * * * * * * * * *
  Инициализация списка из одного элемента
  p - указатель на первый элемент списка
```

```
  * * * * * * * * * * * * * * * * * * * * * * * * */
 if(*pmade) {
    ShowMessage("Список был уже создан ранее!");
    return p;
 }
 if(Edt->Text.IsEmpty()) {
    ShowMessage("Неверный ввод! Повторите.");
    return NULL;
 }
 p = new(Node);
 p->inf = Edt->Text;
 p->next = NULL;
 *pmade = true;
 return p;
}

void addLast(Node *p, TEdit *Edt)
{ /* * * * * * * * * * * * * * * * * * * * * * * * *
  добавление элемента в конец списка
  p - указатель на некоторый элемент списка
  * * * * * * * * * * * * * * * * * * * * * * * * */
 Node *q, *r;

 r = p->next;
 if(r==NULL) {
    if(Edt->Text.IsEmpty()) {
       ShowMessage("Неверный ввод! Повторите.");
       return;
    }
    q = new(Node);
    q->inf = Edt->Text;
    p->next = q;
    q->next = NULL;
 }
```

```
 else  addLast(r, Edt);
}

Node *delet(Node *p, AnsiString a)
{ /* * * * * * * * * * * * * * * * * * * * * * * * * *
  поиск и удаление заданного элемента из списка
  p - указатель на первый элемент списка
  * * * * * * * * * * * * * * * * * * * * * * * * * */
 Node *q, *q0;
 q = p;
 while(q!=NULL)
    { if(q->inf==a)
         { ShowMessage("Удален элемент '" + q->inf + "' Нажми OK");
           if(p==q) p = q->next;  else q0->next = q->next;
           delete q;
           return p;
         }
      q0 = q;
      q = q->next;
    }
}
void disp(Node *p)
{ /* * * * * * * * * * * * * * * * * * * * * * * * * *
   освобождение памяти из под списка
   p - указатель на первый элемент списка
   * * * * * * * * * * * * * * * * * * * * * * * * * */
 Node *q, *r;
 q = p;
 if(q!=NULL) {
    r = q->next;
    delete q;
    disp(r);
 }
 p = NULL;
```

```
}
```

В модуле *SpisokOper* приведенной программы в виде процедур реализованы следующие операции: создание списка из одного элемента (функция *init*), добавление элемента в конец списка (функция *addLast*), поиск и удаление найденного элемента из списка (функция *delet*), вывод значений элементов списка на экран монитора (функция *outp*), удаление списка и освобождение памяти (функция *disp*). При запуске программы на выполнение открывается окно, показанное на рис. 6.3. Прежде чем начать работу со списком, его следует создать. Для этого в поле объекта *Edit1* вводится значение информационного поля первого элемента и нажимается кнопка *Button1*. В результате в динамической памяти будет создан список из одного элемента, значения информационного поля которого будет отображено в объекте *Memo1*. При добавлении элемента в список следует сначала ввести значение его информационного поля в объект *Edit2*, после чего нажать кнопку *Button2*. Новый элемент будет добавлен в конец списка, а в поле объекта *Memo1* отобразится значения информационных полей имеющихся элементов списка. Для удаления элемента сначала в объекте *Edit3* указывается значение заданного элемента, после чего нажимается кнопка *Button3*. В результате будет удален первый из элементов списка, значение информационного поля которого совпадает с заданным, а в объекте *Memo1* будут отображены значения информационных полей всех оставшихся элементов списка.

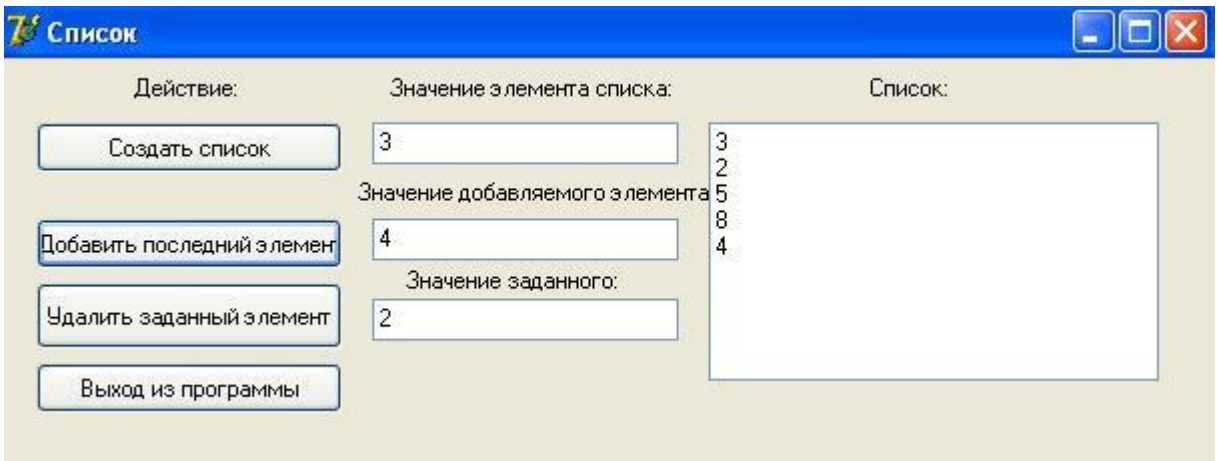


Рис. 6.3. Окно программы для обработки списка записей

6. Варианты заданий

1) Стек: создание, добавление, исключение элементов и функция, определяющая не пуст ли стек (добавление в начало, удаление первого элемента списка).
2) Очередь: создание, добавление, исключение элементов и функция, определяющая не пуста ли очередь (добавление в конец списка, удаление первого элемента).
3) Список: создание, удаление первого элемента, добавление после заданного элемента.
4) Список: создание, удаление последнего элемента, добавление перед заданным.
5) Список: создание, добавление первого элемента, удаление после заданного.
6) Список: создание, добавление последнего элемента, удаление перед заданным.
7) Список: создание, добавление элементов в конец списка, поиск по значению информационного поля максимального и минимального элементов и их перестановка.
8) Двусвязный список: создание, добавление в начало, удаление первого элемента списка.
9) Двусвязный список: создание, добавление в конец списка, удаление первого элемента.
10) Двусвязный список: создание, удаление первого элемента, добавление после заданного.
11) Двусвязный список: создание, удаление последнего элемента, добавление перед заданным.
12) Двусвязный список: создание, добавление в начало списка, удаление после заданного.
13) Двусвязный список: создание, добавление в конец списка, удаление элемента перед заданным.

14) Двусвязный список: создание, добавление элементов в конец списка, поиск по значению информационного поля максимального и минимального элементов и их перестановка.
15) Двусвязный список: создание и упорядочение элементов.
16) Упорядоченный по значению информационного поля список: создание, добавление, элементов с сохранением упорядоченности.

7. Порядок выполнения работы

1) Для указанного варианта задания составить программу на Паскале. В программе предусмотреть меню для выбора операций с элементами списка, а также вывод списка на печать.
2) Отладить программу и проверить правильность ее работы для различных последовательностей выбора операций.
3) Оформить в виде файла отчет о проделанной работе. В отчете привести вариант задания, текст программы с комментариями, краткое описание программы.
4) При сдаче работы студент показывает файл отчета, демонстрирует работу программы и отвечает на контрольные вопросы.

8. Контрольные вопросы

1) Каковы преимущества работы с динамической памятью?
2) Как получить значение элемента, с которым связан указатель?
3) Что такое указатель? Что является его значением?
4) Как описывается тип-указатель?
5) Перечислите способы присвоения значения переменной-указателю.
6) Как получить значение элемента, с которым связан указатель?
7) Опишите процедуры работы с динамической памятью.
8) Что понимается под динамическими структурами данных?
9) Какие связанные структуры Вы знаете?

10) Какова структура элемента связанной структуры?
11) Какие основные операции выполняются при работе с динамическими структурами данных?

ЛАБОРАТОРНАЯ РАБОТА №7. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ. ОБРАБОТКА СПИСКА ОБЪЕКТОВ

1. Общие сведения

Объектный подход к проектированию основан на:

- выделении классов объектов;
- установлении характерных свойств объектов и методов их обработки;
- создании иерархии классов, наследовании свойств объектов и методов их обработки.

Объект – совокупность свойств (параметров) определенных сущностей и методов их обработки (программных средств).

Каждый объект объединяет как данные, так и программы обработки этих данных и относится к определенному классу. С помощью класса один и тот же программный код можно использовать для относящихся к нему различных объектов.

Свойство – характеристика объекта, его параметр. Все объекты наделены определенными свойствами, которые в совокупности выделяют объект из множества других объектов.

Метод – программа действий над объектом или его свойствами.

Метод рассматривается как программный код, связанный с определенным объектом; осуществляет преобразование свойств, изменяет поведение объекта.

Объекты могут объединяться в классы.

Класс – совокупность объектов, характеризующихся общностью свойств и применяемых методов обработки.

При применении объектного подхода проводится объектно-ориентированный анализ предметной области, который направлен на выделение совокупности изучаемых объектов, определении свойств объектов и методов их

обработки, установлении их взаимосвязей. Эта методология базируется на основе понятий классов и объектов, составляющих словарь предметной области.

Объектный подход образует концептуальную основу для объектно-ориентированной методологии: он включает принципы абстрагирования, ограничения доступа, модульности, иерархии, типизации, параллелизма и устойчивости.

Объектно-ориентированные программирование – это методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного класса, а классы образуют иерархию на принципах наследования.

Объект представляет собой особый опознаваемый предмет, блок или сущность (реальную или абстрактную), имеющую важное функциональное назначение в данной предметной области.

Технология объектно-ориентированного программирования обеспечивает выполнение важнейших свойств объектного подхода:

- инкапсуляция;
- наследование;
- полиморфизм.

Инкапсуляция означает совмещение структур данных (свойств) и программ для их обработки (методов) в абстрактных тиках данных – классах объектов.

На основе созданного класса могут создаваться классы потомки, наследующие свойства и методы класса предка. Механизм *наследования* позволяет переопределить или добавить новые данные (свойства) и методы их обработки.

Полиморфизм – способность объекта реагировать на вызов метода сообразно своему типу, при этом одно и то же имя может использоваться для различных классов объектов.

2. Постановка задачи

Пусть требуется составить программу, которая будет обрабатывать информацию о совокупности однотипных объектов из некоторой предметной области. При этом число объектов заранее не известно и может изменяться с течением времени. Пусть каждый объект есть элемент, имеющий одно информационное поле, характеризующее его свойство. Пусть элементы связаны с помощью поля ссылки. Таким образом, пусть задан список элементов. Поведение рассматриваемого списка будет определяться следующими операциями: создание списка, печать списка, удаление списка, добавление новых элементов в список, удаление заданного элемента из списка.

3. Пример программы для обработки списка объектов

Для представления объектов, представляющих элементы списка, определим в программе класс *CNode*:

```
class CNode {
   public:
      AnsiString inf;
      CNode *next;

      CNode(AnsiString &text);
      ~CNode();
      void Display(TMemo *mem);
};
```

Также как при описании типа структура в определении класса задаются поля-характеристики (*inf* – информационное поле элемента и *next* – указатель на следующий элемент списка). Однако для описания поведения объектов класса *CNode* здесь определяются методы *CNode*() (конструктор элемента списка) и *~CNode*() (деструктор элемента списка*), Display*(…) (отображение в *TMemo* содержимого элемента списка).

Методы в объектно-ориентированном программировании (ООП) это обычные функции, заголовки которых перечисляются в описании класса. Таким образом, при определении данных в ООП одновременно определяются и методы для работы с этими данными. Совмещение описания данных и методов называется *инкапсуляцией*.

Для представления списка элементов введем класс *Spisok*:

```
class Spisok {
      CNode *Head;

   public:
      Spisok() { Head=0; }
     ~Spisok() { Clear(); }
      void AddLast(AnsiString text);
      void DelNode(AnsiString text);
      void DelLast();
      void Clear();
      void Display(TMemo *mem);
};
```

Здесь поле `Head` представляет собой указатель на первый элемент списка. Если *Head* равен 0, то в списке нет ни одного элемента, т.е. он пуст. В классе *Spisok* также определены методы: *Spisok*() (конструктор), *~Spisok*`()` (деструктор), *AddLast*(…) (добавление нового элемента в коней списка), *DelNode*(…) (удаление указанного элемента списка), *DelLast*() (удаление последнего элемента списка), *Clear*() (удаление всех элементов списка) и *Display*(…) (отображение содержимого всех элементов списка в объекте *TMemo*).

Реализация методов, определенных в описании класса, может производиться непосредственно при определении класса или разделе описаний (вместе с остальными функциями программы). При этом к имени каждой функции, определенной как метод, присоединяется имя класса, для которого этот метод определен. Вызов метода осуществляется с помощью вызова функции, реализующей

данный метод. При этом перед именем метода добавляется имя экземпляра класса, который будет обрабатываться этим методом.

В качестве примера приведем программу, аналогичную программе из лабораторной работы 6, переделанную с учетом объектно-ориентированного программирования. Здесь для представления списка вместо записи определяется класс *Spisok*. Для представления операций, описывающих поведения объектов класса здесь вместо процедур определяются соответствующие методы. Экземпляр класса *Spisok* определяется с помощью указателя *S*.

Файл *Union1.h*:

```
/* Программа обработки списка записей */
#ifndef Unit1H
#define Unit1H

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>

class TForm1 : public TForm
{
__published:  // IDE-managed Components
        TButton *Button1;
        TButton *Button2;
        TButton *Button3;
        TButton *Button4;
        TLabel *Label1;
        TLabel *Label2;
        TEdit *Edit1;
        TEdit *Edit2;
        TEdit *Edit3;
        TLabel *Label3;
        TLabel *Label4;
```

```
        TMemo *Memo1;
        TLabel *Label5;
        TButton *Button5;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall Button3Click(TObject *Sender);
        void __fastcall Button4Click(TObject *Sender);
        void __fastcall Button5Click(TObject *Sender);
private: // User declarations
public:       // User declarations
        __fastcall TForm1(TComponent* Owner);
};

extern PACKAGE TForm1 *Form1;
#endif
```

Файл *Union1.cpp*:

```
#include <vcl.h>
#pragma hdrstop

#include "SpisokOper.h"
#include "Unit1.h"

#pragma resource "*.dfm"
TForm1 *Form1;

Spisok *S=0;

__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}

void __fastcall TForm1::Button1Click(TObject *Sender)
```

```
{ /* Метод для инициализации списка из одного элемента */
 if(S) { ShowMessage("Список уже создан");  return; }

 S = new Spisok();
 S->AddLast(Edit1->Text);
 S->Display(Memo1);
}

void __fastcall TForm1::Button2Click(TObject *Sender)
{ /* Метод для идобавления элемента в конец списка */
 if(!S) { ShowMessage("Список не был создан");  return; }

 S->AddLast(Edit2->Text);
 S->Display(Memo1);
}

void __fastcall TForm1::Button3Click(TObject *Sender)
{ /* Метод для удаления заданного элемента из списка */
 S->DelNode(Edit3->Text);
 S->Display(Memo1);
}

void __fastcall TForm1::Button4Click(TObject *Sender)
{ /* Метод для удаления последнего элемента из списка */
 S->DelLast();
 S->Display(Memo1);
}

void __fastcall TForm1::Button5Click(TObject *Sender)
{ /* Метод для удаления списка (освобождение паямяти) */
 delete S;
 Close();
}
```

Файл *SpisokOper.h*:

```
 /* * * * * * * * * * * * * * * * * * * * * * * * * * *
      Модуль с процедурами обработки списка:
  * * * * * * * * * * * * * * * * * * * * * * * * * * */
 #ifndef SpisokOperH
 #define SpisokOperH

class CNode {
    public:
       AnsiString inf;
       CNode *next;
       CNode(AnsiString &text) { inf=text;  next=0; }
       ~CNode() {}
       void Display(TMemo *mem);  /* Отображение в TMemo содержимого
элемента списка */
 };

 class Spisok {
       CNode *Head;

    public:
      Spisok() { Head=0; }
     ~Spisok() { Clear(); }
      void AddLast(AnsiString text);  /* Добавление элемента в конец
списка */
      void DelNode(AnsiString text);  /* Удаление заданного элемента
списка */
      void DelLast();  /* Удаление последнего элемента списка */
      void Clear();    /* Удаление всех элементов списка */
      void Display(TMemo *mem);  /* Отображение в TMemo содержимого
всех элементов списка */
 };
 #endif
```

Файл *SpisokOper.cpp*:

```
#include <vcl.h>
#pragma hdrstop

#include "SpisokOper.h"

/* Описание методов класса CNode */
void CNode::Display(TMemo *mem)
{ /* Отображение в объекте Memo содержимого элемента списка */
  mem->Lines->Add(inf);
}

/* Описание методов класса Spisok */
void Spisok::AddLast(AnsiString text)
{ /* Добавление элемента в конец списка */
 if(text.IsEmpty()) {
    ShowMessage("Данные отсутствуют");
    return;
 }
 CNode *np = new CNode(text);     /* Создание нового элемента */

 if(Head) { /* Если список не пустой */
    CNode *p;
    for(p=Head; p->next; p = p->next);  /* Поиск последнего элемента
в списке */
    p->next = np;  /* Добавляем новый элемент в конец списка */
  }
  else  Head = np; /* Если список пустой */
}

void Spisok::DelNode(AnsiString text)
{ /* Поиск и удаление заданного элемента из списка */

 /* Перебираем все элементы списка */
```

```
 for(CNode *prevp=0, *p=Head; p;  p= p->next) { /* Здесь p-указатель
на текущий элемент, prevp - указатель на предыдущий элемент */
    if(p->inf==text) {  /* Если найден элемент с искомым содержимым
*/
       /* Исключаем найденный элемент из списка */
       if(prevp) prevp->next = p->next; /* Если элемент в середине
или конце списка */
          else Head = p->next;    /* Если элемент в начале списка */

       delete p;    /* Удаляем найденный элемент */
       return;
    }
    prevp = p;   /* Указатель на текущий элемент становится
указателем на предыдущий */
 }
   /* Сюда попадаем только если элемент не был найден */
 ShowMessage("Элемент = '"+ text +"' не найден");
}

void Spisok::DelLast()
{ /* Удаление последнего элемента в списке */
 if(Head) { /* Если список не пустой */
    CNode *prevp=0, *p; /* Вводим: p-указатель на текущий элемент,
prevp - указатель на предыдущий элемент */
    p = Head;           /* Устанавливаем p на начало списка */
    while(p->next) { prevp = p;   p = p->next; } /* Доходим до
последнего элемента списка */
    delete p;           /* Удаляем последний элемент списка */
/* Если предпоследний элемент отсутствует (prevp==0), то список
пуст, если существует, то записываем 0 в его поле next */
    if(prevp) prevp->next=0;  else Head=0;
 }
}
```

```
void Spisok::Clear()
{ /* Удаление всех элементов спмска */
 CNode *prevp, *p;  /* Вводим: p-указатель на текущий элемент, prevp
- указатель на предыдущий элемент */
 p = Head;         /* Устанавливаем p на начало списка */
 while(p) { /* Перебираем все элементы списка, начиная с первого */
    prevp = p;     /* Сохраняем указатель на текущий элемент в
качестве в предыдущего */
    p = p->next;  /* Переходим к следующему элементу */
    delete prevp; /* Удаляем предыдущий элемент списка */
 }
 Head = 0;           /* Список пуст */
}

void Spisok::Display(TMemo *mem)
{ /* Отображение в объекте Memo содержимого всех элементов списка */
 mem->Clear();       /* Очистка объекта Memo */
 if(!Head) return;  /* Если список пуст, то завершаем работу */

 for(CNode *p=Head; p; p= p->next)  /* Перебираем все элементы
списка, начиная с первого */
    p->Display(mem);  /* Отображаем в Memo содержимого текущего
элемента списка */
}
```

4. Варианты заданий

1) Стек: создание, добавление, исключение элементов и функция, определяющая не пуст ли стек (добавление в начало, удаление первого элемента списка). Упорядочение списка.
2) Очередь: создание, добавление, исключение элементов и функция, определяющая не пуста ли очередь (добавление в конец списка, удаление первого элемента). Упорядочение списка.

3) Список: создание, удаление первого элемента, добавление после заданного элемента. Упорядочение списка.
4) Список: создание, удаление последнего элемента, добавление элемента перед заданным. Упорядочение списка.
5) Список: создание, добавление первого элемента, удаление после заданного. Упорядочение списка.
6) Список: создание, добавление последнего элемента, удаление перед заданным. Упорядочение списка.
7) Список: создание, добавление элементов в конец списка, поиск по значению информационного поля максимального и минимального элементов и их перестановка. Упорядочение списка.
8) Двусвязный список: создание, добавление в начало, удаление первого элемента списка. Упорядочение списка.
9) Двусвязный список: создание, добавление в конец списка, удаление первого элемента. Упорядочение списка.
10) Двусвязный список: создание, удаление первого элемента, добавление после заданного. Упорядочение списка.
11) Двусвязный список: создание, удаление последнего элемента, добавление элемента перед заданным. Упорядочение списка.
12) Двусвязный список: создание, добавление в начало списка, удаление после заданного. Упорядочение списка.
13) Двусвязный список: создание, добавление в конец списка, удаление элемента перед заданным. Упорядочение списка.
14) Двусвязный список: создание, добавление элементов в конец списка, поиск по значению информационного поля максимального и минимального элементов и их перестановка. Упорядочение списка.
15) Двусвязный список: создание, добавление после первого элемента, удаление первого элемента из пары рядом стоящих одинаковых (равные значения информационного поля) элементов. Упорядочение списка.

16) Двусвязный список: создание, добавление перед последним элементом, удаление всех элементов со значением информационного поля равным заданному значению. Упорядочение списка.
17) Двусвязный список: создание, добавление после второго элемента, удаление элемента со значением информационного поля равным значению информационного поля предыдущего элемента. Упорядочение списка.
18) Двусвязный список: создание и упорядочение элементов.
19) Упорядоченный по значению информационного поля список: создание, добавление, элементов с сохранением упорядоченности.
20) Двусвязный список: создание, добавление элемента между двумя одинаковыми по значению информационного поля, удаление первого элемента, упорядочение элементов.

5. Порядок выполнения работы

1) Для указанного варианта задания составить программу на Паскале. В программе предусмотреть меню для выбора операций с элементами списка, а также вывод списка на печать.
2) Отладить программу и проверить правильность ее работы для различных последовательностей выбора операций.
3) Оформить в виде файла отчет о проделанной работе. В отчете привести вариант задания, текст программы с комментариями, краткое описание программы.
4) При сдаче работы студент показывает файл отчета, демонстрирует работу программы и отвечает на контрольные вопросы.

6. Контрольные вопросы

1) Каковы преимущества объектного подхода?
2) Что такое класс?
3) Какие классы рассматриваются в программе?

4) Какова структура класса *CNode*?
5) Какова структура класса *Spisok*?
6) Зачем заголовки функций приведены в определении класса?
7) Почему и для чего к имени метода при его реализации присоединяется имя класса?
8) Чем являются в программе *inf*, *next*, *p*?
9) Как определить характеристики и методы класса?

СПИСОК ЛИТЕРАТУРЫ

1. Шилд Г. Полный справочник по C++, 4-е издание.: Пер. с англ. – М.: Издательский дом "Вильямс", 2006. – 800 с.

2. Вальпа О.Д. Borland C++ Builder. Экспресс-курс. – СПб.: БХВ-Перербург, 2006. – 224 с.

3. Хьюз Дж., Мичтом Дж. Структурный подход к программированию. – М.: Мир, 1980. – 280 с.

4. Кнут Д. Искусство программирования для ЭВМ. - Т.1. Основные алгоритмы. – М.: Мир, 1976. – 735 с.

5. Вирт Н. Алгоритмы + структуры данных = программа. – М.: Мир, 1989. – 360 с.

6. Морозов В.П., Шураков В.В. Основы алгоритмизации алгоритмические языки и системное программирование. – М.: Финансы и статистика, 1994. – 224 с.

7. Современные численные методы решения обыкновенных дифференциальных уравнений /Под ред. Дж. Холла и Дж. Уатта. - М.: Мир, 1979. – 312с.

8. Болски М.И. Язык программирования Си. Справочник: Пер. с англ. – М.: Радио и связь, 1988. – 96 с.

9. Страуструп Б. Язык программирования C++. Специальное издание.: Пер. с англ. – СПб.: Бином, Невский Диалект, 2008. – 1104 с.

10. Давыдов В.Г. Программирование и основы алгоритмизации: Учеб. пособие. – М.: Высш. шк., 2003. – 447 с.

ОГЛАВЛЕНИЕ