Документ подписан простой электронной подписью
Информация о владельце:
ФИО: Баламирзоев Назим Лиодинович
Должность: И.о. ректора
Дата подписания: 21.08.2023 02:39:16
Уникальный программный ключ:
2a04bb882d7edb7f479cb266eb4aaaedebeea849

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

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

**«ДАГЕСТАНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ »**

**Канаев М.М., Нурмагомедов И.С.**

# Технология программирования на C, C++ и C++Builder

**Учебное пособие**

для выполнения лабораторных работ по дисциплине
«Информатика и программирование»

Махачкала-2023

**УДК.681.3.06(075)**

***Канаев М.М., Нурмагомедов И.С***. Технология программирования на С, С++ и С++Builder. Учебное пособие.
ФГБОУ ВО ДГТУ: -Махачкала, 2023, 318 с.

В учебном пособии рассмотрен язык программирования С. С++, на примерах выполнения лабораторных работ. Излагаются основные теоретические положения языка, принципы объектно- ориентированного программирования, приведены достаточное количество примеров, в объеме, необходимом для изучения языка программирования.

Учебное пособие предназначено для выполнения лабораторного практикума студентами по направлению подготовки бакалавров «Прикладная математика и информатика».

Рекомендуется для студентов всех форм обучения.

Рецензенты: д.т.н. проф. Мелехин В.Б.
к.т.н., проф. Курбанмагомедов К.Д. зав.каф. ИТ, МОГУ.

Печатается согласно постановлению Совета Дагестанского государственного технического университета протокол № от 2022 г. Регистационный № 7231.

**Содержание**

# ВВЕДЕНИЕ

Настоящее учебное пособие предназначенно для расшерения теоретических тем и практического закрепления по программированию на языках высокого уровня.

В учебном пособии рассматривается один из современных языков объектно-ориентированного программирования C++ в среде BUILDER 6.

В первом разделе рассматривается интигрированная среда разработчиков (IDE) C++ Builder 6 и показанно создания первого проекта.

Во втором разделе рассмотренно технология составления блок- схем алгоритмов и программы для линейных структур.

В третьим и четвертом разделе рассмпатривается основные принципы объектно - ориентированного программирования. Рассмотренно основы унифицированного языка моделирования (UML) в том числе и различные диаграммы, а также реализация развевляющих и циклических алгоримов в языков C и C++Builder. Особое внимание удалено объектам и отнощениям в языке программирования.

В последующих разделах подробно рассмотренны создание консольных приложение на языке C++ в среде BUILDER 6, а также работа с матрицамы.

В учебном пособии приведены достоточное количество примеров, которые охватывают все типовые раздели программирование. Приведены блок- схемы алгоритмов.

Учебное пособие предназначено для студентов технических и экономических специальностей вузов, а также может быт использован школьниками, студентами колледжей и другими, кто изучает язык программирования.

# Лабораторная работа 1.
# Интегрированная среда разработчика и проект программы C++ BUILDER 6

**Цель** лабораторной работы: изучение основ интегрированной среды разработчика C++ Builder 6; научиться составлять модули, и изучить каркас простейшего проекта в среде C++ Builder 6; написать и отладить простейший проект.

## 1. 1. Интегрированная среда разработчика (IDE) C++ Builder 6

Интерфейс C++ Builder 6 представляет собой интегрированная среда IDE (англ. Integrated Development Environment) — система программных средств, используемая программистами для разработки программного обеспечения, которая визуально реализуется в виде пяти окон, одновременно раскрытых на экране монитора. Количество, расположение, размер и вид окон может меняться программистом в зависимости от его текущих нужд, что значительно повышает производительность работы. При запуске C++ Builder 6 можно увидеть на экране картинку, подобную представленной на рис. 1.1.

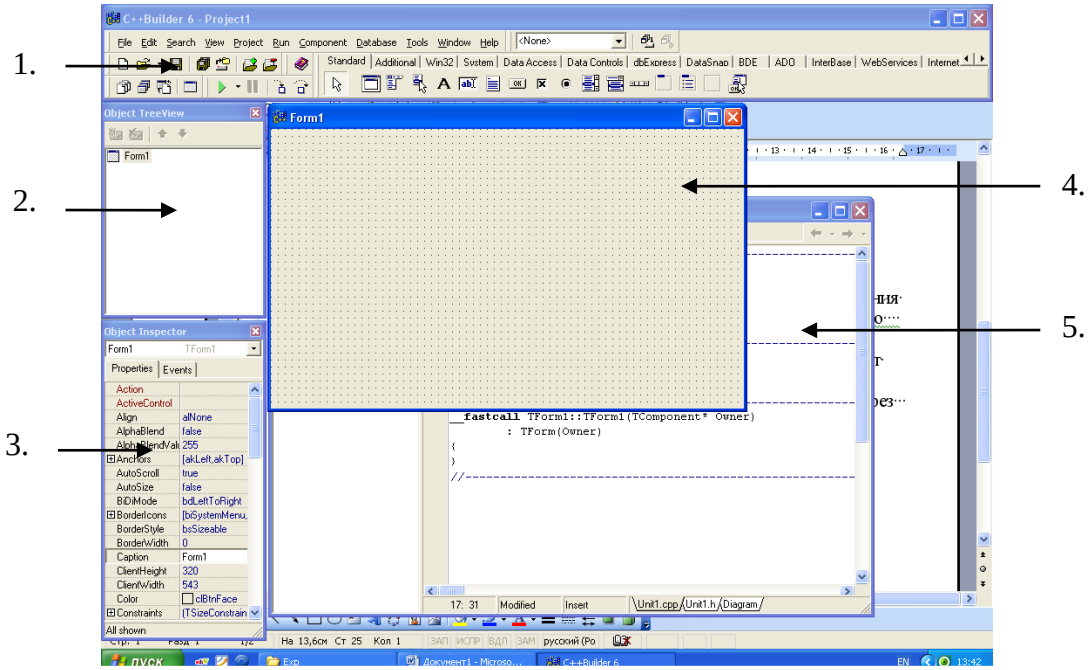



Рис. 1.1. - Главное окно; 2. Окно TreeView;
3. - Окно инспектора объектов; 4. Окно формы;
5 – Окно Редактора кода;

**Главное окно** всегда присутствует на экране и предназначено для управления процессом создания программы. Главное окно (см. рис 1.2.-

1.6.) содержит: Меню главного окна; Главное меню; Панель управления и Палитра компонентов.

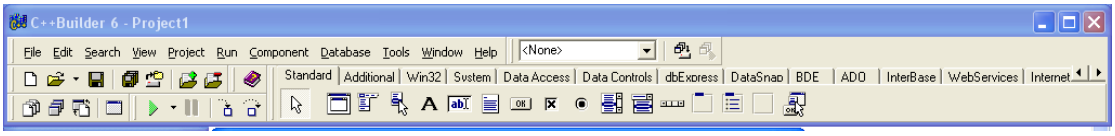



Рис 1.2. Главное окно.

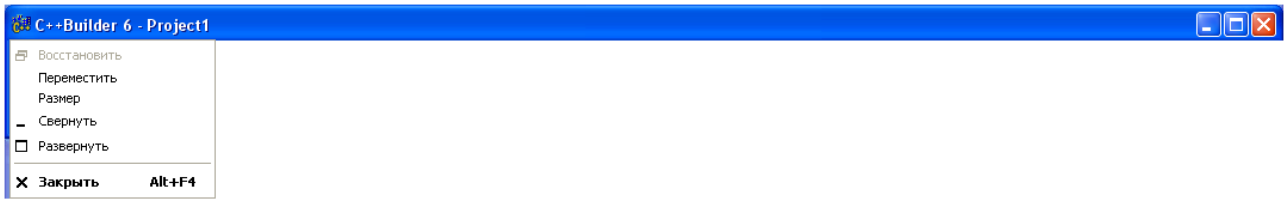



Рис. 1.3. Меню Главного окна и кнопки управления размерами, свертывания и закрытия окна.

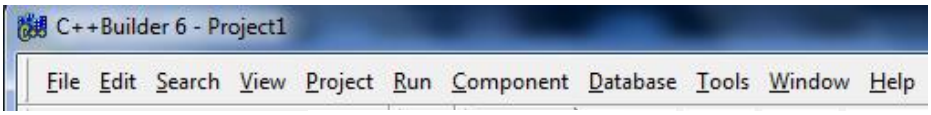



Рис 1.4. Главное меню.

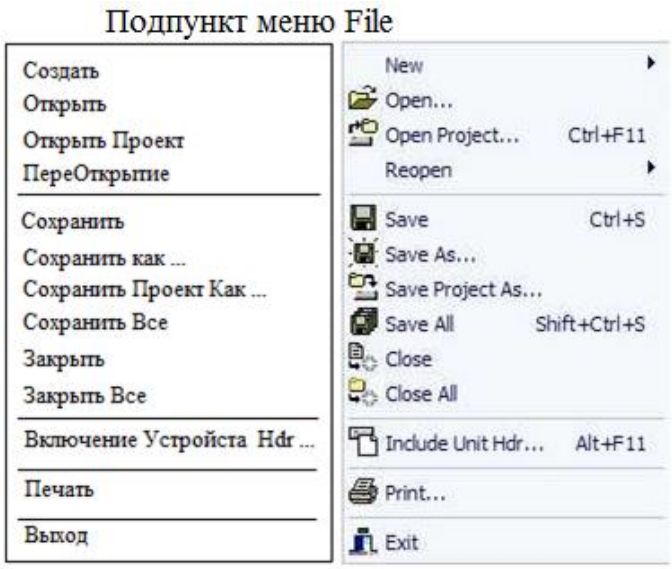



Рис. 1.5. Файловые операция IDE C++ Builder 6

**Главное меню** рис. 1.4. предоставляет доступ к основным командам программы Borland C++ Builder 6.0, объединенным в группы.

Первая группа меню под названием ***File*** (Файл) изображена на рис 1.5.

Команды этого меню осуществляют работу с файлами и предоставляют доступ к операциям создания (New) новых форм (окон) и приложений, открытия (Open), сохранения (Save) и закрытия (Close) файлов и проектов, печати текстов программ (Print) и добавления заголовков модулей (Include Unin Hdr), т.е. файлов с расширением h. Обратите внимание на то, что перед некоторыми командами этого и

последующих меню находятся значки-иконки. Эти же значки изображены на пиктограммах, что позволяет без труда понять их назначение. Кроме того, справа от некоторых команд меню написано соответствие команд горячим клавишам, т. е. клавишам, с помощью которых вызывается данная команда. Еще правее этих надписей кое-где имеются треугольные стрелки, которые говорят о том, что данная команда имеет расширенные возможности, которые становятся доступными после наведения на эту стрелку курсора мышки.

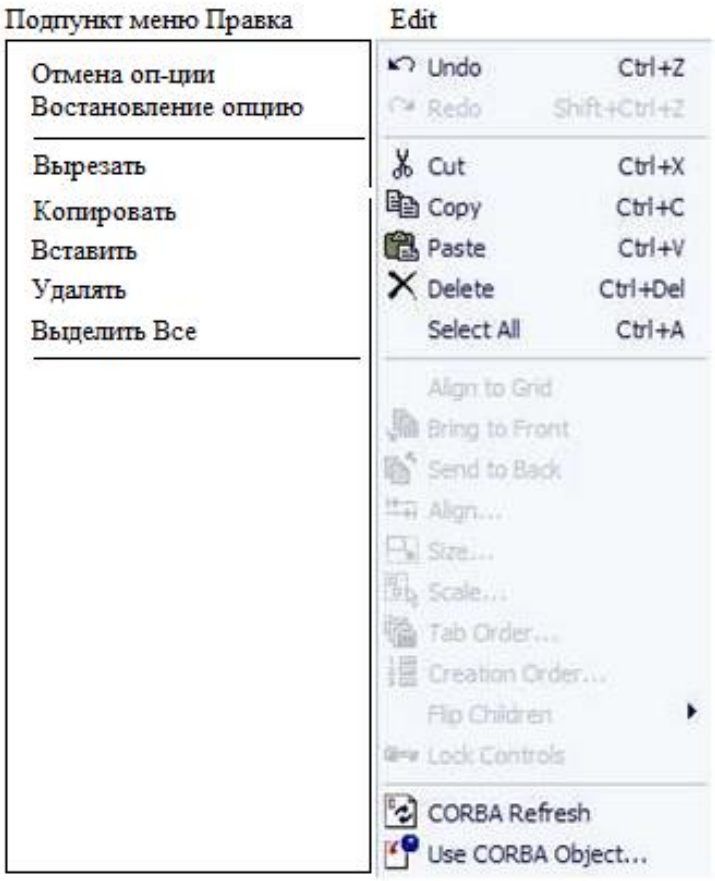



Рис. 1.6. Операции правки IDE C++Builder.

Вторая по порядку группа команд меню называется ***Edit*** (Правка), она изображена на рис. 1.6.

В этом меню собраны команды редактирования, такие как отмена (Undelete) и повторение (Redo) операций, вырезание (Cut), копирование (Copy) вставка (Paste) и удаление (Delete), команды выделения всего текста (Select All), выравнивания компонентов (Align) и настройки редактора кода (текста программы).

А также дает возможность связи с обектами библиотеки CORBA.

Следующая группа команд ***Search*** (Поиск) изображена на рис. 1.7.

Команды данного меню позволяют осуществить поиск текста в файле, продолжить поиск после запуска программы, произвести автоматическую замену, быстро перейти к строке кода, задав ее номер.

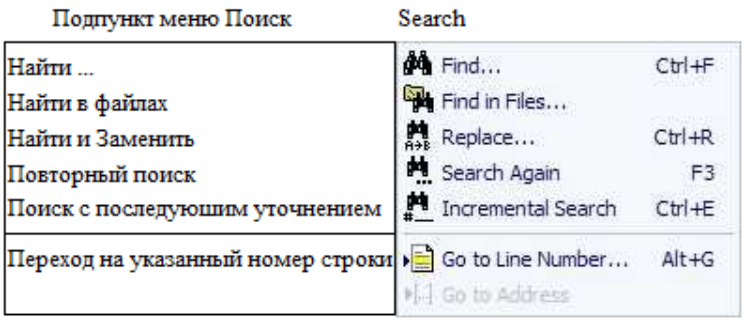



Рис. 1.7. Поисковые оперции.

Группа команд меню ***View*** (Вид) изображена на рис. 1.8. Из этого пункта меню

вызываются основные диалоговые окна управления проектом и компонентами.

Меню разбито на четыре отсека. В первом отсеке содержаться управляющие команды, такие как:

- менеджер проектов для управления проектами Project Manager;
- управление транзакциями при работе с базами данных и другими поддерживающими транзакций устройстваи;
- активизацие и дектавизацие окна инспектора обектов (Object Inspector);
- активизацие и дектавизацие окна дерева объектов (Object TreeView);
- Список последовательных действий, позволяющий планировать группу действий в проекте, и под управлением функции To-Do-List выполнит спиок приведенных действий. Подпунк очень удобен при работе с большими проектами;

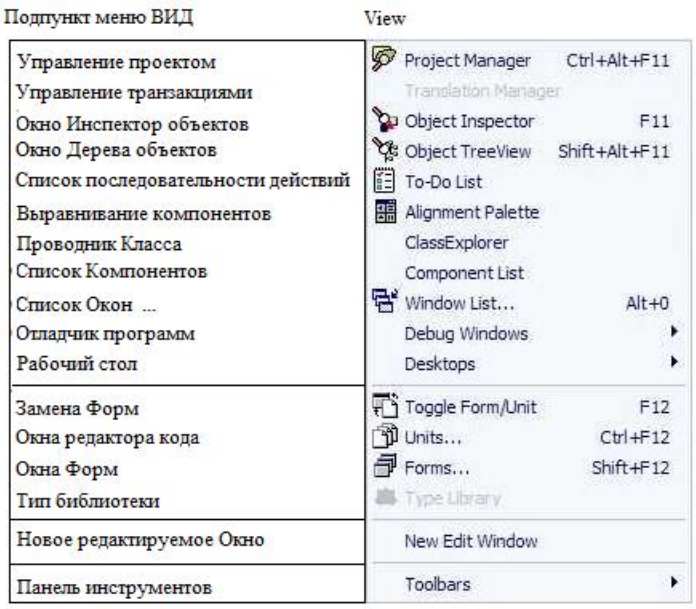



Рис. 1.8. Операции на IDE C++Builder.

- Выравнивание компонент;
- Проводник класса позволяет увидеть дерево наследуемости;
- список компонентов Component List позволяет операции манипуляции компонент и панелей, т.е. устанавка на панели новые компонент и создание новых панели.
- список окон Windows List . из этого пункта меню открываются все окна задействанные в проекте;
- отладки программ Debug Windows, работу с которыми мы рассмотрим позже;
- Рабочий стол Desktops позволяет сохранить сложный проект и его интерфейс в удобном для пользователя виде.

Остальные отсеки рассмотрим позже, последний четвертый отсек служит для работы с панелью управления, можно добавить панели инструментов и пиктограммы на них по потребностям пользователя.

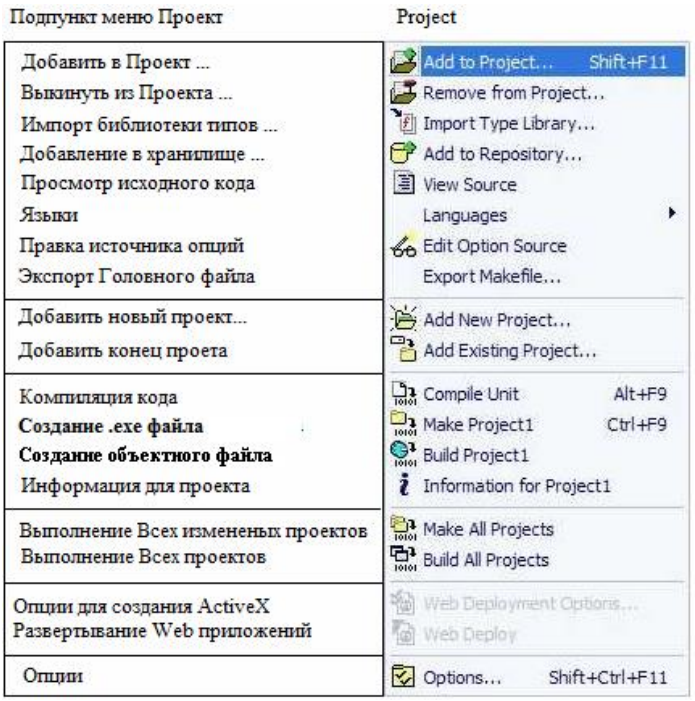



Рис. 1.9. Группа команд проект и операция.

Группа команд ***Project*** (Проект) приведена на рис. 1.9 В этом меню собраны команды управления проектом. С их помощью можно добавлять и удалять модули (файлы с текстами программ), добавить библиотеку компонентов VCL, откомпилировать проект и так далее.

Особый интерес представляет команды находяшиеся в третьем, четвертом и шестом отсеках.

Compile Unit Компиляцию кода программы необходимо выполнять всегда, мы здесь выявляем синтаксические ошибки и предупреждения, если таковых нет, то продолжаем отладку программы, выполняем линкование проекта. Для этой цели можно выполнить Build Project1 Создание объектного файла. Если потом необходимо создание командного файла, то выполняем Make Project1 Создание .exe файла, однако его можно запустить сразу после компиляции проекта, если имеется ошибки, то выполняют те же действия что и Build Project1 Создание объектного файла.

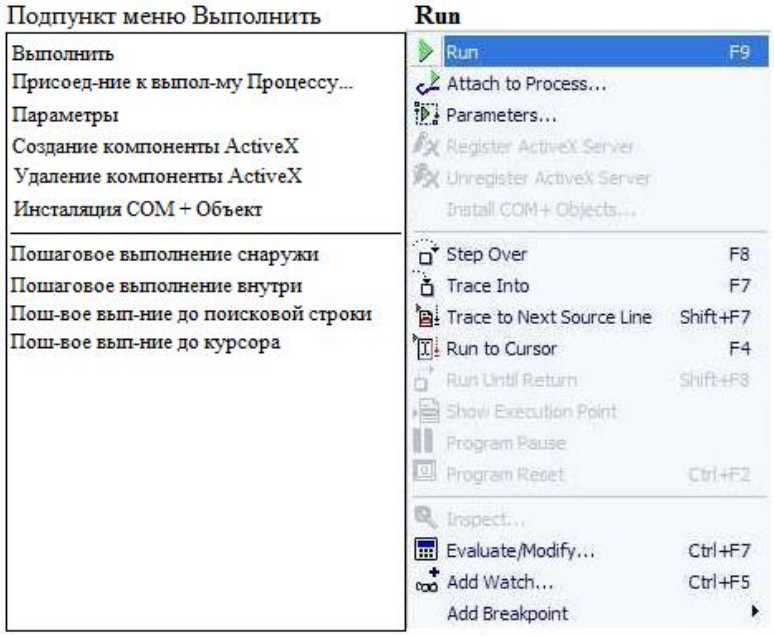



Рис. 1.10. Группа команд отладки и выполнения проекта

Об остальных пунктах мы поговорим позже

Группа команд ***Run*** (Выполнить) изображена на рис.1.10.

С помощью команд этого меню выполняется запуск и останов программ, запуск программ в непрерывном

и пошаговом режимах, добавление переменных для просмотра, установка точек останова и другие действия по отладке программы. Процесс отладки программы мы рассмотрим на конкретных примерах позже.


Рис. 1.11. Операция с компонентами

Группа команд ***Component*** (Компонент) представлена на рис. 1.11.

Из этого меню вызываются команды добавления в систему новых компонентов и конфигурации их палитры. Ее мы также рассмотрим на примерах.


Рис. 1.12. Базы данных

Группа ***Database*** (База данных) представлена на рис. 1.12 она содержит команды для работы с базами данных. Создание псевдонима, таблиц, выбора языка, мастера форм и др. Изучение компонентов этой группы рассмотрим позднее, как и других из-за большого объема.

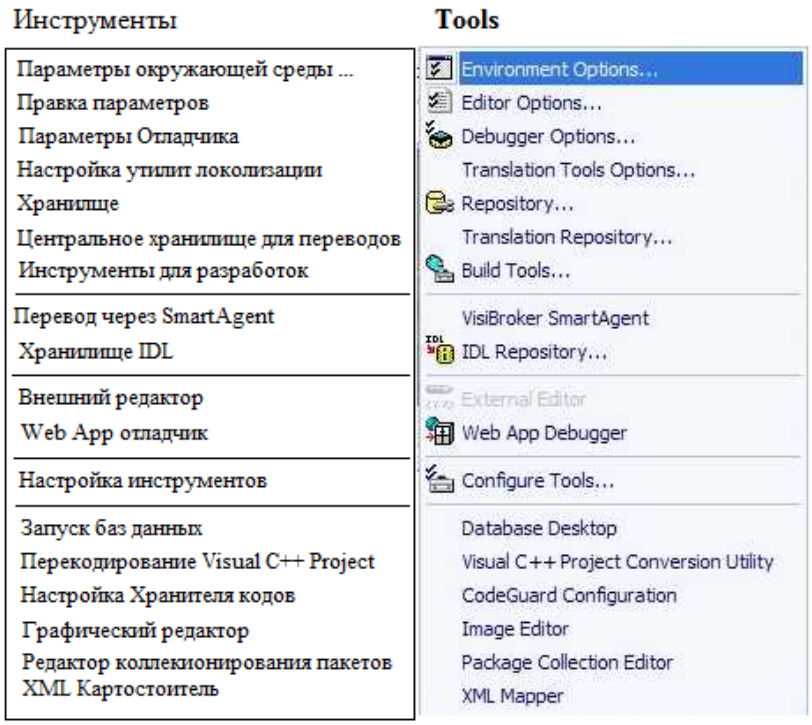



Рис. 1.13. Инструментарий.

Группа ***Tools*** (Инструменты) изображена на рис. 1.13 В этом меню собраны команды как по настройке параметров программы, так и команды вызова различных дополнительных утилит. Полное описание всех утилит выходят далеко за рамки одного учебного пособие, однако кратко можно указать. При помощи параметров окружающей среды можем описать пути, например, нужно прописать путь к опциям в:

Tools->Options->Environment Options->Delphi Options->Library->Library path
Tools->Options->Environment Options->C++ Options->Path and Directories->Include path
Tools->Options->Environment Options->C++ Options->Path and Directories->Library path

Перечисленные параметры можно исправлять, кроме того, можно и создавать приложения словарей, переводчиков, локализовать их и отправлять в хранилище

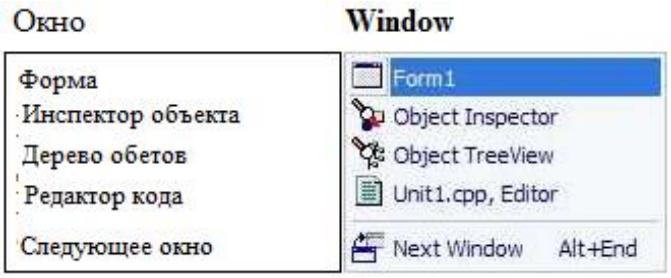



Рис. 1.14. Операции над окнами

Группа ***Windows*** приведена на рис. 1.14.

С помощью команд данного меню производится управление окнами интерфейса.

Замыкает главное меню группа команд ***Help***.

**Панель управления** включают панели: a) ***Standard***; b) ***Help contents***; c) ***View*** d) ***Debug*** см. на рис 1.15. Каждая панель содержит пиктограммы, которые облегчают доступ к наиболее часто применяемым командам основного меню.

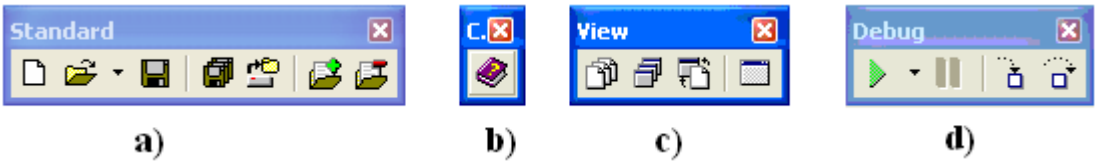



Рис 1.15. Панель управления


Рис. 1.16. Палитра компонентов

Через **палитру компонентов** осуществляется доступ к набору стандартных сервисных программ среды C++ Builder, которые описывают некоторый визуальный элемент (компонент), помещенный программистом в окно формы. Каждый компонент имеет определенный набор свойств (параметров), которые программист может задавать. Например, цвет, заголовок окна, надпись на кнопке, размер и тип шрифта и др. список и назначение подпунктов палитры компонент приведены в таблице 1.1.

Таблица 1.1.

| | |
|---|---|
| ***Standard*** | Стандартная, содержащая наиболее часто используемые компоненты |
| ***Additional*** | Дополнительная, являющаяся дополнением стандартной |
| ***Win32*** | 32-битные компоненты в стиле Windows 95/98 и NT |
| ***System*** | Системная, содержащая такие компоненты, как таймеры, плееры и ряд |

| | |
|---|---|
| | других |
| ***Data Access*** | Доступ к данным, в C++Builder 6 большинство компонентов, размещавшихся ранее на этой странице, перенесено на страницу BDE |
| ***Data Controls*** | Компоненты отображения и редактирования данных |
| ***dbExpress*** | Связь с данными с помощью dbExpress |
| ***DataSnap*** | Компоненты для связи с сервером приложений при построении многопоточных приложений, работающих с данными |
| ***BDE*** | Доступ к данным через Borland Database Engine - BDE |
| ***ADO*** | Связь с базами данных через Active Data Objects (ADO) - множество компонентов ActiveX, использующих для доступа к информации баз данных Microsoft OLE DB |
| ***InterBase*** | Прямая связь с Interbase, минуя Borland Database Engine (BDE) и Active Data Objects (ADO) |
| ***WebServices*** | Компоненты клиентский приложений Web, использующие доступ к службам Web с помощью SOAP |
| ***InternetExpress*** | Построение приложений InternetExpress - одновременно приложений сервера Web и клиента баз данных с параллельными потоками |
| ***Internet*** | Компоненты для создания серверов Web |
| ***WebSnap*** | Компоненты для создания серверов Web, содержащих сложные страницы, управляемые данными |
| ***FastNet*** | Различные протоколы доступа к Интернет |
| ***Decision Cube*** | Компоненты для многомерного анализа данных (не во всех вариантах C++Builder) |
| **QReport** | Компоненты для подготовки отчетов |
| **Dialogs** | Диалоги, системные диалоги типа "Открыть файл" и др. |
| **Win 3.1** | Windows 3.x, компоненты в стиле Windows 3.x |
| **Samples** | Образцы, различные интересные, но не до конца документированные компоненты |
| **ActiveX** | Примеры компонентов ActiveX |
| **COM+** | Компонент, дающий доступ к каталогу COM+, содержащему данные по конфигурации COM+ |
| **Servers** | Компоненты связи с серверами COM (начиная с C++Builder 6, в C++Builder 5 на этой странице размещались компоненты, перенесенные теперь на страницу Office2k) |
| **IndyClients** | Компоненты клиентских приложений Internet Direct (Indy), дающих доступ к различным протоколам Интернет из приложений Delphi, C++Builder, Kylix |
| **IndyServers** | Компоненты серверных приложений Internet Direct (Indy) |
| **IndyMisk** | Различные вспомогательные компоненты приложений Internet Direct (Indy) |
| **InterBase Admin** | Компоненты доступа к службам InterBase |
| **Office2k или Office97** | Оболочки VCL для распространенных серверов COM офисных приложений Microsoft |

## Окно Object TreeView и страница диаграмм Редактора Кода.

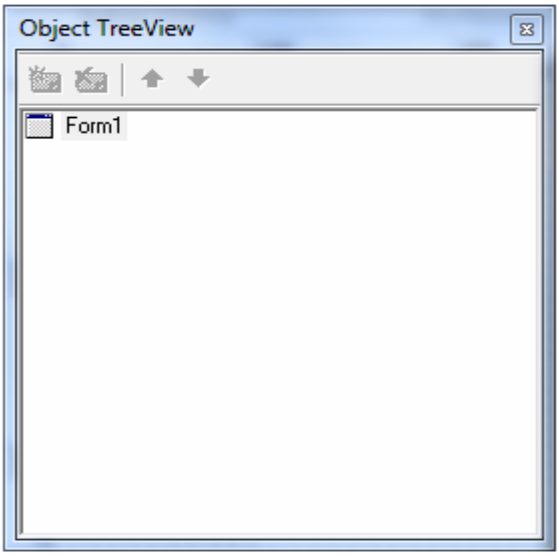



Рис. 1.17. Окно Object Tree Vien

**Object TreeView** - дерево объектов для отображения всех визуальных и невизуальных компонентов приложения в аспекте связей этих компонентов.

Вызов окна Object TreeView см. рис. 1.17. выполняется командой ***View => Object TreeView***, в котором можно перетаскиванием дочернего компонента менять родителя.

Связи между компонентами можно документировать в виде диаграммы:

- Окно Редактора Кода => Страница Diagram => (1), (2), (3):

(1) выпадающий список активной диаграммы и пиктограммы для создания, редактирования, удаления диаграммы; (2) окно Name - имя диаграммы;

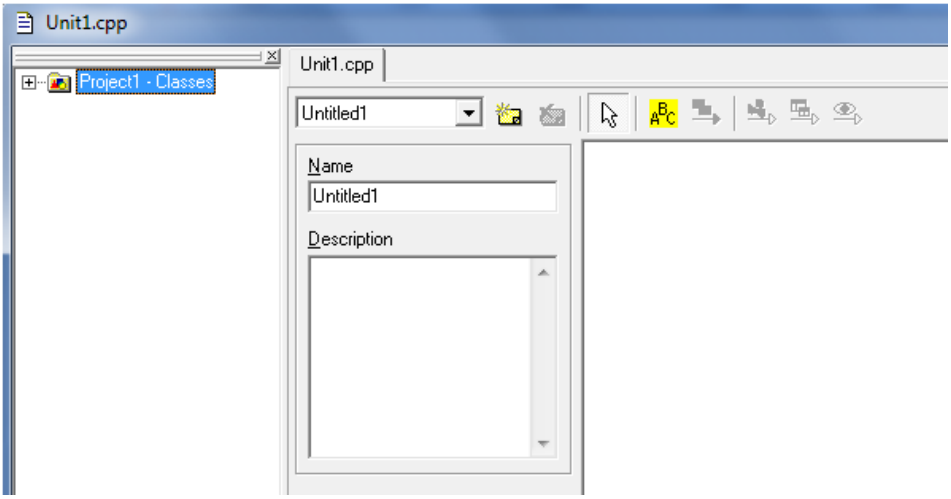


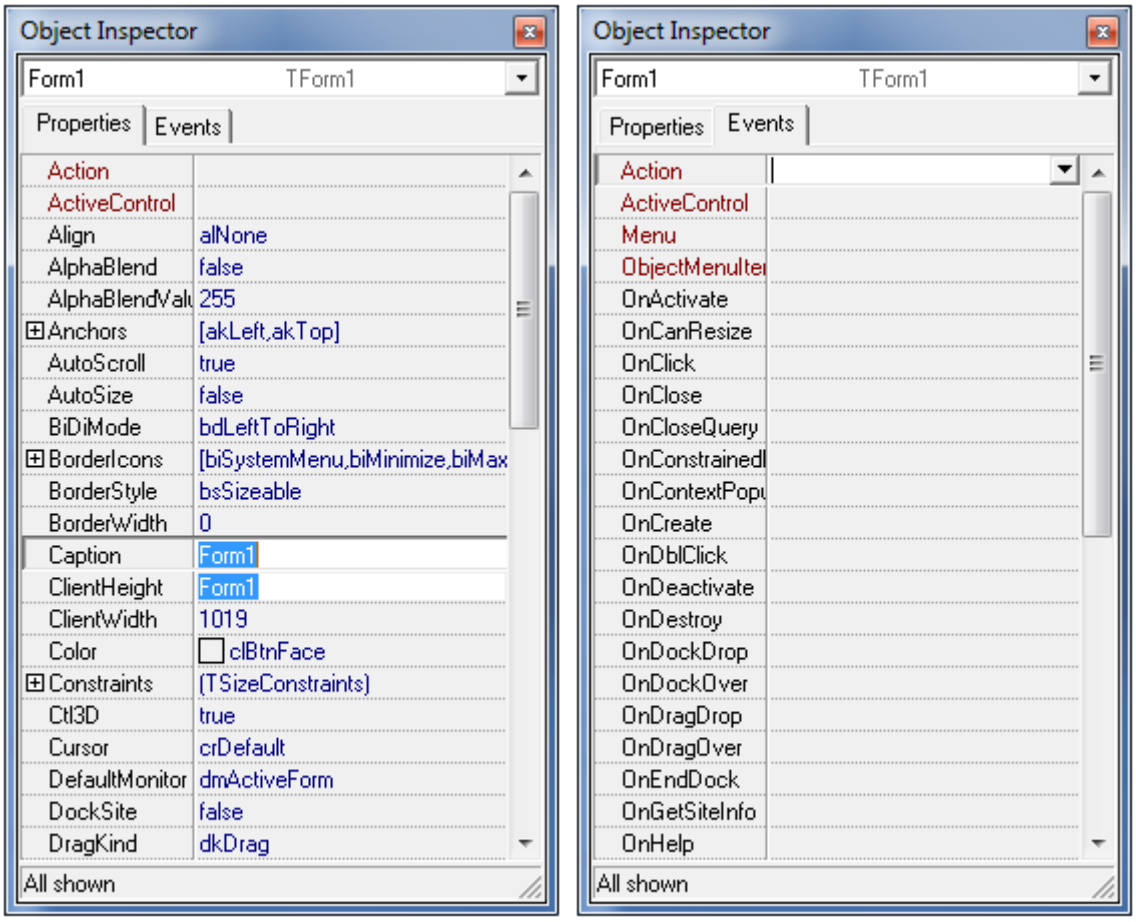

Рис.1.18. Структура диаграмм Редактора Кода

(3) окно Description - текстовое описание диаграммы.

Для создания диаграммы надо из окна Object TreeView перетащить в поле диаграммы интересующие компоненты; автоматически появляются стрелки, показывающие заданные связи между компонентами.

**Окно Object Inspector (инспектора объекта). Окно инспектора объекта** см. рис. 1.19. (вызывается с помощью клавиши F11) предназначено для изменения свойств выбранных компонентов и состоит из двух страниц. Страница Properties (Свойства) предназначена для изменения необходимых свойств компонента, страница Events (События)

– для определения реакции компонента на то или иное событие (например, нажатие определенной клавиши или щелчок по кнопке мыши).




a) Properties b) Events

Рис.1.19. Окно инспектор объекта

**Окно Form (формы).**


Рис.1.20. Атрибуты окна

Окно формы см. рис. 1.21, представляет собой проект Windows-окна, другими словами интерфейс программы.

Как и любое Windows-окна здесь содержатся: икона проекта, имя окна (при старте Form1), кнопки управления окном см. рис.1.20. В это окно в процессе создания программы (проекта) помещаются необходимые компоненты.

Причем при выполнении программы помещенные компоненты будут иметь тот же вид, что и на этапе проектирования, если они видимые,

однако невидимые компоненты не отображаются во время выполнения проекта.

Проект может сдержать одну или множество форм, в зависимости от решаемой задачи. Выделенная компонента (активная) на форме выделяется в окне Object TreeView и его свойства отображаются в окне Object Inspector. Сама форма (окно проекта) может обладать различными свойствами, приведем некоторые наиболее часто используемые свойства.

**Свойства формы:**

***BorderStyle*** - свойство определяет общий вид окна и операции с ним, которые разрешается выполнять пользователю. Рекомендуемые значения свойства BorderStyle: (1), (2)

(1) = ***bsSingle*** - для основного окна приложения с неизменяемыми размерами - наиболее подходящий стиль;

(2) = ***bsDialog*** - для вторичных диалоговых окон - наиболее подходящий стиль.

***BorderIcons*** - свойство определяет набор кнопок, которые имеются в окне заголовка.

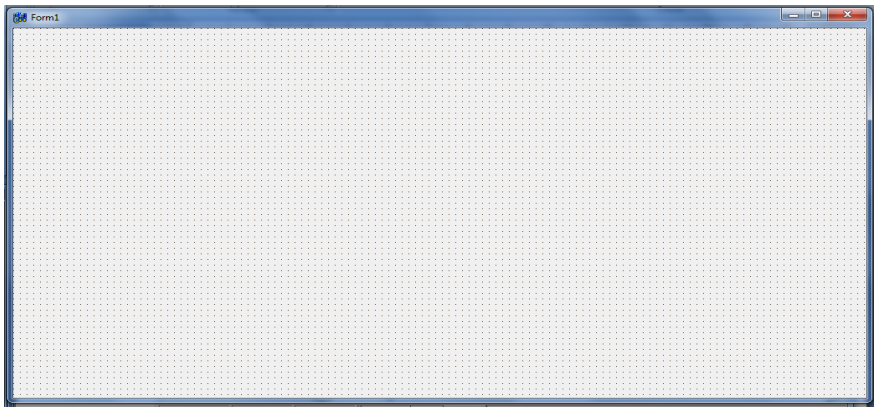



Рис. 1.21. Окно формы.

***WindowState*** - свойство определяет вид, в котором окно первоначально предъявляется пользователю при выполнении приложения. Обычно целесообразно для главной формы приложения задавать значение ***Position poScreenCenter*** или ***poDefaultPosOnly***. В сравнительно редких случаях, когда на экране при выполнении приложения должно определенным образом располагаться несколько окон, имеет смысл оставлять значение ***poDesigned***, принимаемое по умолчанию.

***AutoScrol*** - свойство определяет: будут ли на форме в процессе выполнения появляться автоматически полосы прокрутки.

***Icon*** - свойство задает пиктограмму формы, которая отображается в левом верхнем углу окна приложения в развернутом состоянии (*).

**Изменение пиктограммы формы:**

Команда ***Project*** => ***Options*** => ***страница Application***: (1), (2)

(1) ***Title*** - заголовок, который увидит пользователь в полосе задач при сворачивании приложения.

(2) ***Load Icon*** - позволяет выбрать пиктограмму, которая будет видна в полосе задач при сворачивании приложения или при просмотре пользователем каталога, в котором расположен выполняемый файл приложения.

***FormStyle*** - одно из основных свойств формы, которое может принимать значения: (1) - (4)

(1) ***fsNormal*** - окно обычного приложения, это значение FormStyle принято по умолчанию.

(2) ***fsMDIForm*** - родительская форма приложения MDI, т.е. приложения с дочерними окнами, используемого при работе с несколькими документами одновременно.

(3) ***fsMDIChild*** - дочерняя форма приложения MDI.

(4) ***fsStayOnTop*** - окно, остающееся всегда поверх остальных окон Windows.

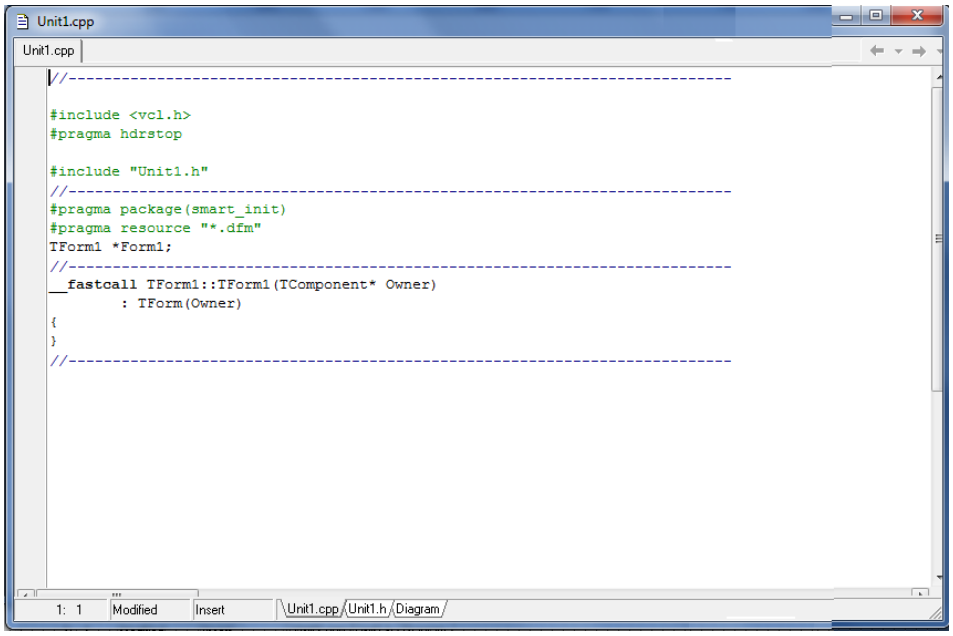



Рис. 1.22. 1[ое] окно Редактора кода, где записываются функции и называют его ***Файлом реализации*** проекта

**Окно Редактора кода.** Окно Редактора кода программы предназначено для просмотра, написания и редактирования текста программы см. рис.1.22. В системе C++ Builder используется язык программирования C++. При первоначальной загрузке в окне текста программы находится текст, содержащий минимальный набор операторов

для нормального функционирования пустой формы в качестве Windows-окна.

Окно Редактора кода состоит из 3 окон, с окном диаграмм мы уже сталкивались, когда рассматривали окно ***Object TreeView***

```
//--  это строка комментарий программа их не обрабатывает -----
/* это тоже комментарий он может быть многострочным */
#include <vcl.h> // через директиву препроцессора (ДПР)   #include
подключено
            // библиотека классов VCL
#pragma hdrstop //Это ДПР #pragma указывает использование
            // опции hdrstop - конец списка общих файлов
#include "Unit1.h" // это  ДПР #include включает 2 окно редактора
кода
//---------------------------------------------------------------------
#pragma package(smart_init) // Это ДПР  #pragma определяет операцию
/*    последовательность    инициализации    пакетов    такой,    какая
устанавливается взаимными ссылками использующих их модулей. */
#pragma resource "*.dfm" // это ДПР сообщает, что для формы
     /* надо использовать файл с расширением *.dfm с таким же именем
*/
TForm1 *Form1;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------
```

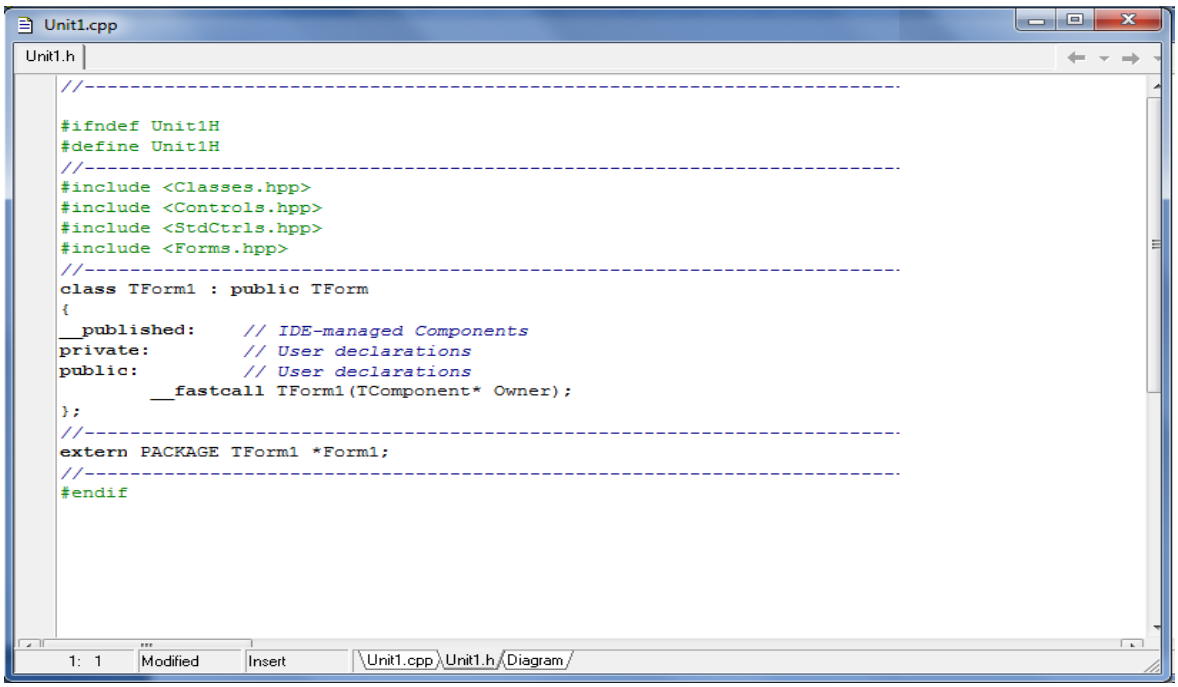



Рис. 1.23. 2[oe] окно Редактора кода, где записываются функции и называют его ***Заголовочный файл*** проекта

Второе окно редактора кода Unit.h служить для блока описаний классов, методов, прототипы функций и др. конструкций проекта, вернее будет сказано-это ***Заголовочный файл*** см. рис. 1.23.

Заголовочный файл состоит из:

```
//---------------------------------------------------------------------------
#ifndef Unit1H //включает условную компиляцию
#define Unit1H //определяет идентификатор для условной компиляции
//---------------------------------------------------------------------------
#include <Classes.hpp> //
#include <Controls.hpp> //
#include <StdCtrls.hpp> //
#include <Forms.hpp> //
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:      // IDE-managed Components-размещенные на форме
компоненты
private:    // User declarations – закрытый раздел классов
public:           // User declarations – открытый раздел классов
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif // завершает заголовочный файл
```

## 1.2. Первый проект

Исторически среди программистов, сложилась такая традиция, первую программу начинают со слов Hello World – Здравствуй Мир! Не будем нарушать его и мы, однако сделаем проект в современном дизайне.

Подготовительный этап создания проекта.

Для того, чтоб создать проект, в хорошем стиле существует последовательность действий, которые в начальный период необходимо не нарушать. Сначала желательно создать папку, с говорящим именем, чтобы легко было ориентироваться в наших проектах.

Так и сделаем! На рабочем столе создадим папку под именем: «Лабораторные работы», а в ней следующую папку «Hello World»

Затем вторым шагом стартуем программу C++ Builder 6. ***Пуск⇒ Программы⇒ Borland C++ Builder 6⇒ C++ Builder 6.***

При создании проекта желательно создать окно с именем, которое удовлетворяет и отвечает назначению проекта. В нашем случае можем назвать «Здравствуй Мир». Для этого в окне инспектора объектов (***Object Inspector***) в окне свойств (***Property***) меняем значение строки **Caption** Form1 на выражение «Здравствуй Мир». Как вы заметили имя окна получила название «Здравствуй Мир».

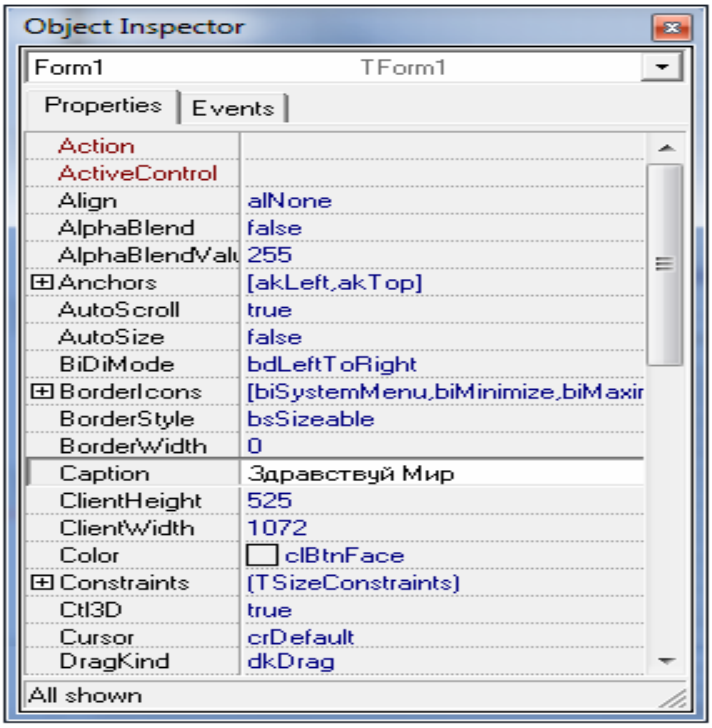
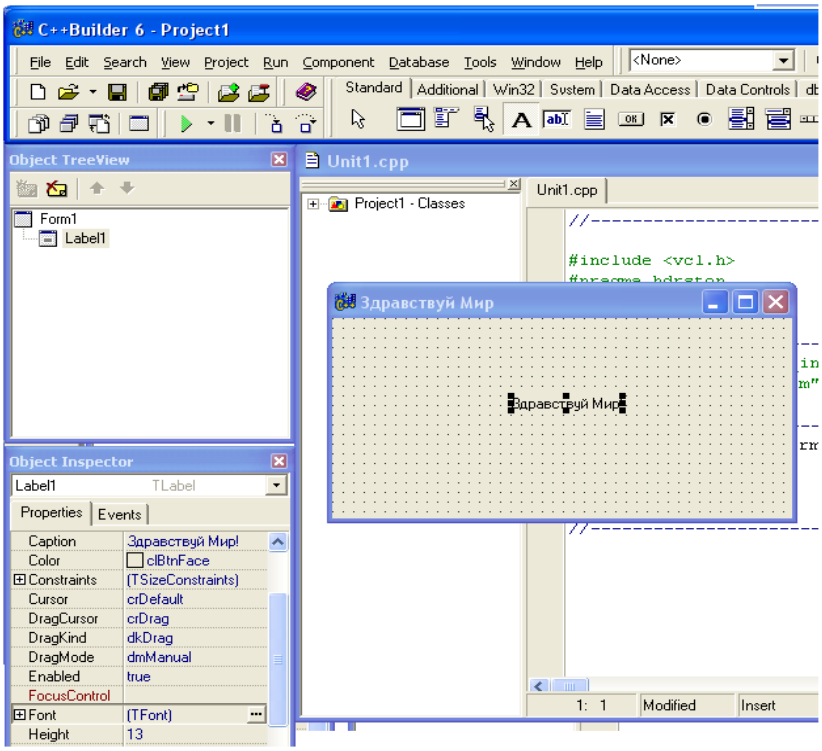


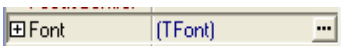

Рис.1.24. Замена содержимого в Caption в свойствах (Proprtis) Objest Inspector для окна Form1.

**Создание первого проекта.** Изменяем размеры окна. Для этого для чего схватив правый край окна Form1, уменьшаем ширину окна до нужного размера, и аналогично, схватив нижний край формы, изменяем высоту. Затем на палитре компонентов активизируем компоненту ***Standard⇒Label,*** как показано на рис 1.25., щелчком устанавливаем его на форме Form1. в нужном месте.




Значение строки ***Object Inspector⇒ Property⇒*** **Caption** Label1 меняем значение на Здравствуй Мир! См. рис. 24.

Затем в пункте Font нажать кнопку с тремя точками.




Такие кнопки и символ + впереди

Рис.1.25. Работа с меткой (Label) и со шрифтом(Font)

названия говорят, что мы имеем дело с субменю. В данном случае можно изменить размеры символов. Процесс можно осуществить, нажав символ + и в соответствующих строках меню изменяя параметры данных, или выбирая в окне Шрифт, которая приведена на рис. 1.26.

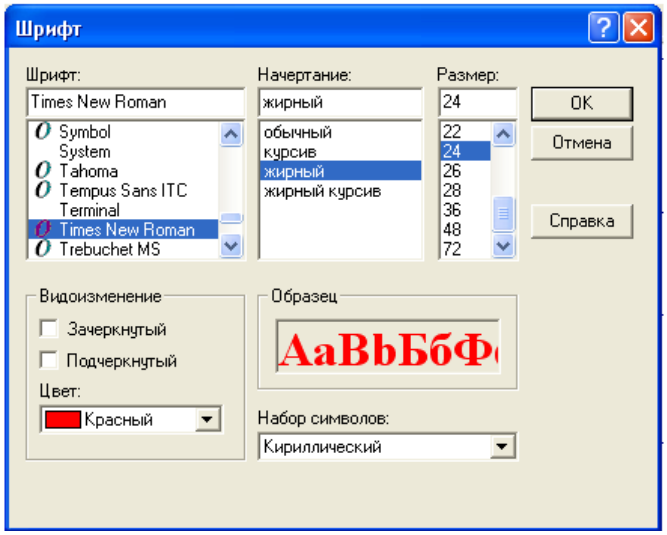



Рис..1.26. Работа со шрифтами

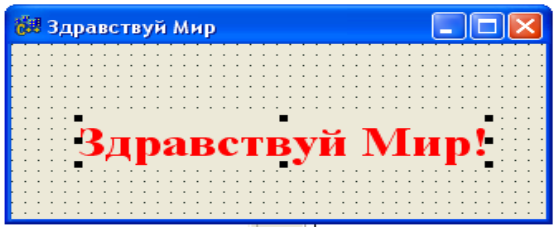



Рис.1.27. Результат работы с меткой (Label)и шрифтом (Font).

Используя соответствующие пункты в подпунктах меню выбираем параметры, соответственно имеем картину Рис. 1.26. где видим почти готовый проект.

Теперь его надо запустить на выполнение и провести отладку проекта. О чем будет сказано ниже.

**Процессы отладки и запуска на выполнение проекта.** Мы почти закончили создание нашего проекта. Хотя мы не написали ни одной строчки программного кода, однако при выполнении указанных действий львиную долю работы на себя взяла сама среда Borland C++ Builder 6, в чем не трудно убедиться. Вызовите на Окне Форм контекстное меню (обратный щелчок мышью), см. рис. 1.28. и подпункт View as Text.

При этом мы видим текст в окне:

```
object Form1: TForm1
  Left = 246
  Top = 226
  Width = 331
  Height = 167
  Caption = 'Здравствуй Мир'
  Color = clBtnFace
  Font.Charset = RUSSIAN_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Times New Roman'
  Font.Style = []
```

```
    OldCreateOrder = False
    PixelsPerInch = 96
    TextHeight = 14
    object Label1: TLabel
      Left = 40
      Top = 56
      Width = 249
      Height = 36
      Caption = 'Здравствуй Мир!'
```

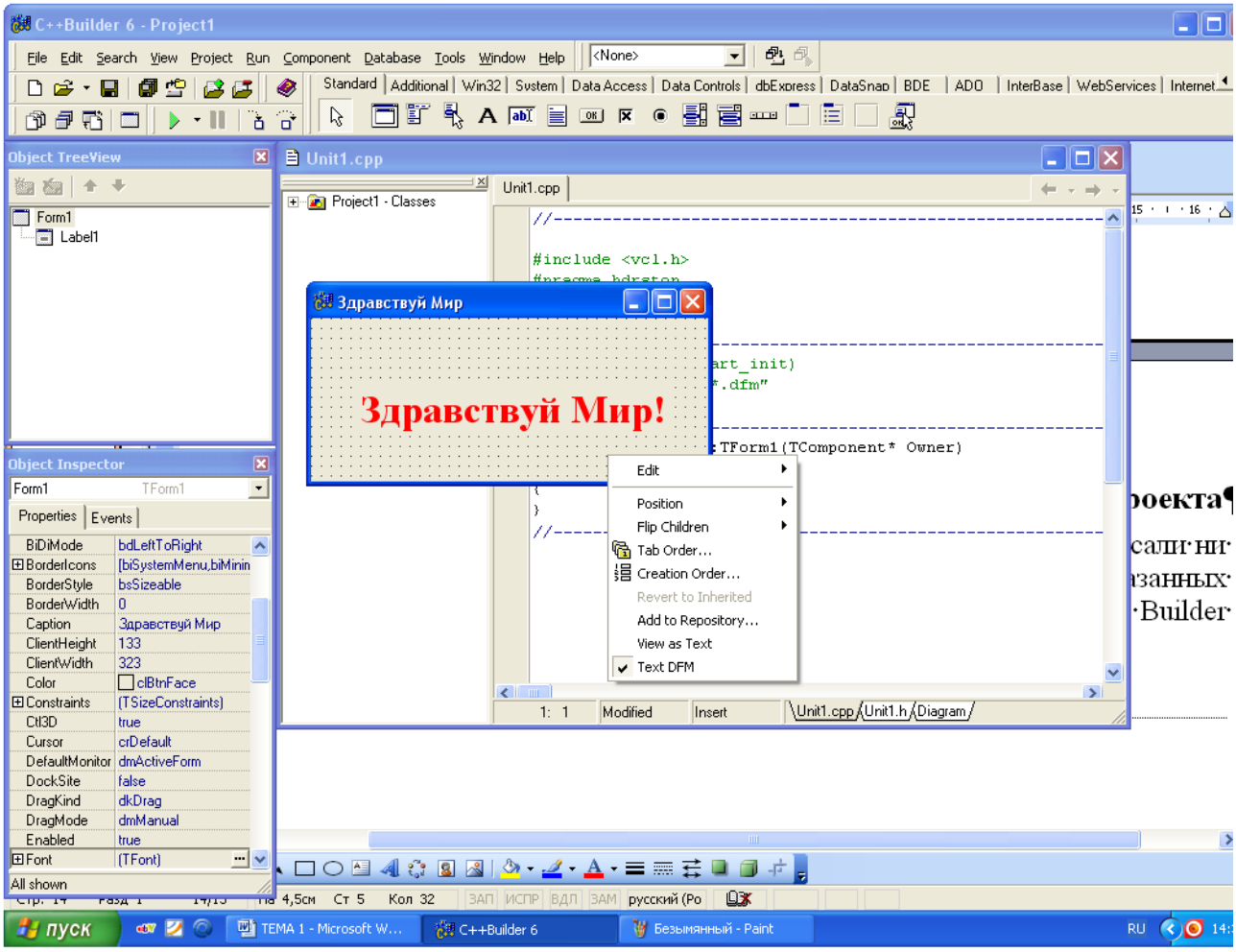



Рис.1.28. Вызов контекстного Меню Окно Форм и просмотр программного кода

```
      Font.Charset = RUSSIAN_CHARSET
      Font.Color = clRed
      Font.Height = -32
      Font.Name = 'Times New Roman'
      Font.Style = [fsBold]
      ParentFont = False
    end
  end
```

Все это за нас написала среда, за нас сделала среда, посмотрите и прочтите выще внимательно, но об этом чуть позже.

Для того чтобы, найти синтаксические ошибки, которых в нашем проекте нет пока, необходимо в Главном меню Главного окна выбрать пункты

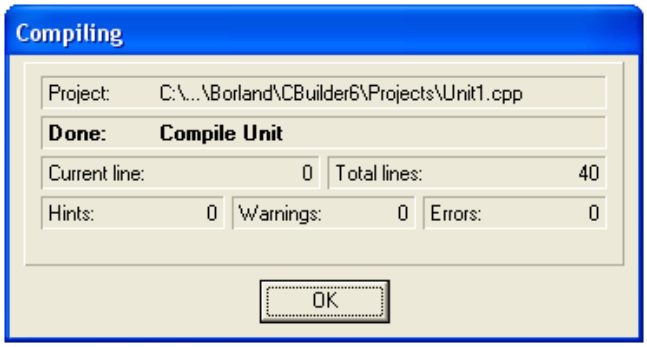



Рис. 1.29. Компиляция проекта

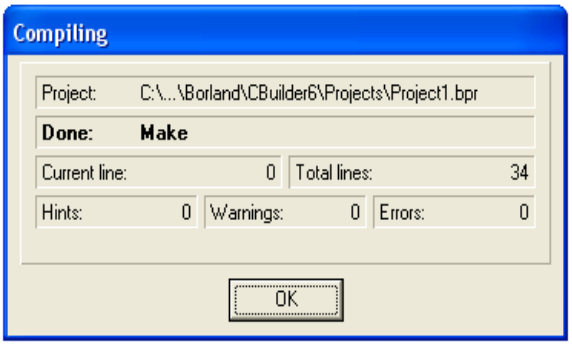



Рис.1.30. Создание Объектного файла.

***Project⇒Compile Unit*** или нажать клавиши ***Alt*** и ***F9*** совместно (обычно кратко записывают ***Alt+F9).***

При этом мы получим окно компиляции проекта, см. рис. 1.29, где приводится адрес проекта, число предупреждений (Warnings) и синтаксических ошибок (Errors). Мы видим рис. 1.29. у нас их отсутствуют.

После компиляции, если нет синтаксических ошибок, приступаем к фазе отладки проекта. Для чего необходимо создание объектного файла (компоновка программы), перед запуском на выполнение проекта см. рис.31. Процесс создания Объектного файла можно выполнить в Главном меню Главного окна выбрать пункты ***Project⇒Make Project1*** или нажатием ***Ctrl+F9***. Процесс отладки программы это довольно громоздкий процесс, и в рамках одной работы его трудно описать, мы об этом поговорим позднее.

После создания объектного файла проекта, если нет ошибок, то можно запустить проект на выполнение. Нажав кнопку  на панели управления, или в Главном меню Главного окна выбрать пункты ***Run⇒Run*** или нажатием функциональной клавиши ***F9***.

**Сохранение проекта.** Когда завершена проект или часть программы уже написано надо временно прервать работу, а затем продолжить, нужно проект сохранить. В первом пункте настоящего параграфа мы указывали на необходимость создания папки Hello World, там и сохраним наш проект. Выбираем в Главном меню Главного окна пункт File⇒***Save Project As*** при этом появится диалоговое окно. Выбираем нужную папку и задаем имя проекта. Для сохранения модуля выбираем в Главном меню Главного окна пункт File⇒***Save As*** при этом появится диалоговое окно. Имена файлов, как указывалось ранее в пункте 1 текущего параграфа желательно, чтобы они были говорящими.

Программа в C++ Builder состоит из множества моделей, которые объединяются в один проект с помощью файла проекта (файл с расширением .bpr). Файл проекта автоматически создается и обрабатывается средой C++ Builder и не предназначен для редактирования. Объявления классов, функций и переменных находятся в заголовочном файле (расширение .h), текст программы, написанный на языке C++, – в файле исходного текста (расширение .cpp). Описание окна формы находится в файле с расширением .dfm. Файл проекта может быть только один, файлов с другими расширениями может быть несколько.

***Внимание!*** *Для того чтобы перенести проект на другой компьютер, необходимо переписать все файлы с расширениями: bpr, h, cpp, dfm.*

При запуске программы на выполнение сначала препроцессор преобразует текст в соответствии с имеющимися директивами. После это займемся созданием иконы для проекта.

## 1.3. Создание иконы для проекта

Хорошим стилем считается для семейства операционных систем Widows 9x, когда проект имеет икону и значок (эмблему проекта).

**Создание эмблемы для иконы.** Итак, мы собираемся создать свою эмблему. Воспользуемся встроенным графическим редактором Image Editor. Выбираем в Главном меню Главного окна пункт ***Tools⇒Image Editor*** при этом появится диалоговое окно см. рис. 1.31., который по своим параметрам и функциям особо не отличается от графического редактора Widows 9x Paint.

Выполняем действия указанные на рис. 1.32., выбираем в меню File⇒***Icon File (.ico),*** и нажимаем ОК. При этом появиться новое окно, где можно рисовать нужную нам эмблему иконы. Рисуем икону см. рис. 1.33.

Теперь можно сохранить полученный файл в нашу папку, где и другие файлы проекта.

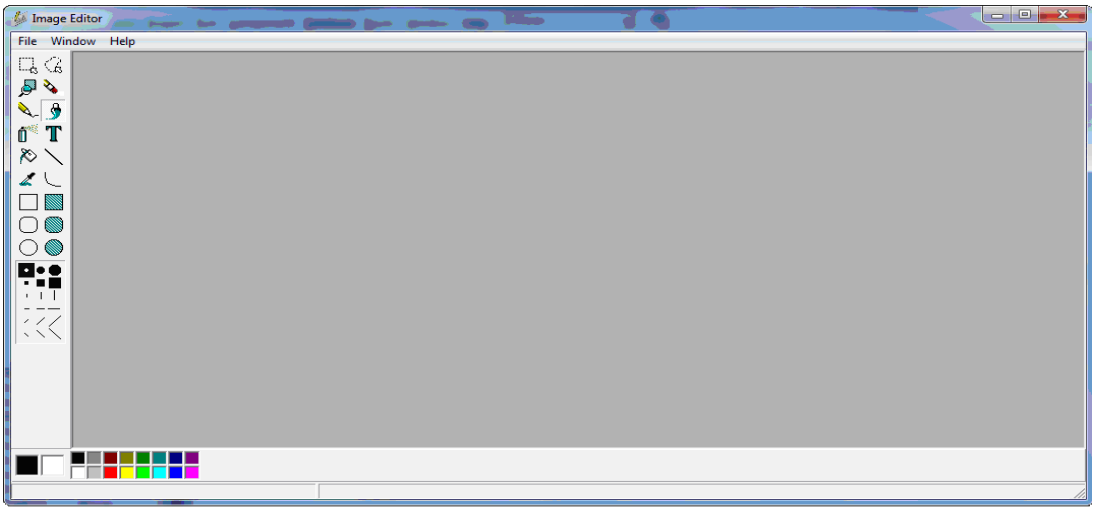



Рис.1.31. Встроенный графический редактор

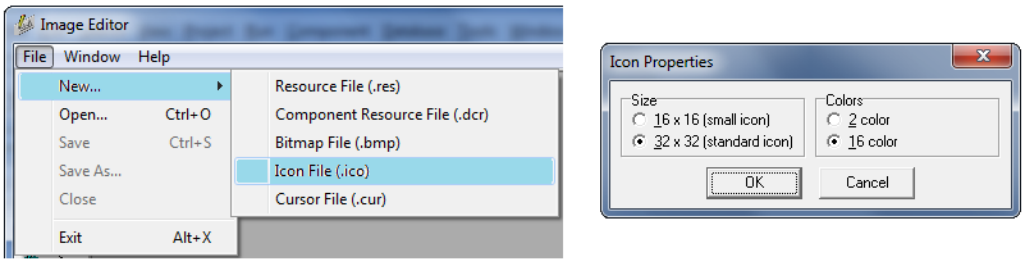
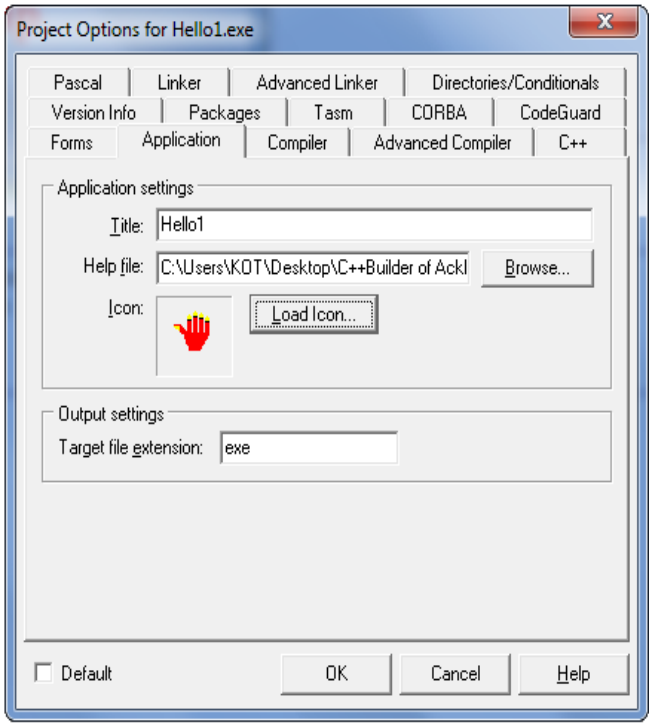



Рис.1.32. Файловые операции для создание иконки



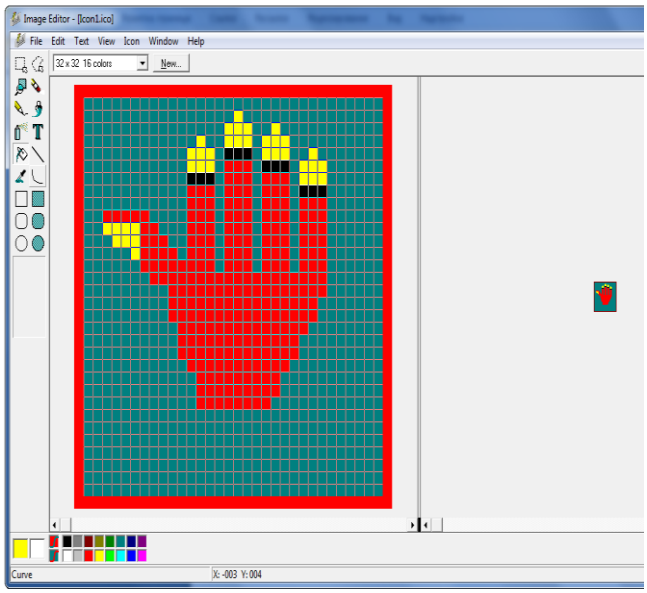



Рис.133. Создание эмблемы иконы

Рис. 134. Включение иконки в проект.

**Включение эмблемы иконы в проект.** После сохранения иконы необходимо чтобы проект знал об иконе, для чего, выбираем в Главном меню Главного окна пункт ***Project⇒Options,*** при этом появится диалоговое окно см. рис. 35., где нужно указать во вкладке Application имя проекта и имя иконы.

Теперь проверим появление эмблемы иконы на окне проекта. Для чего перекомпилируем проект. Выбираем в Главном меню Главного окна пункт ***Project⇒Build Hello1.*** После стартуем программу, и видим окно см. рис. 1.35.

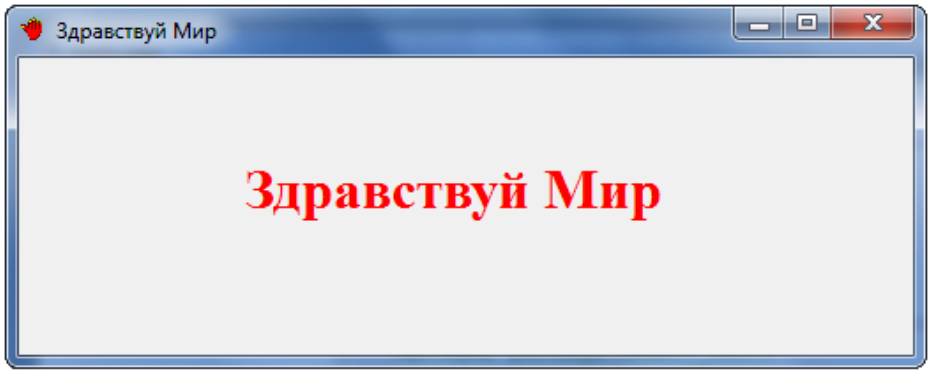



Рис.1.35. Окно с нашей эмблемой.

Ну, вот и завершили наш первый проект. Хотя, ничего особенного мы не сделали, но мы ознакомились с процессом создания простейшего проекта. Мы неоднократно вернемся к данной теме по мере усложнения наших задач, и добавления других более сложных функций, методов и процессов. Удачи ВАМ и трудолюбия дорогие мои.

## 1.4. Контрольные вопросы.

1. Каким образом можно запустить C++ Builder 6?
2. Опишите IDE C++ Builder 6?
3. Что отображает окно ObjectInspector?
3. Что отображает окно ObjectTreeView?
4. Из каких элементов состоит Главное окно?
5. Что представляет собой Палитра компонентов?
6. Из каких пиктограмм состоит панель Standard?
7. Назначение и устройство окна Form?
8. Назначение окна Редактора кода?
9. Что представляет собой проект?
10. Основные файловые операция в C++ Builder 6?
11. Из каких этапов состоит процесс отладки проекта?
12. Из каких этапов состоит процесс создания иконы?

## Лабораторная работа №2.
## Программирования алгоритмов линейной структуры

**Цель лабораторной работы**: изучить различные парадигмы и концепции технологии программирования, научиться состовлять простейщие блок-схем алгоритмов, концепции контексных грамматик, основы языков С и С++ и составлять простейшей программы в среде С++ Builder. Написать и отладить программу линейного алгоритма.

### 2.1. Основные теоретические сведения

ЭВМ служит для обработки информации. Любые процессы, связанные с обработкой информации на ЭВМ, априори связаны с программой. Программа – это один из основных инструментов пользователя компьютером. Образ программы, чаще всего, хранится в памяти машины (например, на диске) как исполняемый модуль (один или несколько файлов). Из образа на диске с помощью специального программного загрузчика может быть построена исполняемая программа уже в оперативной памяти машины. В настоящем курсе нас интересует программы – языков С, С++ и С++ Builder, которые позволяют, используя присущие им языковые правила, операторы, функции, свойства, методы, и другие конструкции для создания ***пользовательских алгоритмических программ или проектов***.

***Алгоритмом называется точное предписание, определяющее последовательность действий исполнителя, направленных на решение поставленной задачи.***

Алгоритм обладает следующими свойствами:

***Однозначность алгоритма****, под которой понимается единственность толкования исполнителем правил и порядка выполнения действий.*

***Конечность алгоритма*** *– обязательность завершения каждого из действий и всего алгоритма в целом.*

***Результативность алгоритма****, предполагающая, что его выполнение завершится получением определённых результатов.*

***Массовость****, т.е. возможность применения алгоритма к целому классу задач, отвечающих общей постановке задачи. Для того, чтобы алгоритм обладал свойством массовости, следует составлять его с использованием обозначения переменных величин и избегая конкретных константных значений.*

**Жизненный цикл и этапы разработки программного продукта.** Жизненный цикл программного продукта (ПП) — это период

времени, начинающийся с момента принятия решения о необходимости создания ПП и заканчивающийся в момент его полного изъятия из эксплуатации.

Структуру жизненного цикла ПП, состав процессов, действия и задачи, которые должны быть выполнены во время создания ПП, определяет и регламентирует международный стандарт ISO/IEC 12207:1995 «Information Technology — Software Life Cycle Processes» (ISO — International Organization for Standardization — Международная организация по стандартизации; IEC — International Electrotechnical Commission — Международная комиссия по электротехнике; название стандарта «Информационные технологии — **Процессы жизненного цикла программ**»).

Под процессом понимают совокупность взаимосвязанных действий, преобразующих входные данные в выходные.

Каждый процесс характеризуется определенными задачами и методами их решения, а также исходными данными, полученными от других процессов и результатами. Каждый процесс разделен на набор действий, каждое действие — на набор задач. Запуск и выполнение процесса, действия или задачи осуществляются другими процессами.

*В соответствии со стандартом ISO/IEC 12207 все процессы жизненного цикла ПП разделены на три базовые группы:*

- ***основные процессы*** *(включают в себя набор определенных действий и связанных с ними задач, которые должны быть выполнены в течение жизненного цикла ПП);*
- ***вспомогательные*** *(поддерживающие) процессы (создание надежного, полностью удовлетворяющего требованиям заказчика ПП в установленные договором сроки. К вспомогательным относятся процессы документирования, управления конфигурацией, обеспечения качества, верификации, аттестации, совместной оценки, аудита, разрешения проблем);*
- ***организационные процессы*** *(организация процесса разработки надежного, полностью удовлетворяющего требованиям заказчика ПП в установленные договором сроки и управление этим процессом. К организационным относятся процессы управления, создания инфраструктуры, усовершенствования, обучения).*

Редько, какая-либо большая программа работает сразу правильно. Обычно большие программы могут содержать ошибки, которые находят во время эксплуатации. Их нужно разработчикам программы ликвидировать,

если получают рекламацию на ошибку. Жизненный цикл ПП включает и процесс разработки, который состоит из следующих этапов:

**Первый этап** – постановка задачи. На этом этапе требуется хорошо изучить предметную область задачи, и осуществить сбор информации. Нужно чётко определить цель задачи, дать словесное описание содержания задачи и продумать общий подход к её решению, и сделать эскизный проект.

**Второй этап** – математическое или информационное моделирование. Цель этого этапа – создать такую математическую модель решаемой задачи, которая может быть реализована на компьютере. Часто математическая постановка задачи сводится к простому перечислению формул и логических условий, однако возможно, что для полученной модели известны несколько методов решения, и тогда следует выбрать лучший из них.

**Третий этап** – алгоритмизация задачи. На основе математического описания необходимо разработать алгоритм решения.

В роли исполнителей могут выступать люди, роботы, компьютеры. Способ записи алгоритма зависит от выбора исполнителя. Наглядно алгоритм представляется в виде блок–схемы, схемы из UML-диаграмм и (или) других конструкций графы и т.д.

**Четвёртый этап** – программирование. Программой называют план действий, подлежащих выполнению исполнителем, в качестве которого может выступать компьютер. От алгоритма программа отличается тем, что записывается на языке понятном для исполнителя. Если исполнителем является компьютер, то программа записывается на одном из языков программирования. В компьютере в конечном итоге данные и команды представляются в виде последовательности нулей и единиц. Поэтому, когда говорят о «понятности» языка программирования для компьютера подразумевают наличие специальной программы, способной перевести инструкции языка программирования в последовательность двоичных компьютерных команд. Язык программирования выполняет функции посредника между человеком и ЭВМ и поэтому с одной стороны должен быть удобен для записи алгоритмов в понятной для человека форме, а с другой стороны легко преобразовываться в машинные коды.

Программа и исходные данные вводятся в ЭВМ с клавиатуры с помощью редактора текстов или загружаются в редактор с внешнего носителя, результаты работы программы выводятся на экран дисплея или записываются на носитель информации.

**Пятый этап** – трансляция программы. Трансляция означает перевод команд языка программирования в двоичные коды (машинный язык). На этом этапе происходит проверка программы на ее соответствие правилам (синтаксису) языка программирования и при отсутствии синтаксических ошибок создается исполняемый файл программы. Исполняемый файл содержит инструкции в двоичном коде. Если транслятор обнаруживает в программе несоответствия синтаксису, то исполняемый файл создать не удастся. Программист должен устранить несоответствия (исправить синтаксические ошибки). После этого необходимо проверить программу на наличие логических ошибок. Для этого нужно подобрать систему тестов (набор исходных данных с заранее известным результатом) и сравнить выдаваемые программой результаты с контрольными тестовыми данными. Подробнее об этом в следующем разделе.

**Шестой этап** – тестирование и отладка программы. На этом этапе происходит исполнение программы на ЭВМ, поиск и исправление логических ошибок, то есть ошибок, приводящих к неправильной работе программы. При этом программисту приходится выполнять анализ работы программы. Для сложных программ этот этап, как правило, требует гораздо больше времени и сил, чем написание первоначального текста программы.

Отладка программы – сложный и нестандартный процесс. Исходный план отладки заключается в том, чтобы проверить программу на контрольных примерах. Под контрольными примерами подразумеваются различные комбинации исходных данных. Контрольные примеры выбираются так, чтобы при работе были задействованы все ветви алгоритма. Дело в том, что некоторые ошибки в программе могут проявиться только при попытке выполнения конкретных действий. Детализация плана зависит от того, как поведёт себя программа на этих примерах: на одном она может зациклиться, на другом дать явно неверный ответ и т.д. Сложные программы отлаживают отдельными фрагментами, или специальными инструментариями, которые предназначены для выполнения отладки программы, о которых мы говорили в 1 теме, обычно обозначают их отдельным пунктом или подпунктом в меню «Debug».

**Седьмой этап** – исполнение отлаженной программы и анализ результатов. На этом этапе программист запускает программу и задаёт исходные данные, требуемые по условию задачи. Полученные результаты анализирует постановщик задачи, на основании анализа

принимаются решения, вырабатываются рекомендации, делаются выводы.

Возможно, что по итогам анализа результатов потребуется возврат ко второму этапу для повторного выполнения всех этапов с учётом приобретённого опыта. Таким образом, в процессе создания программы некоторые этапы повторяют до тех пор, пока не будет получен алгоритм и программа, удовлетворяющие указанным выше свойствам.

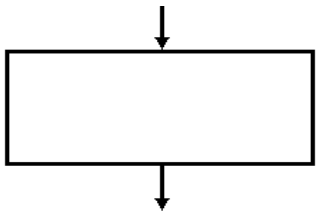
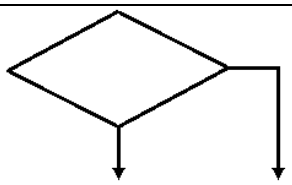
**Основные элементы для построения блок-схемы.** Для создания блок-схем алгоритмов используются Единая система программной документации СХЕМЫ АЛГОРИТМОВ, ПРОГРАММ ДАННЫХ И СИСТЕМ **ГОСТ 19.701-90 (ИСО 5807-85)**. Приведем некоторые положения из ГОСТ. Стандарт распространяется на условные графические обозначения (символы) в схемах алгоритмов, программ, данных и систем и устанавливает правила выполнения блок-схем, используемых для отображения различных видов задач обработки данных и средств их решения.

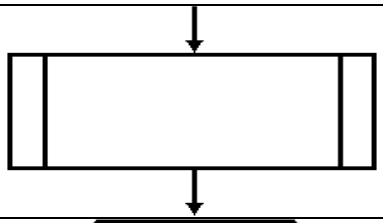
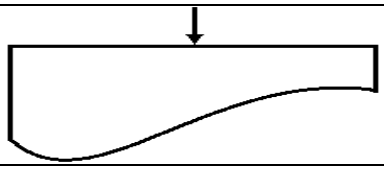
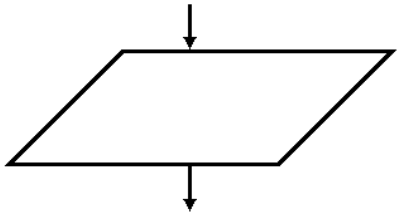
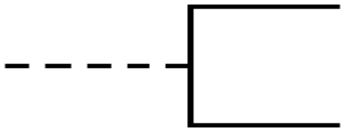
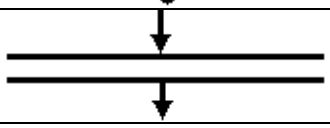
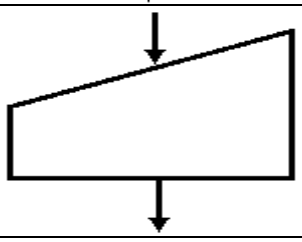
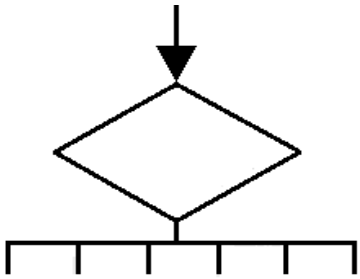
Стандарт не распространяется на форму записей и обозначений, помещаемых внутри символов или рядом с ними и служащих для уточнения выполняемых ими функций.

Требования стандарта являются обязательными.

Описание некоторых графических символов по ГОСТ 19.701-90 (ИСО 5807-85) , показанны в таб. 2.1

Таблица 2.1.

| **Символическое название** | **Обозначение схемы** | **Пояснения** |
|---|---|---|
| Процесс |  | **Вычислительное действие**. Процесс формирование новых значений, выполнение арифметических или логических действий, результаты запоминаются в памяти ЭВМ. |
| Решение |  | **Проверка условия** выбор одного из двух направлений в зависимости от логического условия |
| Пуск останов |  | **Пуск останов.** Начало, конец, вход, выход в программе или подпрограмме. |

| | | |
|---|---|---|
| Предопределенный процесс |  | **Предопределенный процесс**. Вычисление по подпрограмме, используя ранее созданную функцию или процедуру. |
| Модификация |  | **Модификация** Организация циклических конструкций (начало цикла) |
| Документ |  | **Документ.** Вывод результатов на бумажные носители информации |
| Параллелограмм |  | **Параллелограмм** Ввод/вывод данных, по умолчанию на монитор. Осуществить может связь с внешними и внутренними носителями памяти ЭВМ. |
| Комментарий |  | **Комментарий.** Поясняет, что делается в данном блоке или в подпрограмме |
| Соединитель |  | **Соединитель.** Разрыв линий потока |
| Соединитель |  | **Соединитель** перенос на другую страницу |
| Параллельные действия |  | **Параллельные действия** синхронизации |
| Ручной ввод |  | **Ручной ввод** с устройств любого типа (клавиатура, переключатели, кнопки, световое перо, полоски со штриховым кодом) |
| Несколько выходов |  | Несколько выходов из символа следует показывать:<br>1) несколькими линиями от данного символа к другим символам;<br>2) одной линией от данного символа, которая затем разветвляется в соответствующее число линий. |

Полный перечень графических символов приводится в приложении 1.

Приведенные схемы можно использовать при создании алгоритмов решения задач в структурном, императивном, функциональном программировании, однако вызывают огромные трудности для создания блок-схемы в объектно-ориентированном программировании. Мы рассмотрим инструментарии для решения этой проблемы позже.

**Парадигмы программирования.** Термин «парадигма программирования» ввел Роберт Флойд. Сам термин «парадигма» имеет философское толкование. В философии парадигма (с греческого παράδειγμα – «пример, модель, образец») означает совокупность определенных принципов, достижений, аксиом, признаваемых всеми в тот или иной период времени, на основании которых выдвигают определенные научные теории и методологические концепции. Парадигмы в программировании носят диалектический характер, могут со временем опровергаться, дополнятся и изменятся.

Парадигма программирования — это комплекс концепций, принципов и абстракций, определяющих фундаментальный стиль программирования, и априори связано с образом мышления самого программиста. Парадигма задается использованием определенных сущностей:

- состояний программы и команд, изменяющих их (императивное программирование),
- математических функций без состояний (функциональное программирование),
- объектов и взаимодействий между ними (объектно-ориентированное программирование),
- алгоритмов и контейнеров, оперирующих с типами данных, переданными как параметр (обобщенное программирование),
- значений и операций, преобразующих значения (программирование на уровне значений), и т.д.

Следует отметить, что язык программирования не обязательно использует только одну парадигму. Языки, поддерживающие несколько парадигм, называются мультипарадигменными. Создатели таких языков придерживаются точки зрения, гласящей, что ни одна парадигма не может быть одинаково эффективной для всех задач, и следует позволять программисту выбирать лучший стиль программирования для решения каждой отдельной задачи, например, C++ Builder, программный код можно написать, не используя управляющие компоненты, Action, ActionManager возможно и опираясь на них.

К процессу создания программы надо подойти творчески.

Любая парадигма языка программирования реализуется математической формализацией концепции, используя формальные грамматики.

**Формальные языки и грамматики. Формальный язык** является объединением множеств:

- исходных символов, называемых литерами или лексемами, которые включают алфавит языка (**алфавит** *– это счетное множество допустимых символов языка, другими словами на языке формальных грамматик –* ***терминальные символы VT****)*;
- правил, которые позволяют строить из литеров алфавита новые слова, так называемые правила порождения слов или идентификаторов, предопределённых идентификаторов или словаря ключевых слов и прочих идентификаторов называемые именами (на языке *формальных грамматик –* ***нетерминальные символы*** **NT**);
- правил, которые позволяют собирать из имён и ключевых слов выражения, на основе которых строятся простые и сложные предложения, так называемые правила порождения операторов или предложений (*на языке формальных грамматик – множество продукционных правил P вида* $\alpha \to \beta$*, где* $\alpha э (VN \cup VT)$ *и* $\beta э (VN \cup VT)$);
- S –целевой (начальный) символ грамматики S$э$ ***VN***.

*Множество правил порождения слов, выражений и предложений называют* ***грамматикой формального языка*** *или* ***формальной грамматикой***.

Язык, заданный грамматикой Г обозначается как L(Г).

Формальная грамматика Г определяется как четверка **Г(VT,VN,P,S)**.

***Замечание!*** *Алфавиты терминальных и нетерминальных грамматик не пересекаются:* $\mathbf{VT \cap VN = \varnothing}$**.**

Однако грамматика естественного языка, подобно наукам о природе с известной степенью достоверности описывает и обобщает результаты наблюдений за естественным языком как за явлением окружающего мира. Характерные для грамматики естественных языков исключения из правил свидетельствуют о том, что зафиксированная в грамматике языка система правил не может в точности описать все закономерности развития языка.

**Бэкуса-Наура формы (БНФ).** Метаязыки Хомского и Хомского-Щутценберже использовались в математической литературе при описании простых абстрактных языков. Метаязык, предложенный Бэкусом и Науром, впервые использовался для описания синтаксиса

реального языка программирования Алгол 60. Наряду с новыми обозначениями метасимволов, в нем использовались содержательные обозначения нетерминалов. Это сделало описание языка нагляднее и позволило в дальнейшем широко использовать данную нотацию для описания реальных языков программирования. Были использованы следующие обозначения:

- символ «::=» отделяет левую часть правила от правой (иногда вместо знака старых монографий «::=» ставят знак «→» или «⇒»);
- нетерминалы обозначаются произвольной символьной строкой, заключенной в угловые скобки "<" и ">";
- терминалы - это символы, используемые в описываемом языке;
- каждое правило определяет порождение нескольких альтернативных цепочек, отделяемых друг от друга символом вертикальной черты "|".

Пример описания идентификатора с использованием БНФ:

- <буква> :: = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V| W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
- <цифра> :: = 0|1|2|3|4|5|6|7|8|9
- <идентификатор> ::= <буква> | <идентификатор><буква> |
- <идентификатор><цифра>

**Расширенные Бэкуса-Наура формы (РБНФ).** Метаязыки, представленные выше, позволяют описывать любой синтаксис. Однако, для повышения удобства и компактности описания, целесообразно вести в язык дополнительные конструкции. В частности, специальные метасимволы были разработаны для описания необязательных цепочек, повторяющихся цепочек, обязательных альтернативных цепочек. Существуют различные расширенные формы метаязыков, незначительно отличающиеся друг от друга. Их разнообразие зачастую объясняется желанием разработчиков языков программирования по-своему описать создаваемый язык. К примерам таких широко известных метаязыков можно отнести: метаязык PL/I, метаязык Вирта, используемый при описании Модулы-2, метаязык Кернигана-Ритчи, описывающий С см. приложение 2. Зачастую такие языки называются расширенными формами Бэкуса-Наура (РБНФ).

В частности, РБНФ, используемые Виртом, имеют следующие особенности:

- квадратные скобки "[" и "]" означают, что заключенная в них синтаксическая конструкция может отсутствовать;
- фигурные скобки "{" и "}" означают ее повторение (возможно, 0 раз);

- круглые скобки "(" и ")" используются для ограничения альтернативных конструкций;
- сочетание фигурных скобок и косой черты "{/" и "/}" используется для обозначения повторения один и более раз. Нетерминальные символы изображаются словами, выражающими их интуитивный смысл и написанными на русском языке.

Если нетерминал состоит из нескольких смысловых слов, то они должны быть написаны слитно. В этом случае для повышения удобства в восприятии фразы целесообразно каждое ее слово начинать с заглавной буквы или разделять слова во фразах символом подчеркивания. Терминальные символы изображаются словами, написанными буквами латинского алфавита (зарезервированные слова) или цепочками знаков, заключенными в кавычки. Синтаксическим правилам предшествует знак "$" в начале строки. Каждое правило оканчивается знаком "." (точка). Левая часть правила отделяется от правой знаком "=" (равно), а альтернативы - вертикальной чертой "|". Этот вариант РБНФ и будет использоваться для описания синтаксиса языков в лабораторной работе. В соответствии с данными правилами синтаксис идентификатора будет выглядеть следующим образом:

- $ буква =

"A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|"O"|"P"|"Q"|"R"|
"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"|"a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|
"n"| "o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z".

- $ цифра = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
- $ идентификатор = буква {буква | цифра}.

## 2.2. Основы языка C++ Builder

Для передачи своих мыслей человек использует речь в устной или письменной форме. Если в письменной, то необходимо знать буквы, чтобы составить слова, также и грамматические правила для точной передачи мысли в виде предложения. Программа, составленная программистом – продукт позволяющий реализацию его мыслей в виде текста, поэтому начнем с рассмотрения алфавита языка C++ Builder.

**Алфавит языка C++ Builder.** Алфавит языка C++ Builder включает:

- 26 строчных, 26 заглавных символов латинского алфавита и символ подчеркивания «_»;
- Арабские числа 0...9;
- Следующие последовательности специальных символов и букв алфавита см. таблицу 1. образуют множество символов операций

(часть из них в зависимости от контекста может быть использована в качестве разделителей):

Специальные символы, используемые в C, C++ и C++ Builder

Таблица 2.1.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| , | ! | != | \| | \|= | % | %= | & |
| && | &= | () | * | *= | + | ++ | += |
| - | -- | -= | -> | ->* | . | .* | / |
| /= | :: | < | << | <= | <<= | > | >> |
| >= | >>= | == | ?: | [] | ^ | ^= | ~ |
| \|\| | # | ## | sizeof | new | delete | typeid | throw |

**Внимание!** Символы строчные и заглавные в записях на языке C++ Builder значимы в отличие от языка Паскаль и некоторых других языков.

**Лексемы и идентификаторы.** Наименьшая неделимая конструкция языка, построенная из одного или последовательности символов, называют лексемой.

Идентификатором называют лексему, обозначающую имена меток, переменных, констант, функций, конструкторов, деструкторов, классов и других конструкций языка.

Идентификаторы в программе желательно объявлять говорящими (по смыслу что делает или для чего нужен идентификатор) или по венгерской нотации (из начальных букв выражения или сочетания слов).

В идентификаторах не допустимы символ пробел, обычно его замещает символ подчеркивания.

Не допускается, чтобы идентификатор пользователя совпадал с ключевыми словами, или конструкциями языка выдаст сообщение об ошибке, и не желательно, чтобы идентификатор пользователя совпадал со стандартными функциями языка, может привести к непредсказуемым процессам.

Длина идентификатора имеет ограничения, желательно не более 20 символов из различных соображений и из опыта опытных программистов.

Множество лексем, соответствующее множеству символов операций и разделителей, строится на основе набора специальных символов и букв алфавита. Единственное правило словообразования для этих категорий лексем заключается в задании фиксированного множества символов операций и разделителей.

**Переменные и константы.** Имя переменной или константы в программе называют идентификатором.

Любая информация в программе хранится в переменных и константах. Переменные и константы могут хранить как простые данные: числа, символы, строки, — так и сложные структуры, например, домашний адрес с четырьмя составляющими (название города и улицы, номер дома и квартиры) или даже различные объекты.

Перед тем как переменную или константу использовать, ее следует объявить (идентифицировать или декларировать). То есть, переменную надо явно описать в тексте программы, указав при этом ее идентификатор и тип.

Когда программа будет откомпилирована и запущена, для каждой переменной или константе в памяти компьютера будет выделено специальное место. Для чисел это может быть четыре или восемь байтов, а для строк — заданная вами длина.

Если переменная или константа не была объявлена до ее первого использования, например, для записи в нее конкретного числа, то компилятор при трансляции исходного текста выдаст сообщение об ошибке: Undefined symbol <имя переменной>.

Это означает, что переменная с таким-то именем не определена.

**Тип переменной или константы.** Каждая переменная должна иметь тип, определяющий, какого рода информация в ней хранится. Типом переменной может быть число, строка и всевозможные сложные структуры. Пока рассмотрим простые типы.

Таблица 2.2.

| **Тип данных** | | **число бит** | **Диапазон значений** |
|---|---|---|---|
| char | Символьный | 8 | –128 .. 127 |
| unsigned char | Без знака символьный | 8 | 0 .. 255 |
| short int | Без знака целый короткой длины | 16 | –32768 .. 32767 |
| unsigned int | Без знака целый | 32 | 0 .. 4294967295 |
| int | Целый | 32 | –2147483648 .. 2147483647 |
| long | Целый двойной длины | 32 | –2147483648 .. 2147483647 |
| unsigned long | Без знака целый двойной длины | 32 | 0 .. 4294967295 |
| long long int | Целый дважды двойной длины | 64 | $-(2^{63}-1) .. 2^{63}-1$ |
| unsigned long long int | Без знака целый дважды двойной длины | 64 | $0 .. 2^{64}-1$ |
| float | Вещественный | 32 | 3.4E–38 .. 3.4E38 |
| double | Вещественный двойной длины | 64 | 1.7E–308 .. 1.7E308 |
| long double | Вещественный дважды | 80 | 3.4E–4932 .. 3.4E4932 |

| | | | |
|---|---|---|---|
| | двойной длины | | |
| _Bool | Логический | 1 | true(1), false(0) |

Для объявления простых числовых типов см. табл. 2.2. можно использовать ключевые слова: ***signed*** - *знаковый,* ***unsigned*** - *беззнаковый,* ***short*** - *короткое,* ***long*** - *длинно и* ***double*** – *двойное*. Ключевое слово ***signed*** можно не писать, т.к. по умолчанию оно добавляется к основным типам языка. Язык С++ имеет следующие простые типы: ***char*** – *символьный,* ***int*** – *целый,* ***float*** – *вещественный и* ***double*** – *вещественное двойной длины*.

**Простые типы данных алгоритмических языков С и С++.** Отметим, что Язык С++ Builder поддерживает все типы данных языка С++, однако имеет тесные дружеские контакты с языком Delphi, существует правила перехода и преобразования о которых пойдет речь ниже.

С++Builder не позволяет посредством известного ключевого слова typedef просто переопределить некоторые сложные типы данных Объектного Паскаля. С++Builder реализует такие расширенные типы в виде обычных или шаблонных классов (template class). Каждый такой класс (о классах речь пойдет потом) содержит все необходимые конструкторы, деструкторы, свойства и объектные методы. Многие компоненты VCL используют реализацию расширенных типов, а кроме того, они требуются при разработке новых компонент на базе оригиналов из Delphi.

Ниже приводится сводная таблица 2.3. встроенных типов Delphi и соответствующих им типов С++Builder:

Встроенные типы Delphi и соответствующие им типы С++Builder

Таблица 2.3.

| **Delphi** | **Длина и значения** | **C++Builder** | **Реализация** |
|---|---|---|---|
| Shortint | 8-битовое целое | char | typedef |
| Smallint | 16-битовое целое | short | typedef |
| Longint | 32-битовое целое | long | typedef |
| Byte | 8-битовое целое без знака | unsigned char | typedef |
| Word | 16-битовое целое без знака | unsigned short | typedef |
| Integer | 32-битовое целое | int | typedef |
| Cardinal | 32-битовое целое без знака | unsigned long | typedef |
| Boolean | true/false | bool | typedef |
| ByteBool | true/false или 8-битовое целое без знака | unsigned char | typedef |
| WordBool | true/false или 16-битовое целое без знака | unsigned short | typedef |
| LongBool | true/false или 32-битовое целое без знака | unsigned long | typedef |

| | | | |
|---|---|---|---|
| AnsiChar | 8-битовый символ без знака | unsigned char | typedef |
| WideChar | Слово - символ Unicode | wchar t | typedef |
| Char | 8-битовый символ | char | typedef |
| String | Текстовая строка Delphi | AnsiString | typedef |
| Single | 32-битовое плавающее число | float | typedef |
| Double | 64-битовое плавающее число | double | typedef |
| Extended | 80-битовое плавающее число | long double | typedef |
| Real | 32-битовое плавающее число | float | typedef |
| Comp | 64-битовое плавающее число | double | typedef |
| Pointer | 32-битовый указатель | void * | typedef |
| PChar | 32-битовый указатель на символы без знака | unsigned char * | typedef |
| PansiChar | 32-битовый указатель на ANSI символы без знака | unsigned char * | typedef |
| Set | Множество 1..32 байт | set<type, minval, maxval> | template class |
| AnsiString | Текстовая строка Delphi | AnsiString | class |
| Variant | Вариантное значение, 16 байт | Variant | class |
| TdateTime | Значение даты и времени, 64-битовое плавающее число | TDateTime | class |
| Currency | Валюта, 64-битовое плавающее число, 4 цифры после точки | Currency | class |

## 2.3. Операции языка С

Операции языка С

Таблица 2.4.

| Операция | Описание | Приоритет | Ассоциация |
|---|---|---|---|
| **Первичные и постфиксные операции** | | | |
| **[]** | индексация массива | 16 | слева направо |
| **()** | вызов функции | 16 | слева направо |
| **.** | элемент структуры | 16 | слева направо |
| -> | элемент указателя | 16 | слева направо |
| ++ | постфиксный инкремент | 15 | слева направо |
| -- | постфиксный декремент | 15 | слева направо |
| **Одноместные операции** | | | |
| ++ | префиксный инкремент | 14 | справа налево |
| -- | префиксный декремент | 14 | справа налево |
| **sizeof** | размер в байтах | 14 | справа налево |
| **(тип)** | приведение типа | 14 | справа налево |
| **~** | поразрядное NOT | 14 | справа налево |
| **!** | логическое NOT | 14 | справа налево |
| - | унарный минус | 14 | справа налево |
| **&** | взятие адреса | 14 | справа налево |
| * | разыменование указателя | 14 | справа налево |
| **Двухместные и трехместные операции** | | | |

| | | | |
|---|---|---|---|
| **Мультипликативные** | | | |
| * | умножение | 13 | слева направо |
| / | деление | 13 | слева направо |
| % | взятие по модулю | 13 | слева направо |
| **Аддитивные** | | | |
| + | сложение | 12 | слева направо |
| - | вычитание | 12 | слева направо |
| **Поразрядного сдвига** | | | |
| << | сдвиг влево | 11 | слева направо |
| >> | сдвиг вправо | 11 | слева направо |
| **Отношения** | | | |
| < | меньше | 10 | слева направо |
| <= | меньше или равно | 10 | слева направо |
| > | больше | 10 | слева направо |
| >= | больше или равно | 10 | слева направо |
| == | равно | 9 | слева направо |
| ! = | не равно | 9 | слева направо |
| **Поразрядные** | | | |
| & | поразрядное AND | 8 | слева направо |
| ^ | поразрядное XOR | 7 | слева направо |
| \| | поразрядное OR | 6 | слева направо |
| **Логические** | | | |
| && | логическое AND | 5 | слева направо |
| \|\| | логическое OR | 4 | слева направо |
| **Условные** | | | |
| ? : | условная операция | 3 | справа налево |
| **Присваивания** | | | |
| = | присваивание | 2 | справа налево |
| *= | присвоение произведения | 2 | справа налево |
| /= | присвоение частного | 2 | справа налево |
| %= | присвоение модуля | 2 | справа налево |
| += | присвоение суммы | 2 | справа налево |
| -= | присвоение разности | 2 | справа налево |
| <<= | присвоение левого сдвига | 2 | справа налево |
| >>= | присвоение правого сдвига | 2 | справа налево |
| &= | присвоение AND | 2 | справа налево |
| ^= | присвоение XOR | 2 | справа налево |
| \|= | присвоение OR | 2 | справа налево |
| , | запятая | 1 | слева направо |

**Арифметические операции.** Арифметические операции применяются к вещественным, целым числам и указателям. Определены следующие бинарные арифметические операции:

Для арифметических операций действуют следующие правила. Бинарные операции сложения (+) и вычитания (-) применимы к целым и действительным числам, а также к указателям.

В операции сложения указателем может быть только один из двух операндов. В этом случае второй операнд должен быть целым числом.

Указатель, участвующий в операции сложения, должен быть указателем на элемент массива. В этом случае добавление к указателю целого числа эквивалентно сдвигу указателя на заданное число элементов массива.

В операции вычитания указатель на элемент массива может быть первым операндом (тогда второй операнд - целое число) или оба операнда могут быть указателями на элементы одного массива. Вычитание из указателя целого числа эквивалентно сдвигу указателя на заданное число элементов массива. Вычитание двух указателей возвращает число элементов массива, расположенных между теми элементами, на которые указывают указатели.

В операциях умножения (*) и деления (/) операнды могут быть любых арифметических типов. При разных типах операндов применяются стандартные правила автоматического приведения типов. В операции вычисления взятие по модулю (остатка от деления) (%) оба операнда должны быть целыми числами.

В операциях деления и вычисления остатка второй операнд не может быть равен нулю. Если оба операнда в этих операциях целые, а результат деления является не целым числом, то знак результата вычисления остатка совпадает со знаком первого операнда, а для операции деления используются следующие правила:

Если первый и второй операнд имеют одинаковые знаки, то результат операции деления - наибольшее целое, меньшее истинного результата деления.

Если первый и второй операнд имеют разные знаки, то результат операции деления - наименьшее целое, большее истинного результата деления. Округление всегда осуществляется по направлению к нулю.

Определены следующие унарные арифметические операции:

**Некоторые унарные операция алгоритмического языка C++ Builder**

***Замечание.*** *Унарные операция имеет самый высокий приоритет, и при работе с ними желательно всегда учесть это обстоятельство*.

Унарные операции инкремента (++) и декремента (--) сводятся к увеличению (+) или уменьшению (-) операнда на единицу. Операции применимы к операндам, представляющим собой выражения любых арифметических типов или типа указателя. Причем выражение должно быть модифицируемым L-значением, т.е. должно допускать изменение. Например, ошибочным является выражение ++(a+b), поскольку (a+b) не является переменной, которую можно модифицировать.

Операции инкремента и декремента выполняются быстрее, чем обычное сложение и вычитание. Поэтому, если переменная ***a*** должна

быть увеличена на 1, лучше применить операцию (++), чем выражения a=a+1 или оператор a+=1.

Если операция инкремента или декремента помещена перед переменной, говорят о префиксной форме записи инкремента или декремента. Если операция инкремента или декремента записана после переменной, то говорят, о постфиксной форме записи. При префиксной форме переменная сначала увеличивается или уменьшается на единицу, а затем это ее новое значение используется в том выражении, в котором она встретилась. При постфиксной форме в выражении используется текущее значение переменной, и только после этого ее значение увеличивается или уменьшается на единицу.

Например, в результате выполнения операторов:

```
int i = 1, j;
 j = i++ * i++;
```

значение переменной i будет равно 3, а переменной j = 1. Оператор, присваивающий значение переменной j будет работать следующим образом сначала значение i, равное 1, умножится само на себя и присваивается j, затем значение i увеличится на 1 в результате первой операции инкремента и еще раз увеличится на 1 в результате второй операции инкремента.

Если изменить эти операторы следующим образом:

```
int i = 1, j;
j = ++i * ++i;
```

то результат будет другим: значение i будет равно 3, а значение j = 9. В этом случае оператор, присваивающий значение переменной j будет работать следующим образом: сначала выполнится первая операция инкремента, и значение i станет равно 2; затем выполнится вторая операция инкремента, и значение i станет равно 3; а затем это значение i умножится само на себя.

**Поразрядные операции и сдвиги.** Эти операции применяются к целочисленным данным, которые рассматриваются просто как набор отдельных битов.

При *поразрядных операциях* каждый бит одного операнда комбинируется (в зависимости от операции) с одноименным битом другого, давая бит результата. При единственной одноместной поразрядной операции — отрицании (~) — биты результата являются инверсией соответствующих битов ее операнда.

При *сдвиге влево* биты первого операнда перемещаются влево (в сторону старших битов) на заданное вторым операндом число позиций.

Старшие биты, оказавшиеся за пределами разрядной сетки, теряются; справа результат дополняется нулями.

Результат *сдвига вправо* зависит от того, является ли операнд знаковым или без знаковым. Биты операнда перемещаются вправо на заданное число позиций. Младшие биты теряются. Если операнд — целое со знаком, производится *расширение знакового бита* (старшего), т. е. освободившиеся позиции принимают значение 0 в случае положительного числа и 1 — в случае отрицательного. При без знаковом операнде старшие биты заполняются нулями.

Сдвиг влево эквивалентен умножению на соответствующую степень двойки, сдвиг вправо — делению. Например,

```
Num = Num <<4;
```

умножает Num на **16.**

**Операции присваивания.** Операция присваивания (=) не представляет особых трудностей. При ее выполнении значением переменной в левой части становится результат оценки выражения справа. Как уже говорилось, эта операция сама возвращает значение, что позволяет, например, написать:

```
a = b = c = someVar;
```

После исполнения такого оператора все три переменных a, b, c получат значение, равное someVar. Что касается остальных десяти операций присваивания, перечисленных в таблице, то они просто служат для сокращенной нотации присваивании определенного вида. Например, `s += i;` эквивалентно `s = s + i;`

Другими словами, оператор вроде `x *= 3;` означает «текущее значение переменной x умноженное на 3 присвоить x».

**Запятая.** Помимо того, что запятая в C служит разделителем различных списков (как в списке параметров функции), она может использоваться и как операция. Запятая в этом качестве также является разделителем, но обладает некоторыми дополнительными свойствами.

Везде, где предполагается выражение, может использоваться список выражений, возможно, заключенный в скобки (так как операция-запятая имеет наинизший приоритет). Другими словами,

```
Выражение1, выражение2[, ...]
```

также будет выражением, оценкой которого является *значение последнего элемента списка.* При этом операция-запятая *гарантирует,* что оценка выражений в списке будет производиться по порядку слева направо. Вот два примера с операцией-запятой:

```
i++, j++;
// Значение выражения игнорируется.
```

```
x = (j = 4, j += n, j++);
// x присваивается n + 4. j равно n + 5.
```

Операция-запятая применяется довольно редко, обычно только в управляющих выражениях циклов.

**Арифметические и алгебраические функции.** В языке C++ имеется большое количество математических функций, приведем часть наиболее часто используемые. Для их использования в тексте программы необходимо подключить заголовочный файл, например: #include <math.h>.

Некоторые наиболее часто используемые математические функции C++

Таблица 2.5.

| **Функция** | **Описание** | **Синтаксис** | **Файл** |
|---|---|---|---|
| **abs** | Абсолютное значение целого x | int abs(int x) | stdlib.h |
| **cabs** | Модуль комплексного числа z | double cabs(struct complex z)<br>struct complex {double x, y;}; | math.h |
| **cabsl** | Модуль комплексного числа z | long double cabsl(struct _complexl z)<br>struct complex { long double x, y; }; | math.h |
| **Ceil** | Округление вверх, наименьшее целое не меньше x | double ceil(double x); | math.h |
| **Ceil** | Округление вверх , наименьшее целое не меньше X | int Ceil(Extended X); | Math.hpp |
| **Ceil** | Округление вверх , наименьшее целое не меньше x | long double ceill(long double x); | math.h |
| **Div** | Целочисленное деление numer/denom | typedef struct {<br>int quot; // ÷частное<br>int rem; // - остаток<br>} div_t;<br>div_t div(int numer, int denom) | math.h |
| **Exp** | Экспонента | double exp(double x); | math.h |
| **expl** | Экспонента | long double expl(long double x); | math.h |
| **fabs** | Абсолютное значение | double fabs(double x); | math.h |
| **fabsl** | Абсолютное значение | long double fabsl(long double x); | math.h |
| **floor** | Округление вверх, наименьшее целое не меньше x | double floor(double x); | math.h |
| **Floor** | Округление вверх , наименьшее целое не меньше X | int Floor(Extended X); | Math.hpp |
| **floorl** | Округление вверх , наименьшее целое не меньше | long double floorl(long double x); | math.h |

| | | | |
|---|---|---|---|
| | x | | |
| **fmod** | Остаток от деления x/y | double fmod(double x, double y) | math.h |
| **fmodl** | Остаток от деления x/y | long double fmodl(long double x, long double y) | math.h |
| **frexp** | Разделяет x на мантиссу (возвращает) и степень exponent | double frexp(double x, int *exponent); | math.h |
| **Frexp** | Разделяет X на мантиссу Mantissa (возвращает) и степень Exponent | void Frexp(Extended X, Extended &Mantissa, int &Exponent); | Math.hpp |
| **frexp** | Разделяет x на мантиссу (возвращает) и степень exponent | long double frexpl(long double x, int *exponent) | math.h |
| **IntPower** | Возводит **Base** в целую степень **Exponent** | Extended IntPower(Extended Base, int Exponent); | Math.hpp |
| **Labs** | Абсолютное значение | long labs(long int x); | stdlib.h |
| **ldiv_t** | Целочисленное деление numer/denom; quot – результат rem – остаток | typedef struct {<br>long int quot; // целое<br>long int rem; // остаток<br>} ldiv_t<br>ldiv_t ldiv(long int numer, long int denom); | math.h<br>stdlib.h |
| **Log** | Натуральный логарифм | double log(double x); | math.h |
| **LnXP1** | Натуральный логарифм (X+1) | Extended LnXP1(Extended X); | Math.hpp |
| **Log2** | Логарифм по основанию 2 | Extended Log2(Extended X) | Math.hpp |
| **log10** | Десятичный логарифм | double log10(double x) | math.h |
| **Log10** | Десятичный логарифм | Extended Log10(Extended X) | Math.hpp |
| **log10l** | Десятичный логарифм | long double log10l(long double x) | math.h |
| **Logl** | Натуральный логарифм | long double logl(long double x) | math.h |
| **LogN** | Логарифм X по основанию **Base** | Extended LogN(Extended Base, Extended X) | Math.hpp |
| **_lrotl** | Циклический сдвиг влево **val** на **count** битов | unsigned long _lrotl(unsigned long val, int count) | stdlib.h |
| **_rotr** | Циклический сдвиг вправо **val** на **count** битов | unsigned long _lrotr(unsigned long val, int count) | stdlib.h |
| **Max** | Макрос возвращает максимальное значение из a и b любых типов | max(a, b); | stdlib.h |
| **Min** | Макрос возвращает минимальное значение из a и b любых типов | min(a, b) | stdlib.h |
| **modf** | Разделяет x на целую часть **ipart** и возвращает дробную часть. | double modf(double x, double *ipart) | math.h |
| **modfl** | Разделяет x на целую часть | long double modfl(long double x, | math.h |

| | | | |
|---|---|---|---|
| | **ipart** и возвращает дробную часть. | long double *ipart) | |
| **Poly** | Полином $c[n]x^n + c[n-1]x^{n-1} + ... + c[1]x + c[0]$ от x степени **degree** коэффициентами **coeffs** | double poly(double x, int n, double c[ ]); | math.h |
| **Poly** | Полином от X степени C**oefficients_Size** коэффициентами C**oefficients** | Extended Poly(Extended X, const double * Coefficients, const int Coefficients_Size); | Math.hpp |
| **polyl** | Полином от от x степени **degree** коэффициентами **coeffs** | long double polyl(long double x, int degree, long double coeffs[]); | math.h |
| **Pow** | $x^y$ | | math.h |
| **Power** | Возводит **Base** в степень **Exponent** | Extended Power(Extended Base, Extended Exponent); | Math.hpp |
| **powl** | $x^y$ | long double powl(long double x, long double y); | math.h |
| **_lrotl** | Циклический сдвиг влево **value** на **count** битов | unsigned short _rotl(unsigned short value, int count); | stdlib.h |
| **_rotr** | Циклический сдвиг вправо **value** на **count** битов | unsigned short _rotr( unsigned short value, int count); | stdlib.h |
| **Sqrt** | Корень квадратный | double sqrt(double x); | math.h |
| **Sqrtl** | Корень квадратный | long double sqrtl(long double x); | math.h |
| **acos** | Функция арккосинуса. Значение аргумента должно находиться в диапазоне от -1 до +1. | double acos(double x); | math.h, cmath |
| **Asin** | Функция арксинуса. Значение аргумента должно находиться в диапазоне от -1 до +1. | double asin(double x); | math.h, cmath |
| **atan** | Функция арктангенса. | double atan(double x); | math.h, cmath |
| **atan2** | Функция арктангенса от значения y/x. | double atan2(double y, double x); | math.h, cmath |
| **Cos** | Функция косинуса. Аргумент задается в радианах. | double cos(double x); | math.h, cmath |
| **frexp** | Разбивает число с плавающей точкой value на нормализованную мантиссу и целую часть как степень числа 2. Целочисленная степень записывается в область памяти, на которую указывает exp, а мантисса используется как значение, которое возвращает функция. | double frexp(double value, int *exp); | math.h, cmath |
| **hypot** | Вычисляет гипотенузу z прямоугольного треугольника | double hypot(double x, double y); | math.h, cmath |

| | | | |
|---|---|---|---|
| | по значениям катетов x, y: | | |
| **pow10** | Возвращает значение $10^p$ | double pow10(int p); | math.h, cmath |
| **Sin** | Функция синуса. Угол задается в радианах. | double sin(double x); | math.h |
| **Sinh** | Возвращает значение гиперболического синуса для x. | double sinh(double x); | math.h |
| **Tan** | Функция тангенса. Угол задается в радианах. | double tan(double x); | math.h |
| **Tanh** | Возвращает значение гиперболического тангенса для x. | double tanh(double x); | math.h |

## 2.4. Структура программ C++ builder

**Задание 2.1.** Нарисовать блок-схему и составить программу вычисления ***x*** и ***y*** математического выражения

$$\begin{cases} a = \dfrac{\sqrt{x+y}}{\cos(y)}\left(1+e^{x+y}\right) \\ b = xa^2 + ay. \end{cases}$$ для выводимых значений a, b.

Панель диалога программы организовать в виде, представленном на рис. 2.1.

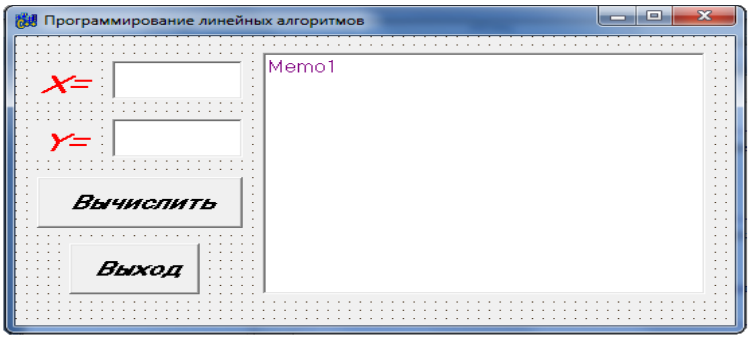



Рис. 2.1. Эскизный проект задания

Составим блок – схему алгоритма для нашего задания. Для создания блок - схемы мы используем рассмотренные выше графические обозначения.

Блок – схемы алгоритма имеет вид

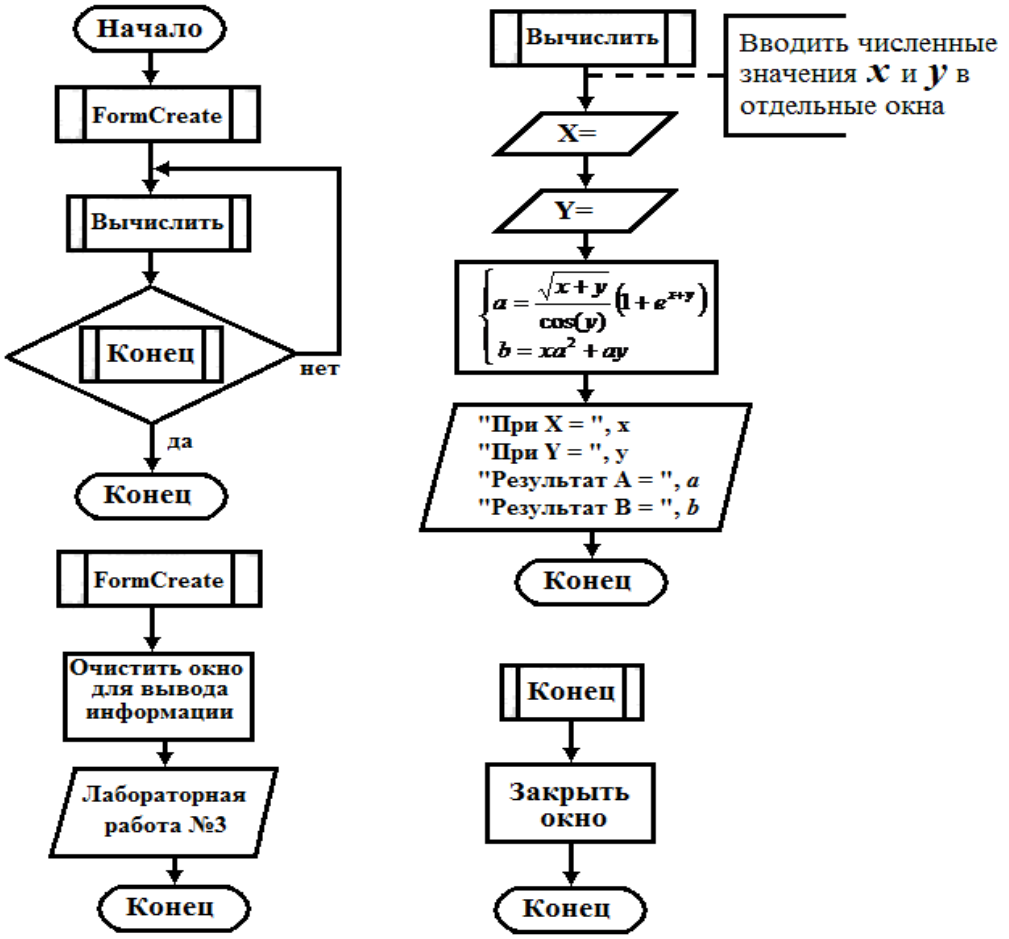



Рис. 2.2. Блок - схема задания

**Настройка формы.** Сначала начнем создавать эскизный проект (визуальный интерфейс пользователя).

В предыдущем примере «Здравствуй мир!», мы задали размеры с помощью мыши, «захватывая» одну из кромок формы или выделенную строку заголовка, и отрегулировали нужные размеры формы и ее положение на экране. Однако чаще всего при выполнении программ иногда нужно бывают определение параметров свойств, либо вручную, либо внутри программы. В нашем примере попытаемся задать свойства окна через Object Inspector вручную.

Некоторые особо значимые свойства мы описали при изучении темы 1.

Расширим тот перечень свойств.

**Некоторые свойства формы (продолжение. Начало см. работа №1)**

**Name -** Имя формы. Используется для обращения к компонентам формы, для управления формой и т.д.

**Caption -** Текст заголовка.

**Width** – Ширина формы.

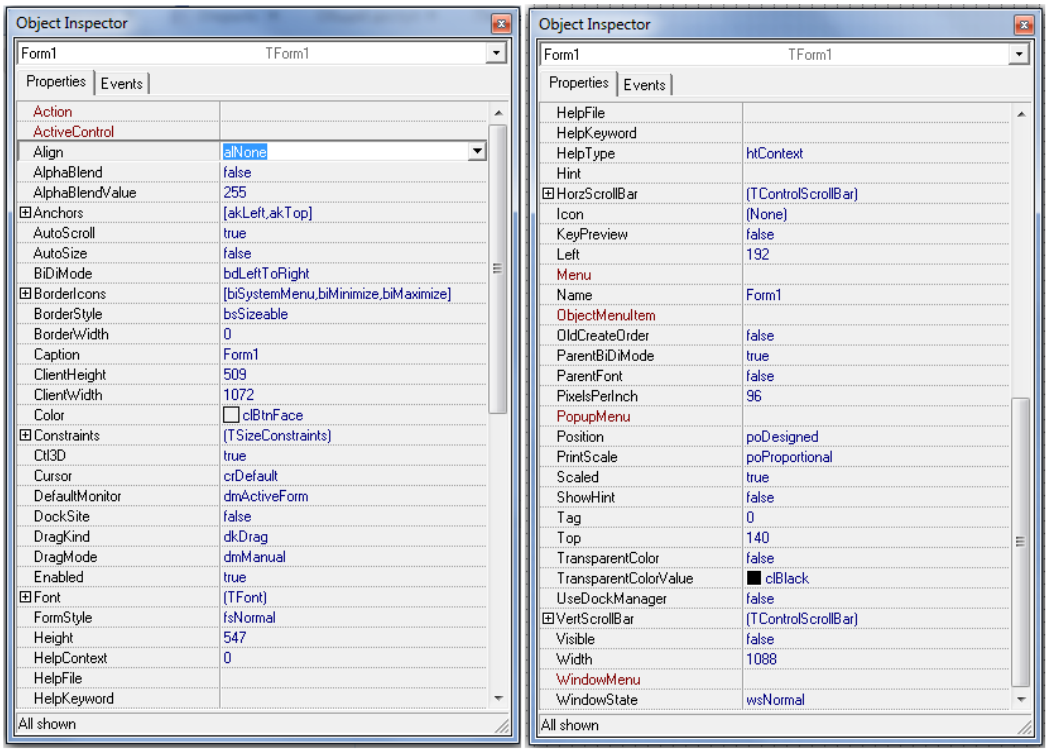
**Height** – Высота формы

**Top** – Расстояние от верхней границы формы до верхней границы

экрана.

**Left** – Расстояние от левой границы формы до левой границы экрана.

**Полный перечень свойств формы (объекта TForm) C++ Builder 6**




**а) начало списка свойств** **б) продолжение списка свойств**

Рис. 2.3. Свойства формы.

Рекомендуемые параметры для формы в задании

Таблица 2.6.

| **Свойства** | **Параметры и значения** |
|---|---|
| Caption | Программирование линейных алгоритмов |
| Width | 551 |
| Height | 317 |
| Left | 196 |
| Top | 108 |
| BorderIcon.biMinimize | False |
| BorderIcon.biMaximize | False |

**Color** – Цвет фона. По умолчанию принимает значение операционной системы, или можно задать нужный цвет.

**Font** – Шрифт. Шрифт может наследовать компоненты расположенные на форме, имеется возможность его запрещения.

Введем теперь некоторые параметры, приведенные в таблице 2.6.

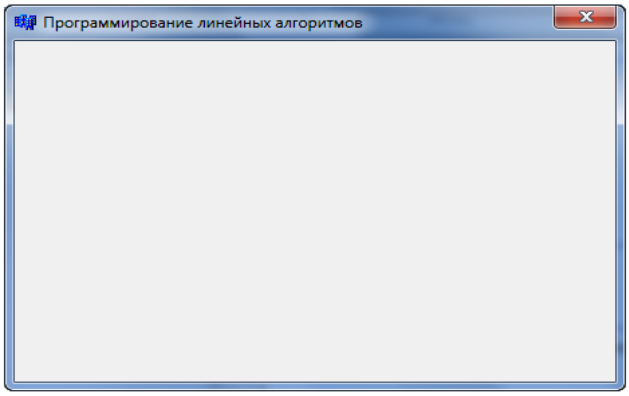



Рис. 2.4. Вид формы с параметрами из таблицы 2.7. после старта программы

**Настройка компонент.** Аналогично можно задавать значения и изменять параметры для свойства для любых компонентов используемых в проекте. Далее мы будем добавлять компоненты Button1, Button2, Label1, Label2, Edit1, Edit1 и Memo1. И расстановим их как указанно в задании.

С компонентой Label мы знакомы при изучении работы №1

**Компонента Edit (*Standard)* - ab| ввод/вывод текстовой информации.** Если необходимо ввести из формы в программу или вывести на форму информацию, которая вмещается в одну строку, используют окно однострочного редактора текста, представляемого компонентой ***Edit***, для чего в меню компонент выбирается пиктограмма ***Edit*** и щелчком кнопкой мыши устанавливается в нужном месте формы. Размер окошка и его положение на форме можно регулировать и в ручном режиме, используя манипулятор мышь.

При этом в заголовочный файл *Unit*1.*h* автоматически вставляется переменная *Edit** (1,2,…) класса *TEdit*. В поле *Text* (*Edit*1–>*Text*) такой переменной будет содержаться строка символов (тип *AnsiString*) и отображаться в соответствующем окне *Edit**.

Щелкнув мышью на Edit1 в инспекторе объектов на свойстве text удалите запись Edit1, и аналогично Edit2

Для работы с компонентой ***Edit*** в C++ Builder существуют стандартные функции, которые позволяют тип *AnsiString* преобразовать в числовые типы. Приведем их в таблице 2.7.

Таблица 2. 7.

| **Наименование** | **Тип** | **Действия** |
|---|---|---|
| ***StrToFloat***(*St*) | float | преобразует строку *St* в вещественное число |
| ***StrToInt***(*St*) | int | преобразует строку *St* в целое число |
| ***FloatToStr***(*W*) | *AnsiString* | преобразует вещественное число *W* в строку символов |

| | | |
|---|---|---|
| ***FormatFloat*** (*формат, W*) | *AnsiString* | преобразует вещественное число *W* в строку |
| ***IntToStr*** (*W*) | *AnsiString* | преобразует целое число *W* в строку символов |
| ***FloatToStrF***( *W, формат, n*1, *n*2) | *AnsiString* | вещественное число *W* в строку символов под управлением *формата*: |
| ***ffFixed*** | фиксированное положение разделителя целой и дробной частей, *n*1 – общее количество цифр числа, *n*2 – количество цифр в дробной части, причем число округляется с учетом первой отбрасываемой цифры | |
| ***fFfExponent*** | *n*1 задает общее количество цифр мантиссы, *n*2 – количество цифр порядка *XX* (число округляется) | |
| ***ffGeneral*** | – универсальный формат, использующий наиболее удобную для чтения форму представления вещественного числа; соответствует формату *ffFixed*, если количество цифр в целой части ≤ *n*1, а само число больше 0,00001, в противном случае соответствует формату *ffExponent*. | |

Например, если значения вводимых из *Edit*1 и *Edit*2 переменны *x* и *y* имеют целый и действительный типы, соответственно, то следует записать:

```
x = StrToInt(Edit1–>Text);
y = StrToFloat(Edit2–>Text);
```

**Внимание!** При записи числовых значений в окошках Edit* не должно быть пробелов, а разделителем целой и дробной частей обычно является «запятая»!

Как и для работы с компонентой Label, смотрите тему 1, и для компоненты *Edit* можно в инспекторе объектов с помощью свойства ***Font*** устанавливать стиль, отражаемого в строке *Edit** текста.

**Компонента Memo(*Standard)* - многострочный ввод/вывод текстовой информации.** Для вывода результатов работы программы обычно используется окно многострочного редактора текста, представленое компонентой *Memo*, для чего выбирается пиктограмма Memo, помещается на форму, регулируется ее размер и местоположение. После установки с помощью инспектора свойства *ScrollBars – SSBoth* в окне появятся вертикальная и горизонтальная полосы прокрутки.

При установки данной компонеты в *Unit*1.*h* прописывается переменная *Memo*1 типа *TMemo*. Информация, выводимая построчно в окне *Memo*1, находится в массиве строк *Memo*1–>*Lines*, каждая из которых имеет тип *String*.

Для очистки окна используется метод *Memo*1–>*Clear*( ).

Для добавления новой строки используется метод *Memo*1–>*Lines*–>*Add*( ).

Если нужно вывести числовое значение, то его надо преобразовать к типу *AnsiString* (см. прил. 4) и добавить в массив *Memo*1–>*Lines*, например, вывести

```
int u = 100;
double w = -256.38666;
```

в результате записей

```
Memo1->Lines->Add (" Значение x  = "+IntToStr(u));
Memo1->Lines->Add (" Значение y = "+FloatToStrF(w,ffFixed,8,2));
```

появятся строки

```
Значение u = 100
Значение w = -256.39
```

При этом под все число отводится восемь позиций, из которых две позиции занимает его дробная часть.

Если выводимая информация превышает размер окна *Memo*1, то для просмотра используются полосы прокрутки.

**Компонента Button (*Standard)* - функция обработчик нажатия Кнопки.** Выбрав в меню *Standard* пиктограмму *Button*, помещаем на форму компоненту *Button*1 (2,3,…). С помощью инспектора объектов изменяем заголовок (*Caption*) на текст, «*Вычислить*» на 1 кнопке и «Выход» на второй, и регулируем положение и размер кнопки. Двойным щелчком кнопкой мыши по компоненте *Button* 1 в текст программы вставляем заготовку ее функции-обработчика **… *Button*1*Click* (…) { }**. Между фигурными скобками набираем соответствующий код. Аналогично и ***Button2Click* (…) { }**

## 2.5. Обработка событий в среде C++ Builder *FormCreate*

При запуске программы возникает событие «создание формы» (*OnCreate*). Оформим функцию-обработчик этого события, которая обычно используется для инициализации начальных установок, таких, как, например, занести начальные значения исходных данных в соответствующие окна *Edit**, очистить окно *Memo*.

Для этого делаем двойной щелчок кнопкой мыши на любом ***свободном*** месте формы, после чего в листинг программы (*Unit*1.*cpp*) автоматически вносится заготовка для создания функции: ее заголовок … ***FormCreate* (…)** *и фигурные скобки*.

Между символами { }, которые обозначают начало и конец функции, соответственно, вставляем нужный текст программы.

**Ввод в окно Редактора кода программный код .** Как, видим из окна Object TreeView Рис. 1, наш интерфейс состоит из следующих компонентов: Button1, Button2, Label1, Label2, Edit1, Edit1 и Memo1. Соответственно это информация отображается в файле заголовка.

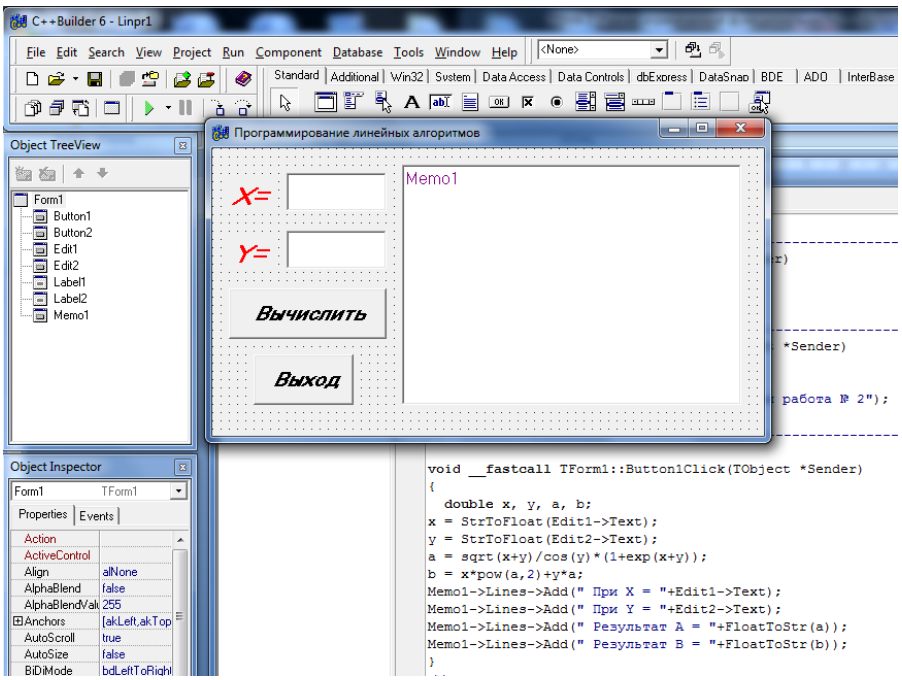



Рис.2.5. Интерфейс для решения задач по линейным алгоритмам.

```
//---------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------
class TForm1 : public TForm
{
__published:    // IDE-managed Components
 TLabel *Label1;      //метка для вывода a=
 TEdit *Edit1;        /откуда берем /значение переменной a
 TLabel *Label2;      //метка для вывода b=
 TEdit *Edit2;        //откуда берем значение переменной b
 TMemo *Memo1;        //
 TButton *Button1;    //
 TButton *Button2;    //
private:   // User declarations
      /* Здесь можно объявить функции, переменные, к которым
      получаем доступ только в данном модуле, поэтому их
      называют закрытыми переменными и функциями*/
public:    // User declarations
      /* Здесь можно объявить функции, переменные, к которым
      получаем доступ, как в данном модуле, так и в других
      модулях, если осуществляем ссылку на объект, поэтому их
      называют открытыми переменными и функциями */
 __fastcall TForm1(TComponent* Owner);
void __fastcall TForm1::FormCreate(TObject *Sender);
void __fastcall TForm1::Button1Click(TObject *Sender);
```

```
void __fastcall TForm1::Button2Click(TObject *Sender):
};
//---------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
      /* Объявления функций, типов, переменных, которые не
      включаются в данный класс */
//---------------------------------------------------------------
#endif
```

Щелкните по кнопкам «Вычислить» и «Выход» на окне Form и в полученных местах запишите программный код.

Файл проекта имеет следующую структуру:

```
// Директивы препроцессора
 #include <vcl.h>
 #pragma hdrstop
 #include "Linpr.h"
//---------------------------------------------------------------
 #pragma package(smart_init)
 #pragma resource "*.dfm" // Подключение файлов форм и файлов
ресурсов
    TForm1 *Form1;
    //---------------------------------------------------------
// Вызов конструктора формы
    __fastcall TForm1::TForm1(TComponent* Owner)
            : TForm(Owner)
    {
    }
void __fastcall TForm1::FormCreate(TObject *Sender)
{
     Memo1->Clear();
Memo1->Lines->Add("  Лабораторная работа № 2");
}
//---------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        double x, y, a, b;
      x = StrToFloat(Edit1->Text);
      y = StrToFloat(Edit2->Text);
      a = sqrt(x+y)/cos(y)*(1+exp(x+y));
      b = x*pow(a,2)+y*a;
      Memo1->Lines->Add(" При X = "+Edit1->Text);
      Memo1->Lines->Add(" при Y = "+Edit2->Text);
      Memo1->Lines->Add(" Результат A = "+FloatToStr(a));
      Memo1->Lines->Add(" Результат B = "+FloatToStr(b));
}
//---------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
 Form1->Close();
}
```

## 2.6. Индивидуальные задания

Для вводимых с клавиатуры действительных чисел a, и b используя математические выражения для своего варианта в таблице вычислить x и y.

Результаты записать в компонент Memo1.

**Примечание:** для ввода a, b подбирать, учитывая конфликтные ситуация в математических функциях, т.е. исследовав область определения и область допустимых значений функций.

### Варианты заданий

| № | Задания | № | Задания |
|---|---|---|---|
| 1 | $a = \dfrac{\lvert x\rvert - \lvert y\rvert}{1+\lvert xy\rvert};\ b = \dfrac{e^{2x}}{\ln\cos y}.$ | 2 | $a = \dfrac{\sqrt{\lvert x - 3 + \sqrt{x}\rvert}}{1 + x/y};$ $b = arctg(x^3) + \cos(a).$ |
| 3 | $a = \dfrac{\cos x}{\cos^2 x}\sqrt{\ln y};\ b = xy\sqrt{\ln x\sqrt{\lvert y\rvert}}.$ | 4 | $a = \dfrac{\lvert x^3\rvert}{1+ctgy};\ b = \arcsin\sqrt[5]{xy}.$ |
| 5 | $a = \dfrac{x\cos\sqrt{\lvert xy\rvert}}{1+xy};\ b = \sin\cos xy.$ | 6 | $a = \dfrac{\left(3+e^{y-3}\right)^2}{7+xy};\ b = arctgx + e^{-(x+y)}.$ |
| 7 | $a = arctg\dfrac{x+y}{\ln x};$ $b = \sin e^{x-y} \big/ (x + e^{tgx}).$ | 8 | $a = \dfrac{\sin\cos x}{\cos^2 x}\sqrt{\ln xy};\ b = xy\sqrt{\ln x}.$ |
| 9 | $a = \dfrac{x\cos\sqrt{\lvert xy\rvert}}{1+xy};\ b = \cos e^x \big/ (x^3 + e^{tgx}).$ | 10 | $a = \dfrac{\left(x+e^y\right)^2}{\sin x + x^y y};\ b = tgx + e^{-(x+y)}.$ |
| 13 | $a = xy\dfrac{\cos x\sqrt{\sin x}}{e^x};\quad b = arctg(\sin x).$ | 14 | $a = \dfrac{\sin(e^{-x})}{tg(e^{3x})};\ b = \dfrac{e^{\sin x}}{tge^{\cos x}}.$ |
| 15 | $a = \cos\left[e^{xy}\right];\ b = -\sqrt{\sin\sin x}.$ | 16 | $a = \cos x + x^7 y^3;$ $b = \left[\sin x + \sin(x^3 + y^3)\right].$ |
| 17 | $a = \dfrac{x - y^2}{x^4};\ b = \sin x + \cos^2 x^2.$ | 18 | $a = \sqrt{xy\lvert e^x + y^3 e^{3x}\rvert};$ $b = \cos xy + \sin xy.$ |
| 19 | $a = \dfrac{\sin x - y^2}{e^x};$ $b = \left[\sin x^x + \sin(x^3 + y^3)\right].$ | 20 | $a = \dfrac{\left(\sin x + e^{yx}\right)^2}{\cos x + x^y};$ $b = arctg(\sin^y xy).$ |
| 21 | $a = \dfrac{x\sqrt{\lvert xy\rvert}}{1+xy};\ b = xye^x \big/ (x^3 + e^{tgx}).$ | 22 | $a = \sqrt[4]{\lvert x - y\rvert}\cos x;\ b = \dfrac{\cos yx}{e^{\sin x}}$ |
| 23 | $a = \dfrac{\lvert x^3 + \sin x\rvert}{1+tgy};\ b = \dfrac{x + \arcsin y}{arctgx}.$ | 24 | $a = \dfrac{x^3 + y^5}{2 + x^2};\ b = \dfrac{\lvert x - \cos x^2\rvert}{\lvert yx - x^2 + y^3\rvert}.$ |

| | | | |
|---|---|---|---|
| 25 | $a=\frac{\lvert x^3+y\rvert}{1+tge^{y}};\ b=\arccos\sqrt[3]{xy}.$ | 26 | $a=\frac{\cos\cos x^3}{\cos^2 x+sixy}\sqrt{\ln y};$ $b=xy\sqrt{x\sqrt[3]{\lvert y\rvert}}.$ |
| 27 | $a=\frac{x\sqrt{\lvert xy\rvert}}{\sin x+xy};\ b=e^{x}\Big/(x^3+e^{tgx}).$ | 28 | $a=\frac{xy\sqrt{\lvert\cos xy\rvert}}{e^{x}+xy};$ $b=\cos e^{x}\Big/(x^3+e^{tgx}).$ |
| 29 | $a=\frac{x\sqrt[5]{x^3}}{y^2};\ b=\sqrt{xy}+\frac{\sin\cos x}{\cos\sin y}.$ | 30 | $a=\frac{x^{y}}{y^{x}};\ b=x+\left\lvert\frac{x-y}{3x-\sin x}\right\rvert.$ |

## 2.7. Контрольные вопросы

1. Понятия алгоритма и его основные свойство.
2. Какие базовые группы жизненого цикла программы?
3. Основные этапы разработки программы.
4. Основные понятия формальных языков и грамматик.
5. Алфавит языка C++ Builder.
6. Типы переменных и констант в языке C++ Builder.
7. Унарные операция в языке C++ Builder.
8. Арифметические и алгебраические функции в языке C++ Builder.
9. Основные свойство формы в C++ Builder.
10. Основные копоненты C++ Builder.

## Лабораторная работа № 3.
## Алгоритмы разветвляющихся структур

**Цель лабораторной работы**: изучение некоторых основных диаграмм UML; основы языка C++ Builder, научиться пользоваться простейшими компонентами организации переключений (классов TCheckBox, TRadioGroup и другими); научиться составлять алгоритмы разветвляющихся структур простейшей программы в среде C++ Builder и научиться осуществлять обработку конфликтных ситуаций в линейных алгоритмах.

### 3.1. Технологии программирования

**Технология быстрой разработки приложений.** Современный C++Builder – это громадный набор различных средств, которые позволяют повышать производительность труда программистов и сокращают продолжительность цикла разработки. Многофункциональная интегрированная среда разработки C++Builder включает компилятор, удовлетворяющий стандарта ANSI/ISO, встроенный дизайнер форм, богатейший набор средств для разработки проектов, с которыми мы познакомились в предыдущих темах. Однако C++Builder имеет преимущество перед другими языками при использовании традиционных интерфейсных оболочек других систем. В-первых язык C++Builder предназначен как для быстрой разработки приложений (RAD), построенных на современном фундаменте объектно-ориентированного программирования (ООП), так и сам постепенно помогает овладевать различными возможностями RAD, ООП и языка C++, поначалу, требуя лишь минимальных предварительных знаний языка. Во - вторых, наверное, вы заметили, код программы был довольно прост, но мы создали интерфейсную оболочку с различными кнопками, компонентами для ввода и вывода информации, используя готовые компоненты, лишь изменяя свойства. В-третьих, навыки программирования расширяются по мере роста сложности ваших разработок: чем сложнее задача, тем больший по объему программный код потребуется написать для ее реализации. В-четвертых, получение знаний перестает быть самоцелью, в этом убедитесь сами. Программирование на C++Builder в корне меняет процесс разработки проекта, и насколько легче и быстрее вы сможете получать работающие и надежные программы, выполняющие различные пользовательские потребности.

Однако по мере роста опыта в программировании, появляется задачи новой ступени сложности, где возможно и необходимо использовать различные парадигмы и концепции, и в данном случае, язык C++Builder остается на высоте, т.к. поддерживает несколько парадигмов: структурного, процедурного и др.

**Парадигма объектно-ориентированного программирования.** Введем историческую справку. Парадигма ООП возникло в результате развития идеологии процедурного программирования, где данные и подпрограммы (процедуры, функции) их обработки формально не связаны.

Первым языком программирования, в котором были предложены принципы объектной ориентированности, была Симула. В момент своего появления (в 1967 году), этот язык программирования предложил поистине революционные идеи: объекты, классы, виртуальные методы и др., однако это всё не было воспринято современниками как нечто грандиозное. Благодаря трудам и инженеринга мыслей в области ООП, большинство концепций были развиты Аланом Кэйем, Дэном Ингаллсом и в языке Smalltalk, который стал первым широко распространённым объектно-ориентированным языком программирования.

В ООП часто большое значение имеют понятия события (так называемое событийно-ориентированное программирование) и компонента (компонентное программирование).

Хотя теперь количество алгоритмических языков программирования, реализующих объектно-ориентированную парадигму, является наибольшим по отношению к другим парадигмам, однако в области системного программирования до сих пор применяется парадигма процедурного программирования, и общепринятым языком программирования является язык C, а для написания более компактных и быстродействующих модулей ассемблер. При взаимодействии системного и прикладного уровней для операционных систем заметное влияние стали оказывать языки ООП. Например, одной из наиболее распространенных библиотек мультиплатформенного программирования является объектно-ориентированная библиотека Qt, написанная на языке C++. Поэтому в дальнейшем мы будем рассматривать и создание консольных приложений.

## 3.2. Основные принципы объектно-ориентированного программирования

**Абстрагирование** — это способ выделения набора значимых характеристик или признаков объекта, исключая незначимые

характеристики и признаки. Следовательно, абстракция — это набор всех таких характеристик и признаков, некоторого объекта, которая отличает его от других видов объектов, четко определяя присущие данному объекту характеристики и признаки для анализа и решения задачи.

**Класс** – это структурный тип, используемый для описания некоторого множества сущностей предметной области, имеющих общие свойства, признаки и поведение. Говорят, что объект — это экземпляр класса. Обычно классы разрабатывают таким образом, чтобы их объекты соответствовали объектам предметной области.

**Объект -** сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса или копирования прототипа.

**Инкапсуляция** — это свойство системы, позволяющее объединить данные и методы, работающие с ними в классе, и скрыть детали реализации от пользователя. Для чего описывают данные в разделах public (данные, методы и свойства общедоступны), private (данные, методы и свойства скрыты, доступны только пользователю и дузьям класса), в C++ Builder дополнительно имеется и раздел published (где описываются все компоненты, естественно они визуальны).

**Наследование** — это свойство системы, позволяющее описать новый класс на основе существующих классов, с частично или полностью заимствующейся функциональностью. Класс или классы, от которого производится наследование, называется базовым (и), родительским (и) или суперклассом. Новый класс — потомком, наследником, дочерним или производным классом.

**Прототип** — это объект-образец, по образу и подобию которого создаются другие объекты. Объекты-копии могут сохранять связь с родительским объектом, автоматически наследуя изменения в прототипе; эта особенность определяется в рамках конкретного языка.

**Полиморфизм** — это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. Можно указать, что полиморфизм свойственно синонимам и объектам класса. Например, класс млекопитающиеся, ему характерно свойства: живородящиеся, позвоночные, питаются молоком и т.д.

**Перегрузка** — отличается от полиморфизма, тем, что описывают действия для различных объектов класса, например, сложение, хотя процесс один, надо складывать, однако вполняется по-разному для сложения целых, вещественны, комплексных чисел и матриц.

### 3.3. Инструментарий визуализации ООП - унифицированный язык моделирования (Unified Modeling Language, UML)

**Унифицированный язык моделирования** (UML) является одним из способов при описании (документирования) результатов проектирования и разработки объектно-ориентированных систем. Хотя их можно описать и используя принципы, описанные нами при изучении работы №2, однако описание новых, перечисленных выше парадигмов ООП, испытывают затруднения, связанные с громоздкостью изложения.

Общая структура UML показана на рис. 3.1.

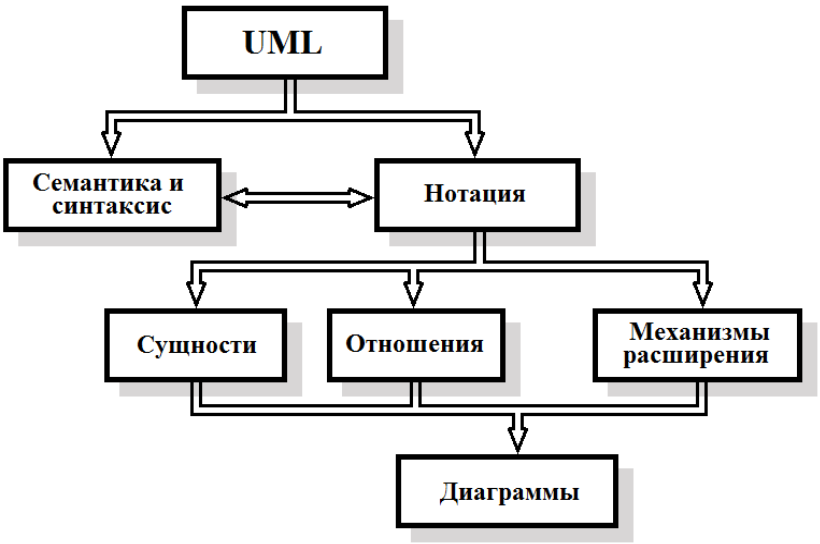



Рис.3.1. Общая структура UML.

**Семантика и синтаксис UML** **Семантика** – раздел языкознания, изучающий значение единиц языка, прежде всего его слов и словосочетаний.

**Синтаксис** – способы соединения слов и их форм в словосочетания и предложения, соединения предложений в сложные предложения, способы создания высказываний как части текста, опираясь на грамматические правила языка.

Таким образом, применительно к UML, семантика и синтаксис определяют стиль изложения (построения моделей), который объединяет естественный и формальный языки для представления базовых понятий (элементов модели) и механизмов их расширения.

**Нотация UML.** Нотация представляет собой графическую интерпретацию семантики для ее визуального представления.

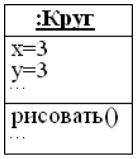
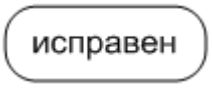
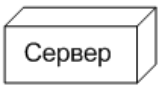
В UML определено три ***типа сущностей***:

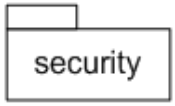
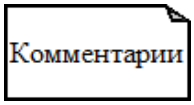
- Структурная сущность – абстракция, являющаяся отражением концептуального или физического объекта;

- Группирующая сущность – это элемент, используемый для некоторого смыслового объединения элементов диаграммы;
- Поясняющая (аннотационная) сущность – комментарий к элементу диаграммы.

**Сущности.** В табл. 3.1 приведено краткое описание сущностей, используемых в графической нотации.

Таблица 3.1.

| Тип | Наименование | Обозначение | Определение (семантика) |
|---|---|---|---|
| **1** | **2** | **3** | **4** |
| Структурная | Класс (class) |  | Множество объектов, имеющих общую структуру и поведение |
| | Объект (object) |  | Абстракция - реальная или воображаемая сущность с четко выраженными концептуально-существенными признаками. С точки зрения UML объекты являются экземплярами класса (экземплярами сущности) |
| | Интерфейс (interface) |  <br>iРасчет | Совокупность операций, определяющая сервисные операция или услуги, предоставляемые классом или компонентой |
| | Действующее лицо (actor) |  <br>Инженер | Внешняя по отношению к системе сущность, которая взаимодействует с системой и использует ее функциональные возможности для достижения определенных целей, решения частных задач или приемник информации |
| | Вариант использования (use case) |  | Способ описание последовательности действий выполняемых системой, что приводит к значимому для актера результату |
| Динамическая | Состояние (state) |  | Описание момента в ходе функционирования сущности, когда она удовлетворяет некоторому условию, при выполнении некоторой подпрограммы или ожидает наступления некоторого события |
| Динамическая | Кооперация (collaboration) |  | Описание совокупности экземпляров актеров, объектов и их взаимодействия в процессе решения некоторой задачи |
| | Компонент (component) |  | Физическая часть системы (файл), в том числе модули системы, обеспечивающие реализацию согласованного набора интерфейсов |
| | Узел (node) |  | Физическая часть системы (компьютер, принтер и т. д.), предоставляющая ресурсы для решения задачи |

| | | | |
|---|---|---|---|
| Группирующая | Пакет (packages) |  | Общий механизм группировки элементов. В отличие от компонента, пакет – чисто концептуальное (абстрактное) понятие. Частными случаями пакета являются система и модель |
| Поясняющая | Примечание (comment) |  | Комментарий к элементу |

В некоторых источниках выделяют также поведенческие сущности *взаимодействия* и *конечные автоматы*, но с логической точки зрения их следует отнести к диаграммам.

**Отношения.** Взаимосвязи в UML — это особые типы между сущностных логических отношений, отображаемых на диаграммах классов и объектов. В UML представлены следующие виды отношений см. рис 3.2.

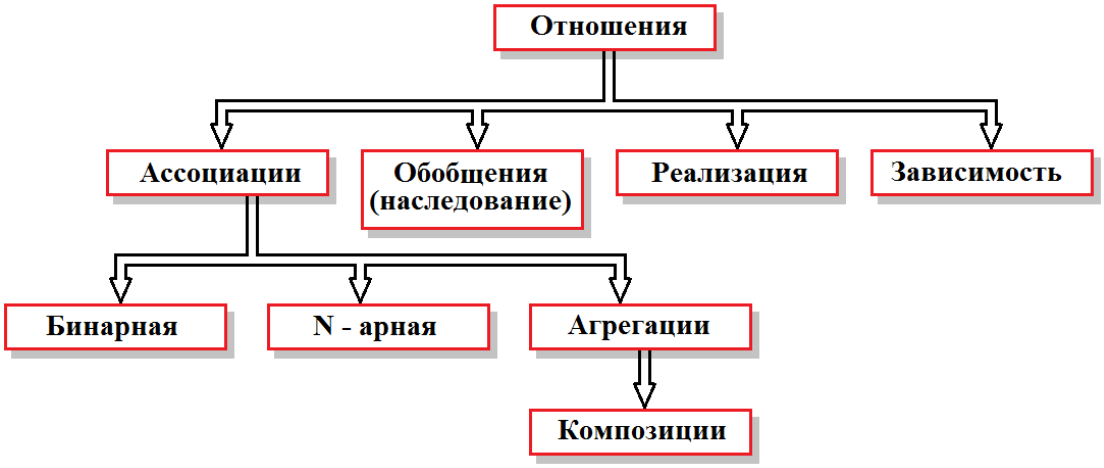
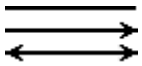
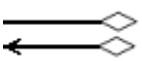
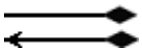


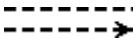

Рис. 3.2. Отношения между сущностями.

В табл. 3.2 приведено описание всех видов ***отношений*** UML, используемых на диаграммах для указания связей между сущностями.

Виды отношений для связи между сущностями

Таблица 3.2.

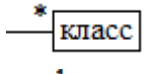
| Наименование | Обозначение | Определение (семантика) |
|---|---|---|
| Ассоциация (association) |  | Отношение, описывающее значимую связь между двумя и более сущностями. Наиболее общий вид отношения |
| Агрегация (aggregation) |  | Подвид ассоциации, описывающей связь «часть»–«целое», в котором «часть» может существовать отдельно от «целого». Ромб указывается со стороны «целого». Отношение указывается только между сущностями одного типа |
| Композиция (composition) |  | Подвид агрегации, в которой «части» не могут существовать отдельно от «целого». Как правило, «части» создаются и уничтожаются одновременно с «целым» |

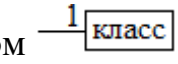
| | | |
|---|---|---|
| Зависимость (dependency) |  | Отношение между двумя сущностями, в котором изменение в одной сущности (независимой) может влиять на состояние или поведение другой сущности (зависимой). Со стороны стрелки указывается независимая сущность |
| Обобщение (наследование generalization) |  | Отношение между обобщенной сущностью (предком) и специализированной сущностью (потомком). Треугольник указывается со стороны предка. Отношение указывается только между сущностями одного типа |
| Реализация (realization) |  | Отношение между сущностями, где одна сущность определяет действие, которое другая сущность обязуется выполнить. Отношения используются в двух случаях: между интерфейсами и классами (или компонентами), между вариантами использования и кооперациями. Со стороны стрелки указывается сущность, определяющее действие (интерфейс или вариант использования) |

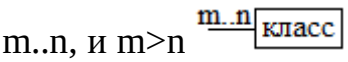
**Ассоциации.** Существует пять различных типов ассоциации:

- **бинарная связь,** например, классы «Маршруты автопарка №1» и «Автопарк №1». «Маршруты автопарка №1» без «Автопарка №1» не бывают;
- **однонаправленная связь**, например, «пушка» и «снаряды» связаны однонаправленной. Так «Маршруты автопарка №1» без «Автопарка №1» не бывают, в отличие от случая «пушка»/«снаряды»;
- **двойные ассоциации** (с двумя концами) представляются линией, соединяющей два классовых блока;
- **высокой степени** имеют более двух концов и представляются линиями, один конец которых идет к классовому блоку, а другой к общему ромбику. В представлении однонаправленной ассоциации добавляется стрелка, указывающая на направление ассоциации;
- **именованная**, и тогда на концах представляющей её линии будут подписаны роли, принадлежности, индикаторы, мультипликаторы, видимости или другие свойства.

Для ассоциации, агрегации и композиции можно указать кратность (multiplicity), определяющие количество экземпляров сущностей, в отношении следующими способами:

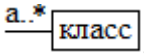
- * - неограниченное количество экземпляров 0..n 

;
- выразить любым целым положительным числом 

;
- диапазон чисел – начальное число и конечное, m..n, и m>n 

;
- перечисление положительных целых чисел и диапазонов по возрастанию через символ запятую 

;

- диапазон чисел от начального числа до произвольного конечного, a..* 

.

По умолчанию кратность принимается равной 1.

**Механизмы расширения.** Приведем описание механизмов расширения, используемых для уточнения сущностей и отношений таб. 3.3

Таблица 3.3.

| **Наименование** | **Обозначение** | **Определение (семантика)** |
|---|---|---|
| Стереотип (stereotype) | ” ” | Уточнение семантики элемента нотации (например: “include” рассматривают, как отношение включения, а класс “boundary” – граничный класс) |
| Сторожевое условие (guard condition) | [ ] | Логическое условие [a&&b] или [цель достигнута]. |
| Ограничение (constraint) | { } | Ограничение семантику элемента (например {только иррациональные числа}) |
| Помеченное значение (tagged value) | { } | Новое или уточняющее свойство элемента нотации (например {version=6.2}) |

**Диаграммы.** Однако помимо стереотипов, указываемых в виде строки текста в кавычках, на диаграммах могут, использованы графические стереотипы смотрите рис. 3.3.

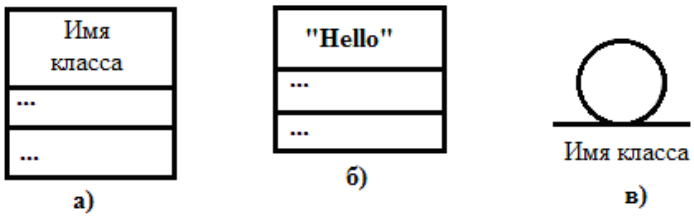



Рис. 3.3. Примеры стандартного и стереотипного отображения класса:
а) стандартное обозначение; б) стандартное обозначение с текстовым стереотипом; в) графический стереотип.

***Диаграмма*** *– связанный граф, в котором сущности являются вершинами, а отношения дугами*. Сущностями могут быть группировки элементов нотации, которые отображают некоторый объект или аспект разрабатываемой информационной системе. Дадим краткую характеристику диаграммам в таблице 3.4.

Диаграммы UML-2.4.1.

Таблица 3.4

| **Диаграмма** | **Назначение** | **Тип диаграммы (модели ИС)** | | |
|---|---|---|---|---|
| | | по степени физической реализации | по отображению динамики | по отображению аспекту |
| **Структурные диаграммы (Structure Diagrams):** | | | | |
| **Классов** (class) | Отображает наборы интерфейсов, классов, атрибутов и отношений между ними | Логическая или физическая | Статистическая или динамическая | Функционально-информационная и/или структурная |
| **Объектов** (object) | Отображает объекты, экземпляры классов системы с указанием текущих значений их атрибутов и связей между объектами | | | |
| **Композитной/составной структуры (**composite structure diagram**)** | Отображает внутреннюю структуру класса и, по возможности, их взаимодействие. Его подвид в UML 2.0 - диаграммы кооперации collaboration, показывающие роли и взаимодействие классов в кооперации. | | | |
| **Развёртывания** (deployment) | Служит для моделирования работающих узлов, артефактов, логических элементов и связей, устанавливающих между ними зависимость манифестации | | Статистическая | Функционально-информационная |
| **Компонентов** (component) | Отображает компоненты системы и связи между компонентами, исполняемыми файлами, модулями, пакетами и т.д. | Физическая | Статистическая | Компонентная (структурная) |
| **Профилей (UML2.2)** (profile (UML2.2)) | Стереотипы используются, как части профилей в метомоделях. Профиль берет часть UML и расширяет с помощью связанной группы стереотипов для | Логическая или физическая | | |

| | | | | | |
|---|---|---|---|---|---|
| | | определенной цели. | | | |
| **Диаграмма пакетов (Package diagram)** | | Отображает структурную диаграмму отношений и взаимосвязи между пакетами. | | Статистическая | Структурная |
| **Behavior Diagrams (Диаграммы поведения):** | | | | | |
| **Вариантов использования (прецедентов)** (use case) | | Отображает взаимодействия между актерами и функциями, а также функции системы | Логическая | Статистическая | Функциональная |
| **Деятельности** (activity) | | Отображает деятельность в виде алгоритма поведения | Логическая | Динамическая | Поведенческая |
| **Состояний** (statechart) | | Отображает состояние сущности в процессе ее жизненного цикла и переходы между ними, если как таковые имеются | | | |
| **Взаимодействия:** (integration) | **Последовательности** (sequence) | Отображает последовательный диалог (сообщения) системы с актером | Логическая | Динамическая | Поведенческая |
| | **Обзора взаимодействия (UML2.0)** Interaction overview diagram (UML2.0) | Отображает деятельности, включающая фрагменты диаграммы последовательности и конструкции потока управления. | | | |
| | **Синхронизации (UML2.0)** Timing diagram (UML2.0) | Отображает регулирование процессов, как параллельных, так и последовательных, протекающих в реальном времени, и допускает их шкалирования. | | | |
| | **Коммуникации /кооперации** (UML1.x)Communication diagram (UML2.0)/Collaboration (UML1.x) | Отображаются взаимодействия между частями композитной структуры или ролями кооперации с явно указанными адресными отношениями между элементами. | | | |

Структуру диаграмм UML 2.x можно представить на диаграмме классов UML см. рис. 3.4. (для исключения неоднозначности приведены также обозначения на английском языке):

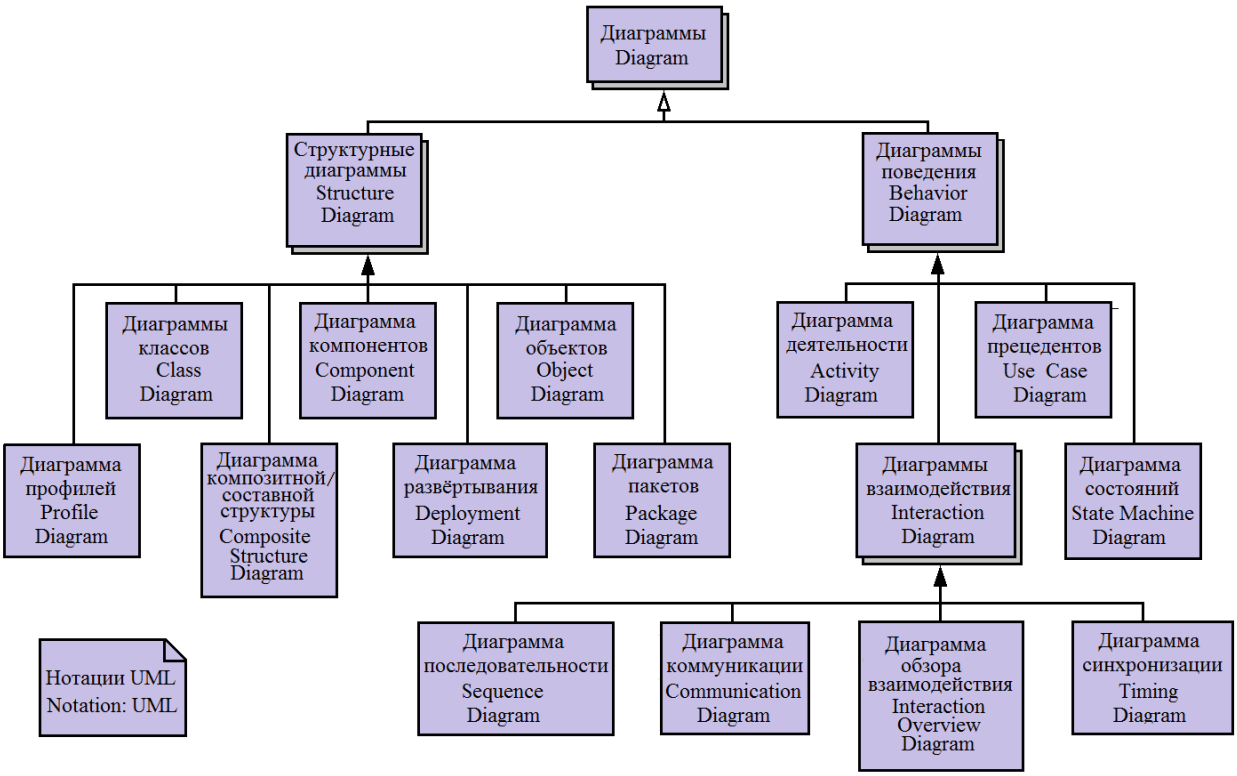



Рис. 3.4. Структура диаграмм.

Связь моделей и диаграмм

Таблица 3.5

| **Модель** | **Диаграмма** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **Вариантов использования** | **Состояний** | **Классов** | **Кооперации** | **Последовательности** | **Деятельности** | **Компонентов** | **Развертывания** |
| **Вариантов использования** | + | + | + | + | + | | | |
| **Анализа** | + | + | + | + | + | | | |
| **Проектирования** | | + | + | + | + | + | | |
| **Реализации** | | | + | | | | + | + |
| **Тестирование на полноту и непротиворечивость** | + | + | + | + | + | + | + | + |

Напоминаем по своему построению язык UML – это язык точных определений и спецификаций, поэтому при моделировании диаграмм рекомендуется избежание всякого рода двусмысленности и не полноты описываемого процесса, и поэтому, при разработке модели системы

строят несколько видов диаграмм, а для сложных систем, охарактеризовать поведение строят дополнительно диаграммы состояния, используя рекомендации таблицы 3.5. При выполнения работы №2, мы ощутили с вами, что не все процессы, выполняемые нашей задачей, мы сумели отобразить.

Приведем некоторые наиболее часто используемые диаграммы UML 2.x:, к остальным мы вернемся позже при изучении соответствующих тем.

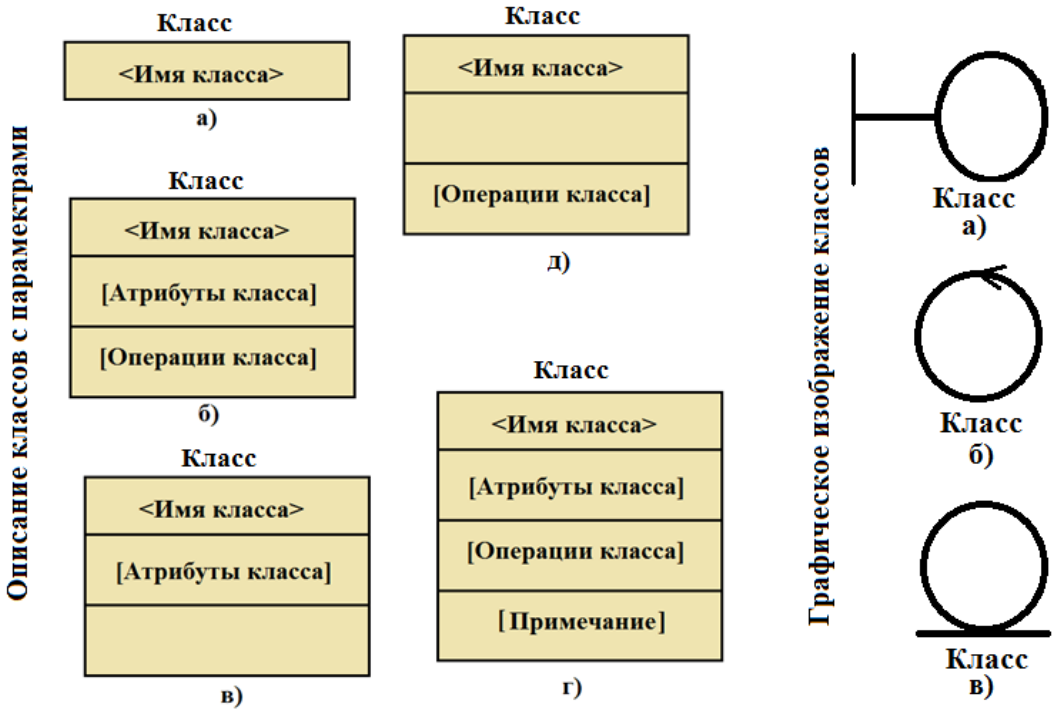



Рис. 3.5. Диаграммы класса.

**Диаграмма классов (Class diagram)** — статическая структурная сущность: описывающая структуру системы, демонстрирующая классы системы, их атрибуты, методы и зависимости между классами.

Существуют разные точки зрения на построение диаграмм классов в зависимости от целей их применения:

- концептуальная точка зрения — диаграмма классов описывает модель предметной области, в ней присутствуют только классы прикладных объектов;
- точка зрения спецификации — диаграмма классов применяется при проектировании информационных систем;
- точка зрения реализации — диаграмма классов содержит классы, используемые непосредственно в программном коде (при использовании объектно-ориентированных языков программирования).

Диаграммы класса некоторых вариантов приведены на рис 3.5.

**Графическое представление одного класса (или нескольких классов со связями между ними) называют диаграммой классов**

**Диаграмма компонентов (Component diagram)** — статическая структурная сущность, показывающая разбиение программной системы на структурные компоненты и связи (зависимости) между компонентами. В качестве физических компонент могут выступать файлы, библиотеки, модули, исполняемые файлы, пакеты и т.п.

Диаграмма компонентов - предназначена для распределения классов и объектов по компонентам при физическом проектировании системы, и графически изображается диаграммой, показанной на рис. 3.6.


Рис. 3.6. Диаграмма компонентов.

Диаграмма компонентов разрабатывается для следующих целей:

- визуализации общей структуры исходного кода программной системы;
- спецификации исполняемого варианта программной системы;
- обеспечения многократного использования отдельных фрагментов программного кода;
- представления концептуальной и физической схем баз данных и/или других разделов обработки информации.

Диаграмма компонентов, в отличие от ранее рассмотренных диаграмм, описывает особенности физического представления системы. Она позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых может выступать исходный и исполняемый код. Основными графическими элементами диаграммы компонентов являются компоненты, интерфейсы и зависимости между ними

**Диаграмма деятельности (Activity diagram)** — диаграмма, на которой показано разложение некоторой деятельности на её составные части. Под деятельностью (англ. activity) понимается спецификация исполняемого поведения в виде координированного последовательного и параллельного выполнения подчинённых элементов — вложенных видов деятельности и отдельных действий (англ. action), соединённых

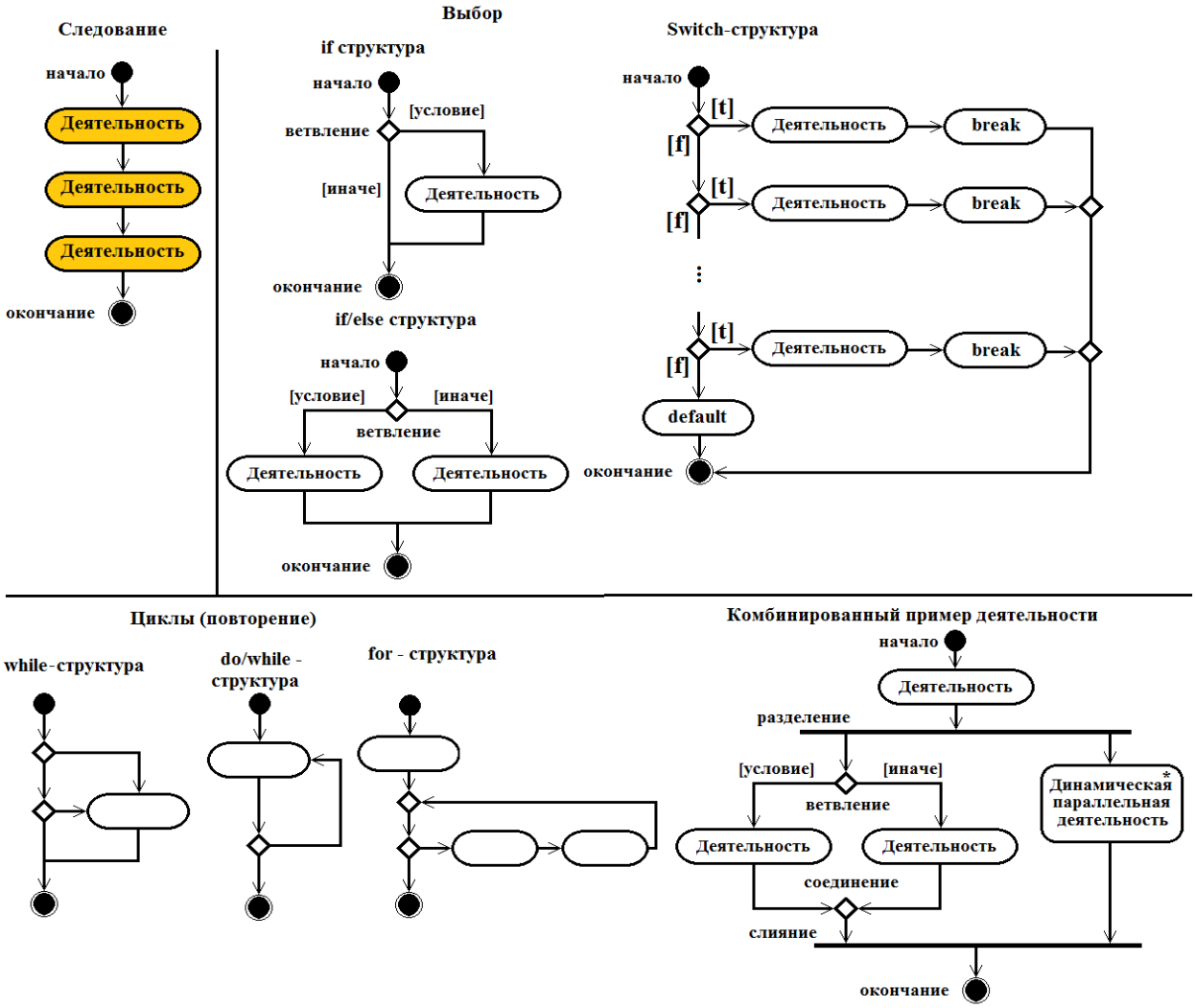



Рис. 3.7. Диаграмма деятельности.

между собой потоками, которые идут от выходов одного узла к входам другого.

Диаграммы деятельности используются при моделировании бизнес-процессов, технологических процессов, последовательных и параллельных вычислений.

Аналогом диаграмм деятельности являются схемы алгоритмов по ГОСТ 19.701-90.

Диаграмма деятельности (активности) – это графическое представление прямых и параллельных процессов деятельности, которая условно можно обозначить в виде приведенной на рис. 3.7.

При формировании структурированных программ можем курировать следующими правилами:

- Начинать с простейшей диаграммы деятельности;
- Каждое состояние действия может быть замещено двумя последовательными состояниями действия;

- Каждое состояние действия может быть замещено одной из управляющей структурой (if, if/else, switch, while, do/while или for);
- Последние два пункта может повторяться неограниченно.

**Диаграмма зависимости (связей)** - отношение между двумя сущностями, в котором изменение в одной сущности (независимой) может влиять на состояние или поведение другой сущности (зависимой). Со стороны стрелки указывается независимая сущность;

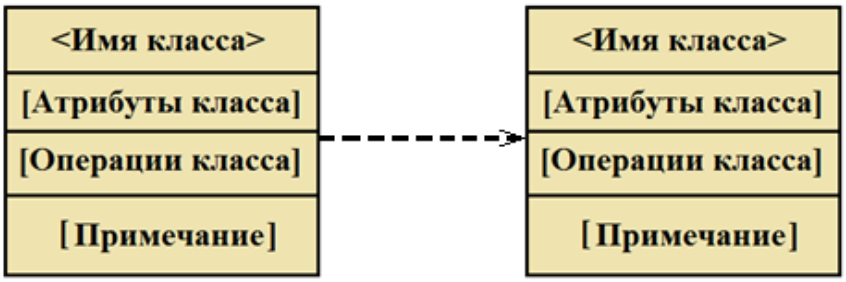



Рис.3.8. Диаграмма зависимости классов.

## 3.4. Подробно язык C++ Builder
## Типы данных

**Переменные перечисляемого типа.** Переменная, которая может принимать значение из некоторого списка значений, называется переменной перечисляемого типа или перечислением.

Объявление перечисления начинается с ключевого слова **enum** и имеет два формата представления.

***Формат 1***. **enum** [имя-тега-перечисления] {список-перечисления} описатель[,описатель...];

***Формат 2***. **enum** имя-тега-перечисления описатель [,описатель.];

Объявление перечисления задает тип переменной перечисления и определяет список именованных констант, называемый списком-перечисления. Значением каждого имени списка является некоторое целое число.

Переменная типа перечисления может принимать значения одной из именованных констант списка. Именованные константы списка имеют тип ***int***. Таким образом, память, соответствующая переменной перечисления, это память необходимая для размещения значения типа ***int***.

Переменная типа ***enum*** могут использоваться в индексных выражениях и как операнды в арифметических операциях и в операциях отношения.

В формате 1 имена и значения перечисления задаются в списке перечислений. Необязательное имя-тега-перечисления, это идентификатор, который именует тег перечисления, определенный списком перечисления. Описатель именует переменную перечисления. В объявлении может быть задана более чем одна переменная типа перечисления.

Список-перечисления содержит одну или несколько конструкций вида:

*идентификатор [= константное выражение]*

Каждый идентификатор именует элемент перечисления. Все идентификаторы в списке *enum* должны быть уникальными. В случае отсутствия константного выражения первому идентификатору соответствует значение 0, следующему идентификатору - значение 1 и т.д. Имя константы перечисления эквивалентно ее значению.

Идентификатор, связанный с константным выражением, принимает значение, задаваемое этим константным выражением. Константное выражение должно иметь тип **int** и может быть как положительным, так и отрицательным. Следующему идентификатору в списке присваивается значение, равное константному выражению плюс 1, если этот идентификатор не имеет своего константного выражения. Использование элементов перечисления должно подчиняться следующим правилам:

1. Переменная может содержать повторяющиеся значения.

2. Идентификаторы в списке перечисления должны быть отличны от всех других идентификаторов в той же области видимости, включая имена обычных переменных и идентификаторы из других списков перечислений.

3. Имена типов перечислений должны быть отличны от других имен типов перечислений, структур и смесей в этой же области видимости.

4. Значение может следовать за последним элементом списка перечисления.

Пример:

```
enum nedelja { SUB = 0,     /*    0   */
               VOS = 0,     /*    0   */
               POND,        /*    1   */
               VTOR,        /*    2   */
               SRED,        /*    3   */
               HETV,        /*    4   */
               PJAT         /*    5   */
               }   rab_ned ;
```

В данном примере объявлен перечисляемый тег **nedelja,** с соответствующим множеством значений, и объявлена переменная `rab_ned` имеющая тип **nedelja.**

В формате 2 используется имя тега перечисления для ссылки на тип перечисления, определяемый где-то в другом месте. Имя тега перечисления должно относится к уже определенному тегу перечисления в пределах текущей области видимости. Так как тег перечисления объявлен где-то в другом месте, список перечисления не представлен в объявлении.

Пример:

```
enum nedelja rab1;
```

В объявлении указателя на тип данных перечисления и объявляемых typedef для типов перечисления можно использовать имя тега перечисления до того, как данный тег перечисления определен. Однако определение перечисления должно предшествовать любому действию используемого указателя на тип объявления typedef. Объявление без последующего списка описателей описывает тег, или, если так можно сказать, шаблон перечисления.

**Массивы.** Массивы - это группа элементов одного и того же типа (**double, float, int** и т.п.), при объявлении, которого компилятор получает информацию о типе элементов и их количестве, и при этом он выделяет память под эти элементы. Объявление массива имеет два формата:

```
спецификатор-типа описатель [константное – выражение];
спецификатор-типа описатель [ ];
```

Описатель - это идентификатор массива.

Спецификатор-типа задает тип элементов объявляемого массива.

***Примечание:*** *Элементами массива не могут быть функции и элементы типа* ***void****.*

Константное-выражение в квадратных скобках задает количество элементов массива. Константное-выражение при объявлении массива может быть опущено в следующих случаях:

- при объявлении массив инициализируется;
- массив объявлен как формальный параметр функции;
- массив объявлен как ссылка на массив, явно определенный в другом файле.

В языке С определены только одномерные массивы, однако элементом массива может быть и массив, следовательно, можно определить и многомерные массивы. Они формализуются списком константных -выражений следующих за идентификатором массива,

причем каждое константное - выражение заключается в свои квадратные скобки.

Каждое константное - выражение в квадратных скобках определяет число элементов по данному измерению массива, так что объявление двухмерного массива содержит два константных - выражения, трехмерного - три и т.д. Отметим, что в языке С первый элемент массива имеет индекс равный 0.

Примеры:

```
int a[2][3]; /* представлено в виде матрицы
                  a[0][0]  a[0][1] a[0][2]
                  a[1][0]  a[1][1] a[1][2]      */
double b[7]; //вектор из 7 элементов имеющих тип double
int w[3][3] = { { 1, 3, 4 },
                { 8, 4, 8 },
                { 1, 2, 9 } };
```

В последнем примере объявлен массив w[3][3]. Списки, выделенные в фигурные скобки, соответствуют строкам массива, в случае отсутствия скобок инициализация будет выполнена неправильно.

В языке С можно использовать сечения массива, как и в других языках высокого уровня, однако на использование сечений накладывается ряд ограничений. Сечения формируются вследствие опускания одной или нескольких пар квадратных скобок. Пары квадратных скобок можно отбрасывать только справа налево и строго последовательно. Сечения массивов используются при организации вычислительного процесса в функциях языка С, разрабатываемых пользователем.

Пример: **`int s[2][3];`**

Если при обращении к некоторой функции написать s[0], то будет передаваться нулевая строка массива s. **`int`** `b[2][3][4];`

При обращении к массиву `b` можно написать, например, `b[1][2]` и будет передаваться вектор из четырех элементов, а обращение `b[1]` даст двухмерный массив размером 3 на 4. Нельзя написать `b[2][4]`, подразумевая, что передаваться будет вектор, потому что это не соответствует ограничению, наложенному на использование сечений массива.

Пример объявления символьного массива.

```
char str[] = "объявление символьного массива";
```

*Следует учитывать, что в символьном литерале находится на один элемент больше, так как последний из элементов является управляющей последовательностью '\0'.*

**Структуры.** Структуры - это составной объект, в который входят элементы любых типов, за исключением функций. В отличие от массива, который является однородным объектом, структура может быть неоднородной. Тип структуры определяется записью вида:

```
struct { список определений }
```

В структуре обязательно должен быть указан хотя бы один компонент. Определение структур имеет следующий вид:

```
тип-данных описатель;
```

где тип-данных указывает тип структуры для объектов, определяемых в описателях. В простейшей форме описатели представляют собой идентификаторы или массивы.

Пример:

```
struct { double x,y; } s1, s2, sm[9];
struct {  int   year;
          char  moth, day; } date1, date2;
```

Переменные s1, s2 определяются как структуры, каждая из которых состоит из двух компонент x и y. Переменная sm определяется как массив из девяти структур. Каждая из двух переменных date1, date2 состоит из трех компонентов year, moth, day. >p> Существует и другой способ ассоциирования имени с типом структуры, он основан на использовании тега структуры. Тег структуры аналогичен тегу перечислимого типа. Тег структуры определяется следующим образом:

```
struct тег { список описаний; };
```

где тег является идентификатором.

В приведенном ниже примере идентификатор student описывается как тег структуры:

```
struct student { char name[25];
                 int  id, age;
                 char  prp;        };
```

Тег структуры используется для последующего объявления структур данного вида в форме:

```
struct тег список-идентификаторов;
```

Пример: `struct studeut st1,st2;`

Использование тегов структуры необходимо для описания рекурсивных структур. Ниже рассматривается использование рекурсивных тегов структуры. **struct** node { **int** data;

```
        struct node * next; }   st1_node;
```

Тег структуры node действительно является рекурсивным, так как он используется в своем собственном описании, т.е. в формализации указателя next. Структуры не могут быть прямо рекурсивными, т.е.

структура node не может содержать компоненту, являющуюся структурой node, но любая структура может иметь компоненту, являющуюся указателем на свой тип, как и сделано в приведенном примере.

Доступ к компонентам структуры осуществляется с помощью указания имени структуры и следующего через точку имени выделенного компонента, например:

```
st1.name="Иванов";
st2.id=st1.id;
st1_node.data=st1.age;
```

**Объединения (смеси).** Объединение подобно структуре, однако в каждый момент времени может использоваться только один из элементов объединения. Тип объединения может задаваться в следующем виде:

```
union {   описание элемента 1;
          ...
          описание элемента n; };
```

Главной особенностью объединения является то, что для каждого из объявленных элементов выделяется одна и та же область памяти, т.е. они перекрываются. Хотя доступ к этой области памяти возможен с использованием любого из элементов, элемент для этой цели должен выбираться так, чтобы полученный результат не был бессмысленным. Доступ к элементам объединения осуществляется тем же способом, что и к структурам. Тег объединения может быть формализован точно так же, как и тег структуры.

Объединение применяется для следующих целей:

- инициализации используемого объекта памяти, если в каждый момент времени только один объект из многих является активным;
- интерпретации основного представления объекта одного типа, как если бы этому объекту был присвоен другой тип.

Память, которая соответствует переменной типа объединения, определяется величиной, необходимой для размещения наиболее длинного элемента объединения. Когда используется элемент меньшей длины, то переменная типа объединения может содержать неиспользуемую память. Все элементы объединения хранятся в одной и той же области памяти, начиная с одного адреса.

Пример:

```
union {   char  fio[30];
          char  adres[80];
          int   vozrast;
```

```
            int   telefon;    } inform;
union {     int ax;
            char al[2];       }   ua;
```

При использовании объекта infor типа **union** можно обрабатывать только тот элемент который получил значение, т.е. после присвоения значения элементу inform.fio, не имеет смысла обращаться к другим элементам. Объединение ua позволяет получить отдельный доступ к младшему ua.al[0] и к старшему ua.al[1] байтам двухбайтного числа ua.ax.

**Поля битов.** Элементом структуры может быть битовое поле, обеспечивающее доступ к отдельным битам памяти. Вне структур битовые поля объявлять нельзя. Нельзя также организовывать массивы битовых полей и нельзя применять к полям операцию определения адреса. В общем случае тип структуры с битовым полем задается в следующем виде:

```
struct { unsigned идентификатор 1 :  длина-поля  1;
         unsigned идентификатор 2 :  длина-поля  2; }
```

длинна - поля задается целым выражением или константой. Эта константа определяет число битов, отведенное соответствующему полю. Поле нулевой длинны обозначает выравнивание на границу следующего слова.

Пример:

```
struct { unsigned a1 :  1;
         unsigned a2 :  2;
         unsigned a3 :  5;
         unsigned a4 :  2;  } prim;
```

Структуры битовых полей могут содержать и знаковые компоненты. Такие компоненты автоматически размещаются на соответствующих границах слов, при этом некоторые биты слов могут оставаться неиспользованными.

Ссылки на поле битов выполняются точно так же, как и компоненты общих структур. Само же битовое поле рассматривается как целое число, максимальное значение которого определяется длиной поля.

**Переменные с изменяемой структурой.** Очень часто некоторые объекты программы относятся к одному и тому же классу, отличаясь лишь некоторыми деталями. Рассмотрим, например, представление геометрических фигур. Общая информация о фигурах может включать такие элементы, как площадь, периметр. Однако соответствующая информация о геометрических размерах может оказаться различной в зависимости от их формы.

Рассмотрим пример, в котором информация о геометрических фигурах представляется на основе комбинированного использования структуры и объединения.

```
struct figure {
    double area,perimetr;  // общие компоненты
    int type;              // признак компонента
union                      // перечисление компонент
    { double radius;       // окружность
    double a[2];    // прямоугольник
    double b[3];    // треугольник
    } geom_fig;
              }  fig1, fig2  ;
```

В общем случае каждый объект типа `figure` будет состоять из трех компонентов: `area, perimetr, type`. Компонент `type` называется меткой активного компонента, так как он используется для указания, какой из компонентов объединения `geom_fig` является активным в данный момент. Такая структура называется переменной структурой, потому что ее компоненты меняются в зависимости от значения метки активного компонента (значение `type`).

Отметим, что вместо компоненты `type` типа **`int`**, целесообразно было бы использовать перечисляемый тип. Например, такой

```
enum figure_chess { CIRCLE,
                    BOX,
                    TRIANGLE    } ;
```

Константы `CIRCLE, BOX, TRIANGLE` получат значения соответственно равные 0, 1, 2. Переменная `type` может быть объявлена как имеющая перечислимый тип:

```
enum figure_chess type;
```

В этом случае компилятор С предупредит программиста о потенциально ошибочных присвоениях, таких, например, как

```
figure.type = 40;
```

В общем случае переменная структуры будет состоять из трех частей: набор общих компонент, метки активного компонента и части с меняющимися компонентами. Общая форма переменной структуры, имеет следующий вид:

```
struct { общие компоненты;
         метка активного компонента;
           union { описание компоненты 1 ;
                   описание компоненты 2 ;
                   … … …
                    описание компоненты n ;
```

```
                    } идентификатор-объединения ;
          } идентификатор-структуры ;
```

Пример определения переменной структуры с именем helth_record

```
struct { // общая информация
          char  name [25];   // имя
          int   age;         // возраст
          char  sex;         // пол
          /* метка активного компонента
              (семейное положение)            */
          enum marital_status ins;
          // переменная часть
          union  {   // холост
                     // нет компонент
                   struct {  // состоит в браке
                            char marripge_date[8];
                            char spouse_name[25];
                            int  no_children;
                           }  marriage_info;
                    /* разведен */
                    char date_divorced[8];
                  }   marital_info;
           } health_record;
  enum marital_status { SINGLE, // холост
                        MARRIGO, // женат
                        DIVOREED // разведен
                        } ;
```

Обращаться к компонентам структуры можно при помощи ссылок:

```
helth_record.neme,
helth_record.ins,
helth_record.marriage_info.marriage_date.
```

## 3.5. Операторы и операция разветвляющихся алгоритмов языка С и С++Builder

**Условная операция (тернарная альтернатива).** В алгоритмическом языке С имеется одна ***тернарная операция*** – которая является условной операцией и имеет следующий формат:

```
Операнд_1 ? операнд_2 : операнд_3
```

Операнды должны быть целого или плавающего типа или быть указателем см. тема 2, таблица 2.2. Он оценивается с точки зрения его эквивалентности 0. Если `операнд_1` не равен `0`, то вычисляется `операнд_2` и его значение является результатом операции. Если

операнд_1 равен 0, то вычисляется операнд_3 и его значение является результатом операции.

**Примечание:** Следует отметить, что вычисляется либо операнд_2, либо операнд_3, но не оба, т.е. альтернативный выбор. Тип результата зависит от типов операнда_2 и операнда_3, следующим образом:

1. Если операнд_2 или операнд_3 имеет целый или плавающий тип (отметим, что их типы могут отличаться), то выполняются обычные арифметические преобразования. Типом результата является тип операнда после преобразования.
2. Если операнд_2 и операнд_3 имеют один и тот же тип структуры, объединения или указателя, то тип результата будет тем же самым типом структуры, объединения или указателя.
3. Если оба операнда имеют тип **void**, то результат имеет тип void.
4. Если один операнд является указателем на объект любого типа, а другой операнд является указателем на void, то указатель на объект преобразуется к указателю на **void**, который и будет типом результата.
5. Если один из операндов является указателем, а другой константным выражением со значением 0, то типом результата будет тип указателя.

Пример:
```
int a=6,b=7;
 max = (a<=b) ? b : a;
```

Переменной max присваивается максимальное значение переменных a и b. max=7.

**Операторы условий if и if/else.** Для программирования разветвляющихся алгоритмов в языках C и C++ Builder используются переменные типа **bool** см. тема 2, таблица 2.2., которые могут принимать только два значения – true - истина и false - ложь (истина (да), ложь (нет), или 0 и 1). Также могут быть использованы в условном операторе результаты логических операций: не - !=, и - &&, или - || и исключающего или - ^, которых приводятся в таблице 3.6.

Таблица 3.6.

| a | b | !a | a&&b | a\|\|b | Поразрядные | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | ~a | a&b | a\|b | a^b |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |

Условный оператор имеет 2 формата описания, которые приведены на рис. 3.9. и 3.10.

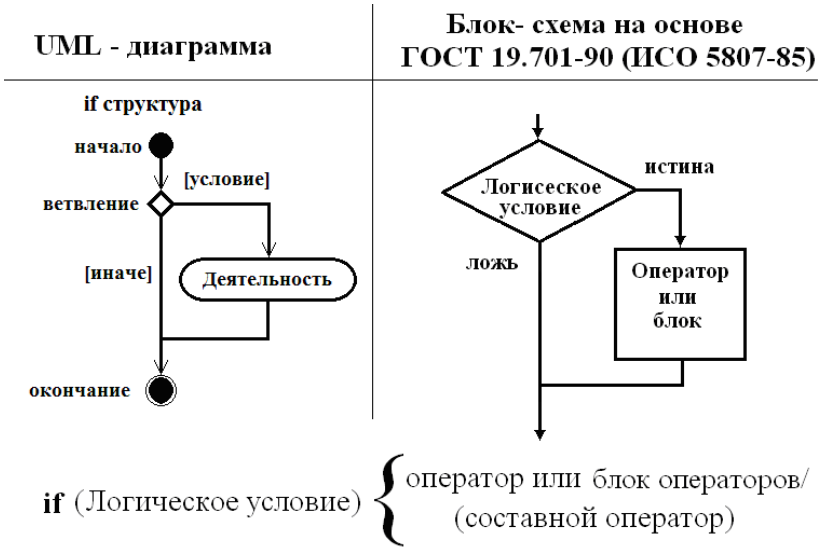



Рис. 3.9. Формат if структуры, и диаграмма UML и блок-схема.

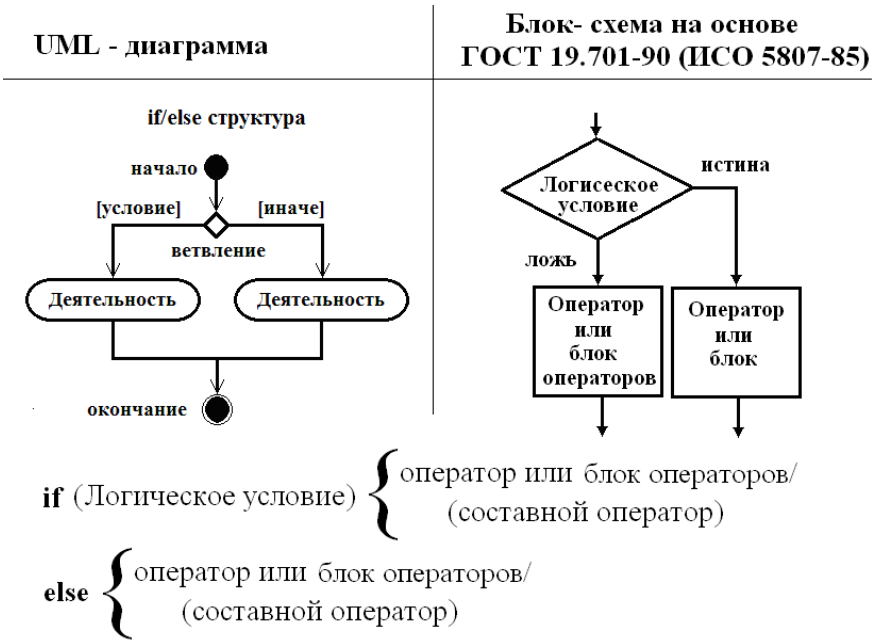



Рис. 3.10. Формат if/else структуры, и диаграмма UML и блок-схема.

Оператор if проверяет результат логического выражения или значение переменной типа **`int`** либо `bool` и организует разветвление вычислений.

Например, если **`bool`** `varbl;` **`double`** `x,y,z;` то фрагмент программы с оператором **`if`** может быть таким:

```
varbl=x>y;
if  (varbl)  z=x-y;
else
 z=x+y;
```

**Оператор выбора условия switch.** Оператор выбора **switch** организует разветвления в зависимости от значения некоторой переменной перечисляемого типа см. рис. 3.11.

Например, если **double** x,y,z; int varin, то после выполнения

```
 switch (varin )
  {
 case 0: z=x+y; break;
 case 1: z=x-y;  break;
 case 2: z=x*y;  break;
 case 3: z=x/y;  break;
 default z=0;
}
```

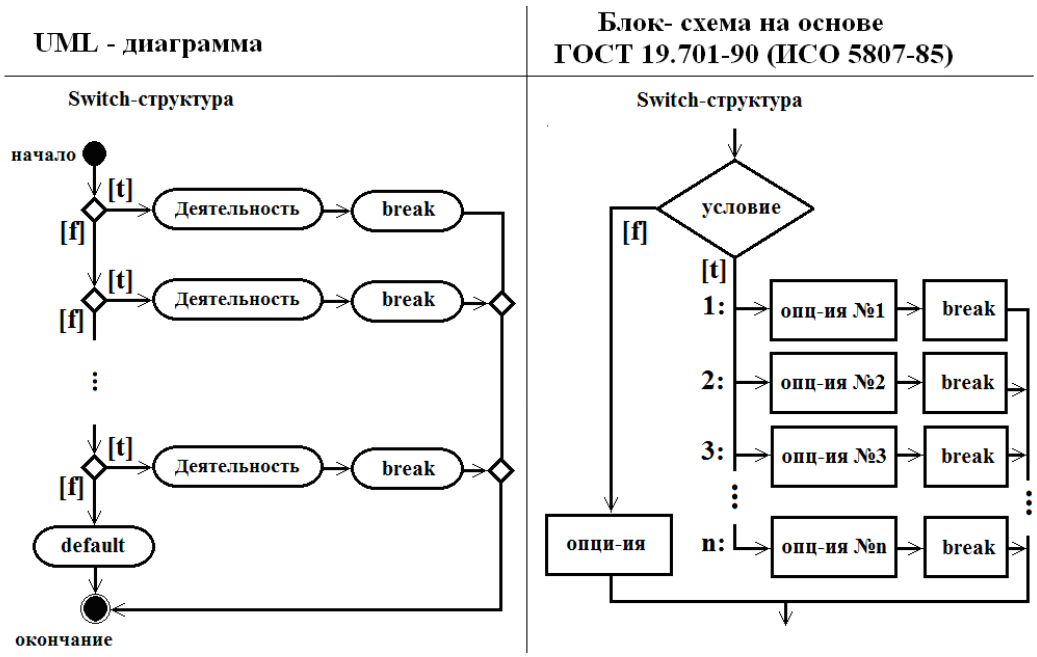



Рис. 3.11. Структура switch – оператор выбора условий.

В соответствии со значением varin вычисляется z. Если varin=0, то z=x+y, если varin=1, то z=x-y, если varin=2, то z=x*y и z=0 при любых значениях varin отличных от 0, 1, 2 или 3. Ветвь **default** можно при необходимости опустить.

Чаще всего с оператором выбора условия в согласии работает перечисляемые типы данных. Перечисляемые типы определяют упорядоченное множество идентификаторов, представляющих собой возможные значения переменных этого типа. Вводятся эти типы для того, чтобы сделать код более понятным.

Определяются перечисляемые переменные следующим образом:

**enum** {константа 1, ... , константа n} <имена переменных>;

Например : **enum** {mRed, mYellow, mGreen} mcolor;

Перечисляемые переменные можно проверять и сравнивать с

возможными значениями.

Например:

```
      if (mcolor > mRed)  …
       switch (mcolor) {
      case mRed  …
                    break;
    case mYellow  …
                    break;
    …
                              }
```

По умолчанию перечисляемые значения интерпретируются как целые числа, начиная с -1.

*Оператор* ***break*** *является не обязательным параметром в операторе, при отсутствии этого оператора выполняется далее последовательно.*

Например:

```
if (mcolor > mRed)  …
 switch (mcolor) {
case mRed  …
case mYellow  …
              break;
```

после выполнения **mRed** выполнится **mYellow,** а затем будет прерывание **break.**

***Вниманме!*** *Следует отметить внутри* ***switch*** *ключевое слово* ***case*** *является меткой и допускается до 16384 раза применений.*

**Обработка исключительных ситуаций .** В работе 2 при исследовании функций вы, наверное, заметили, что не любые численные значения можно вводить, а существует ограничения, которые необходимо накладывать на вводимые данные. Во время работы приложения могут возникать различного рода ошибки: переполнение, деление на нуль, попытка открыть несуществующий файл и т.д.

При возникновении таких исключительных ситуаций программа генерирует так называемое исключение и выполнение дальнейших вычислений в данном блоке прекращается. Исключение – это объект специального вида, выявляющий и характеризующий возникшую исключительную ситуацию. При проектировании программного кода разработчикам следует в проекте рассматривать возникновение всякого рода исключений. Особенностью исключений является то, что это сугубо временные объекты. Как только они обработаны, каким то, обработчиком, они разрушаются.

Наиболее кардинальный путь борьбы с исключениями – отлавливание и обработка их с помощью блоков **try**…**catch**. Синтаксис этих блоков следующий:

```
try
    {
 …//операторы, которые могут вызвать исключение
 }
 catch(TypeToCatch)
 {
 …//операторы, выполняемые в случае ошибки
 }
```

Операторы блока catch представляют собой обработчик исключения. Параметр TypeToCatch может быть или ссылкой на класс исключения, или одним из целых типов (int, char и т. п.), или многоточием, что означает обработку любых исключений.

Операторы обработчика выполняются только в случае генерации в операторах блока try исключения типа, указанного в заголовке catch. После блока try может следовать несколько блоков catch для разных типов исключений. Таким образом, в обработчиках catch можно предпринять какие-то действия: известить пользователя о возникшей проблеме и подсказать ему пути ее решения, принять какие-то меры к исправлению ошибки и т. д. Важно то, что можно определить тип сгенерированного исключения и дифференцированно реагировать на различные исключительные ситуации. В связи с этим следует отметить некоторую опасность применения многоточия в качестве параметра блока catch. Перехват всех исключений catch(…) способен замаскировать какие-то непредвиденные ошибки в программе, что затруднит их поиск и снизит надежность работы.

В табл. 3.7 перечислены наиболее часто используемые классы исключения и указаны причины, которые могут привести к возникновению этих исключений.

Таблица 3.7.

| **Исключение** | **Возникает** | |
|---|---|---|
| EConvertError – ошибка преобразования | При выполнении преобразования, если преобразуемая величина не может быть приведена к требуемому типу. Наиболее часто возникает при преобразовании строки символов в число. | |
| EZeroDivide – деление на ноль | При выполнении операции деления над дробными операндами, если делитель равен нулю. | |
| EDivByZero – целочисленное деление на ноль | При выполнении операции целочисленного деления, если делитель равен нулю. | |
| EVariantError | Ошибка, связанная с типом данных вариант | |

| | | |
|---|---|---|
| EOverflow | Переполнение при целочисленных операциях. | |

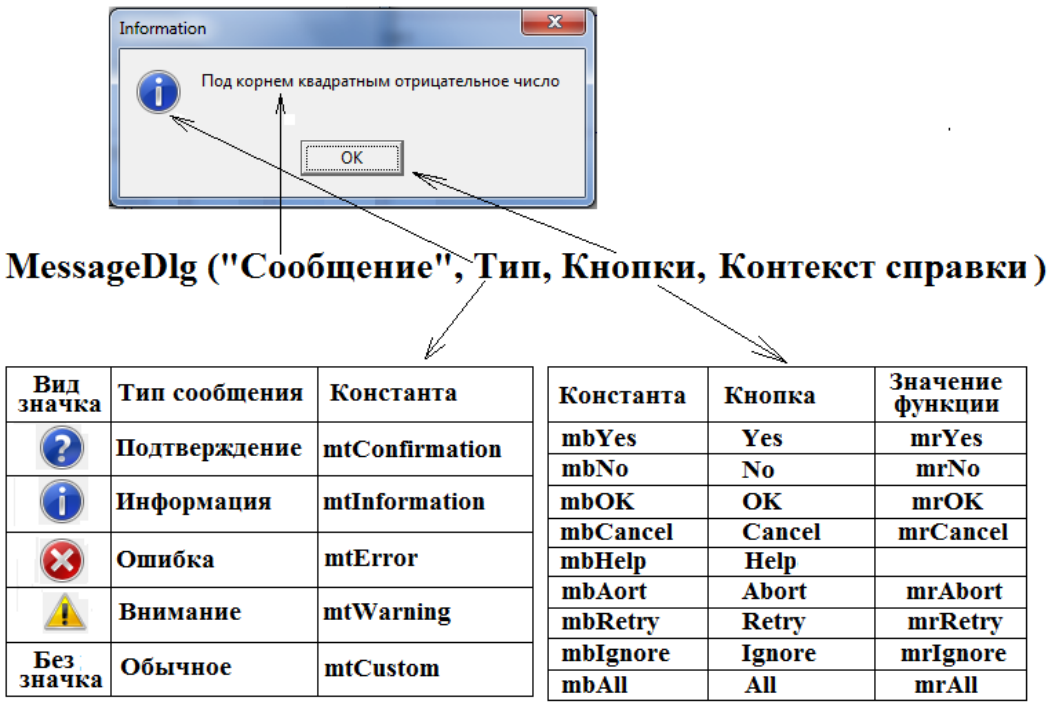
Ниже приведен пример обработки исключения для случая, когда пользователь может ввести символы вместо числа.

```
 float a;
try
{
a=StrToFloat(Edit1->Text);
}
catch(EConvertError&)
{
Application->MessageBox("Вы ввели ошибочное
число","Повторите ввод",MB_OK);
return;
}
```

Для вывода сообщения можно использовать функцию `MessageDlg`, формат которого выглядит так:

`MessageDlg("Сообщение",Тип, Кнопки, Контекст справки)`, где: "Сообщение", текст AnsiString; Тип – тип сообщения см. на рис. 3.12; Кнопка – вид кнопки, со значком отображаемый в окне сообщения см. на рис.3.12.; Контекст справки – параметр который определяет раздел справочной информации, если нажмет F1, если раздела Help нет, то его значение равно 0.

Как и любая функция `MessageDlg` возвращает значения которые приведены на рис. 3.12.




| Вид значка | Тип сообщения | Константа |
|---|---|---|
| | Подтверждение | mtConfirmation |
| | Информация | mtInformation |
| | Ошибка | mtError |
| | Внимание | mtWarning |
| Без значка | Обычное | mtCustom |

| Константа | Кнопка | Значение функции |
|---|---|---|
| mbYes | Yes | mrYes |
| mbNo | No | mrNo |
| mbOK | OK | mrOK |
| mbCancel | Cancel | mrCancel |
| mbHelp | Help | |
| mbAort | Abort | mrAbort |
| mbRetry | Retry | mrRetry |
| mbIgnore | Ignore | mrIgnore |
| mbAll | All | mrAll |

Рис. 3.12. Вид окна задания темы 2 после выполнения инструкции – `MessageDlg("Под корнем квадратным отрицательное число", mtInformation, TMsgDlgButtons()<<mbOK,0)`.

### 3.6. Программирование в C++ Builder
### Усовершенствование программы из работы №2.

**Задание 3.1.** Используя, данные задания 2.1, составьте программу, учитывая исключения.

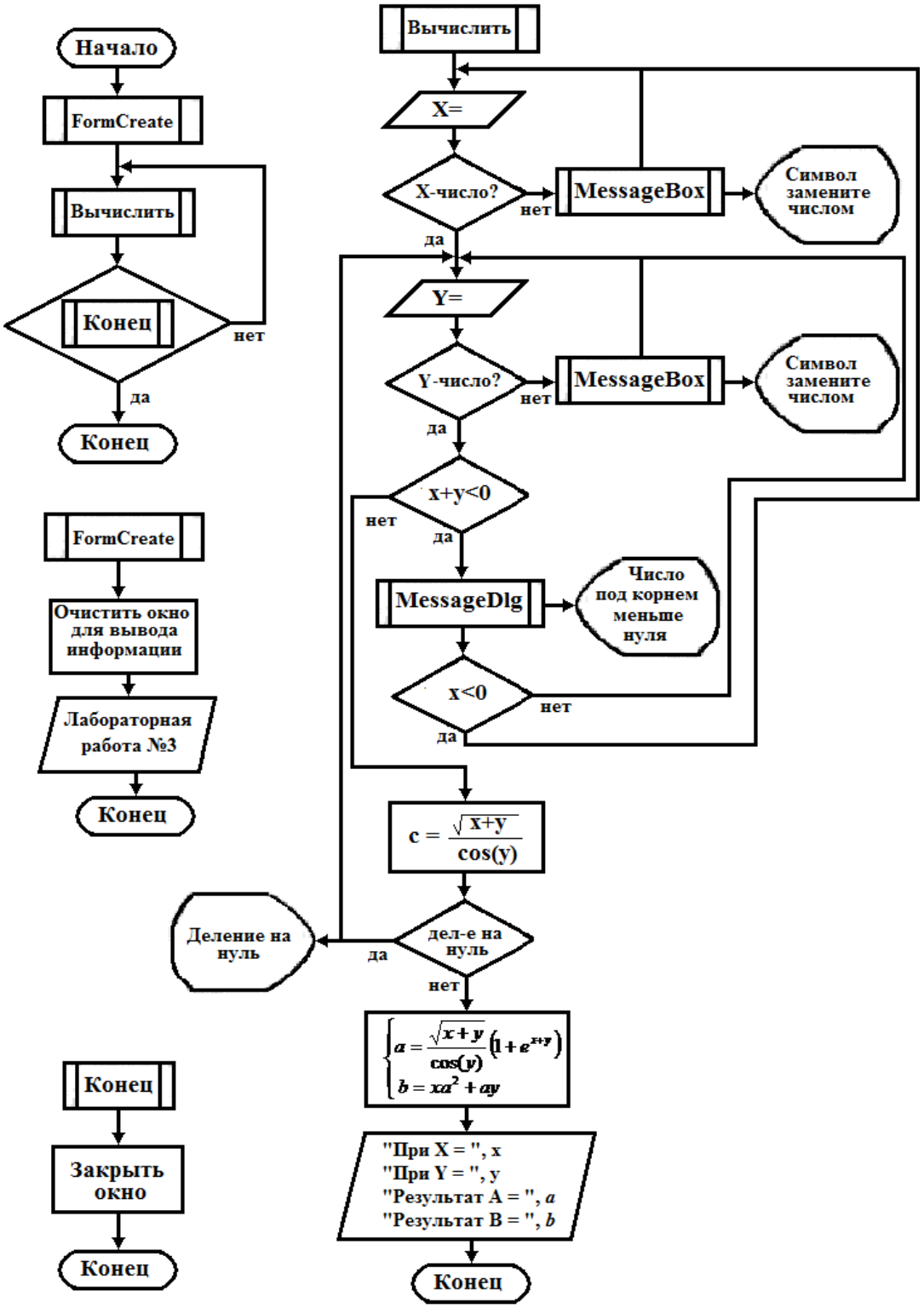



Рис. 3.13. Блок-схема для задания из темы 2 после дополнения возможных исключений в функционировании жизненного цикла проекта.

Вернемся к условию $\begin{cases} a = \dfrac{\sqrt{x+y}}{\cos(y)}\left(1+e^{x+y}\right) \\ b = xa^2 + ay. \end{cases}$ задания из работы №2. Как видим из условия при значениях x и y, удовлетворяющих условию $x + y < 0$ появляются комплексные числа, а они нужны в некоторых научных приложениях и выкладках, обычно пользуются действительными числами. И еще если результирующее значение при вычислении функции *cos(y)=0*, то деление на 0, также может случайно вместо числа включить произвольный символ. Попробуем устранить указанные недочеты, перепишем программу и составим соответствующие диаграммы и блок-схемы см. рис.3.13.

Дополним программу новыми строками для обработки исключений.

```
#include <vcl.h>
#pragma hdrstop
#include <math.h>
#include "Linpr.h"
//---------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
     Memo1->Clear();
    Memo1->Lines->Add("       Лабораторная работа№ 3");
}
//---------------------------------------------------------------

void __fastcall TForm1::Button1Click(TObject *Sender)
{
  double x, y, a, b,c;
 try
 {
   x = StrToFloat(Edit1->Text);
 }
 catch(EConvertError&)
 {
  Application->MessageBox("Символ замените числом", "Повторите ввод
",MB_OK);
  return;
 }
 try
 {
   y = StrToFloat(Edit2->Text);
 }
 catch(EConvertError&)
```

```
      {
       Application->MessageBox("Символ замените числом", "Повторите ввод
",MB_OK);
       return;
      }
       if (x+y<0)
       {
          MessageDlg("Число под корнем меньше нуля",
          mtInformation, TMsgDlgButtons()<<mbOK,0);
          if ((Edit1->Text).Length()<0) Edit1->SetFocus(); //Курсор в X=
          else Edit2->SetFocus();                          //Курсор в Y=
         return;
       }
       try
       {
          c=sqrt(x+y)/y;
       }
       catch(EZeroDivide &e)
        {
           ShowMessage("Деление на 0");
           Edit2->SetFocus();
        }
           a = c*(1+exp(x+y));
           b = x*pow(a,2)+y*a;
     Memo1->Lines->Add(" При X = "+Edit1->Text);
     Memo1->Lines->Add(" При Y = "+Edit2->Text);
     Memo1->Lines->Add(" Результат A = "+FloatToStr(a));
     Memo1->Lines->Add(" Результат B = "+FloatToStr(b));
     }
     //---------------------------------------------------------------
     void __fastcall TForm1::Button2Click(TObject *Sender)
     {
      Form1->Close();
     }
     //---------------------------------------------------------------
```

Приведенная блок-схема на рис. 3.1. является укрупненным, т.к. далеко не отражает все нюансы, выполняющие программной средой алгоритмического языка ООП C++Builder. Множество положений еще нами не изучены, а некоторые, чтобы отобразить составить трудность и начинающему специалисту, опираясь на предлагаемые элементы ГОСТа, (придется многие положения ООП и соответственно отображающие им конструкция разработать самому) обычно профессионалы пользуются диаграммами UML.

***Преимущества UML.*** UML объектно-ориентирован, в результате чего методы описания результатов анализа и проектирования семантически близки к методам программирования на современных объектно-ориентированных языках;

UML позволяет описать систему практически со всех возможных точек зрения и разные аспекты поведения системы;

Диаграммы UML сравнительно просты для чтения после достаточно быстрого ознакомления с его синтаксисом;

UML позволяет вводить авторские текстовые и графические стереотипы, что способствует универсализации его применение;

UML получил широкое распространение и динамично развивается.

Пытливый читатель спросить, тогда зачем их мы изучаем?

В первых это стандарты нашей страны других нет пока.

Во вторых мы собираемся изучать, как создавать консольные приложения, они далеко не всегда опираются на ООП, следовательно, знать их специалисту так же желательно и нужно.

В третьих UML имеет и недостатки, которых не мало, одна из которых избыточность или разночтение и т.д., что не скажешь относительно ГОСТом предлагаемого варианта изложения.

Для полного изложения нашей задачи в диаграммах UML, пока не достаточно материала, изученного нами, однако начальный этап мы уже можем показать, что и сделаем.

**Постановка задачи и анализ (словесное описание задания) на UML**

На рис. 2.1. работы 2 задан эскизный проект интерфейса калькулятора и некоторые формулы преобразования данных. Используя эти формулы и данные, которые получают после ввода в соответствующие окна под маркерами X и Y, и при нажатии кнопки с названием «Вычислить», необходимо провести процедуру вычисления и записать полученные результаты в соответствующее окно, а при нажатии кнопки «Выход» процесс завершить.

***Произведем декомпозицию (детализацию или редукцию) процесса:***

- Ввод данных осуществляет сущность см. табл. 1. «Действующее лицо» – Actor;
- Визуальная часть программы – это одноконный интерфейс с: двумя окнами для ввода данных; два маркера (идентификатора или метки) для окон ввода данных; две кнопки с названиями «Вычислить» и «Выход»; окно для вывода расчетных данных;
- Анализ визуализации показывает, все перечисленные элементы можно заменить соответствующими компонентами C++ Builder и окно рассмотреть, как один класс - абстрактный объект выполняющий процесс калькуляции нашего задания;
- Для приведения в действие кнопок необходимо использование сущности см. табл. 3.1. «Вариант использования» - Use case.

Класс пока оставим в покое, слишком мало мы пока знаем о нем. Мы продолжим изложение после изучения некоторых деталей.

## 3.7. Вычисления значения функции заданной графически оператором if

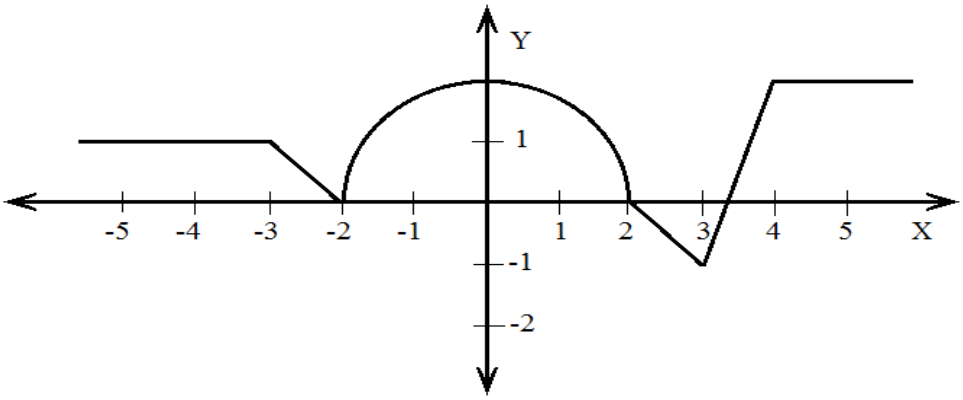



Рис. 3.13. Функция для задачи 3.2.

**Задание 3.2.** Создать проект на C++Builder, и написать программный код, который по введенному значению аргумента вычисляет значение функции, заданный в виде графика на рисунке 3.13.

Прежде всего, мы сделаем анализ графической функции, и представим ее в виде аналитической функции.

Область определения функции – совокупность вех действительных чисел $[-\infty, \infty]$, область изменения [-1, 2], следовательно, можем определить, аргумент - x и абсцисса - y должны быть вещественными. Аналитический вид функции приведен ниже

$$Y = \begin{cases} 1 & x < -3 \\ -x - 2 & -3 \le x < -2 \\ \left|\sqrt{4 - x^2}\right| & -2 \le x \le 2 \\ -x + 2 & 2 < x < 3 \\ 3x - 4 & 3 \le x \le 4 \\ 2 & x > 4 \end{cases}.$$

**Эскизный проект задания 3.2.** Анализ задания показывает, для визуализации проекта, необходимо отображение самого графика и компоненты для ввода и вывода данных, а так же кнопки управления проектом.

Возможный эскизный проект изображен на форме см. рис 3.14.

Компоненты см. приложение 2. Edit и Button нами рассмотрены в предыдущих проектах, однако компонента Label имеет возможности отображать не только информацию с Caption, но и можем дополнить

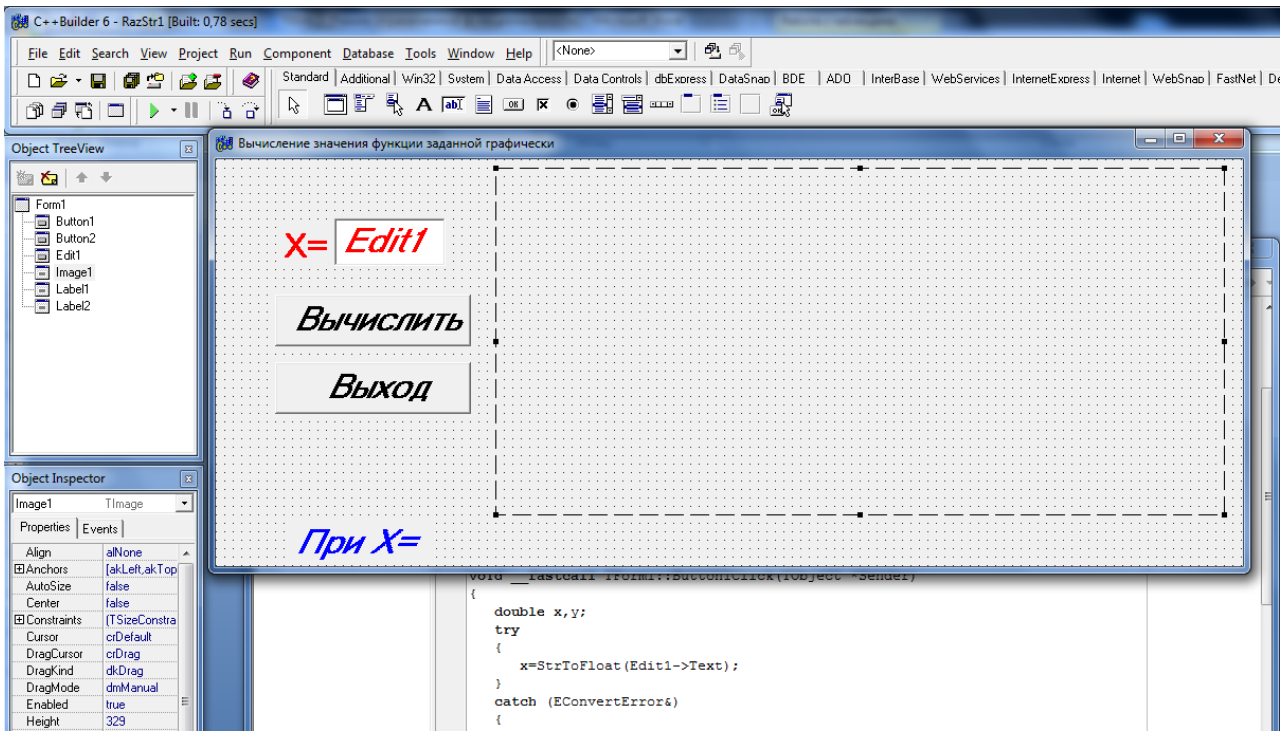



Рис. 3.14. Эскизный проект задания 3.2.

информацию, что, и делаем в нашем программном коде в строке

```
Label2->Caption ="При X = "+FloatToStr(x)+" Результат Y = " +
FloatToStrF(y,ffGeneral,5,2);
```

Компонента Image (холст) (см. приложение 2) является контейнером графического изображения, в нашем случае, мы отображаем в нем рисунок из файла, что делает командная строка

```
Image1->Picture->LoadFromFile("GrFun.bmp")
```

Боле подробно об элементах графики поговорим в соответствующих темах позже.

Далее проведем детализацию алгоритма. Мы создадим алгоритм задания в словесной неформальной форме

**Алгоритм решения графического задания и программный код.**

Мы делаем упор сейчас на создание алгоритма задания в словесной неформальной форме, из следующих соображений, т.к. любая задача, когда четко описана, и понято, только тогда он может ее записать в виде программного кода.

Алгоритм решения графического задания для нашего простейшего случая заключается в следующем:

1. Расстановка и укладка в соответствии с размерами и положением всех компонентов на форму, таких как:

- две метки Label1 и Label2, первая для указания имени поля Edit1, откуда читается, и вводят переменную x, а вторая, куда выводим результаты наших исследований;
- холст Image1 для отображения графика функции;
- две кнопки Button1 и Button2, первая с названием «Вычислить», а вторая «Выход»;
- и, наконец, компонента Edit1 для ввода информации.

2. Создать обработчик событий FormCreate, и указать в нем очистить текст в компоненте Edit1, а также отображение на холсте Image1 информацию с файла `GrFun.bmp` находящегося в текущей папке.

***Внимание!*** *Напоминаем, обработчик событий создается двойным щелчком мыши на объекте, и программа в среде Builder представляет собой набор функций, выполняющих обработку событий, связанных с формой, например, щелчок кнопкой мыши – событие* ***OnClick****, создание формы – событие* ***OnCreate****.*

3. Написать функцию завершающую работу при нажатии кнопки «Выход» (Создать обработчик кнопки и в той функции написать программный код).

4. Написать функцию для вычисления вводимого значения аргумента x (создать сначала обработчик событий кнопки «Вычислить» и открывшейся окне писать), учитывая следующие положения:

- проверка условия на не цифру, если не цифра потребовать повторный ввод, иначе продолжить вычисление;
- определить какому интервалу принадлежит функция;
- вычислить значение абсциссы y по соответствующей формуле;
- вывести значение y.

После детализации и изложения алгоритма приводим код программы.

В проекте есть еще 1 файл – ***главной функции*** его тоже пишет среда ***C++ Builder***, обычно его не видят и малоопытные пользователи его не трогают, который управляет работой всего проекта на основе событий и других конструкций языка, приведем и его

```
/* Это файл главной функции */
//---------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
//---------------------------------------------------------
USEFORM("RazStr.cpp", Form1);
//---------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
```

```
        try
        {
            Application->Initialize();
            Application->CreateForm(__classid(TForm1), &Form1);
              Application->Run();
        }
        catch (Exception &exception)
        {
               Application->ShowException(&exception);
        }
        catch (...)
        {
              try
              {
                      throw Exception("");
              }
              catch (Exception &exception)
              {
                     Application->ShowException(&exception);
              }
        }
        return 0;
}
//---------------------------------------------------------------------
```

На сей раз и достаточно того, что мы уже знаем, что такой файл существует.

```
//---------------------------------------------------------
```

***Внимание! Напоминаем заголовочный файл, писала сама среда C++ Builder.***

```
//Заголовочный файл
#ifndef RazStrH
#define RazStrH
//---------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ExtCtrls.hpp>
//---------------------------------------------------------------------
class TForm1 : public TForm
{
__published:     // IDE-managed Components
        TImage *Image1;
        TEdit *Edit1;
        TLabel *Label1;
        TButton *Button1;
        TButton *Button2;
        TLabel *Label2;
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall FormCreate(TObject *Sender);
private:    // User declarations
public:           // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------
```

```
#endif
```

*Обратите внимание после двойного щелчка на любом объекте, если еще не создан обработчик событий, то создается обработчик событий, и в заголовочном файле появляется и соответствующий прототип функции,* например **void __fastcall** `FormCreate(TObject *Sender);`, а при установке на форму какого либо компоненты, то соответствующая компонента с указателем на его класс. Например для `Image1` соответствующая строка `TImage *Image1.`

`TImage` - класс которому принадлежит компонента `Image` и `*Image1` ссылается на него является указателем.

```
      //Программный код

  #include <vcl.h>
  #pragma hdrstop
#include <math.h>
#include "RazStr.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
   Edit1->Clear();
   Image1->Picture->LoadFromFile("GrFun.bmp");
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
   Form1->Close();
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   double x,y;
   try
   {
      x=StrToFloat(Edit1->Text);
   }
   catch (EConvertError&)
   {
      Application->MessageBox("Символ замените числом»", "Повторите
      ввод ",MB_OK);
      return;
   }
   if (x<-3) y=1;
   if ((x>=-3)&&(x<-2)) y=-x-2;
   if ((x>=-2)&&(x<=2)) y=fabs(sqrt(4-pow(x,2)));
```

```
        if ((x>2)&&(x<3)) y=-x+2;
        if ((x>=3)&&(x<=4)) y=3*x-10;
        if (x>4) y=2;
      Label2->Caption ="При X = "+FloatToStr(x)+" Результат Y = " +
FloatToStrF(y,ffGeneral,5,2);
     }
```

Приведем результаты работы нашего проекта см. рис. 3.15.

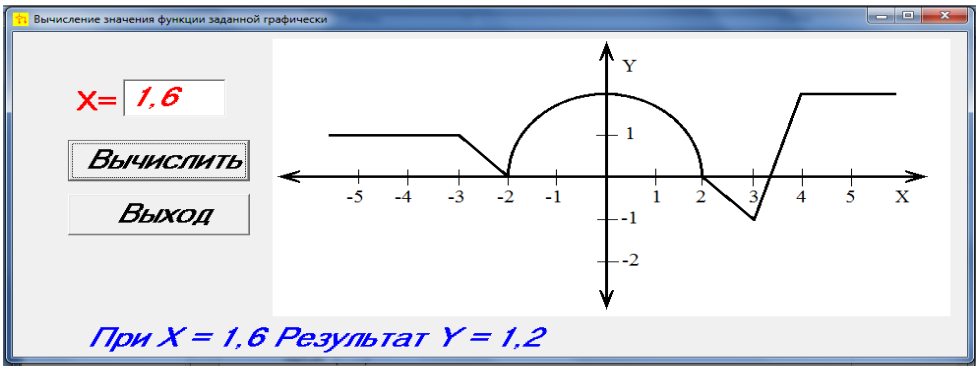
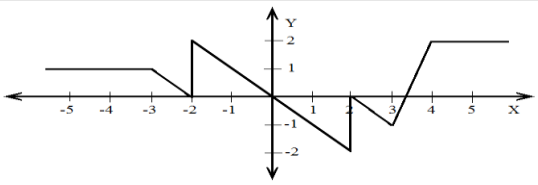
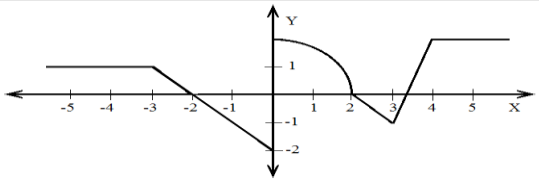
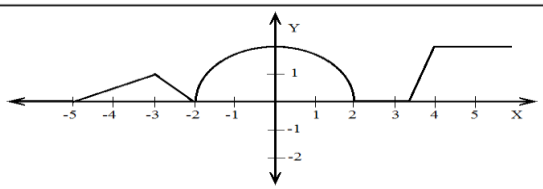
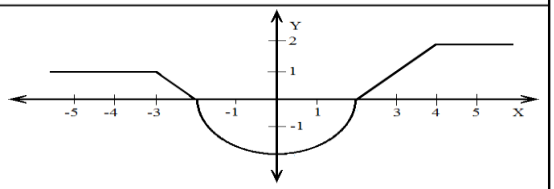
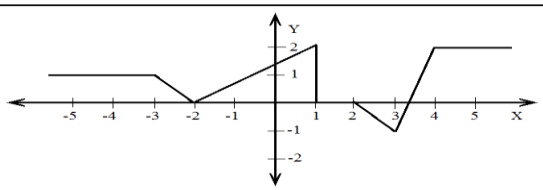
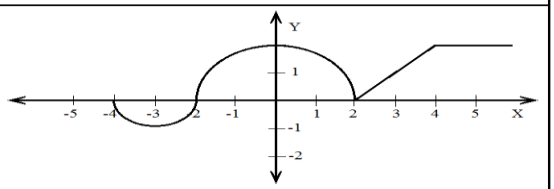
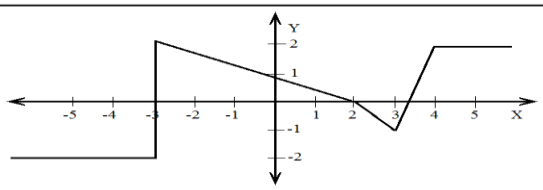
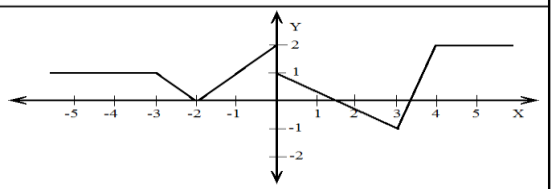
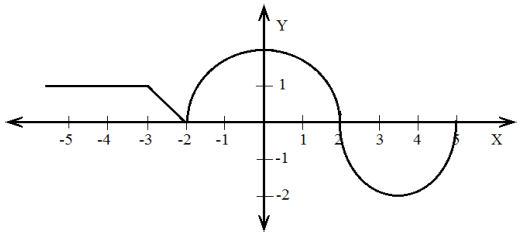
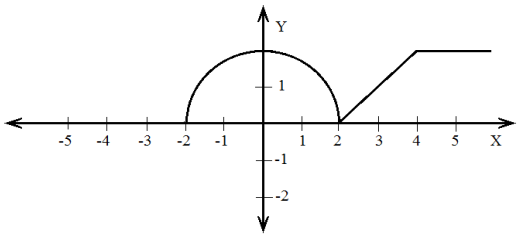
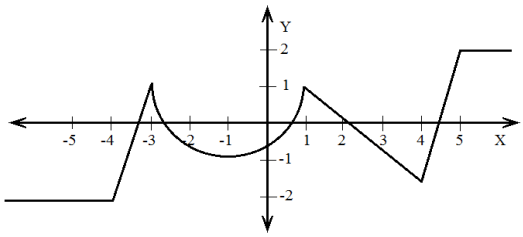
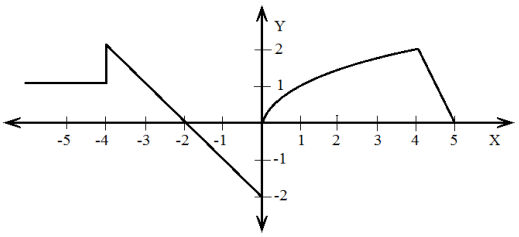
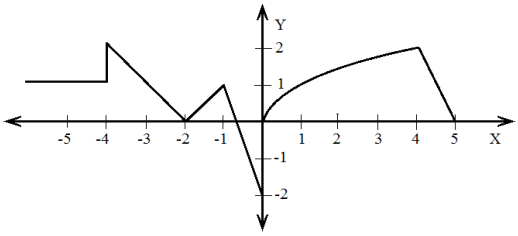
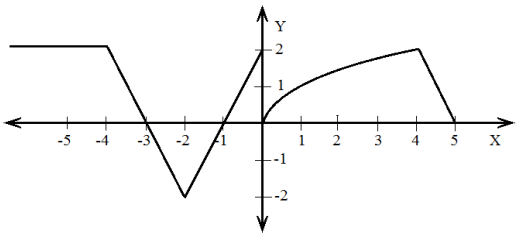
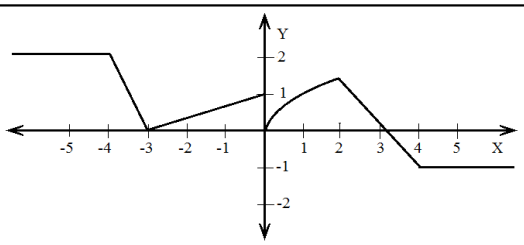
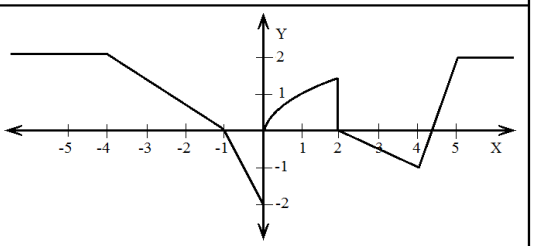
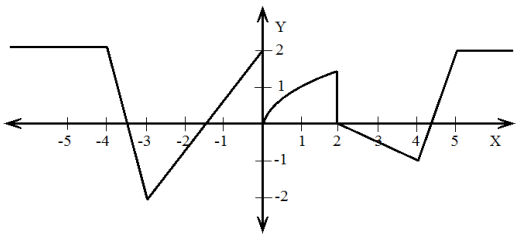
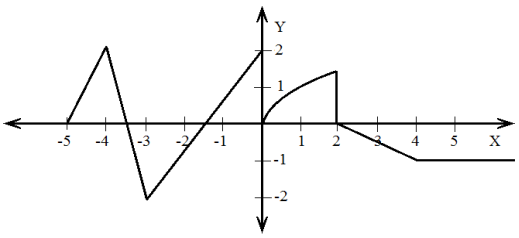
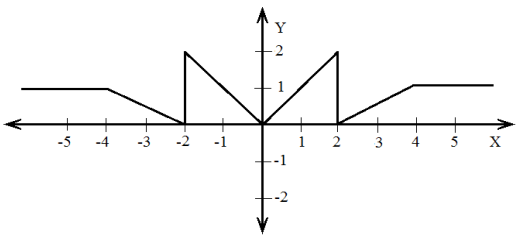
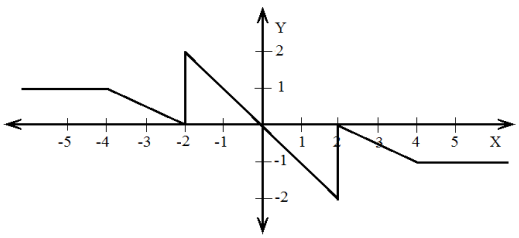
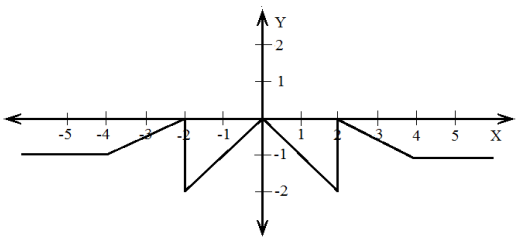
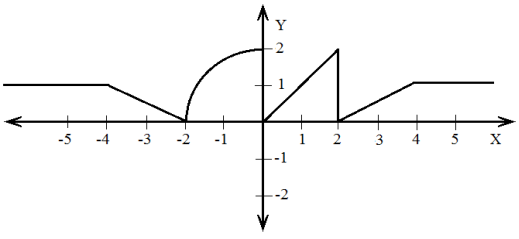



Рис. 3.15. Результаты задания 3.2.

## 3.8. Индивидуальные задания для выполнения первой части лабораторной работы

**Задание 3.2.** Создать проект на C++Builder, и написать программный код, который по введенному значению аргумента x вычисляет значение функции абсциссы y, заданный в виде графика. График функции взять из .

| № | Варианты решать используя оператор **if** | № | Варианты решать используя оператор **if/else** |
|---|---|---|---|
| 1 |  | 2 |  |
| 3 |  | 4 |  |
| 5 |  | 6 |  |
| 7 |  | 8 |  |

| № | Варианты решать используя оператор **if** | № | Варианты решать используя оператор **if /else** |
|---|---|---|---|
| **9** |  | **10** |  |
| **11** |  | **12** |  |
| **13** |  | **14** |  |
| **15** |  | **16** |  |
| **17** |  | **18** |  |
| **19** |  | **20** |  |
| **21** |  | **22** |  |

## 3.9. Оператор switch в C++ Builder

Существует класс задач, когда необходимо выбрать одно или несколько условий среди множества вариантов, писать на все случаи оператор условия **if** оказывается не целесообразным. Для решения такого класса проблем и создан оператор switch, о чем мы писали ранее.

Для реализации таких проектов C++ Builder имеет множество компонентов, следует заметить, при желании пользователь и сам может создавать свои компоненты, и такую возможность предоставляет среда проектирования. Мы изучим наиболее часто применяемые и существующие компоненты для этих целей.

**Кнопки-переключатели - группа радиокнопок (RadioGroup, RadioButton и GroupBox) и индикаторы (CheckBox, CheckListBox).**

При создании программ в C++ Builder для организации разветвлений часто используются компоненты в виде кнопок-переключателей. Состояние такой кнопки (включено - выключено) визуально отражается на форме. На форме (рис.3.16) представлены кнопки-переключатели двух типов (классов TCheckBox, TRadioGroup ) CheckBox, RadioGroup полную информацию о них можно получить в приложении 2.

Компоненты CheckBox и CheckListBox создает кнопку независимого переключателя, с помощью которой пользователь может указать свое решение типа да/нет. В программе состояние кнопки связано со значением булевой переменной, которая проверяется с помощью оператора **if**.

Компонент Radiogroup создает группу кнопок - зависимых переключателей. При нажатии одной из кнопок группы все остальные кнопки отключаются. В программу передается номер включенной кнопки (0,1,2,...), который анализируется с помощью оператора switch.

**Задание 3.3.** Ввести четыре числа – a,b,c,d, и сделать контроль на не цифру. Вычислить по усмотрению v=sin(a), v=cos(a), v=arctg(a) или $v=e^{-a}$ и указать выбранный элемент v. Найти по выбору максимальное из четырех чисел: max(v,b,c,d), или max (|v|,|b|,|c|,|d|). Создать форму, представленную на рис. 3.16, начертить блок схему и написать соответствующую программу.

**Интерфейс проекта (создание формы с компонентами), или так называемый эскизный проект.** Создайте форму с указанными компонентами, (кто затрудняется вернитесь к темам 1 и 2), как в предыдущих заданиях, скорректировав текст надписей и положение окон Edit и других компонент из эскизного проекта.

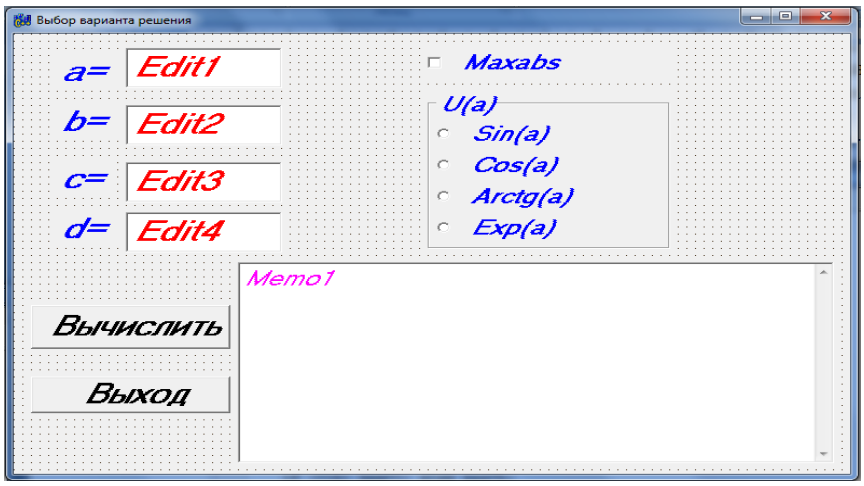



Рис. 3.16. Эскизный проект задания 3.3.

**Работа с компонентом CheckBox.** Выберите в меню компонентов Standard пиктограмму компонента CheckBox и поместите ее в нужное место формы. С помощью инспектора объектов измените заголовок (Caption) на Maxabs. В тексте программы появилась переменная CheckBox1 типа TCheckBox. Теперь в зависимости от того, нажата или нет кнопка, булева переменная CheckBox1->Checked будет принимать значение true или false.

**Работа с компонентом RadioGroup .** Выберите в меню компонентов Standard пиктограмму компонента RadioGroup и поместите ее в нужное место формы. На форме появится обрамленный линией чистый прямоугольник с заголовком RadioGroup1. Замените заголовок (Caption) на U(a). Для того чтобы разместить на компоненте кнопки, необходимо свойство Columns установить равным единице (кнопки размещаются в одном столбце). Дважды щелкните по правой части свойства Items мышью, появится строчный редактор списка заголовков кнопок. Наберите четыре строки с именами: в первой строке - Sin(a), во второй - Cos(a), в третьей - Arctg(a), в четвертой Exp(-a) нажмите ОК.

После этого на форме внутри обрамления появится четыре кнопки-переключателя с введенными надписями.

Обратите внимание на то, что в тексте программы появится переменная RadioGroup1, который указателем ссылается на класс *TRadioGroup. Теперь при нажатии одной из кнопок группы в переменной целого типа RadioGroup1->ItemIndex будет находиться номер нажатой клавиши (отсчитывается от нуля), что используется в тексте приведенной программы.

**Создание обработчиков событий FormCreate и BottonClick .**

Функции для обработки событий FormCreate и Botton1Click Botton2Click создаются аналогично тому, как и в предыдущих заданиях. Блок схема и текст функций и процедур приведены ниже.

Запустите программу и убедитесь в том, что все ветви алгоритма выполняются правильно. Форма в работе приведена на рис.3.17.

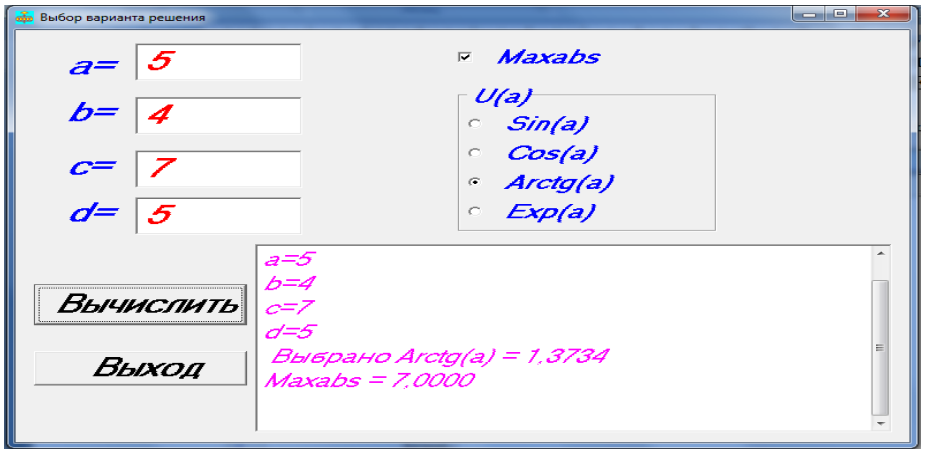



Рис. 3.17. Задание 3.3. Выбор варианта.

Блок – схема алгоритма к заданию 3.3.показан на рис.3.18.

## Программный код

**Внимание!** Здесь и далее без особой необходимости, (если мы не вмешивались) заголовочный файл будем пропускать.

```
//---------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include <math.h>
#include "Caseproj.h"
//---------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Edit1->Clear();
    Edit2->Clear();
    Edit3->Clear();
    Edit4->Clear();
    Memo1->Clear();
    Memo1->Lines->Add("  Лабораторная работа. Задание 3.3.");
}
```

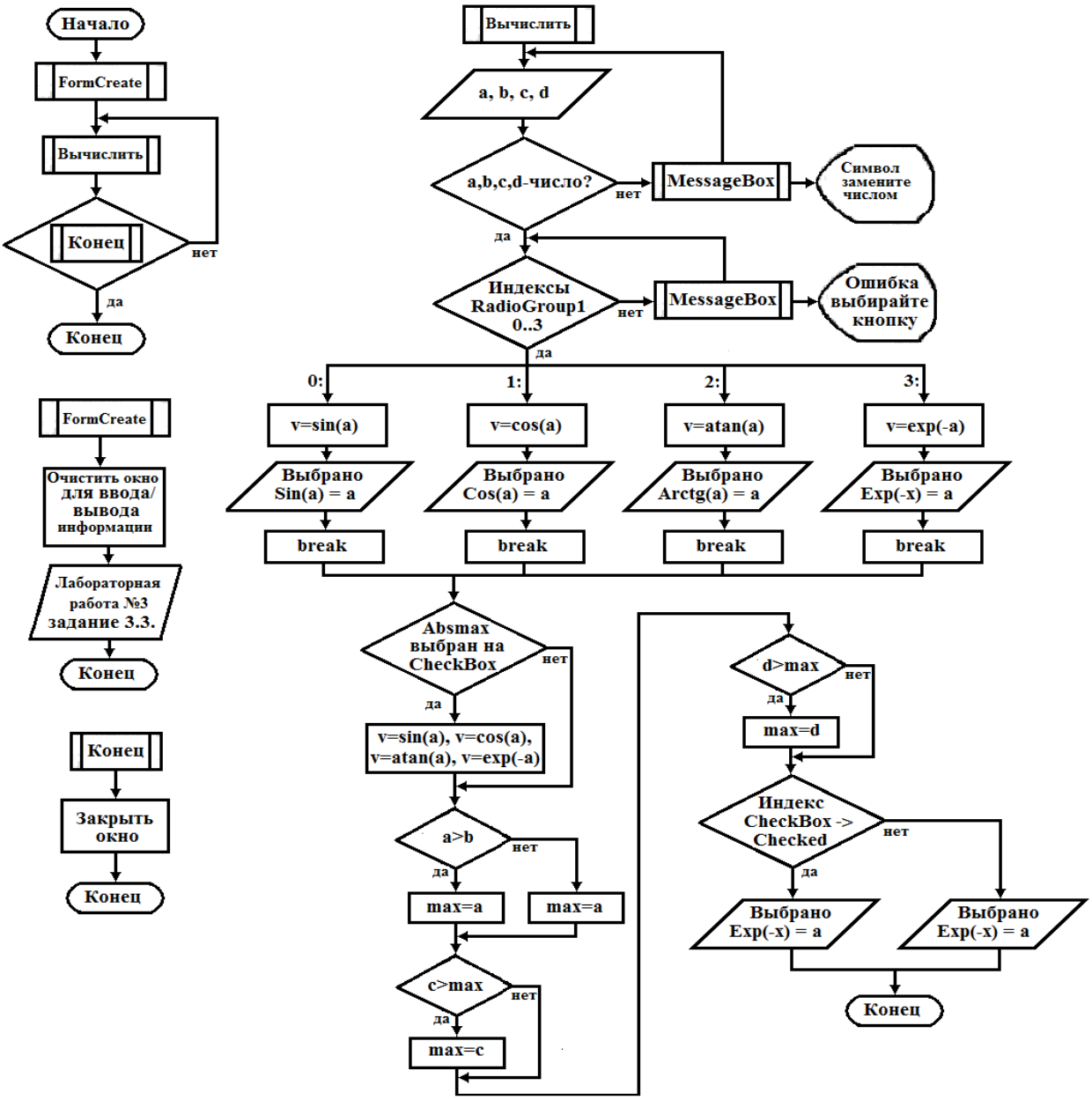



Рис.3.18. Блок-схема к заданию 3.3.

```
//---------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  Form1->Close();
}
//---------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   double v,a,b,c,d,max;
   try
   {
     a = StrToFloat(Edit1->Text);
     Memo1->Lines->Add("a="+Edit1->Text);
     b = StrToFloat(Edit2->Text);
     Memo1->Lines->Add("b="+Edit2->Text);
     c = StrToFloat(Edit3->Text);
```

```
     Memo1->Lines->Add("c="+Edit3->Text);
     d = StrToFloat(Edit4->Text);
     Memo1->Lines->Add("d="+Edit4->Text);
    }
    catch (EConvertError&)
    {
     Application->MessageBox("Символ замените числом ",
                             "Повторите ввод ",MB_OK);
     return;
    }
    switch (RadioGroup1->ItemIndex)
     {
       case 0:{
            v=sin(a); Memo1->Lines->Add("Выбрано Sin(a) = "
            +FloatToStrF(v,ffFixed,6,4));
              }
             break;
       case 1:{
              v=cos(a); Memo1->Lines->Add("Выбрано Cos(a) = "
            +FloatToStrF(v,ffFixed,6,4));
               }
              break;
       case 2:{
           v=atan(a); Memo1->Lines->Add(" Выбрано Arctg(a) = "
            +FloatToStrF(v,ffFixed,6,4));
               }
              break;
       case 3: {
            v=exp(-a); Memo1->Lines->Add(" Выбрано Exp(-a) = "
            +FloatToStrF(v,ffFixed,6,4));
                }
               break;
       default:
                {
      Application->MessageBox("Ошибка! Выберите РадиоКнопку",
      "Повторите ввод",MB_OK);
      return;
                }
     }
     if (CheckBox1->Checked)
        {
          v=fabs(v);
          b=fabs(b);
          c=fabs(c);
          d=fabs(d);
        }
     if (v>b) max=v; else max=b;
     if (c>max) max=c;
     if (d>max) max=d;
     if (CheckBox1->Checked) Memo1->Lines->Add("Maxabs = "
                             +FloatToStrF(max,ffFixed,6,4));
     else Memo1->Lines->Add("Max = "+FloatToStrF(max,ffFixed,6,4));
}
//---------------------------------------------------------
```

## 3.10. Индивидуальные задания ко второй части лабораторной работы.

Получите индивидуальное задание у преподавателя. Отредактируйте вид формы и текст программы в соответствии с полученным заданием. Предусмотрите вывод информации, показывающий, по какой ветви производились вычисления.

| № | Задание | Примечание |
|---|---|---|
| 1 | Дано число C. распечатать величину этого числа в словесном формате, учитывая его знак | Предусмотреть, что -9 ≤ C ≤ 9 |
| 3 | Дано число M. Напечатать фразу "Мне M лет", учитывая, что при некоторых значениях M слово "лет" надо заменить на слово "год" или "года" | Предусмотреть, что M<100, M-целые числа. |
| 4 | Дано число M. Напечатать фразу "Мы успешно сдали M экзаменов", согласовав окончание слова "экзамен" с числом M. | Предусмотреть, что 1 ≤ M ≤ 25. |
| 5 | Определить время года, к которому относится месяц M и определить количество дней в этом месяце. | Предусмотреть, что 1 ≤ M ≤ 12 год високосный. |
| 6 | В григорианском календаре каждый год, номер которого делится на 4, является високосным, за исключением тех, которые делятся на 100 и не делятся на 400 нацело. Определить число дней в году по номеру года. | Т.о. 1900г.-невисокосный 2000 г.-високос. |
| 7 | Определить D и M -дату K-го по счету дня високосного года. Месяц вывести числом и в словесной форме. | D-день, M-месяц. Предусмотреть, что 1 =< K > = 366 |
| 8 | Даны два числа D и M. Определить день недели с датой D и M, считая, что год високосный и 1 Января приходится на день недели W. | D-день, M-месяц. |
| 9 | Даны два числа D и H. Определить K -порядковый номер того дня високосного года, который имеет дату D и M. | D-день, M-месяц. |
| 10 | Даны три числа D, M, и G,определяющие день, месяц и год. Проверить образуют ли они правильную дату и вывести соответствующее сообщение. | 29.02.90- не правильная дата. Определение вис. Г. см. в условии № 6. |
| 11 | Даны три целых числа определяющие дату: год, месяц, день. Считая, что год невисокосный, определить дату следующего дня. | |

| | | |
|---|---|---|
| 12 | В японском календаре был принят 60-ти летний цикл, состоящий из пяти 12-ти летних подциклов. Внутри подцикла года носили названия животных: мыши, коровы, тигра, зайца, дракона, змеи, лошади, овцы, обезьяны, курицы, собаки и свиньи. Попарно года в цикле обозначались названиями цвета: зеленый, красный, желтый, белый и черный и другие года по японскому календарю.<br>По номеру года определить название его по японскому календарю. | Начало очередного цикла: 1984 год-год зеленой мыши.<br>1985-год зеленой коровы.<br>1986-год красного тигра.<br>1987-год красного зайца. |
| 13 | Даны два целых числа: D (день) и M (месяц), определяющие дату рождения индивида. Вывести знак Зодиака в компонент Image, соответствующий этой дате.<br>Знаки можно создать, используя графический редактор, или взять файл, созданный на любом графическом редакторе. | «Водолей» (20.1–18.2),<br>«Рыбы» (19.2–20.3),<br>«Овен» (21.3–19.4),<br>«Телец» (20.4–20.5),<br>«Близнецы» (21.5–21.6),<br>«Рак» (22.6–22.7),<br>«Лев» (23.7–22.8),<br>«Дева» (23.8–22.9),<br>«Весы» (23.9–22.10),<br>«Скорпион» (23.10–22.11),<br>«Стрелец» (23.11–21.12),<br>«Козерог» (22.12–19.1). |
| 14 | Вывести строку-описание данного числа, например: 256 — «двести пятьдесят шесть», 811 — «восемьсот одиннадцать». В эскиз проекта рекомендуем компоненты Label, Memo, Button. | Дано целое число в диапазоне 100–999 |
| 15 | Распределите слово кочерга в указанном диапазоне. | 1-7 |
| 16 | Вывести строку-описание указанного количества заданий, обеспечив правильное согласование числа со словами «учебное задание», например: 18 — «восемнадцать учебных заданий», 23 — «двадцать три учебных задания», 31 — «тридцать одно учебное задание». | Дано целое число в диапазоне 10–40, определяющее количество учебных заданий |
| 17 | Вывести строку-описание указанного возраста, обеспечив правильное согласование числа со словом «год», например: 20 — «двадцать лет», 32 —«тридцать два года», 41 — «сорок один год». | Дано целое число в диапазоне 1–100, определяющее возраст (в годах). |
| 18 | Даны два целых числа: N — достоинство ($0 \le N \le 10$) и M — масть карты ($1 \le M \le 4$). Вывести название соответствующей карты вида «шестерка бубен», «Joker червей» и т. п. | Мастям игральных карт присвоены порядковые номера:<br>1 — пики, 2 — трефы, 3 — бубны, 4 — червы. Достоинству карт, номера:0-10 |
| 19 | Даны два целых числа: N — достоинство ($11 \le N \le 14$) и M — масть карты ($1 \le M \le 4$). Вывести название соответствующей карты вида «шестерка бубен», «дама червей», «туз треф» и т. п. | Мастям игральных карт присвоены порядковые номера:<br>11 —14 валет — туз.<br>1 — пики, 2 — трефы, 3 — бубны, 4 — червы.. |

| | | |
|---|---|---|
| 20 | Элементы равностороннего треугольника пронумерованы следующим образом: 1 — сторона a, 2 — радиус R1 вписанной окружности (R1=$a\sqrt{3/6}$), 3 — радиус R2 описанной окружности ($R2 = 2 \times R1$), 4 — площадь S =$\sqrt{3}/4 \times a^2$. Дан номер одного из этих элементов и его значение. Вывести значения остальных элементов данного треугольника (в том же порядке). | |
| 21 | Элементы равнобедренного прямоугольного треугольника пронумерованы следующим образом: 1 — катет a, 2 — гипотенуза c = $a\sqrt{2}$, 3 —высота h, опущенная на гипотенузу (h = c/2), 4 — площадь S = c·h/2. Дан номер одного из этих элементов и его значение. Вывести значения остальных элементов данного треугольника (в том же порядке). | |
| 22 | Элементы окружности пронумерованы следующим образом: 1 — радиус R, 2 — диаметр D = 2R, 3 — длина L = 2πR, 4 — площадь круга S = π·R2. Дан номер одного из этих элементов и его значение. Вывести значения остальных элементов данной окружности (в том же порядке). В качестве значения π использовать 3.14. | |
| 23 | Локатор может принимать три цифровые команды поворота: 1 — поворот налево, −1 — поворот направо, 2 — поворот на 180◦. Дан символ C — исходная ориентация локатора и целые числа N1 и N2 — две посланные команды. Вывести ориентацию локатора после выполнения этих команд. | Локатор ориентирован на одну из сторон света («С» — север, «З» — запад, «Ю» — юг, «В» — восток) |
| 24 | Робот может принимать три цифровые команды: 0 — продолжать движение, 1 — поворот налево, −1 — поворот направо. Дан символ C — исходное направление робота и целое число N — посланная ему команда. Вывести направление робота после выполнения полученной команды. | Робот может перемещаться в четырех направлениях («С» — север, «З» — запад, «Ю» — юг, «В» — восток) |
| 25 | Единицы массы пронумерованы следующим образом: 1 — килограмм, 2 — миллиграмм, 3 — грамм, 4 — тонна, 5 — центнер. Дан номер единицы массы (целое число в диапазоне 1–5) и масса тела в этих единицах (вещественное число). Найти массу тела в килограммах. | |
| 26 | Единицы длины пронумерованы следующим образом: 1 — дециметр, 2 — километр, 3 — метр, 4 — миллиметр, 5 — сантиметр. Дан номер единицы длины (целое число в диапазоне 1–5) и длина отрезка в этих единицах (вещественное число). Найти длину отрезка в метрах. | |

| | | |
|---|---|---|
| 27 | Арифметические действия над числами пронумерованы следующим образом: 1 — сложение, 2 — вычитание, 3 — умножение, 4 — деление. Дан номер действия N (целое число в диапазоне 1–4) и вещественные числа A и B (B не равно 0). Выполнить над числами указанное действие и вывестирезультат. | |
| 28 | Если сумма трех попарно различных действительных чисел x, y, меньше единицы, то наименьшее из этих трех чисел заменить полусуммой двух других; в противном случае заменить меньшее из x и y полусуммой двух оставшихся значений. | |
| 29 | По заданному номеру месяца определить сезон (весна, лето, осень или зима). | |
| 30 | Для натурального числа k вывести фразу “мы выпили k бутылок фанты”, согласовав окончание слова “бутылка” с числом k. | |

## 3.11. Контрольны вопросы

1. Основные принципы объектно-ориентированного программирования.
2. Унифицированный язык моделирования (UML).
3. Что такое диаграмма классов (Class diagram)?
4. Основы языка C++ Builder. Типы данных.
5. Основы языка C++ Builder. Массивы.
6. Основы языка C++ Builder. Структуры.
7. Переменные с изменяемой структурой.
8. Условная операция (тернарная альтернатива).
9. Операторы условий `if` и `if/else`.
10. Оператор выбора условия switch .
11. Оператор switch в C++ Builder.
12. Понятия об интерфейсе проекта.

## Лабораторная работа № 4.
## Циклические алгоритмы

***Цель работы:*** изучить основные диаграммы UML; основы языка ***C++*** и ***C++ Builder*** и освоить циклические операторы *while, do-while, for*, научиться реализовывать циклические алгоритмы, используя меню и панель управления. Изучив простейшие средства отладки программ в среде C++ *Builder*, составить и отладить программу.

### 4.1. Объекты и отношения в C++ и C++ Builder

В прошлой теме мы дали определения объекту и классам, однако, что понимаем под объектом, и как определяются простейшие классы в C++и C++ Builder и как их описать или создать не указали. Есть много общего в функционировании и декларации классов в языках C++ и C++ Builder, но существуют незначительные различия имеются свойственные самому C++ Builder, связанные с компонентами и характеристиками среды. Мы будем рассматривать для языка C++, а при наличии дополнительных устройств или объектов будем конкретизировать C++ Builder.

Прежде чем заняться конкретным изучением классов C++, хотелось бы немного поговорить об объектно-ориентированном програмировании (ООП) вообще, как таковой.

В предыдущих темах говоря о парадигмах программирования, немного говорили об ООП, однако сказанное пока недостаточно, чтобы овладеть технологиями программирования, пополним наши знания в этой области. Известный теоретик Грейди Буч парадигму ООП формулирует так:

*«ООП - методология программирования, основанная на организации программы, в виде совокупности объектов, каждый из которых является представителем определенного класса, а классы образуют иерархию наследования».*

Однако язык C++ таковым, не является хотя бы потому, что он сохраняет все возможности процедурного языка C. В C++ можно создать, например, совершенно отдельно стоящую глобальную переменную, да и сама функция main — совершенно «внеклассовая». Подобная универсальность C++ может быть, как преимуществом, так и недостатком, если ею злоупотреблять. У программиста, впервые применяющего объектный подход, всегда имеется тенденция мыслить старыми, процедурными категориями.

Итак, центральным элементом абстракции объектно-ориентированной методологии является, очевидно, объект.

**Объект**. Понятие объекта тесно связано с понятиями класса, интерфейса и других программных конструкций, реализуемых в парадигмах ООП. Объекты программного мира моделируют объекты реального или воображаемого (виртуального) мира. Как модель, программный объект представляет собой некоторую абстракцию реального объекта, предполагая выделение ее существенных свойств, и игнорирование тех, что безразличны с насущной, «сиюминутной», точки зрения.

Как *нечто,* как единичность, объект (реальный или программный) отличен от всего остального.

Объект обладает *индивидуальностью и принадлежность какому-либо классу*, который описывается как переменная, и при присвоении некоторых численных значений о нем можно говорить, как об объекте см. рис 4.1.

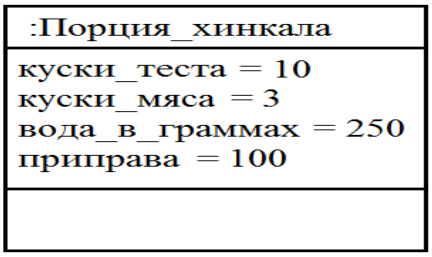



Рис. 4.1. Объект порции хинкала.

Понятие индивидуальности программного объекта определить сложно, но интуитивно ясно, что объект, как бы его ни переименовали, и как бы ни менялось его состояние, остается той же самой единичной сущностью (о сущностях говорили в теме 3, при изучении UML диаграмм) с момента создания и до своего уничтожения. Заметим, что о программном объекте можно *говорить* точно так же, как о реальном объекте. В целом, как видно, объект является *моделью* предмета реального мира в программе.

**Состояние объекта.** Объект, прежде всего, характеризуется своим *состоянием.* Возьмем, к примеру, конкретный самолет. Он — объект, поскольку есть нечто цельное. С точки зрения летчика у него есть приборный щиток, отражающий скорость, высоту полета и др. параметры, и органы управления см. рис. 4.2. Но в самолете масса частей, спрятанных под капотом, состояние которых не исчерпывается показаниями приборов.

| Самолет |
| --- |
| экипаж_самолета |
| система_навигации |
| система_управления_полетом |
| взлетно-посадочная_система |
| система_корпуса |
| система_жизнеобеспечения |

Рис.4.2. UML диаграмма для класса самолет.

На самом деле состояние, как таковое вообще, может быть скрыто (*инкапсулировано)* от внешнего взгляда. Однако в зависимости от своего состояния объект по-разному взаимодействует со своим окружением, что приводит нас к поведению объекта.

**Поведение объекта.** Поведение — это то, взаимодействие объекта с окружающими объектами. Воздействия объекта на окружение или окружения на него. Объект может рассматриваться как аналог предмета, а поведение — как реакция на манипулирование и ответные действия самим объектом. Для каждого объекта существует определенный набор отношений, возможностей и возможных действий над ним, как указывали мы при изучении темы 3 в отношениях, а для примера самолет можем рассмотреть воздушный транспорт.

Действия в отношении к объектам иногда называют передачей *сообщений* между ними. Операции над объектами называют обычно *методами или функция - элемента*. Эти функции являются структурными элементами определения класса, к которому принадлежит объект см. рис.4.3. диаграмма объекта «Воздушная транспорт.


Рис. 4.3. Диаграмма объекта «Воздушная транспорт».

**Отношения между объектами.** Об отношениях мы подробно говорили при изучении диаграмм UML, в предыдущей теме, теперь конкретизируем их. Нас интересует, как их отобразить в программном коде. Естественно, программа, т.е. цельная система, реализуется только во взаимодействии всех ее объектов. Здесь можно выделить в основном два типа взаимодействий, или отношений: *связь* и *агрегацию.*

Связь является довольно очевидной разновидностью взаимодействий — один объект может воздействовать на другой, являющийся в известном смысле автономной сущностью. Тут существует отношение подчинения — «А использует В». Один объект является активным, другой — пассивным. Понятно, что в системе один и тот же объект может выступать как в активной, так и в пассивной роли по отношению к различным объектам.

Другой тип отношений — агрегация, когда один объект является составной частью, т.е. элементом класса другого — «А содержит В» см. рис. 4.4. Агрегация может (однако не всегда) означать физическое вхождение одного объекта в другой; в C++ это соответствует описанию первого объекта в качестве элемента данных другого объекта.

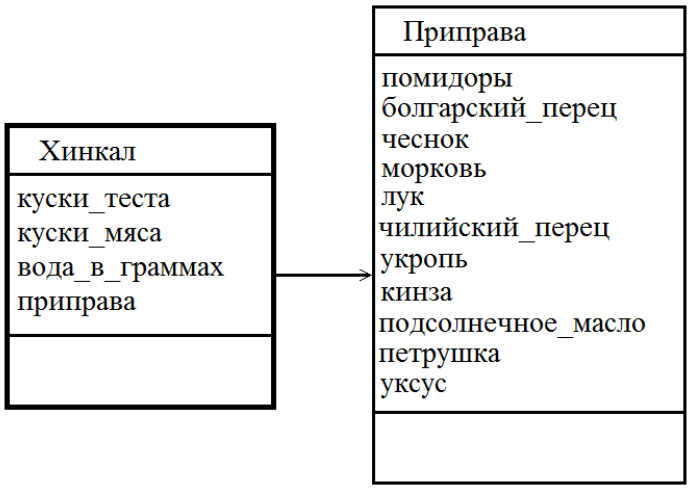



Рис. 4.4. Отношения агрегации.

**Наследование.** Если существенными видами отношений между объектами являются связь и агрегация, то фундаментальное отношение между классами — это *наследование.* Один класс может наследовать другому. В C++ в таком случае говорят, что один класс является *базовым,* а другой (который наследует первому) — *производным*. Еще их называют соответственно классом-предком и классом-потомком *родительским – дочерним.* Наследование может быть *прямым,* когда один класс является непосредственным предком (потомком) другого, или *косвенным,* когда имеют место промежуточные наследования см. рис. 4.5.

Родительский класс наследует всю структуру характеристик и поведение базового, однако может дополнять или модифицировать их. Если класс является дочерним по отношению к А, то с логической точки зрения «В есть А». Например, понятие, или класс, «самолет» (В) является производным по отношению к понятию «воздушный транспорт» (А) см. рис. 4.3 и 4.4.


**(а)**. Наследование признаков 1 родителя в одном поколении.

**(б)**. Наследование признаков 2 родителей в одном поколении.

**(в)**. Наследование признаков n родителей в одном поколении.

**(г).** Наследование признаков n родителей в m поколениях.

Рис. 4.5. Иерархия классов при различных видах наследования.

Как и в логике, здесь существует взаимосвязь между «содержанием» и «объемом» понятия. Производный класс имеет большее содержание, но меньший объем, чем базовый.

Родительский класс наследует всю структуру характеристик и поведение базового, однако может дополнять или модифицировать их. Если класс В является дочерним по отношению к А, то с логической точки зрения «В есть А». Например, понятие, или класс, «самолет» (В) является производным по отношению к понятию «воздушный транспорт» (А).

Как и в логике, здесь существует взаимосвязь между «содержанием» и «объемом» понятия. Производный класс имеет большее содержание, но меньший объем, чем базовый.

**Полиморфизм.** *Полиморфизм,* наряду с наследованием, является фундаментальной концепцией объектной модели программирования. Без него ООП потеряло бы значительную долю своего смысла.

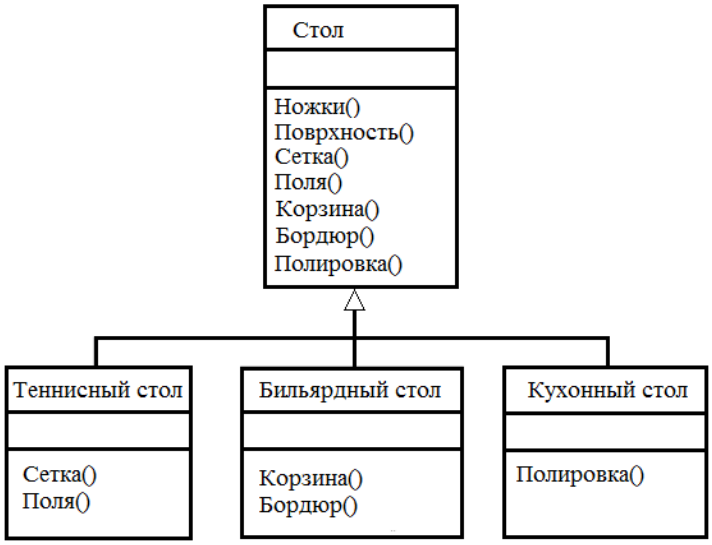



Рис.4.6. Представление виртуального метода в UML.

Суть полиморфизма в том, что с объектами различных классов, имеющих один и тот же базовый класс, можно при определенных условиях обращаться, как с объектами базового класса. Однако объект, являющийся, по видимости, объектом базового класса, будет вести себя по-разному в зависимости от того, что он такое на самом деле, т.е. представитель какого из производных классов.

В C++ полиморфное поведение объектов обеспечивается механизмом *виртуальных функций-элементов* поздним связыванием. Допустим, программа должна выводить на экран различные геометрические фигуры. Она определяет класс «фигура», в котором предусмотрен виртуальный метод «нарисовать» (в C++ это был бы *абстрактный класс).* От данного класса можно произвести несколько классов — «точка», «линия», «круг» и т. д., — каждый из которых будет *по-своему* определять этот метод.

Указатель на класс «фигура» может ссылаться на объект любого из производных классов (поскольку все они *являются* фигурами), и для указываемого им объекта можно вызвать метод «нарисовать», не имея представления о том, что это на самом деле за фигура.

Приведем еще пример, связанный полиморфизмом, например, стол, как класс предок, а потомки обеспечиваются поздним связыванием см. рис. 4.6. это теннисный стол, бильярдный стол и кухонный стол

**Абстракция и инкапсуляция.** Об этих принципах объектного подхода мы уже упоминали. На самом деле это принципы программирования, присущие не только объектно-ориентированной модели. Но хотелось бы несколько уточнить их в отношении к организации классов.

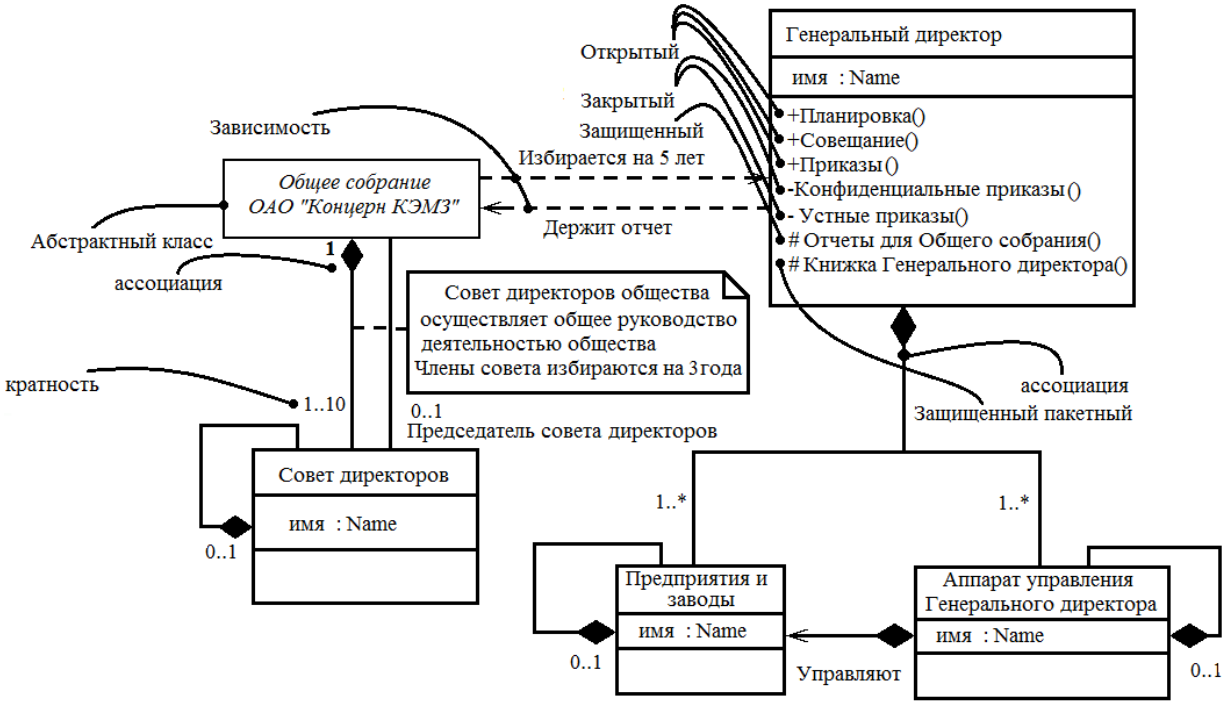



Рис. 4.7. Диаграммы UML. Абстракция и инкапсуляция.

Чтобы *абстрагировать* объект, он должен быть сравнительно «слабо связан» с окружающим миром. Он должен обладать сравнительно небольшим набором (существенных) свойств, характеризующих его отношения с другими объектами. С другой стороны, выделение класса как некоторого понятия, охватывающего целый ряд различных объектов, также является моментом абстракции.

Поэтому абстрагирование, как таковое, имеет два аспекта: выделение *общих* и в тоже время *существенных* свойств, описывающих поведение ряда схожих предметов.

С абстракцией неразрывно связан принцип *инкапсуляции,* которая осуществляет сокрытие второстепенных деталей объекта. Для чего важно выделить сначала существенные его свойства, однако для выделения существенных свойств, необходимо отвлечься мысленно от второстепенных свойств. Так что в действительности речь может идти

только о едином акте, что отвлеченно выделяет всего два отдельных момента. И с технической точки зрения абстракция и инкапсуляция выражаются в том, что классы состоят из *интерфейса* и *реализации.* Интерфейс представляет абстрагированную сущность объектов (описывает множество операций, определяющих, что может делать класс или компонент). Реализация скрыта в своих деталях от пользователя класса.

Сделав такие предварительные замечания о понятиях объектно-ориентированного подхода, перейдем к конкретному рассмотрению связей и классов, как они реализованы в языке C++.

**Связи в C++.** В предыдущем пункте мы рассмотрели отношение между объектами, и, говоря об объекте или сущности в целости, мы мысленно (у себя в памяти) представляли его, используя различные связи, и он в единственном числе. Однако люди полиглоты знают, описание этой сущности на различных языках приходится по-разному, внеся лепту особенностей языка. Язык C++ имеет свои особенности также. Дополним сказанное различными примерами. Рассмотрим реализацию понятия даты с использованием **struct** для того, чтобы определить представление даты date и множества функций для работы с переменными этого типа:

```
struct date { int month, day, year; }; // дата: месяц, день,
 год }
date today;
date tomorrow;
date tonight;
void set_date(date*, int, int, int);
void next_date(date*);
void print_date(date*);
  // ...
```

Никакой явной связи между функциями и типом данных нет. Такую связь можно установить, описав функции как элементы:

```
struct date {
    int month, day, year;
void set(int, int, int);
void get(int*, int*, int*);
void next();
void print();
};
```

Функции, описанные таким образом, называются функциями элементов и могут вызываться только для специальной переменной соответствующего типа с использованием стандартного синтаксиса для доступа к элементам структуры, один из способов передачи связей. Например:

```
date today;         // сегодня
date my_burthday;   // мой день рождения
```

```
  date tomorrow;     // завтра
  date tonight;      //  сегодня ночь
  void f()
  {
      my_burthday.set(12,05,1956);
      today.set(18,8,2021);
      tonight.set (17,8,2021);
      tomorrow.set(19,8,2021);

      my_burthday.print();
      today.next();
  }
```

Поскольку разные структуры могут иметь функции элементы с одинаковыми именами, при определении функции элемента необходимо для уточнения связи указывать имя структуры:

```
  void date::next()
  {
      if ( ++day > 28 ) {
     // делает сложную часть работы
      }
  }
```

В функции элементе имена элементов могут использоваться без явной ссылки на объект. В этом случае имя относится к члену того объекта, для которого функция была вызвана.

Другой пример, где передаются связи.

```
    /* Базовый класс Фигура для него заданы методы, реализация
которых отложена для выведенных классов.*/

  class Shape
{// Родительский (базовый) класс
  public:
// Внешняя часть класса
  virtual bool Draw() = 0; // Перерисовать фигуру
  virtual bool Move(int _x, int _y) = 0; // Сдвинуть фигуру
     virtual bool Zoom(int scale) = 0; // Масштабировать
  protected:
/* Защищенная часть класса, доступная только выведенным из
него классам */
     int coordX; // Атрибут - координата X
     int coordY; // Атрибут - координата Y
};

// Класс Круг выведенный из класса Фигура
  class Circle : public Shape
{  // Дочерний (производный) класс
  public:
   virtual bool Draw() {...}; // Реализация перерисовки
   virtual bool Move(int _x, int _y) {...}; // Реализация
сдвига
   virtual bool Zoom(int scale) {...};
                  //Реализация операции масштабирования
```

```
  private:
     // Внутренняя часть доступная только самому классу
     double radius; // Атрибут - длина радиуса
};

// Класс Квадрат выведенный из класса Фигура
  class Square : public Shape
{   // Дочерний (производный) класс
  public:
   virtual bool Draw() {...}; // Реализация перерисовки
   virtual bool Move(int _x, int _y) {...}; // Реализация
сдвига
   virtual bool Zoom(int scale) {...};
                  //Реализация операции масштабирования
  private:
// Внутренняя часть доступная только самому классу
     int side; // Атрибут - длина стороны квадрата
};
```

К процессу осуществления связей мы вернемся позже.

## 4.2. Введение в классы

**Класс** — это множество объектов, имеющих одинаковую структуру. Класс в ООП является аналогом *понятия,* или категории. В то время как объект представляет собой конкретную сущность, класс является абстракцией сущности объекта. Конкретный объект является *представителем* класса, где структура характеристик и все *потенциальные* отношения между объектами заложены в классе.

Каждый объект имеет независимую память, которая состоит из других объектов, и является представителем класса, который выражает общие свойства объектов (таких, как целые числа или списки).

В классе задаётся поведение (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.

Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память и поведение, связанное с экземплярами определённого класса, автоматически доступно любому классу, расположенному ниже в иерархическом дереве.

Таким образом, программа представляет собой набор объектов, имеющих состояние и поведение. Объекты взаимодействуют посредством сообщений, что можно увидеть в примерах, приводимых ниже. Естественным образом выстраивается иерархия объектов: программа в целом — это объект, для выполнения своих функций она обращается к входящим в неё объектам, которые, в свою очередь, выполняют

запрошенное путём обращения к другим объектам программы. Естественно, чтобы избежать бесконечной рекурсии в обращениях, на каком-то этапе объект трансформирует обращённое к нему сообщение в сообщения к стандартным системным объектам, предоставляемым языком и средой программирования.

С позиции программирования класс в ООП предназначен, чтобы предоставить пользователю инструмент для создания новых типов, столь же удобных в обращении сколь и встроенные типы. В идеале при функционировании тип, определяемый пользователем, не должен отличаться от встроенных типов. Диаграммы UML см. в работе №3.

Тип есть конкретное представление некоторого объекта или концепции (понятия). Например, имеющийся в C++ тип int с его операциями +, -, * и т.д. обеспечивает ограниченную, но конкретную версию математического понятия целого числа. Новый тип класс создается для того, чтобы дать специальное и конкретное определение понятия, которому ничто прямо и очевидно среди встроенных типов не отвечает, поскольку ему присуще некоторое поведение, кроме того классы могут находиться еще и в специфических отношениях между собой используя различные связи.

Объявляются классы в C++ следующим образом:

```
    class <имя класс>
{private:     <внутренние (недоступные) компоненты класса>
protected:   <защищенные компоненты класса>
                    public:      <общие (доступные) компоненты
класса>
    };
```

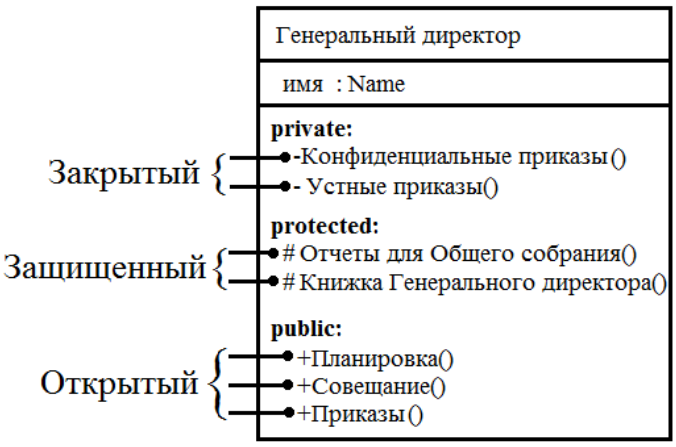



Рис. 4.8. Пример обозначения компонентов в UML диаграммах.

Сначала мы приведем простейший пример определений классов, которые можно было бы разместить в заголовочном файле. Так обычно и

делается, если классы должны быть доступны нескольким модулям программы.

После этого мы, опираясь на показанный пример, объясним элементарные моменты строения класса в C++.

Приведенный ниже код определяет два класса, которые могли бы применяться в графической программе. Это классы точек и линий.

```
// My_Class.h: Пример двух геометрических классов. //
const int MaxX = 2200; //Максимальные значения координат.
const int MaxY = 1640;
//
struct, Point { // Определяем класс точек.
int GetX(void) ;
int GetY(void) ;
     void SetPoint(int, int);
private:
     int x;
     int y;
};
class Line
{ // Определяем класс линий.
     Point p0;
     Point p1;
public:
Line(int x0, int y0, int xl, int yl);
// Конструктор.
~Line(void); // Деструктор.
     void Show(void);
};
```

Ну вот, такие вот классы. Теперь разберем различные моменты этих определений.

*Иногда может потребоваться предварительное объявление класса, если нужно, например, объявить указатель на объект класса прежде, чем будет определен сам класс. Предварительное объявление в этом смысле подобно прототипу функции и выглядит так:*

```
class SomeClass;
```

**Заголовок определения.** Определение класса начинается с ключевых слов **class**, **struct** или **union**. Правда, **union** применяется крайне редко. И структуры, и классы, и объединения относятся к «классовым» типам C++. Разницу между этими типами мы рассмотрим чуть позже.

**Спецификации доступа.** Ключевые слова отображенные в виде меток **private** и **public** называются *спецификаторами доступа.* Спецификатор **private** означает, что элементы данных и элементы-функции, размещенные под ним, доступны только функциям-элементам данного класса. Это так называемый *закрытый доступ.*

Спецификатор **public** означает, что размещенные под ним элементы доступны как данному классу, так и функциям других классов и вообще любым функциям программы через представителя класса.

Есть еще спецификатор *защищенного* доступа **protected**, означающий, что элементы в помеченном им разделе доступны не только в данном классе, но и для функций-элементов классов, производных от него.

Допускается умолчание спецификаторов **private** и **protected** в этом случае метка **public** делит тело класса на две части, до метки как **private**, а после **public**.

**Структуры, классы и объединения.** Типы, определяемые с ключевыми словами **struct, class** и **union**, являются классами. Отличия их сводятся к следующему:

- Структуры и классы отличаются только доступом по умолчанию. Элементы, не помеченные никаким из спецификаторов, в структурах имеют доступ **public** (открытый); в классах — **private** (закрытый).
- В объединениях по умолчанию принимается открытый доступ.
- Элементы (разделы) объединения, как и в С, перекрываются, т. е. начинаются с одного и того же места в памяти.

**Элементы данных и элементы-функции.** Элементы данных класса совершенно аналогичны элементам структур в С, за исключением того, что для них специфицирован определенный тип доступа. Объявления элементов-функций аналогичны прототипам обычных функций.

**Конструктор и деструктор.** В классе могут быть объявлены две специальные функции-элемента. Это *конструктор* и *деструктор.* Класс Line в примере объявляет обе эти функции.

Конструктор отвечает за создание представителей данного класса. Его объявление записывается без типа возвращаемого значения и ничего не возвращает, а имя должно совпадать с именем класса. Конструктор может иметь любые параметры, необходимые для *конструирования,* т. е. создания, нового представителя класса. Если конструктор не определен, компилятор генерирует *конструктор по умолчанию,* который просто выделяет память, необходимую для размещения представителя класса. Класс может иметь несколько конструкторов.

Деструктор отвечает за уничтожение представителей класса. Это происходит либо в случае, когда автоматический объект данного класса выходит из области действия, либо при удалении динамических

объектов, созданных операцией **new**. И один деструктор может уничтожить любое количество конструкторов в одном классе.

Деструктор объявляется без типа возвращаемого значения, ничего не возвращает и не имеет параметров. Если деструктор не определен, генерируется деструктор по умолчанию, который просто возвращает системе занимаемую объектом память. Более подробно о конструкторах и деструкторах мы поговорим ниже.

***Вид класса для C++ Builder***

```
class <имя класс>
{
        public:   <общие (доступные) компоненты
класса>
    // Данные, методы, свойства, события.
        _published <видны в инспекторе объектов и
изменяемы>
    // Данные, свойства.
        private:  <внутренние (недоступные) компоненты
класса>
    // Данные, методы, свойства, события
        protected:     <доступно только потомкам>
    // Данные, методы, свойства, события
    };
```

***Примечание!*** *Имя класса в C++ Builder может быть любым идентификатором. Идентификатор классов, наследующих классам библиотеки VCL в C++ Builder, принято начинать с символа T.*

***Элементы данных (поля)*** – переменные и константы доступные через объекты класса. У любого экземпляра объекта имеются индивидуальные (собственные без связей) переменные поля.

***Методы*** – функция элемента введенные в классе, которые доступны через объект класса, и обрабатывают значения полей. Возможно создание методов класса и через идентификатор, которые не связаны, с конкретными объектами.

***Свойства*** – это механизм доступа к данным (полям) через функции чтения и записи. Следует заметить, в стандартном языке ***C++*** понятие свойства отсутствует.

**Ссылки на себя.** В функции элементе на элементы объекта, для которого она была вызвана, можно ссылаться непосредственно. Приведем отлаженный пример консольного варианта, где все это очевидно:

**Пример 4.1**. Ссылки и указатели на себя

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

```
#include <iostream.h>
#include <conio.h>
//---------------------------------------------------------
class My_cout {
public:
   int x;
void print_date() {
   cout << x << "\n";
 }
};
#pragma argsused
int main(int argc, char* argv[])
{
   My_cout vyvod_znach,
           *vyvod_znachPtr = &vyvod_znach,
           &vyvod_znachRef = vyvod_znach;
    cout << " x = 1 and print of Name object X =";
    vyvod_znach.x = 1;
    vyvod_znach.print_date();
    cout << " x = 2 and print Reference X =";
    vyvod_znachRef.x = 2;
    vyvod_znachRef.print_date();
    cout << " x = 3 and print of Pointers X =";
    vyvod_znachPtr->x = 3;
    vyvod_znachPtr->print_date();
    cout << " Press any key" << "\n";
    getch();
        return 0;
}
```

О классах мы поговорим позже после выполнения темы 5.

## 4.3. Инструментарий визуализации ООП – универсальный язык моделирования (Unified Modeling Language, продолжение)

**Диаграмма композитной/составной структуры.***Диаграмма композитной/составной структуры (Composite structure diagram)* — статическая структурная диаграмма, демонстрирует внутреннюю структуру классов и, по возможности, взаимодействие элементов (частей) внутренней структуры класса.

Подвидом диаграмм композитной структуры являются диаграммы кооперации (Collaboration diagram, введены в UML 2.0), которые показывают роли и взаимодействие классов в рамках кооперации. Кооперации удобны при моделировании шаблонов проектирования.

Диаграммы композитной структуры могут использоваться совместно с диаграммами классов.

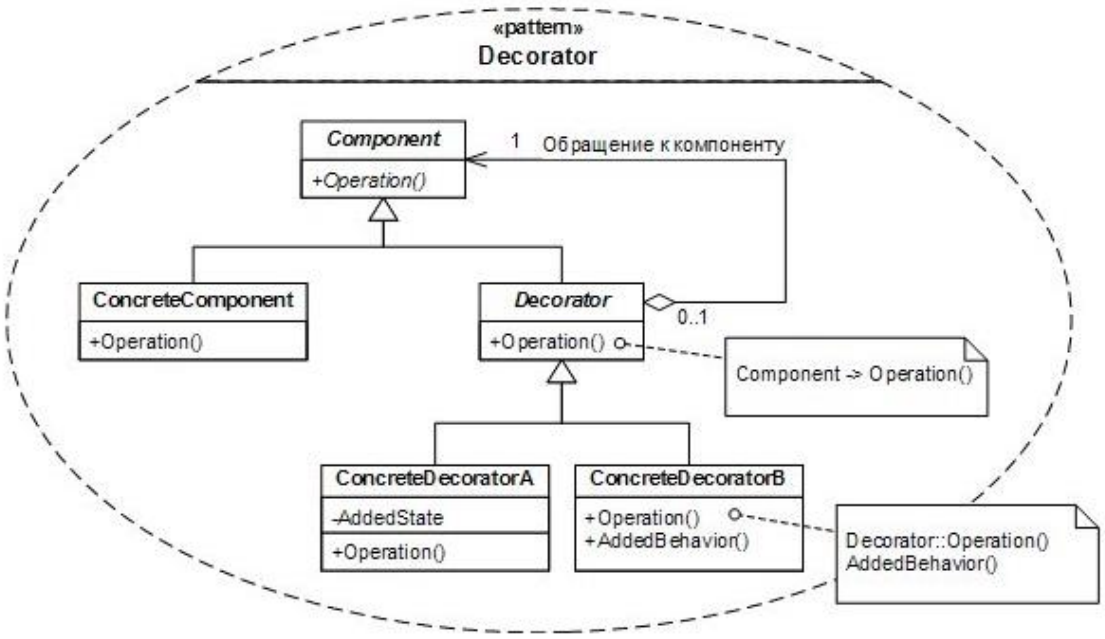



Рис. 4.9. Диаграмма композитной/составной структуры.

**Диаграмма развёртывания.** Диаграмма развёртывания (Deployment diagram, диаграмма размещения) — служит для моделирования работающих узлов и артефактов, развёрнутых на них. В UML 2 на узлах разворачиваются артефакты (англ. artifact), в то время как в UML 1 на узлах разворачивались компоненты. Между артефактом и логическим элементом (компонентом), который он реализует, устанавливается зависимость манифестации.

Итак, перечислим цели, преследуемые при разработке диаграммы развертывания:

- Определить распределение компонентов системы по ее физическим узлам.
- Показать физические связи между всеми узлами реализации системы на этапе ее исполнения.
- Выявить узкие места системы и реконфигурировать ее топологию для достижения требуемой производительности.

Для обеспечения этих требований диаграмма развертывания разрабатывается совместно системными аналитиками, сетевыми инженерами и системотехниками.

Узел (node) представляет собой некоторый физически существующий элемент системы, которая может включать в себя не только вычислительные устройства (процессоры), но и другие механические или электронные устройства, такие как датчики, принтеры, модемы, цифровые камеры, сканеры и манипуляторы. Кроме собственно

изображений узлов на диаграмме развертывания указываются отношения между ними. В качестве отношений выступают физические соединения между узлами и зависимости между узлами и компонентами, изображения которых тоже могут присутствовать на диаграммах развертывания.

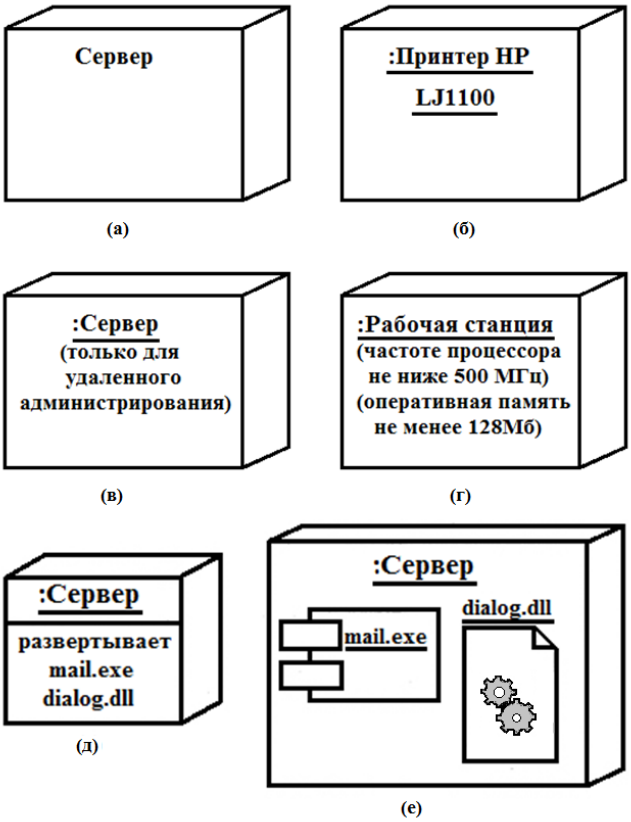



Рис. 4.10. Графические изображения узлов на диаграммах развертывания

Приведем перечень некоторых возможных видов узлов и соединений которые изображены на рис. 4.10 - 4.12:

- графическое изображение узла на диаграмме развертывания на рис. 4.10. (а) и (б);
- графическое изображение узла-экземпляра с дополнительной информацией в форме помеченного значения на рис. 4.10. (в) и (г);
- варианты графического изображения узлов-экземпляров с размещаемыми на них компонентами на рис. 4.10. (д) и (е);

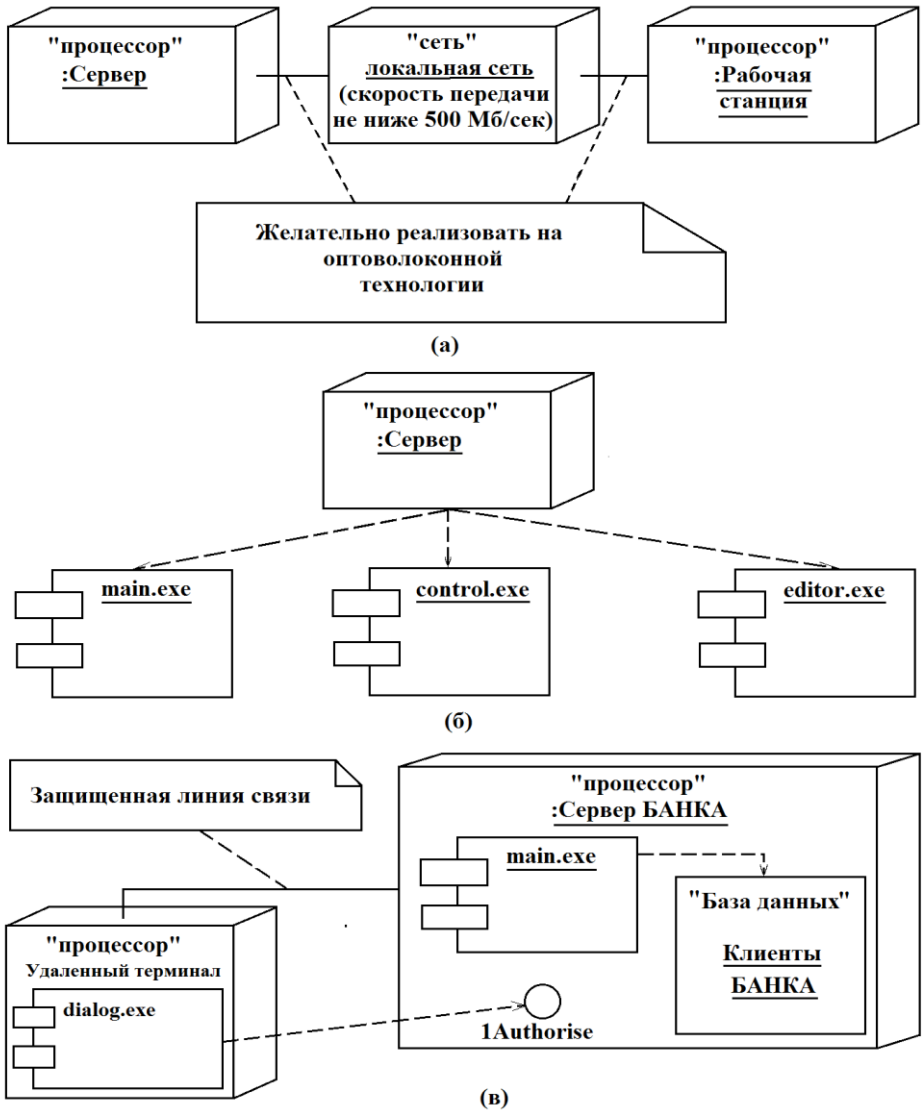



Рис. 4.11. Фрагменты диаграмм развертывания с соединениями между узлами

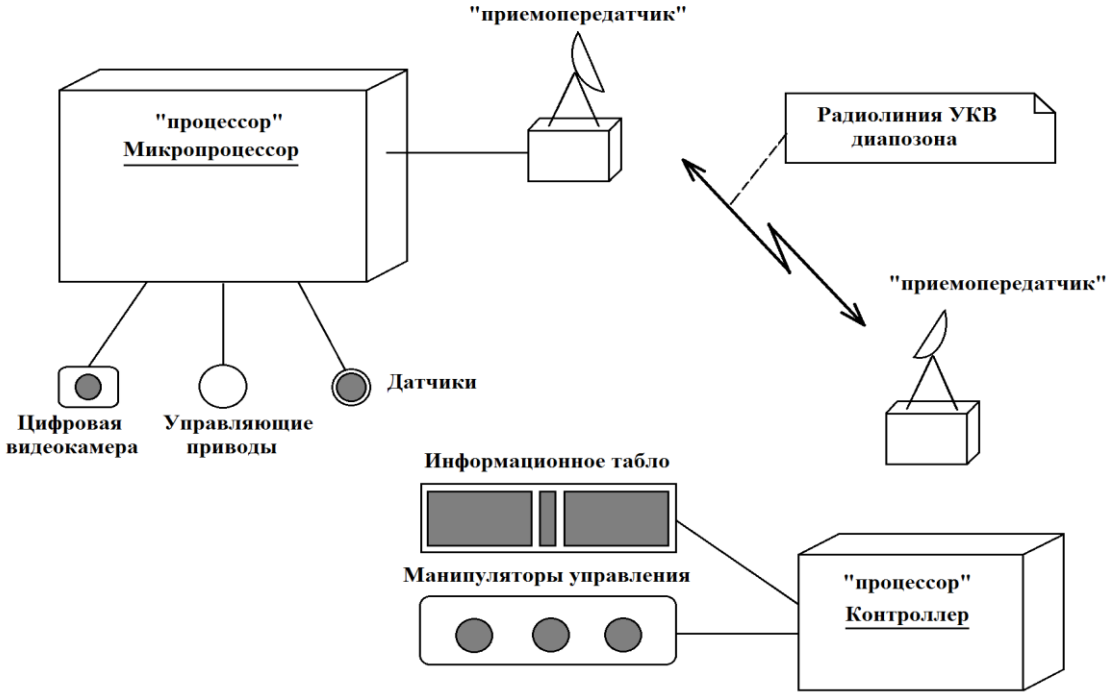


Рис. 4.12. Диаграмма развертывания для модели системы управления транспортной платформой.

- Фрагмент диаграммы развертывания с соединениями между узлами рис.4.11.(а);
- Диаграмма развертывания с отношением зависимости между узлом и развернутыми на нем компонентами рис. 4.11.(б);
- Диаграмма развертывания для системы удаленного обслуживания клиентов банка на рис. 4.11.(в);
- Диаграмма развертывания для модели системы управления транспортной платформой на рис. 4.12.

**Диаграмма объектов.** *Диаграмма объектов (Object diagram)* — демонстрирует полный или частичный снимок моделируемой системы в заданный момент времени см. рис. 4.13. На диаграмме объектов отображаются экземпляры классов (объекты) системы с указанием текущих значений их атрибутов и связей между объектами.

Процесс моделирования диаграммы объектов разработчики UML Г. Буч, Д. Рамбо и А. Джекобсон предлагают выполнять по следующему алгоритму:

1. Идентифицируйте механизм, который собираетесь моделировать. Механизм представляет собой некоторую функцию или поведение части этой системы, которая является результатом взаимодействия сообщества классов, интерфейсов и других сущностей (о том, как механизмы связаны с прецедентами, см. тему 3);
2. Для каждого обнаруженного механизма идентифицируйте классы, интерфейсы и другие элементы, участвующие в кооперации, а также отношения между ними;
3. Рассмотрите один из сценариев использования работы механизма. Заморозьте этот сценарий в некоторый момент времени и изобразите все объекты, участвующие в механизме;
4. Покажите состояние и значения атрибутов каждого такого объекта, если это необходимо для понимания сценария;
5. Покажите также связи между этими объектами, которые представляют экземпляры существующих ассоциаций.

Очень часто программист попадает в ситуацию, когда у него есть программный код, нет UML диаграмм, для полного понимания кода в больших задачах желательно иметь и проектную часть задачи. Тогда перед программистом появляется проблема обратного проектирования объектной диаграммы.

Алгоритм обратного проектирования объектной диаграммы осуществляется, придерживаясь следующей последовательности операций:

1. Выберите, что именно вы хотите реконструировать. Обычно базовой точкой является какая-либо операция или экземпляр конкретного класса;
2. С помощью инструментальных средств или просто пройдясь по сценарию, зафиксируйте выполнение системы в некоторый момент времени;
3. Идентифицируйте все интересующие объекты, сотрудничающие в данном контексте, и изобразите их на диаграмме объектов;
4. Если это необходимо для понимания семантики, покажите состояния объектов;
5. Чтобы обеспечить понимание семантики, идентифицируйте связи, существующие между объектами;
6. Если диаграмма оказалась слишком сложной, упростите ее, убрав объекты, не существенные для прояснения данного сценария. Если диаграмма слишком проста, включите в нее окружение некоторых представляющих интерес объектов и подробнее покажите состояние каждого объекта.

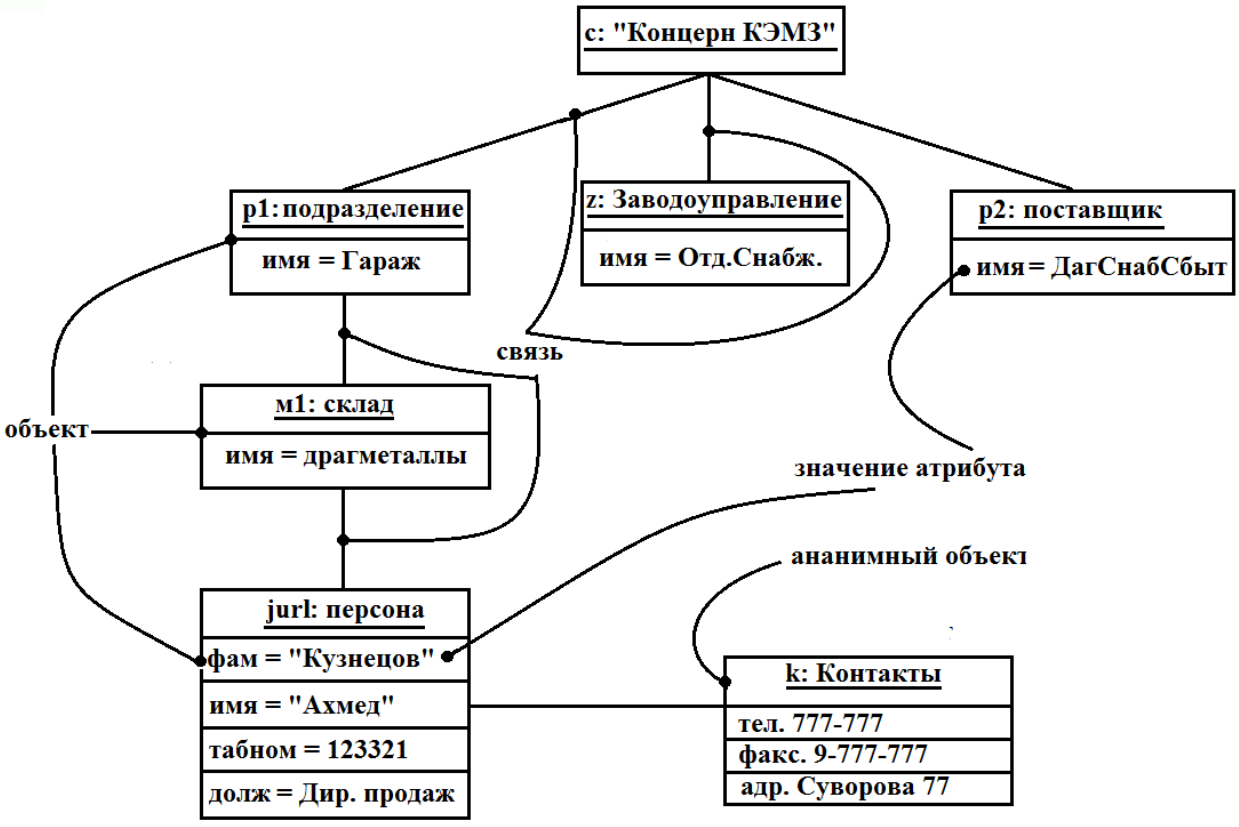



Рис..4.13. Диаграмма объектов

***Рекомендация.*** *Создавая диаграммы объектов на языке UML, помните, что каждая такая диаграмма - это всего лишь графическое представление статического вида системы с точки зрения проектирования или процессов. Это означает, что ни одна отдельно взятая диаграмма объектов не в состоянии передать всю заключенную в этих видах информацию. На самом деле во всех системах, кроме самых тривиальных, существуют сотни, а то и тысячи объектов, большая часть которых анонимна. Полностью специфицировать все объекты системы и все способы, которыми они могут быть ассоциированы, невозможно. Следовательно, диаграммы объектов должны отражать только некоторые конкретные объекты или прототипы, входящие в состав работающей системы.*

Хорошо структурированная диаграмма объектов характеризуется следующими свойствами:

- акцентирует внимание на одном аспекте статического вида системы с точки зрения проектирования или процессов;
- представляет лишь один из кадров динамического сценария, показанного на диаграмме взаимодействия;
- содержит только существенные для понимания данного аспекта элементы;
- уровень ее детализации соответствует; уровню абстракции системы. (Показывайте только те значения атрибутов и дополнения, которые существенны для понимания);
- не настолько лаконична, чтобы ввести читателя в заблуждение относительно важной семантики.

Изображая диаграмму объектов, придерживайтесь следующих правил:

- давайте ей имя, соответствующее назначению;
- располагайте элементы так, чтобы число пересечений было минимальным;
- располагайте элементы так, чтобы семантически близкие сущности оказывались рядом;
- используйте примечания и цвет для привлечения внимания к важным особенностям диаграммы;
- включайте в описания каждого объекта значения, состояния и роли, если это необходимо для понимания ваших намерений.

Для обозначения группы объектов, которым адресован один и тот же сигнал, вводится понятие мультиобъекта, изображаемого графически двумя прямоугольниками, наложенными друг на друга (рис. 4.14).

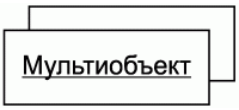



Рис. 4.14. Графическое изображение мультиобъекта.

В отличие от мультиобъекта составной объект действительно состоит из других объектов (рис. 4.15).


Рис. 4.15. Графическое изображение составного объекта.

**Диаграмма пакетов.** *Диаграмма пакетов (Package diagram)* — структурная диаграмма, основным содержанием которой являются пакеты и отношения между ними. Жёсткого разделения между разными структурными диаграммами не проводится, поэтому данное название предлагается исключительно для удобства и не имеет семантического значения (пакеты и диаграммы пакетов могут присутствовать на других структурных диаграммах). Диаграммы пакетов служат, в первую очередь, для организации элементов в группы по какому-либо признаку с целью упрощения структуры и организации работы с моделью системы. Детализируем сказанное.

В языке UML определены пять стандартных стереотипов, применимых к пакетам:

- *facade* (фасад) - определяет пакет, являющийся всего лишь представлением какого-то другого пакета;
- *framework* (каркас) - определяет пакет, состоящий в основном из образцов;
- stub (стаб, заглушка) - определяет пакет, служащий заместителем открытого содержимого другого пакета;
- *subsystem* (подсистема) - определяет пакет, представляющий независимую часть моделируемой системы;
- *system* (система) - определяет пакет, представляющий всю моделируемую систему.
- *import* (импорт) - помечают отношения зависимости между пакетами, один из которых импортирует другой. Его нет в стандартах.

Стереотипы facade и stub предназначены для работы с очень большими моделями. *Стереотип facade* используется для показа

свернутого представления сложного пакета. Предположим, что ваша система содержит пакет BusinessModel. Некоторым пользователям вы захотите показать одно подмножество его элементов (например, только те, что связаны с клиентами), другим - другое (связанное с продуктами). С этой целью определяют фасады, которые импортируют только подмножества элементов другого пакета (но никогда не владеют ими).

*Стереотип stub* используют, если в вашем распоряжении имеются инструменты, расщепляющие систему на пакеты для последующей работы с ними в разное время и в разных местах. Например, если есть две группы разработчиков, работающие в двух разных офисах, то первая из них может создать стабы (Stubs) для пакетов, необходимых другой группе. Такая стратегия позволяет командам работать независимо и не мешать друг другу, причем пакеты-стабы будут описывать стыковочные узлы системы, требующие особенно тщательной проработки.

Если говорить о каркасе (Framework - это, как правило, архитектурный образец, предлагающий расширяемый шаблон для приложений в одной конкретной области), то внешне можно обнаружить типичные механизмы (механизм - это образец проектирования, применимый к сообществу классов), с помощью которых привыкли организовывать свои классы и другие абстракции в иерархии организации программных продуктов. Например, в управляемой событиями системе применение образца проектирования, который можно было бы назвать "цепочка ответственности", - это типичный способ организации обработчиков событий. Теперь поднимитесь чуть выше уровня таких механизмов, и можно увидеть типичные каркасы, образующие архитектуру всей системы. Например, использование трехзвенной архитектуры в информационных системах - это широко распространенный способ четко разделить обязанности между пользовательским интерфейсом, хранением информации и бизнес - объектами и правилами.

Все хорошо структурированные системы изобилуют образцами. Образец (Pattern) предлагает типичное решение типичной проблемы в данном контексте. Образцы используются для специфицирования механизмов и каркасов, образующих архитектуру системы см. рис. 4.16. Его делают доступным, ясно идентифицируя те элементы управления и стыковки, с помощью которых пользователь настраивает образец для применения в имеющемся контексте.

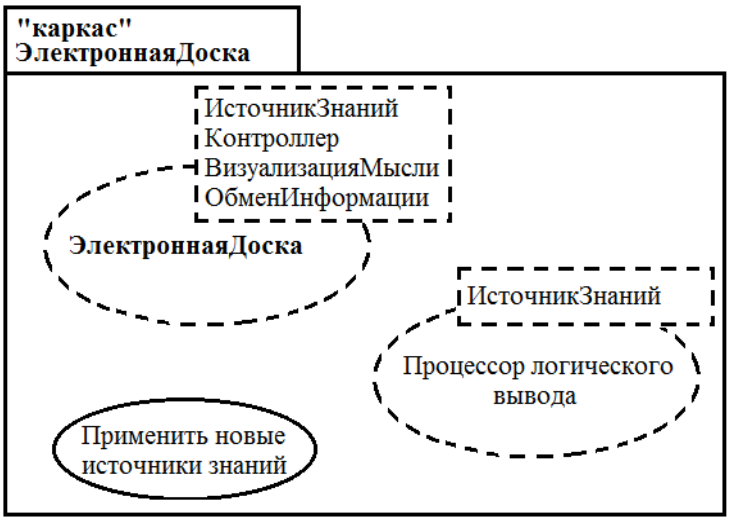



Рис.4.16. Моделирование архитектуры образца.

***Примечание:*** *Каркасы отличаются от обычных библиотек классов. Библиотека классов содержит абстракции, конкретизируемые или вызываемые абстракциями программы, однако не содержит абстракции, которые вызывают или конкретизируют эти абстракции. Оба вида соединений образуют те элементы управления и стыковки, с помощью которых каркас настраивается на данный контекст.*

Модели образцов в UML обычно связаны с процессом функционирования программной продукции, и они работают на многих уровнях абстракции, начиная от отдельных классов и кончая системой в целом. Самые интересные виды образцов - это механизмы и каркасы. Хорошо структурированный образец обладает следующими свойствами:

- решает типичную проблему типичным способом;
- включает структурную и поведенческую составляющие;
- раскрывает элементы управления и стыковки, с помощью которых его можно настроить на разные контексты;
- является атомарным, то есть не разбивается на меньшие образцы;
- охватывает разные индивидуальные абстракции в системе.

Изображая образец в UML, руководствуйтесь следующими правилами:

- раскрывайте те его элементы, которые необходимо адаптировать для применения в конкретном контексте;
- делайте образец доступным, приводя прецеденты его использования, а также способы настройки.

*Подсистема* (Subsystem) - это объединение элементов, ряд которых составляет спецификацию поведения, предложенного другими ее

элементами. Подсистема изображаются в виде пиктограммы стереотипного пакета. *Модель* (Model) - это упрощение реальности, абстракция, создаваемая для лучшего восприятия системы. *Вид*, или *представление* (View), - это модель, рассматриваемая под определенным углом зрения: в ней отражены одни сущности и опущены другие, которые с данной точки зрения не представляют интереса.

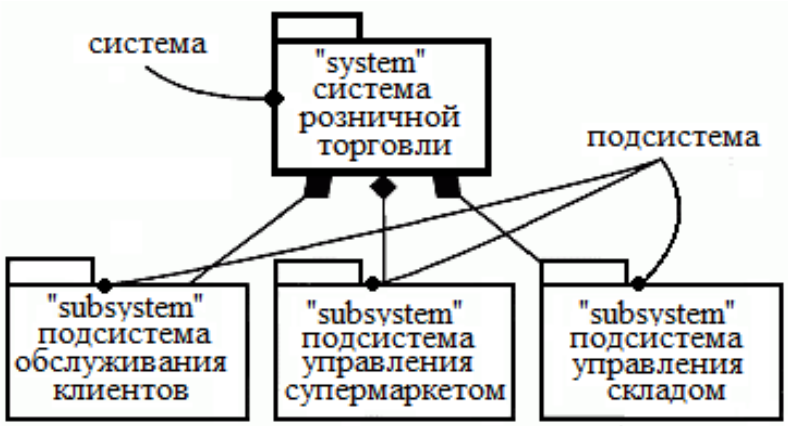



Рис. 4.17. Системы и подсистемы.

Подсистемы - это образования, которые используются для декомпозиции сложной системы на более простые, слабо зависящие друг от друга составляющие. То, что на одном уровне абстракции выглядит системой, на другом - более высоком - может рассматриваться как подсистема.

В UML подсистема изображается в виде пиктограммы стереотипного пакета (см. рис. 4.17). Семантически подсистема - это одновременно и разновидность пакета, и разновидность классификатора.

*Система (system)*, зачастую разложенная на ряд подсистем см. рис.4.15., - это множество элементов, организованных некоторым образом для выполнения определенной цели. Система описывается набором моделей, по возможности рассматривающих ее с различных точек зрения. Важными составными частями модели являются такие сущности, как классы, интерфейсы, компоненты и узлы. В UML модели применяются для организации подобных абстракций системы. По мере возрастания сложности обнаруживается, что сущность, на одном уровне абстракции представлявшаяся как система, на другом - более высоком - оказывается лишь подсистемой. В UML можно моделировать системы и подсистемы как единое целое и тем самым органично решать проблему масштабирования.

*Импорт и экспорт.* Предположим, у вас есть два класса одного уровня A и B, расположенных рядом друг с другом. A может "видеть" B и наоборот, то есть любой из них может зависеть от другого. Два таких класса составляют стандартную систему, и для них не надо задавать никаких пакетов.

Допустим теперь, что у вас имеется несколько сотен равноправных классов. Размер "паутины" отношений, которую вы можете соткать между ними, не поддается воображению. Более того, столь огромную неорганизованную группу классов просто невозможно воспринять в ее целостности. Описанная ситуация порождает реальную проблему больших систем: простой неограниченный доступ не позволяет осуществить масштабирование. В таких случаях для организации абстракций приходится применять пакеты.

Итак, допустим, что класс A расположен в одном пакете, а класс B - в другом, причем оба пакета равноправны. Допустим также, что как A, так и B объявлены открытыми частями в своих пакетах. Эта ситуация коренным образом отличается от двух предыдущих. Хотя оба класса объявлены открытыми, свободный доступ одного из них к другому невозможен, ибо границы пакетов непрозрачны. Однако если пакет, содержащий класс A, импортирует пакет-владелец класса B, то A сможет "видеть" B, хотя B по-прежнему не будет "видеть" A. Импорт дает элементам одного пакета односторонний доступ к элементам другого. На языке UML отношения импорта моделируют как зависимости, дополненные стереотипом import. Упаковывая абстракции в семантически осмысленные блоки и контролируя доступ к ним с помощью импорта, вы можете управлять сложностью систем, насчитывающих множество абстракций.

***Примечание*: *Фактически в рассмотренной ситуации применяются два стереотипа - import и access, - и оба они показывают, что исходный пакет имеет доступ к элементам целевого. Стереотип import добавляет содержимое целевого пакета в пространство имен исходного. Таким образом, возникает вероятность конфликта имен, которой необходимо избегать, если вы хотите, чтобы модель была хорошо оформлена. Стереотип access не изменяет пространство имен цели, поэтому содержащиеся в исходном пакете имена необходимо квалифицировать. Чаще используют стереотип import.***

Открытые элементы пакета называются экспортируемыми. Так, на рис.4.18. пакет GUI экспортирует два класса - Windows и Form. Класс

EventHandler является защищенной частью пакета. Довольно часто экспортируемыми элементами пакета оказываются интерфейсы.

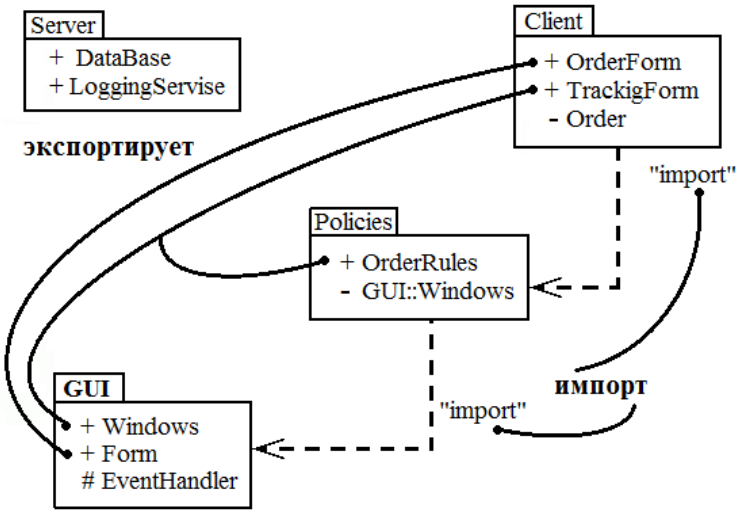



Рис. 4.18. Импорт и экспорт

Экспортируемые элементы будут видны только тем пакетам, которые явно импортируют данный. В примере на рис. 4.18. пакет Policies импортирует GUI. Таким образом, классы GUI:: Window и GUI:: Form будут из него видны. Но класс GUI:: EventHandler виден не будет, так как является защищенным. С другой стороны, пакет Server не импортирует GUI, и потому элементы Server не имеют права доступа к элементам GUI. По той же причине у GUI нет доступа к содержимому пакета Server.

Зависимости импорта и доступа не являются транзитивными. В данном примере пакет Client импортирует Policies, а Policies - GUI, однако это не значит, что Client импортирует GUI. Таким образом, доступ из пакета Client к экспортируемым частям пакета Policies разрешен, а к экспортируемым частям пакета GUI - нет. Чтобы получить такой доступ, надо явно указать, что Client импортирует GUI.

***Примечание*:** *Элемент, видимый внутри пакета, будет видим также и внутри всех вложенных в него пакетов. Вообще, вложенные пакеты могут "видеть"- все, что "видит" объемлющий их пакет.*

**Диаграмма автомата (State Machine diagram)** — диаграмма, на которой представлен конечный автомат с простыми состояниями, переходами и композитными состояниями.

*Автомат* (State machine) определяет, используя различные датчики и детекторы поведение последовательности состояний, через которые проходит объект в течение своего жизненного цикла, отвечая на различные события, а также его реакции на эти события. *Состояние*

(State) - это ситуация, которая возникает при функционировании в жизни объекта, на протяжении которой он удовлетворяет некоторому условию, выполняя определенную деятельность, или ожидая некоторое событие.

*Событие* (Event) - это спецификация существенного факта, как стимула осуществляющего переход из одного состояния в другое, который детектирует автомат, используя датчики или детекторы, имеющего место в пространстве и во времени. *Переход* (Transition) -это отношение между двумя состояниями, отображающее, что объект, находящийся в первом целевом состоянии, должен выполнить определенные действия и перейти во второе целевое состояние, при возникновении ожидаемого события и удовлетворении всех детектируемых условий.

*Деятельность* (Activity) - это алгоритм или вычислительный процесс внутри автомата в жизненном цикле, который выполняет определенные действия.

*Действие* (Action) - это атомарное вычисление, которое приводит к изменению состояния модели или возврату значения. Состояние изображается в виде прямоугольника с закругленными углами. Переход обозначается линией со стрелкой.

Теория автоматов охватывает функционирования всех видов вычислительной техники (аналоговых и цифровых), рассмотрим ее безотносительно конкретного вида машин и приведем для выделенных курсивом понятий алгоритмы, советы и другие виды указаний.

***Примечание. Алгоритм*** *может быть реализован, как программно, используя инструментарии (языки программирования) для цифровой, так и аппаратно для аналоговой и нейрокомпьютерной вычислительной техники.*

***Состояние*** определяют следующие элементы:

- ***имя*** - уникальный идентификатор, которое отличает одно состояние от всех остальных внутри пакета, целевая критическая точка на графике, состояние хромосомы (или популяции) в генетическом алгоритме или расхождение корреляционного состояния в двух режимах при обучении Больцмана на нейрокомпьютерах и т.д.;
- ***действия при входе/выходе*** - действия, выполняемые при достижении цели и входе в состояние и выходе из него;
- ***внутренние переходы*** – логические переходы внутри программного или подпрограммного кода проекта, обрабатываемые без выхода из состояния;

- ***подсостояния*** – в алгоритме программного или подпрограммного кода проекта внутри состояния могут существовать подсостояния, как непересекающиеся (активизируемые последовательно), так и параллельные (активные одновременно);
- ***отложенные события*** - список событий, которые не обработаны в этом состоянии, а отложены и поставлены в очередь для обработки объектом в некотором другом состоянии.

**Примечание:** *Первую букву каждого слова в имени состояния принято делать заглавной, например* ***Ожидание*** *или* ***Идет Отключение***.

***Переход*** определяют пять элементов:

- ***исходное состояние*** - состояние, из которого происходит переход. Если объект находится в исходном состоянии, то исходящий переход может сработать, когда объект получит событие-триггер, инициирующее этот переход, причем должно быть выполнено сторожевое условие (если оно задано);
- ***событие-триггер*** - событие, при получении которого объектом, находящимся в исходном состоянии, может сработать переход (при этом должно быть выполнено сторожевое условие);
- ***сторожевое условие*** - булевское выражение, которое вычисляется при получении события-триггера. Если значение истинно, то переходу разрешено сработать, если ложно - переход не срабатывает. Если при этом не задано никакого другого перехода, инициируемого тем же самым событием, то событие теряется;
- ***действие*** - атомарное вычисление, которое может непосредственно воздействовать на объект, владеющий автоматом, или оказать косвенное воздействие на другие объекты, находящиеся в области видимости;
- ***целевое состояние*** - состояние, которое становится активным после завершения перехода.

Сейчас приведем пример работы автомата кондиционера см. рис. 4.19.

**Примечание:** *У перехода может быть несколько исходных состояний (в этом случае он представляет собой слияние нескольких параллельных состояний), а также несколько целевых состояний (в этом случае он представляет собой разделение на несколько параллельных состояний).*

**Диаграмма вариантов использования.** *Диаграмма вариантов использования (Use case diagram, диаграмма прецедентов)* — диаграмма, на которой отражены отношения, существующие между актёрами и вариантами использования.

Основная задача — представлять собой единое средство, дающее возможность заказчику, конечному пользователю и разработчику совместно обсуждать функциональность и поведение системы.

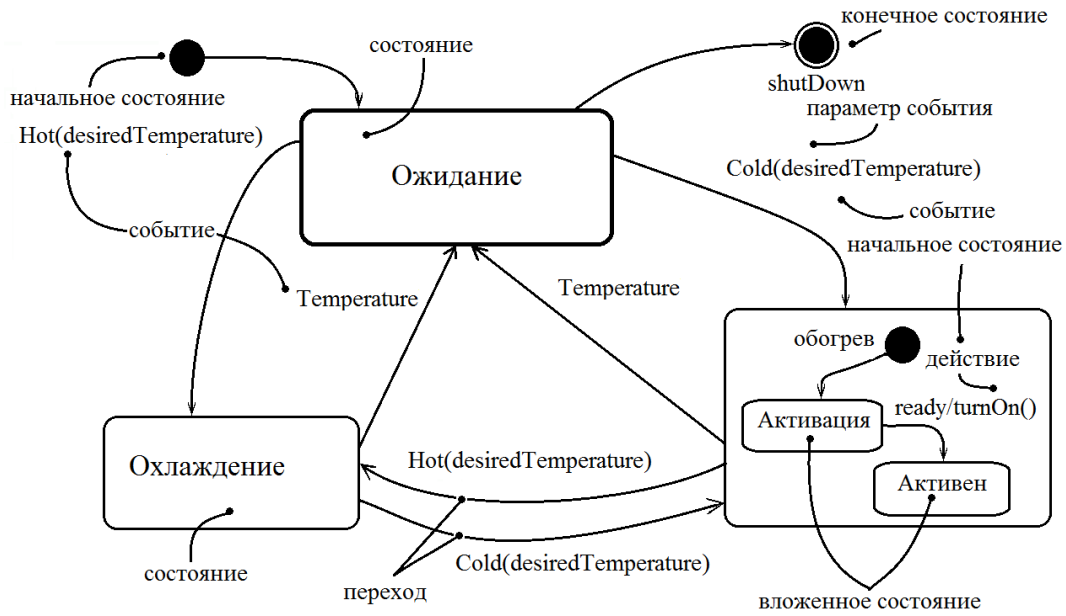



Рис. 4.19. Автомат кондиционера.

При выполнение работы №2 мы рассмотрели этапы проектирования жизненного цикла проекта. Разработка диаграммы вариантов использования следует отнести к первому этапу, т.е. как некоторый процесс поуровневого спуска от наиболее обшей и абстрактной концептуальной модели исходной системы к логической, а затем и к физической модели соответствующей программной системы. Для достижения этих целей, если пользоваться диаграммами UML (напоминаем можно использовать и другие виды графического изложения проекта) вначале строится модель в форме, так называемой диаграммы вариантов использования (use case diagram), которая описывает функциональное назначение системы или, другими словами, то, что система будет делать в процессе своего функционирования. Диаграмма вариантов использования является исходным концептуальным представлением или концептуальной моделью системы в процессе ее проектирования и разработки.

Разработка диаграммы вариантов использования преследует цели:

- Определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы;
- Сформулировать общие требования к функциональному поведению проектируемой системы;

• Разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей;

• Подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

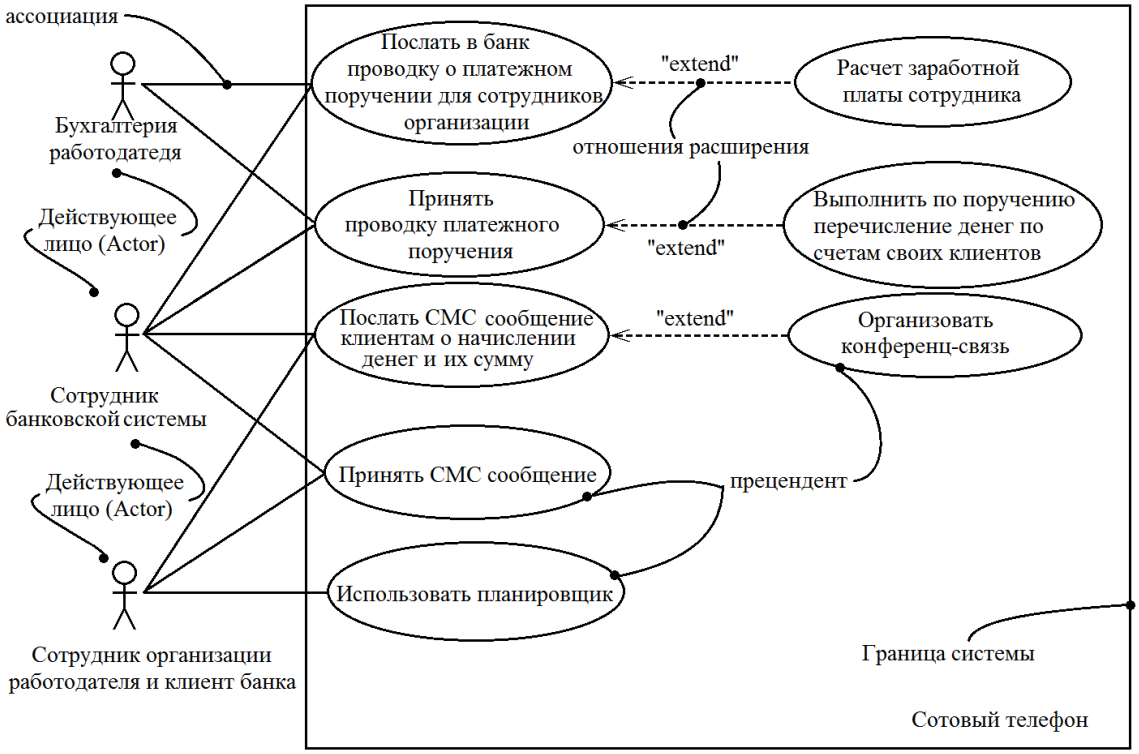



Рис. 4.20. Диаграмма вариантов UML, начисленя зарплаты через банк.

Приведем для примера, часть задачи, используя диаграмму вариантов UML для начисления зарплаты бухгалтерией работодателя через банковскую систему, т.е. начисление зарплаты на расчетный счет клиента в банке, и оповещение банком, клиента СМС сообщением, через сотовую связь см. рис. 4.20. Суть данной диаграммы состоит в следующем: проектируемая система представляется в виде множества сущностей или актеров, взаимодействующих с системой с помощью, так называемых вариантов использования. При этом актером (actor) или действующим лицом называется любая сущность, взаимодействующая с системой извне.

Это может быть человек, техническое устройство, программа или любая другая система, которая может служить источником воздействия на моделируемую систему так, как определит сам разработчик. В свою очередь, вариант использования (use case) служит для описания сервисов, которые система предоставляет актеру. Другими словами, каждый

вариант использования определяет некоторый набор действий, совершаемый системой при диалоге с актером. При этом ничего не говорится о том, каким образом будет реализовано взаимодействие актеров с системой.

Остальные диаграммы UML мы рассмотрим позже после изучения консольных приложений.

### 4.4. Теория и практика для решения задач с циклами

Оператор цикла используется, обычно, если необходимо повторить некоторое число одних и тех же операторов с различными промежуточными значениями. Если число повторений заранее известно, то подходящей конструкций является оператор for. В противном случае следует использовать конструкции операторов while или do/while. Для управления в блоке последовательностью операторов внутри цикла можно использовать операторы **`break`** и **`continue`**. Оператор break завершает оператор цикла, а continue продолжает последовательность операторов внутри блока в операторе цикла со следующей итерации см. рис 4.21.

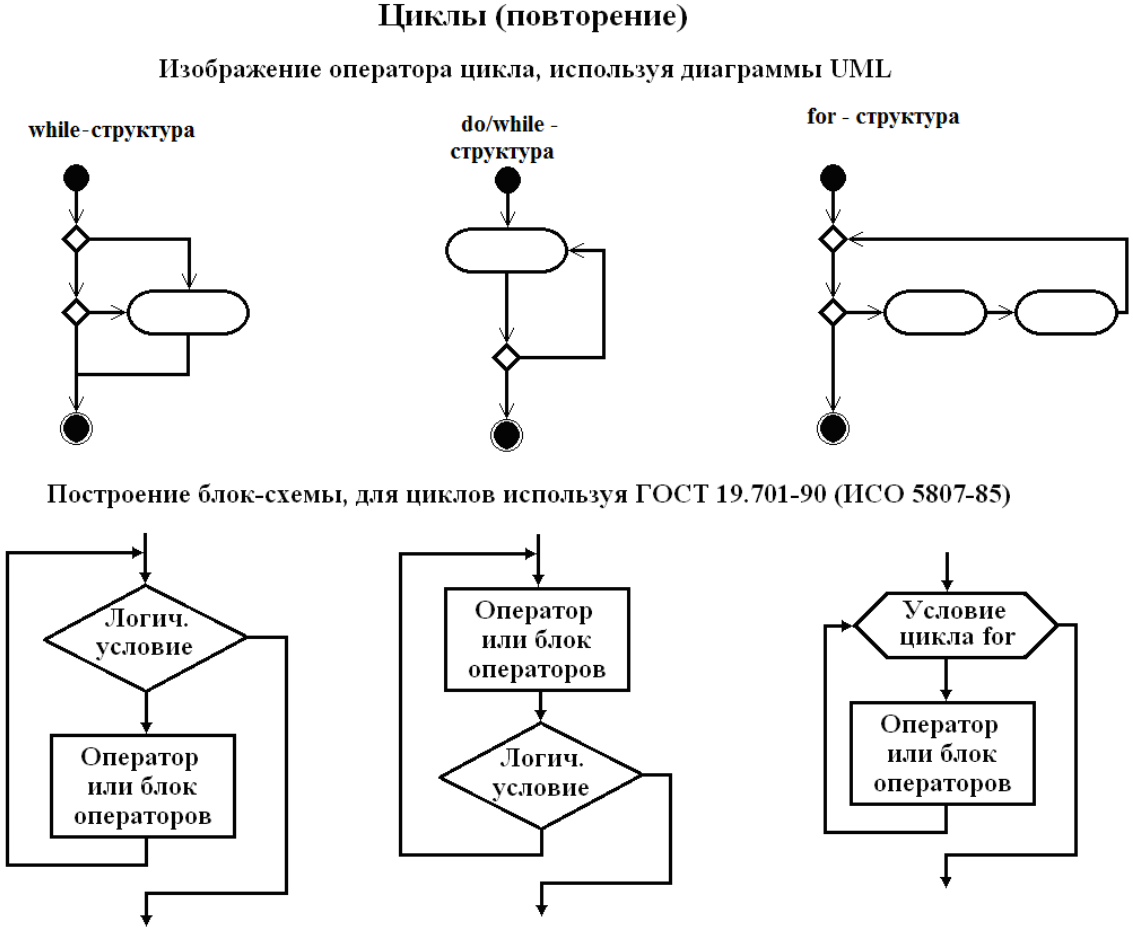



Рис. 4.21. Блок – схемы и UML диаграммы для циклов.

Напоминаем при изучении темы 2 и 3 мы рассматривали элементы для построения блок схемы и диаграммы (диаграмма деятельности) для циклов. Приведем их в более наглядной и компактной форме теперь.

### 4.4.1. Операторы циклов

**Оператор цикла с предусловием – *while.*** Формат оператора while имеет вид ***while*** (**выражение**) ***оператор***

Выполнение подоператора повторяется, пока значение выражения остается ненулевым. Проверка выполняется перед каждым выполнением оператора.

В следующем примере ввод с клавиатуры происходит до тех пор, пока пользователь не введет символ A:

```
char wait_for_char(void)
{
  char ch;
  ch = '\0';  /* инициализация ch */
  while(ch != 'A') ch = getchar();
  return ch;
}
```

Переменная `ch` является локальной, ее значение при входе в функцию произвольно, поэтому сначала значение `ch` инициализируется нулем. Условие цикла `while` истинно, если `ch` не равно `A`. Поскольку `ch` инициализировано нулем, условие истинно и цикл начинает выполняться. Условие проверяется при каждом нажатии клавиши пользователем. При вводе символа `A` условие становится ложным и выполнение цикла прекращается.

**Оператор цикла с постусловием – *do.*** Формат оператор имеет вид ***do*** *код цикла* ***while*** **(***выражение***);**

Выполнение подоператора повторяется до тех пор, пока значение выражения не станет нулем. Проверка выполняется после каждого выполнения оператора, т.е. организует повторение *кода цикла* до тех пор, пока выполнится *выражение* истинно, после чего управление передается следующему за циклом оператору. Данный оператор в отличие от оператора **while** гарантирует выполнение *кода цикла* хотя бы один раз.

В следующем примере в цикле `do-while` числа считываются с клавиатуры, пока не встретится число, меньшее или равное 100:

```
do {
  scanf("%d", &num);
} while(num > 100);
```

**Оператор цикла – `for`.** Оператор цикла `for`, имеет большую гибкость относительно оператора for в других языках программирования, например, Pascal, Modula, PL 1 и др.

Формат оператора имеет вид

```
for (выражение 1; выражение 2; выражение 3)
    тело цикла,
```

где ***выражение* 1** – начальное значение параметра цикла; ***выражение* 2** – проверка условия на продолжение цикла; ***выражение* 3** – изменение (коррекция) параметра цикла.

Схема работы оператора цикла ***for***:

1. вычисляется ***выражение* 1**;
2. вычисляется ***выражение* 2**;
3. Если значения ***выражения* 2** равно нулю (т.е. значение выражения ложно) тогда **конец цикла** *и управление передается на оператор, следующий за оператором* ***for***;
4. выполняется тело цикла, вычисляется ***выражение 3***
5. переход к пункту 2.

***Примечание!*** *Сначала выполняют выражение 1***.** *Проверка условия всегда выполняется в начале цикла, следовательно, тело цикла может ни разу не выполниться, если условие выполнения сразу будет ложным.*

Учитывая примечание, можем заявить, что этот оператор эквивалентен следующим форматам:

```
выражение_1;
while     (выражение_2)
                {
                 оператор
                    выражение_3;
                 }
```

```
 //или for (инициализация; условие; приращение)
оператор;
```

Цикл ***for*** допускает и следующий запись **`for (;;)`** – бесконечный цикл. Завершение такого цикла возможно, используя, оператор ***break*.**

Пример:

```
for (;;)
{ ...
  ...  break;
  ...
}
```

Бесконечность цикла вытекает из следующих соображений, учитывая, синтаксис языка С, допускает, что оператор может быть

пустым, тело оператора **for** также может быть пустым. Такая форма оператора может быть использована для организации поисковых операций.

Некоторые варианты использования оператора **for** повышают его гибкость за счет возможности использования нескольких переменных, управляющих циклом. Один из распространенных способов усиления мощности цикла **for** — применение оператора "запятая" для создания двух параметров цикла.

Пример:

```
char str[100],  ch;
for (int t=0, b=100 ; t < b ; t++, b--)
    {  ch= str[t];
       str[b]=ch;
    }
```

В этом примере, реализующем запись строки символов в обратном порядке, для управления циклом используются две переменные t и b, которые видны внутри цикла, в отличие от str и ch. Отметим, что на месте выражение 1 и выражение 3 здесь используются несколько выражений, записанных через запятую, и выполняемых последовательно.

В следующем примере цикл **for** удаляет начальные пробелы в строке str:

```
for( ; *str == ' '; str++) ;
```

В этом примере указатель str переставляется на первый символ, не являющийся пробелом. Цикл не имеет тела, так как в нем нет необходимости.

**Операторы break и continue.** Часто возникают ситуации, когда необходимо управлять выходом из цикла до проверки условия вначале или в конце. Оператор **break** позволяет выйти из операторов **for, while** и **do** до окончания цикла, вспомните, аналогичную операцию выполняли для оператора переключателя **switch**. Оператор **break** приводит к немедленному выходу из самого внутреннего охватывающего его цикла (или переключателя).

Оператор **continue** родственен оператору **break**, но используется реже; он приводит к началу следующей итерации охватывающего цикла (**for,while** и **do**). В циклах **for,while** и **do** это означает непосредственный переход к выполнению проверочной части; в цикле **for** управление передается на шаг реинициализации.

Оператор **continue** применяется только в циклах, но не в переключателях, однако ее можно использовать и в переключателе, т.е.

внутри переключателя внутри цикла, который вызывает выполнение следующей итерации цикла.

**Пример выполнения задания.** Написать и отладить программу вывода всех значений функции $S(x)$ для аргумента $x$, изменяющегося в интервале от $a$ до $b$ с шагом $h$ и заданном $n$.

$$S(x) = \sum_{k=0}^{N} (-1)^k \frac{x^k}{k!}.$$

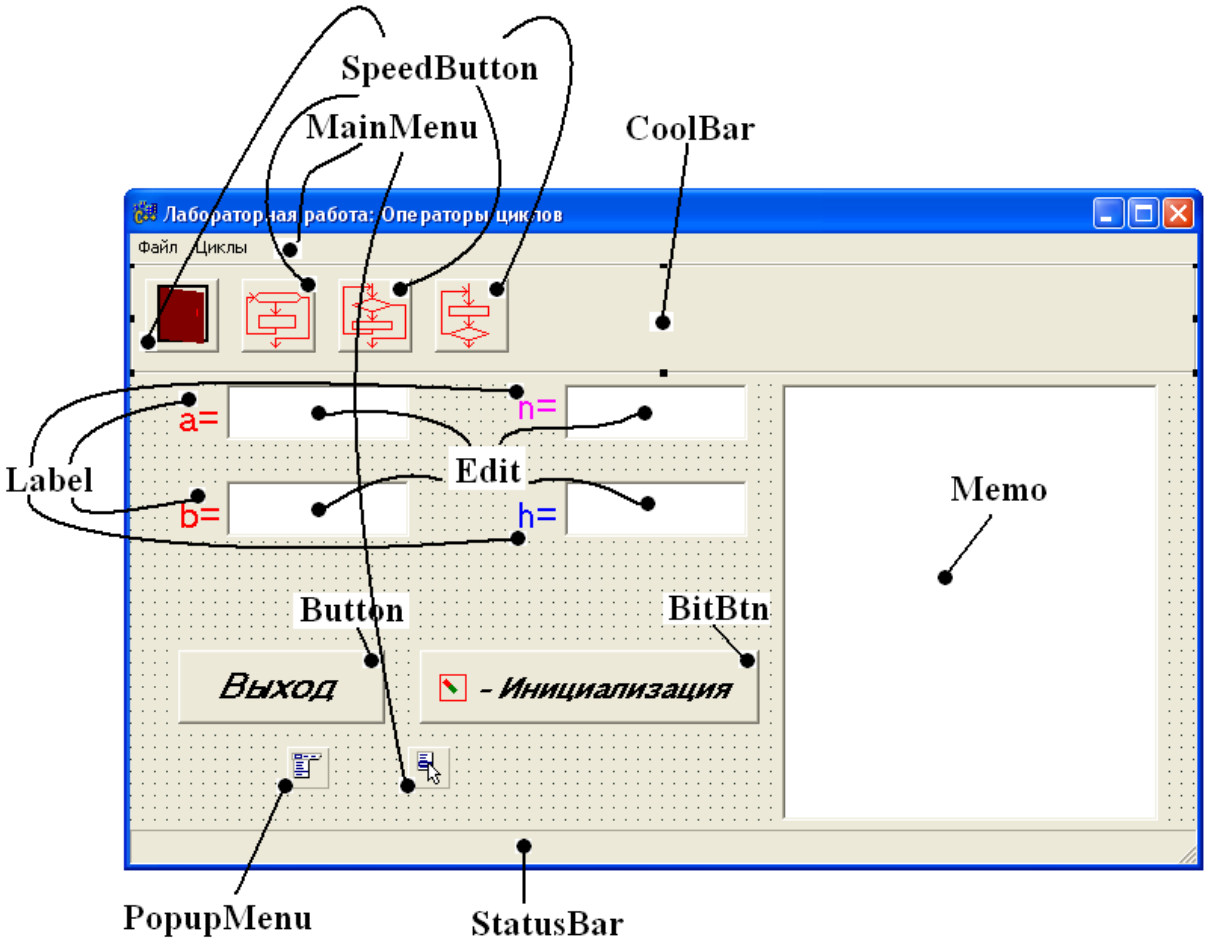



Рис.4.22. Эскизный проект окна проекта Лабораторная работа: «Операторы циклов».

Построить проект, интерфейс которого состоит из меню, контекстного меню, строки состояния, где отображает наименование оператора цикла и из панели инструментов с ответствующих пиктограмм. Эскизный проект интерфейса представлен на рис. 4.22.

В эскизном проекте отображено информация об используемых компонентах.

## 4.5. Невидимые компоненты MainMenu и PopupMenu

Компоненты на языке C++ Builder подразделяются на 2 группы: видимые (визуальные) и невидимые (невизуальные). Визуальные

компоненты появляются во время выполнения точно так же, как и во время проектирования. Невизуальные компоненты появляются во время проектирования как пиктограммы на форме. Они никогда не видны во время выполнения, но обладают определенной функциональностью (например, обеспечивают доступ к данным, вызывают стандартные диалоги Windows и др.) Компоненты **MainMenu** и **PopupMenu** являются невидимыми, сравните рисунки 4.22., где видно невидимые компоненты. С рисунком 4.23. во время процесса работы, где компоненты не видны.

Поместите на форму компонент MainMenu и PopupMenu из вкладки Standard.

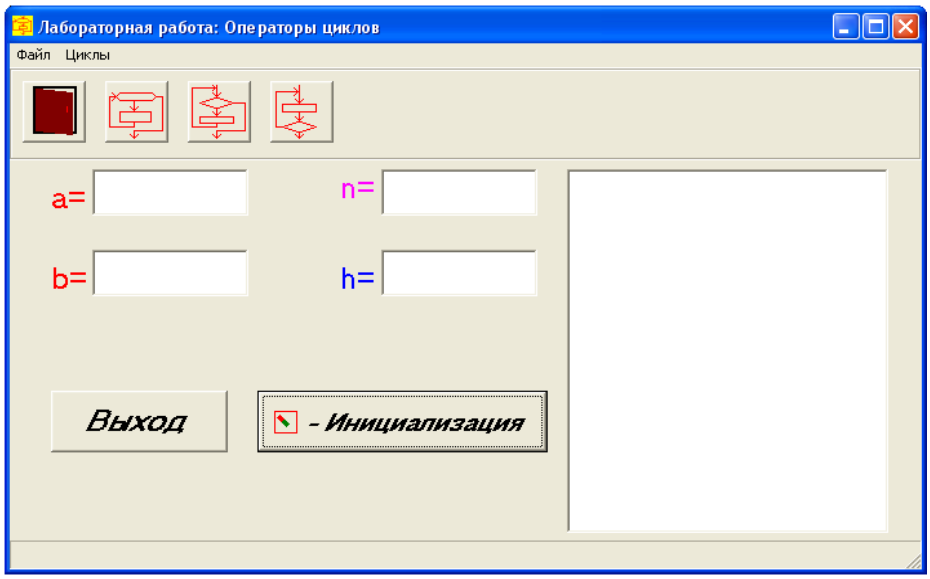



Рис. 4.23. Вид эскизного проекта Лабораторная работа: Операторы циклов после старта проекта на выполнение.

Двойным щелчком щелкните по компоненте MainMenu1, и вы увидите картину, приведенную на рис. 4.24. Окно для создания меню.

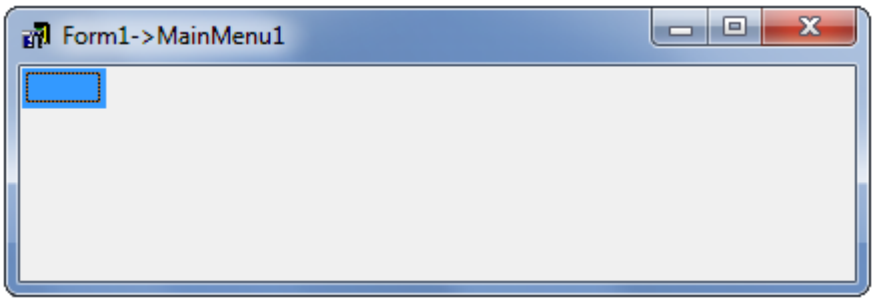



Рис. 4.24. Диалоговое окно MainMenu.

Наберите слово Файл в Caption свойствах Object Inspector. Переместите стрелку вниз и наберите слово Выход, а затем стрелку

вправо Циклы и соответственно Oper_for, Oper_while и Oper_do как показано на рис. 4.25.

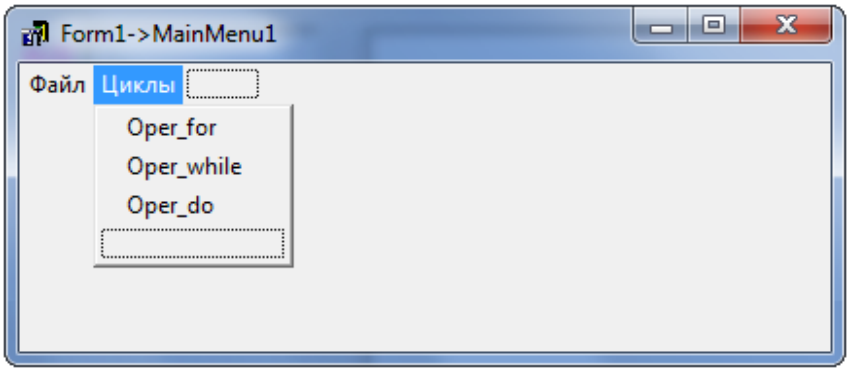



Рис.4.25. Создание меню проекта.

Затем закройте окно MainMenu1. Как видим, в окне появилось меню.

Теперь приступим к построению контекстного меню, появляющегося при нажатии правой кнопки мыши. Двойным щелчком щелкните по компоненте PopupMenu1. После щелка появляется диалоговое окно, изображенное на рис. 4.26.

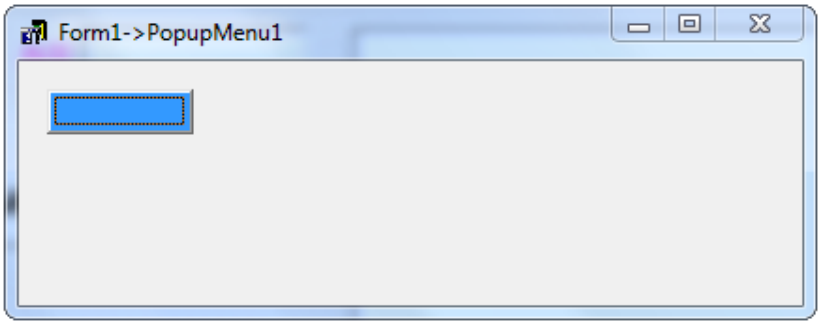



Рис.4.26. Диалоговое окно PopupMenu1.

В пункте Caption свойствах Object Inspector наберите слово Выход, а затем щелкнете по синей рамке в диалоговом окне PopupMenu1, как вы заметили, в рамке отобразился запись из Caption. Аналогично запишите слова Oper_for, Oper_while и Oper_do как показано на рис. 4.27.

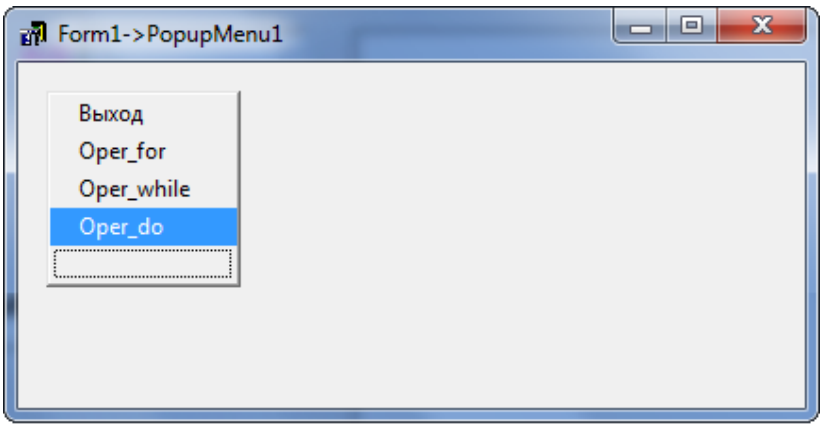


Рис.4.27. Создание контекстного меню для проекта.

Теперь выходите из диалогового окна PopupMenu1.

Сначала создадим макет функций необходимых для функционирования проекта.

Создаем ***FormCreate***, напоминаем, для этого надо сделать двойной щелчок на любом свободном месте формы.

Для создания макета функций подпунктам: Выход; Oper_for; Oper_while и Oper_do необходимо их выделять на окне Object TreeView и сделать двойной щелчок. После чего мы в редакторе кода увидим следующие функции.

```
//---------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "Cirkle.h"
//---------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
}
//---------------------------------------------------------
void __fastcall TForm1::N2Click(TObject *Sender)
{
}
//---------------------------------------------------------
void __fastcall TForm1::for1Click(TObject *Sender)
{

}
//---------------------------------------------------------
void __fastcall TForm1::OperWhile1Click(TObject *Sender)
{
```

```
}
//---------------------------------------------------------------
void __fastcall TForm1::Operdo1Click(TObject *Sender)
{

}
//---------------------------------------------------------------
```

**Внимание!** В контекстном меню имеются подпункты, совпадающие по выполняемым действиям, следовательно, мы их можем привязать к одним и тем же функциям.

Выделим пункт «Выход» в окне Object TreeView и в окне Object Inspector в событиях (Events) мы видим активизировано событие щелчка, и при этом вызывается функция N2Click из макета программного кода приведенного выше см. рис.4.28.

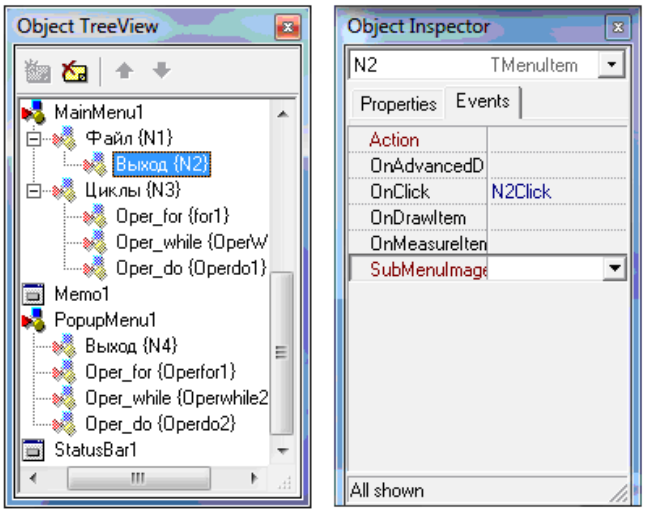



Рис. 4.28. Взаимосвязь функции N2Click с подпунктом меню Выход.

Если возникает событие OnClick (щелчок) по подпункту Выход, тостартует функция N2Click.

Выделим подпункт «Выход» из PopupMenu1 в окне Object TreeView, и в окне Object Inspector в событиях (Events) мы в пункте OnClick выберем N2Clic см. рис. 4.29. Таким образом, при выборе в контекстном меню пункта «Выход», будут выполняться те же действия, что в обычном меню. Так как они выполняют одну и ту же функцию.

Аналогичные операции выполняйте самостоятельно и для пунктов Oper_for; Oper_While и Oper_do.

Теперь согласованно работают меню и контекстное меню. А программный код для функционирования подпунктов меню смотрите ниже.

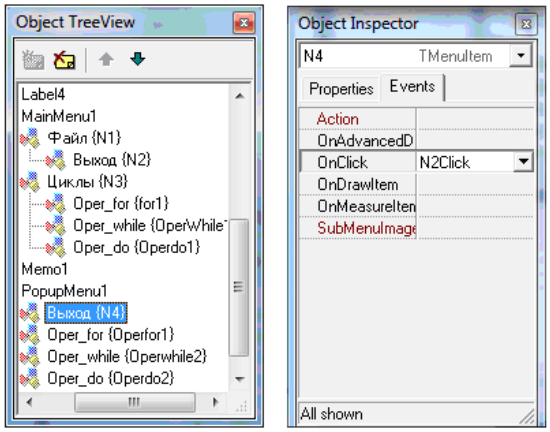



Рис. 4.29. Привязка подпункта «Выход» контекстного меню к функции N2Click в событиях (Events) окна Object Inspector к пункту OnClick.

**Панель инструментов и строка состояния.** Все окна проектов, используемые, в современных операционных системах Windows, обычно сопровождаются неотъемлемыми атрибутами «Панелью инструментов» и «Строкой состояния». Панель инструментов служит для формирования в программе органов управления, которая отображает наиболее часто выполняемые команды в виде пиктограмм. Строка состояния отображает состояние активных объектов окна.

Перетащите и установите компоненты CoolBar для «Панели управления» под меню и StatusBar для «Строки состояния» у нижней границы окна Form с панели Win32, как показано на эскизном проекте, и остальные компоненты.

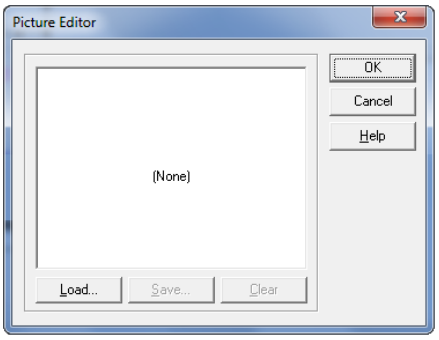
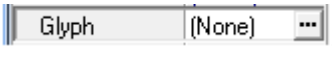



Рис. 4.30. Диалоговое окно для привязки графического файла к кнопкам BitBtn и SpeedButton.

Рисунки на кнопках BitBtn и SpeedButton создаются графическим редактором Image Editor, с принципом работы вы уже знакомы по первой теме. Только здесь нужно создать графический файл с расширением .bmp.

После сохранения иконы необходимо выделить кнопку и в окне Object Inspector выбрать свойство Glyph 

. Сделав

щелчок на кнопке с тремя точками, мы получаем диалоговое окно изображенное на рисунке 4.30.

Затем в диалоговом окне выберите нужный файл и нажмите ОК, а данные о свойствах компонент найдете в приложении 2.

**Полный текст примера создания оконного приложения**

```
//---Заголовочный файл----------------------------------------
#ifndef CirkleH
#define CirkleH
//---------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Menus.hpp>
#include <Buttons.hpp>
#include <ComCtrls.hpp>
#include <ToolWin.hpp>
//---------------------------------------------------------------
class TForm1 : public Tform
{
__published:      // IDE-managed Components
        TmainMenu *MainMenu1;
        TpopupMenu *PopupMenu1;
        TmenuItem *N1;
        TmenuItem *N2;
        TmenuItem *N3;
        TmenuItem *for1;
        TmenuItem *OperWhile1;
        TmenuItem *Operdo1;
        Tedit *Edit1;
        Tedit *Edit2;
        Tedit *Edit3;
        Tedit *Edit4;
        Tlabel *Label1;
        Tlabel *Label2;
        Tlabel *Label3;
        Tlabel *Label4;
        Tmemo *Memo1;
        TstatusBar *StatusBar1;
        Tbutton *Button1;
        TbitBtn *BitBtn1;
        TcoolBar *CoolBar1;
        TspeedButton *SpeedButton1;
        TspeedButton *SpeedButton2;
        TspeedButton *SpeedButton3;
        TspeedButton *SpeedButton4;
        TmenuItem *N4;
        TmenuItem *Operfor1;
        TmenuItem *OperWhile2;
        TmenuItem *Operdo2;
        void __fastcall N2Click(Tobject *Sender);
        void __fastcall for1Click(Tobject *Sender);
        void __fastcall OperWhile1Click(Tobject *Sender);
        void __fastcall Operdo1Click(Tobject *Sender);
```

```
        void __fastcall FormCreate(Tobject *Sender);

private:    // User declarations
public:     // User declarations
        __fastcall Tform1(Tcomponent* Owner);

};
//---------------------------------------------------------------------
extern PACKAGE Tform1 *Form1;
//---------------------------------------------------------------------
#endif
```

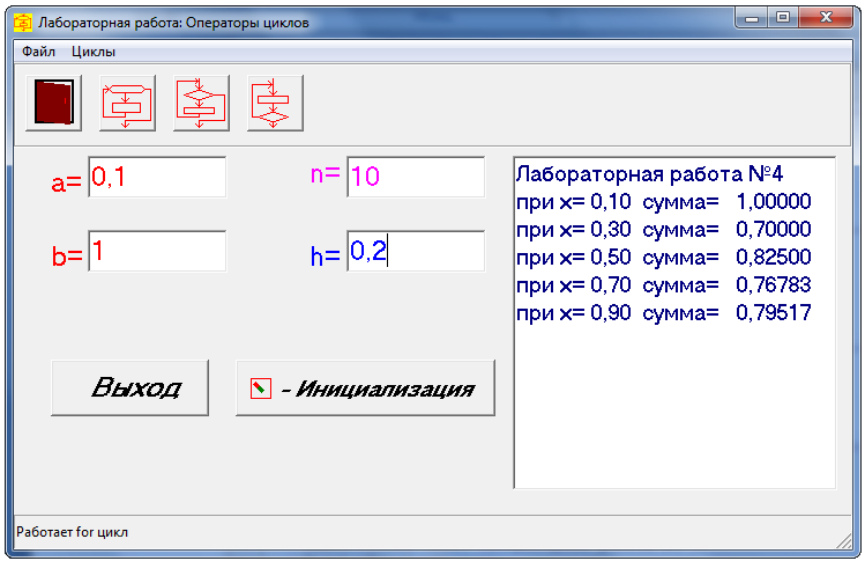



Рис. 4.31. Экспериментальный пример

```
    //-- Файл программы -----------------------------------

    #include <vcl.h>
    #pragma hdrstop
    #include "math.h"
    #include "Cirkle.h"
//---------------------------------------------------------------------
    #pragma package(smart_init)
    #pragma resource "*.dfm"
    TForm1 *Form1;
    double myfak(int k);

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

    void __fastcall TForm1::FormCreate(TObject *Sender)
    {
        Form1->Memo1->Clear();
        Memo1->Lines->Add("Лабораторная работа №4");
        Form1->Edit1->Clear();
```

```
         Form1->Edit2->Clear();
         Form1->Edit3->Clear();
         Form1->Edit4->Clear();
         Form1->StatusBar1->SimpleText="Введите данные и выберите
вид цикла";
     }
//---------------------------------------------------------------------
     void __fastcall TForm1::N2Click(TObject *Sender)
     {
      Form1->Close();
     }
//---------------------------------------------------------------------
     void __fastcall TForm1::for1Click(TObject *Sender)
     {
      double a, b, x, h, r, p,st, s;
      int n, k=0, zn=-1;
       Form1->StatusBar1->SimpleText="Работает for цикл";
      try
        {
           a=StrToFloat(Edit1->Text);
        }
        catch (EConvertError&)
        {
           Application->MessageBox("Симво замените числом.
","Повторите ввод",MB_OK);
           return;
         }
        try
        {
           b=StrToFloat(Edit2->Text);
        }
        catch (EConvertError&)
        {
           Application->MessageBox("Симво замените числом.
","Повторите ввод",MB_OK);
           return;
        }
        try
        {
           n=StrToInt(Edit3->Text);
        }
        catch (EConvertError&)
        {
           Application->MessageBox("Симво замените числом.
","Повторите ввод",MB_OK);
           return;
        }
        try
        {
           h=StrToFloat(Edit4->Text);
        }
        catch (EConvertError&)
        {
           Application->MessageBox("Симво замените числом.
","Повторите ввод",MB_OK);
           return;
         }
```

```
                for(x = a; x<=b; x+=h) {
                        p=myfak(k);
                        st=pow(x,k);
                        r =pow(zn,k)*st/p;
                           s+=r;
            if(n<k) {
           ShowMessage("Увеличте шаг в задании, и повторно
введите");
            break;}
            Memo1->Lines->Add("при x= "+FloatToStrF(x,ffFixed,8,2)+
                          "  сумма =
"+FloatToStrF(s,ffFixed,8,5));
                k++;
        }
    }
    //---------------------------------------------------------------
    void __fastcall TForm1::OperWhile1Click(TObject *Sender)
    {
     double a, b, h;
     int n;
      Form1->StatusBar1->SimpleText="Работает while цикл";
          a=StrToFloat(Edit1->Text);
          b=StrToFloat(Edit2->Text);
          n=StrToInt(Edit3->Text);
          h=StrToFloat(Edit4->Text);
     double r=1,p,st=1, x, s=1;
     int k, zn=1;
                    x=a;  k=0; p=myfak(k);
         Memo1->Lines->Add("при x= "+FloatToStrF(x,ffFixed,8,2)+
                       "  сумма =   "+FloatToStrF(s,ffFixed,8,5));
                       k=1;
       while(x<b-h)
          {
          zn=zn*-1;
           x+=h;
           st=pow(x,k);
           p=myfak(k);
           r =zn*st/p;
           s+=r;
         Memo1->Lines->Add("при x= "+FloatToStrF(x,ffFixed,8,2)+
                       "  сумма =   "+FloatToStrF(s,ffFixed,8,5));
             if (k>n){
          ShowMessage("Уменьшайте шаг в задании, и повторно
введите");
            break;}
            k++;
          }
    }
//---------------------------------------------------------------
    void __fastcall TForm1::Operdo1Click(TObject *Sender)
    {
        Form1->StatusBar1->SimpleText="Работает do цикл";
    double a, b, x, h, r, s, st, p=0;
    int n, zn=-1, k=0;
           a=StrToFloat(Edit1->Text);
           b=StrToFloat(Edit2->Text);
           n=StrToInt(Edit3->Text);
```

```
            h=StrToFloat(Edit4->Text);
                x=a;
         do {
                    p=myfak(k);
                    st=pow(x,k);
                    r = pow(zn,k)*st/p;
                    s+=r;
                    k++;
               Memo1->Lines->Add("при x=
"+FloatToStrF(x,ffFixed,10,2)+
                    "    сумма = "+FloatToStrF(s,ffFixed,10,8));
                    x+=h;
           if (k>n)
            {
            ShowMessage("Увеличте шаг в задании, и повторно введите
");
                    break;
             }
            }
         while (x<=b);
      }
//---------------------------------------------------------------
     double myfak(int k)
     {
        int i;
        double p=1;
       if (k<=1){
          p=1;
          return p;
         }
       else
        for(i=2; i<=k; i++)
          p=p*i;
      return p;
     }
//---------------------------------------------------------------
```

**Средства отладки программ в C++ Builder.** Практически в каждой вновь написанной программе после запуска обнаруживаются ошибки.

Синтаксические ошибки связаны с неправильной записью операторов. При обнаружении таких ошибок компилятор *Builder* останавливается напротив оператора, в котором она обнаружена. В нижней части экрана появляется текстовое окно, содержащее сведения обо всех ошибках, найденных в проекте, в каждой строке которого указаны имя файла, номер строки и характер ошибки.

Для быстрого перехода к интересующей ошибке нужно дважды щелкнуть кнопкой мыши по строке с ее описанием, а для получения более полной информации – обратиться к *HELP* нажатием клавиши *F*1.

Ошибки *семантические* связаны с неверным выбором алгоритма решения или с неправильной программной реализацией задачи. Эти ошибки проявляются обычно в том, что результат расчета оказывается

неверным (переполнение, деление на ноль и др), поэтому перед использованием отлаженной программы ее надо протестировать, т.е. выполнить при значениях исходных данных, для которых заранее известен результат. Если тестовые расчеты указывают на ошибку, то для ее поиска следует использовать встроенные средства отладки среды *Builder*.

В простейшем случае для поиска места ошибки рекомендуется выполнить следующие действия.

В окне редактирования текста установить курсор в строке перед подозрительным участком и нажать клавишу *F*4 (выполнение до курсора) – выполнение программы будет остановлено на строке, содержащей курсор. Чтобы увидеть, чему равно значение интересующей переменной, нужно поместить на переменную курсор – на экране будет высвечено ее значение, либо нажать *Ctrl+F*7 и в появившемся диалоговом окне указать эту переменную (с помощью данного окна можно также изменить значение переменной во время выполнения программы).

Нажимая клавишу *F*7 (пошаговое выполнение), построчно выполнять программу, контролируя изменение тех или иных переменных и правильность вычислений. Если курсор находится внутри цикла, то после нажатия *F*4 расчет останавливается после одного выполнения кода цикла;

Для продолжения расчетов следует нажать <*Run*> в меню *Run*.

### 4.5. Индивидуальные задания

Для каждого ***x***, изменяющегося от ***a*** до ***b*** с шагом ***h***, найти значения функции $Y(x)$, суммы $S(x)$ и $|Y(x)–S(x)|$ и вывести в виде таблицы. Значения *a, b, h* и *n* вводятся с клавиатуры. Так как значение $S(x)$ является рядом разложения функции $Y(x)$, при правильном решении значения *S* и *Y* для заданного аргумента *x* (для тестовых значений исходных данных) должны совпадать в целой части и в первых двух-четырех позициях после десятичной точки.

Работу программы проверить для $a = 0{,}1$; $b = 1{,}0$; $h = 0{,}1$; значение параметра *n* выбрать в зависимости от задания.

Варианты заданий

Таблица 4.1.

| № вар. | Задание для составления цикла | Итерационная формула контроля | Примечание |
|---|---|---|---|
| **1** | **2** | **3** | **4** |
| 1 | $S(x)=\sum_{k=0}^{n}(-1)^k\frac{x^{2k+1}}{(2k+1)!}$ | $Y(x)=\sin(x)$ | |

| | | | |
|---|---|---|---|
| 2 | $S(x)=\sum_{k=1}^{n}(-1)^{k+1}\frac{x^{2k}}{2k(2k-1)}$ | $Y(x)=x\cdot arctg(x)-\ln\sqrt{1+x^2}$ | |
| 3 | $S(x)=\sum_{k=0}^{n}\frac{x^{2k-1}}{(2k-1)!}$ | $Y(x)=sh(x)$ | |
| 4 | $S(x)=\sum_{k=0}^{n}\frac{\cos(k\pi/4)}{k!}x^k,$ | $Y(x)=e^{x\cos\frac{\pi}{4}}\cos(x\sin(\pi/4))$ | |
| 5 | $S(x)=\sum_{k=1}^{n}(-1)^k\frac{x^{2k+1}}{2k+1},$ | $Y(x)=arctg\,x$ | $\lvert x\rvert<1$ |
| 6 | $S(x)=\sum_{k=0}^{n}(-1)^k\frac{x^{2k}}{(2k)!}$ | $Y(x)=\cos(x)$ | |
| 7 | $S(x)=\sum_{k=1}^{n}\frac{x^k\cos(k\pi/3)}{k}$ | $Y(x)=-\frac{1}{2}\ln(1-2x\cos\frac{\pi}{3}+x^2)$ | |
| 8 | $S(x)=\sum_{k=0}^{n}\frac{2k+1}{k!}x^{2k},$ | $Y(x)=(1+2x^2)e^{x^2}$ | |
| 9 | $S(x)=\sum_{k=0}^{n}\frac{\cos(kx)}{k!},$ | $Y(x)=e^{\cos x}\cos(\sin(x))$ | |
| 10 | $S(x)=2\sum_{k=1}^{n}\frac{x^{2k+1}}{2k+1}$ | $Y(x)=\ln\frac{(1+x)}{(1-x)}$ | $\lvert x\rvert<1$ |
| 11 | $S(x)=\sum_{k=1}^{n}(-1)^{k+1}\frac{x^{2k+1}}{4k^2-1}$ | $Y(x)=\frac{1+x^2}{2}arctg(x)-x/2$ | |
| 12 | $S(x)=\sum_{k=0}^{n}\frac{x^{2k}}{(2k)!}$ | $Y(x)=\frac{e^x+e^{-x}}{2}$ | |
| 13 | $S(x)=\sum_{k=0}^{n}\frac{(2x)^k}{k!}$ | $Y(x)=e^{2x}$ | |
| 14 | $S(x)=\sum_{k=0}^{n}\frac{k^2+1}{k!}(x/2)^k$ | $Y(x)=(x^2/4+x/2+1)e^{x/2}$ | |
| 15 | $S(x)=-\sum_{k=1}^{n}\frac{x^k}{k},$ | $Y(x)=\ln(1-x)$ | $\lvert x\rvert<1$ |
| 16 | $S(x)=\sum_{k=0}^{n}(-1)^k\frac{2k^2+1}{(2k)!}x^{2k}$ | $Y(x)=(1-\frac{x^2}{2})\cos(x)-\frac{x}{2}\sin(x)$ | |
| 17 | $S(x)=\sum_{k=1}^{n}(-1)^k\frac{(2x)^{2k}}{(2k)!}$ | $Y(x)=2(\cos^2 x-1)$ | |
| 18 | $S(x)=\sum_{k=0}^{n}\frac{x^{2k+1}}{(2k+1)!}$ | $Y(x)=\frac{e^x-e^{-x}}{2}$ | |
| 19 | $S(x)=\sum_{k=1}^{n}(-1)^{k+1}\frac{x^{2k}}{2k(2k-1)}$ | $Y(x)=-\ln\sqrt{1+x^2}+x\,arctg(x)$ | |

| | | | |
|---|---|---|---|
| 20 | $S(x)=\sum_{k=0}^{n}\frac{\cos(k\pi/4)}{k!}x^k,$ | $Y(x)=\cos[x\cdot\sin(\pi/4)]e^{x\cos\frac{\pi}{4}}$ | |
| 21 | $S(x)=\sum_{k=1}^{n}(-1)^{k-1}\frac{x^k}{k},$ | $Y(x)=\ln(1+x).$ | |
| 22 | $S(x)=\sum_{k=1}^{n}\frac{(-1)^{k-1}}{k}$ | $Y(x)=\ln 2$ | |

### 4.6. Контрольные врпросы

1. Понятие объекта,
2. Отношения между объектами.
3. Введение в классы.
4. Диаграмма объектов.
5. Диаграмма пакетов.
6. Диаграмма автомата
7. Операторы циклов.
8. Невидимые компоненты MainMenu и PopupMenu.

## Лабораторная работа №5.
## Создание консольных приложений

**Цель лабораторной работы**: изучить основы языка С и С ++ и научиться создавать простейшие консольные программы в среде С ++Builder, на языках С и С++.

### 5.1. Введение в процесс подготовки создания консольного приложения на алгоритмических языках С и С++

Что такое - консольное приложение? Приложение, в Windows эти компьютерные программы Word, Excel, пасьянс "Косынка" и Internet Explorer и ряд других. Приложения бывают разными. Не только в том смысле, что Word отличается от "Блокнота", но и по принципу организации своего пользовательского интерфейса. Интерфейс - это внешний вид программы, и в Windows он бывает двух типов. Первый - это графический, второй - текстовый. Все программы, имеющие красивые разноцветные окна, имеют и графический интерфейс. Даже Word и "Блокнот", хоть и работают с текстом, но интерфейс имеют графический. Где же тогда в наши дни можно увидеть программу, работающую в режиме текстового интерфейса?

Для того, чтобы увидеть текстовый пользовательский интерфейс (его ещё называют интерфейсом командной строки), не нужно делать каких-то особенно сложных действий. Нажмите кнопку "Пуск", выберите пункт "Выполнить", наберите в строке появившегося окна текст "command" и нажмите кнопку "Выполнить". Вы увидите перед собой окно с серым текстом на чёрном фоне - примерно такое же, как изображено на Рис. 5.1. Это - командная строка, одна из самых старых вещей в современном персональном компьютере.


Рис. 5.1.

Дело в том, что ещё не так давно, около двадцати лет назад, возможности компьютеров не хватало на отображение даже такой несложной графики, как окна (что и говорить о трёхмерных видеоиграх!). Поэтому компьютеру и пользователям приходилось общаться с помощью текста. Пользователь вводил специальные команды - например, команда dir в первой из систем производства Microsoft, DOS, позволяла просмотреть список файлов и папок в определённой директории, а команда ver показывала версию операционной системы, с которой работал пользователь. Программы, которые запускал пользователь, тоже, естественно, работали в текстовом режиме.

Командная строка, как вы видите, сохранилась в Windows до сих пор. Команды, которые можно выполнить из неё, подробно описаны в справке Windows. А приложения, которые выполняются в текстовом режиме, теперь называются консольными.

Почему же командная строка продолжает жить, несмотря на свой почтенный возраст? Во-первых, есть множество программ, которым не нужен графический интерфейс - например, программы по взлому паролей, которые просто подбирают комбинации символов, пока то, что закрыто паролем, не откроется. Кроме того, командная строка приходит на помощь тем пользователям, которые выучили назубок её команды и быстро набирают с клавиатуры, - для них она гораздо более быстрый способ работать с компьютером, чем графический интерфейс.

Консольных программ масса, и они используются часто для того, чтобы упростить рутинные действия пользователя. Дело в том, что с ними можно обращаться точно так же, как и с обычными командами. А те, в свою очередь, можно записать в специальный текстовый файл с расширением BAT или CMD (такой файл называется командным), и их можно потом выполнить все залпом как обычную программу - достаточно в "Проводнике" дважды кликнуть по этому файлу мышью.

Чтобы создать проект в консольном приложении, выполняем следующую последовательность действий: *File* $\Rightarrow$ *Close All*, т.е. закрываем окно Форм (Form) и Редактора кода (Units).

При этом появляется картина, приведенная на рис. 5.2., где остались открытыми три окна C++ Builder.

Следующая команда *File* $\Rightarrow$ *New* $\Rightarrow$ *Other* позволяет открыть нам различные группы, свойственные C++ Builder.

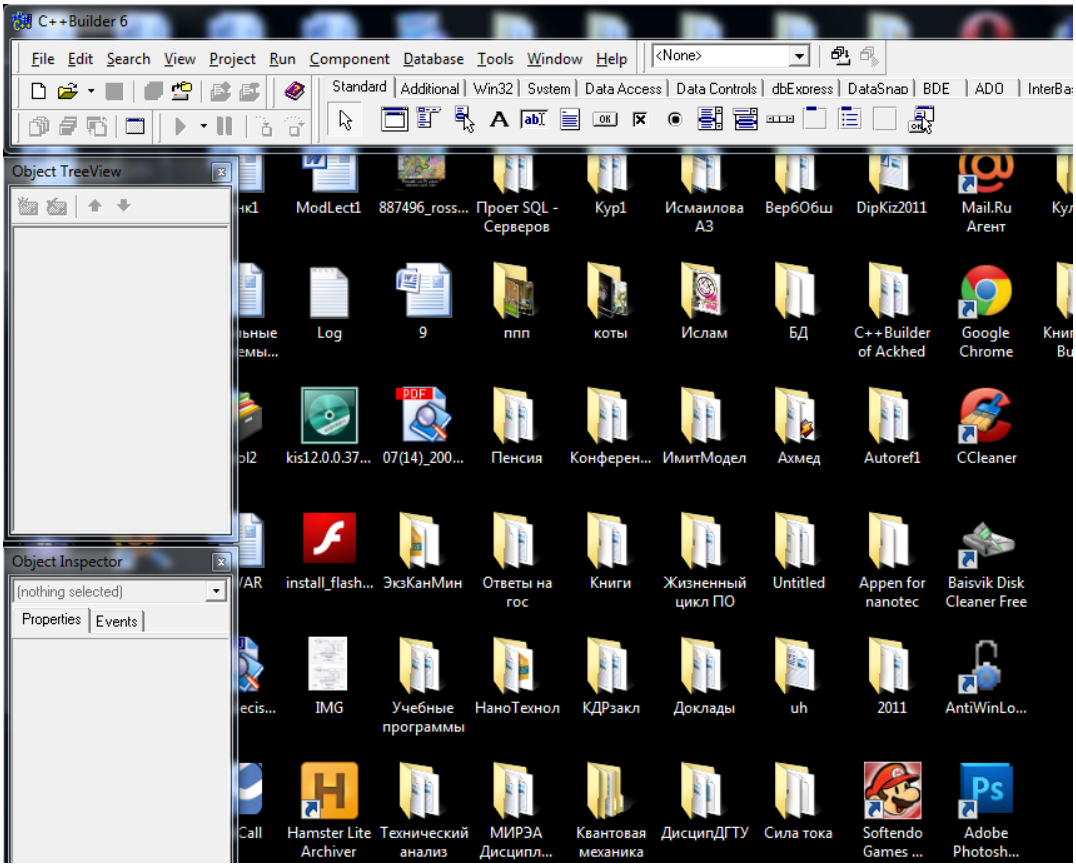



Рис. 5.2. Закрытие всех окон для приложений и проектов.

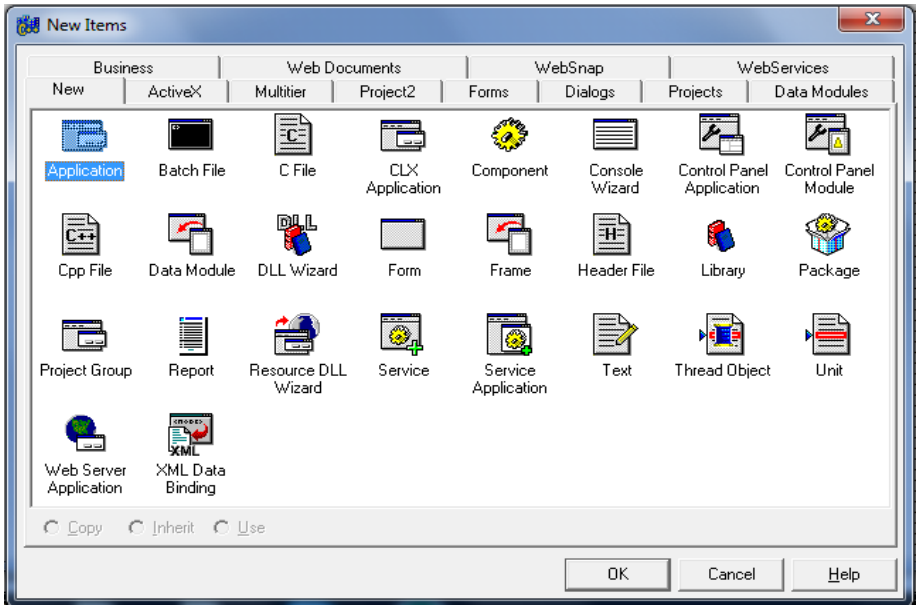



Рис. 5.3. Главное меню файловых операций для создания новых приложений на C++ Builder.

Все объекты объединены в следующие группы:

- New - встроенные базовые объекты, используемые при разработке приложений.

- ActiveX - объекты COM и OLE, элементы ActiveX, активные серверные страницы (ASP).
- Multiter - объекты многопоточного приложения (CORBA и др.)
- Project1 - формы создаваемого приложения.
- Forms - формы.
- Dialogs - диалоговые окна (открытие файла, диалог печати, сохранение и т.д.)
- Projects - проекты одно- и многодокументных приложений.
- Data Modules - модули данных.
- Business - Мастера форм баз данных и Web-приложений.
- Web Documents - Web-документы (HTML, XHTML, WML, XSL).
- WebSnap - WebSnap-приложения и модули.
- WebServices - приложение, модуль и интерфейс для SOAP.
- IntraWeb - приложения и формы Web.
- Corba - CORBA-приложения.

Нас интересует вопрос создания консольного приложения, и мы выбираем команду (объект) *Console Wizard* $\Rightarrow$ *Ok*, при этом получаем окно мастера создания консольных приложений.

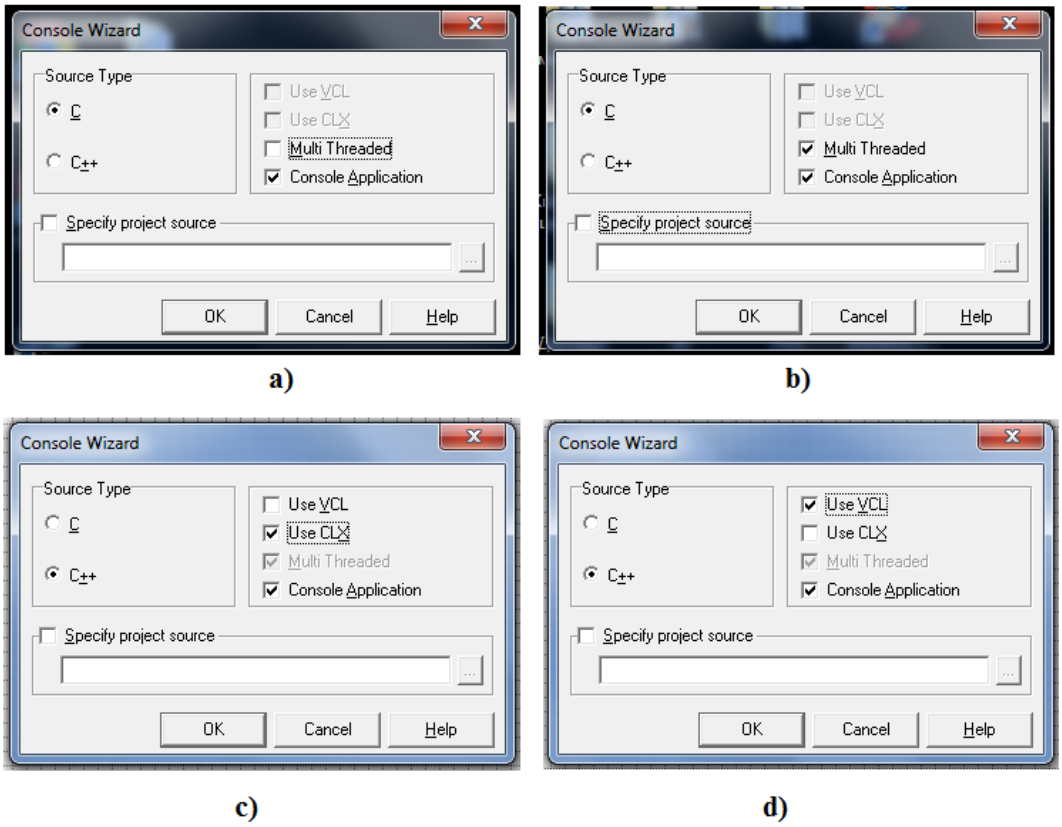
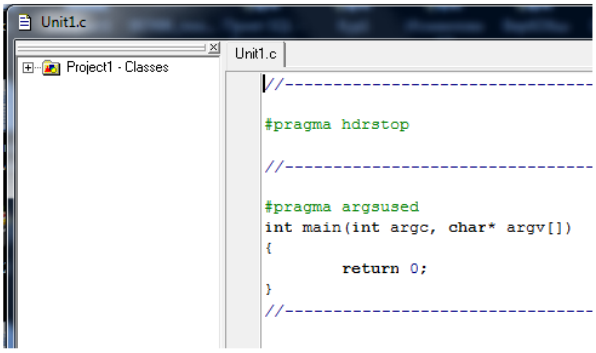



Рис. 5.4. Варианты консольных приложений, допускаемые мастером создания консольных приложений

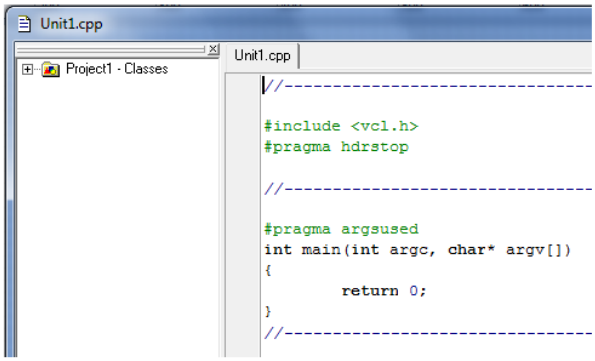
Итак, мы можем выбирать один из вариантов создания консольных приложений, приведенных на рис. 5.4.

Из рисунка 5.3. видим, консольные приложения можно создать, как на языке С, так и на С ++. Дополнительно можем сказать, консольные приложения на языке С могут быть как в однопоточном режиме рис.4. a), так и в многопоточном b), а для С ++ возможно подключение платформ VCL c) или CLX d).

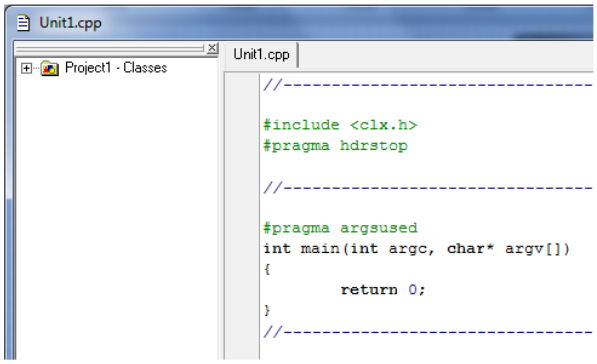
IDE C++ Builder находясь в среде OS Windows позволяет создавать приложения см. рис.5.5., как для среды OS Windows на кросс - платформенной библиотеке компонентов VCL см. b) и d) рис. 5.4, так и для сред OS Windows и для OS Linux см. c) на кросс - платформенной библиотеке компонентов CLX, а также для OS DOS см. a).




**a) при выборе вариантов консольных приложений a) и b)**




**b) при выборе варианта консольного приложения d)**




**c) при выборе варианта консольного приложения c)**

Рис. 5.5. Редакторы кодов при выборе вариантов консольных приложений изображенных на рис. 5.3.

## 5.2. Примеры программ на языке С

**Элементы и структура программы на языке С.** Основы языка С, как С++, и С++ Builder мы начали изучать, начиная с первой темы наряду с другими разделами технологии программирования, теперь, продолжим изложение справочного материала по алгоритмическому языку С. Понимание языка С, является неотъемлемым и необходимым

атрибутом для успешного программирования на C++ и C++ Builder, которые поддерживает стандарт ANSI C и некоторые другие версии языка.

В данной теме, опираясь на короткие примеры, расскажем об элементах программы и технологии программирования на C, а также о синтаксисе различных его конструкций и попутно будем создавать коды для консольного приложения рис. 5.6.

**Элементы простой программы.** Давайте немного поближе познакомимся со строением консольной программы Hello World, аналогичная программа написано нами в первой теме:

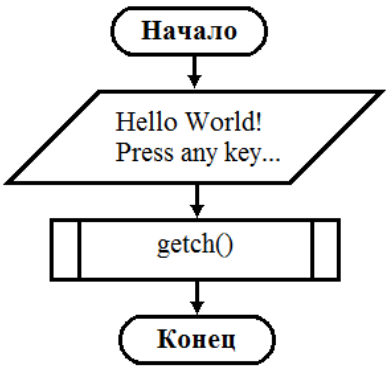



Рис. 5.6. Блок схема задачи ***"Hello World"***

Задача 5.1. ***"Hello World"***

```
/* Простейшая консольная программа C++Builder.
Выводит на экран "Hello World" и ждет, пока
пользователь не нажмет любую клавишу.
*/
//------ Hello World ----------------------------
#pragma hdrstop
#include <stdio.h> //для работы с функцией printf
#include <conio.h> //для работы с функцией getch
//---------------------------------------------------
#pragma argsused
int main(int argc, char* argv[ ])
{
printf("Hello World!\n");
printf("Press any key...");
getch();  // Ожидание нажатия клавиши.
     return 0;
}
```

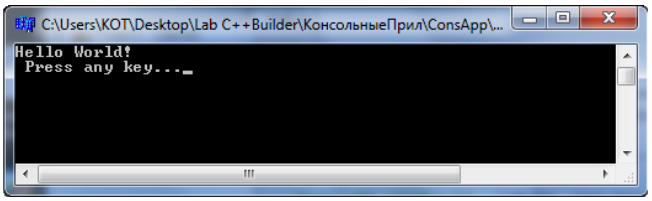



Рис. 5.7 Результат выполнения задачи 5.1.

**Комментарии.** Вот пример комментария в стиле C, который можно было бы поместить в самое начало исходного файла:

Он занимает, как видите, пять строк многострочных и еще строка обычных комментариев. А вот комментарий в стиле C++:`#include <stdio.h> //Файл, где находится функция printf.`

В данном случае комментарий размещен в конце строки и поясняет смысл расположенного в ней оператора, подробно о комментариях мы говорили и при выполнение работы №1. Приведем советы для создания комментарий.

***Для создания и добавления комментарий:***

1. *Не вставляйте комментарий, если это очевидно.*
2. *Комментируйте функции и глобальные данные.*
3. *Не комментируйте плохой код программы — переписывайте его.*
4. *Не допускайте противоречье в программном коде и комментариях к нему.*
5. *В комментариях проясняйте человеку программный код, а не запутывайте его.*

**Директивы # pragma.** Строки исходного кода, начинающиеся со знака шарп #, которые являются, как правило, *директивами препроцессора,* т.е. управляют обработкой текста программы еще до его передачи собственно компилятору (сюда относятся текстовые подстановки, вставка содержимого других файлов и некоторые специальные операции). Директивы #pragma в этом смысле являются исключением, поскольку они адресованы непосредственно компилятору и служат для передачи ему различных указаний. Например, *#pragma argsused* говорит компилятору, что следует подавить выдачу предупреждающего сообщения о том, что параметры функции ***main ()*** никак в ней не используются.

Часто директивы #pragma эквивалентны некоторым установкам компилятора, задаваемым в диалоге Project Options. Например, упомянутые выше сообщения о неиспользуемых параметрах можно было бы запретить, открыв этот диалог (Project | Options... в главном меню) на странице Compiler и нажав кнопку Warnings..., после чего будет открыто окно со списком всех возможных предупреждений; в нем следует сбросить флажок напротив сообщения “Parameter 'parameter' is never used (-wpar)”.

Правда, тем самым в проекте будут запрещены все такие предупреждения, в то время как директива argsused позволяет управлять ими для каждой из функций в отдельности.

Подробнее о #pragma и других директивах мы поговорим в следующих темах.

**Директивы #include**. Директива #include заменяется препроцессором на содержимое указанного в ней файла. Это *заголовочные файлы с* расширением *.h для программ С, а для С++ и С++ Builder расширения *.cpp. Они содержат информацию, обеспечивающую раздельную компиляцию файлов исходного кода и корректное подключение различных библиотек. Имя файла может быть заключено либо в угловые скобки (<> знаки меньше — больше), либо в обычные двойные кавычки (""). Эти случаи различаются порядком поиска включаемых файлов; если использованы угловые скобки, поиск будет сначала производиться в стандартных каталогах С++ Builder, если кавычки — в текущем каталоге.

**Функция `main()`.** После всех директив в программе расположено определение функции **`main()`**. Как уже говорилось, в строгом смысле любая программа на С содержит эту функцию, которая является ее входной точкой, главной функцией. Однако в среде Windows вместо main () часто используется WinMain().

Функция ***`main()`*** — это, конечно, частный случай *функции* вообще. Функции являются основными «строительными блоками» программы, или *подпрограммами.* Они, в свою очередь, строятся из *операторов,* составляющих *тело функции.* Каждый оператор оканчивается точкой с запятой (;). В общем виде формат функции определяется таким образом:

```
Возвращаемый_ тип_ имя функции(список_ параметров)
{
// В фигурных скобках заключено тело функции,
// составленное из отдельных операторов.
    тело_функции
}
```

Функция — единственный тип подпрограмм С, в отличие, от языков программирования PL-1, Modula, Pascal и др., который различает функции и процедуры. Под процедурой обычно понимают подпрограмму, не возвращающую никакого значения. В С формально любая функция возвращает какой-либо тип, хотя в ANSI С этот тип может быть пустым (***void***).

В нашем случае тело функции состоит из четырех операторов, первые три из которых являются, в свою очередь, *вызовами функций.* Значения, возвращаемые функциями, здесь игнорируются, т.е. функции вызываются аналогично процедурам языка Pascal и др. Применяемые здесь функции содержатся в стандартной (исполнительной) библиотеке С.

**Параметры функции main().** Параметры функции **main()** служат для передачи программе аргументов командной строки, т.е. имен файлов, ключей, опций и вообще всего, что вы вводите с клавиатуры после подсказки DOS, запуская программу. Конечно, программа не обязана воспринимать какие-либо команды, указываемые в строке запуска, однако в любом случае функции main() передаются два параметра — число аргументов/включая имя, под которым запущена программа (argc), и массив указателей (argv) на отдельные аргументы (выделенные элементы командной строки). Забегая вперед, приведем пример, который распечатывает по отдельности все “аргументы” строки, введенной пользователем при запуске

```
#include <stdio.h>
int main(int argc, char *argv[])
{
int i;
for (i=0; i<argc; i++)
printf ( "%s\n", argv[i]);
return 0;
}
```

Мы вернемся к подобным примерам, когда мы начнем выполнять действия с массивами. Теперь мы займемся более последовательным изучением основ языка С.

**Представление данных в С.** Любая программа напоминаем, так или иначе, обрабатывает *данные.* Наша программка обрабатывает свои данные — строку сообщения “Hello World!”, выводя ее на консоль (монитор). Рассмотрим, какие возможны варианты представления информации в С.

**Литералы.** Прежде всего, данные могут присутствовать непосредственно в тексте программы. В этом случае они представляются в виде *литеральных констант.* Литералы бывают числовыми, символьными и строковыми. В программе Hello World! мы пользовались строковыми литералами. Это — последовательность символов, заключенная в двойные кавычки.

Символьный литерал служит для представления одиночного знака. Это символ, заключенный в одиночные кавычки (апострофы).

Числовые литералы бывают вещественными (с плавающей точкой) и целыми. В случае целого литерала он может быть записан в десятичной, восьмеричной или шестнадцатеричной системе исчисления. Вещественный литерал поддерживает запись в десятичном формате (с плавающей точкой) или в экспоненциальном.

В таблице 5.1 перечислены все упомянутые выше виды литеральных констант и даны соответствующие примеры.

Таблица 5.1.

| Литерал | Описание | Примеры |
|---|---|---|
| Символьный | Одиночный символ, заключенный в апострофы | 'W', '&', 'Ф' |
| Строковый | Последовательность символов, заключенная в обычные (двойные) кавычки | "Это строка \n" |
| Целый | Десятичный — последовательность цифр, не начинающаяся с нуля | 123, 1999 |
| | Восьмеричный — последовательность цифр от нуля до семерки, начинающаяся с нуля | 011, 0177 |
| | Шестнадцатеричный — последовательность шестнадцатеричных цифр (0 - 9 и A - F), перед которой стоит 0X или Ox | OX9A, Oxffff |
| Вещественный | Десятичный — [цифры].[цифры] | 123., 3.14, .99 |
| | Экспоненциальный — [цифры]E\|e[+\|-] цифры | 3e-10, 1.17e6 |

Можно дать литеральной константе некоторый идентификатор (имя), определив ее в качестве макроса препроцессора. После этого можно вместо литерала использовать имя. Это особенно удобно в том случае, когда одна и та же константа встречается в различных частях программы; используя имя вместо литералов, вы гарантированы от опечаток и, кроме того, гораздо проще вносить в код изменения, если значение константы нужно модифицировать. Макросы определяются директивой препроцессора #define:

```
#define PI 3.14159
#define TRUE 1
#define FALSE 0
#define MAX 100
```

При обработке исходного кода препроцессором выполняется просто текстовая подстановка: каждое вхождение имени макроса заменяется соответствующим ему литералом. Макросы называют также символическими константами (не путайте с символьными).

**Встроенные типы данных (стандартные типы С)**. Однако данные могут не только вписываться в текст программы, но и храниться в памяти во время ее выполнения. С физической точки зрения любая информация в памяти машины выглядит одинаково — это просто последовательности нулей и единиц, сгруппированных в байты. Поэтому наличные в памяти данные должны как-то интерпретироваться процессором; этой интерпретацией управляет, естественно, компилятор С. Любая информация рассматривается компилятором как принадлежащая к некоторому *типу данных.* В языке имеется несколько *встроенных,* или *простых,* типов (возможны и другие типы данных, например, определяемые пользователем). Простые типы перечислены в таблице 2.2. при выполнения работы №2.

**Переменные.** При изучении предыдущих тем мы встречались с переменными. Итак, переменная - отдельная единица данных, которая должна обязательно иметь определенный тип. Для хранения, которой во время работы программы мы должны, во-первых, отвести соответствующее место в памяти, а во-вторых, идентифицировать ее, присвоив некоторое *имя.* Если мы указываем тип и некоторый идентификатор (имя) происходит процесс *декларации* (идентификации или объявления) переменной.

Именованная единица памяти для хранения данных, значения которых в ходе выполнения программы, могут изменяться называется, переменной. Прежде чем использовать переменную в программном коде ее надо декларировать (идентифицировать или объявить).

Синтаксис оператора объявления можно описать примерно так:

*тип имя_переменной [= инициализирующее_значение][, ...];*

Имена переменных при необходимости могут иметь начальные значения, которыми переменные *инициализируются.* Вот несколько примеров:

```
  short i;      //Объявление короткой целой
переменной.
  char quit = 'Q'; //Инициализация символьной
переменной.
  float fl, factor = 3.0, f2; /* Три переменных типа
                 float, одна из которых
                 инициализируется.*/
```

Как и любой другой оператор С, он оканчивается точкой с запятой.

Инициализирующее значение должно быть литеральной (или символической) константой либо выражением, в которое входят только

константы. Инициализация происходит при создании переменной, один раз за все время ее существования (об этом мы будем говорить ниже).

**Типизированные константы.** Разновидностью переменных являются *типизированные константы.* Это переменные, значение которых (заданное при инициализации) нельзя изменить. Создание типизированной константы ничем не отличается от инициализации переменной, за исключением того, что перед оператором объявления ставится ключевое слово const:

const *тип имя_константы = значение [, ...];*

Например:

```
const double Ch_Nep = 2.718281828;
```

Ранее мы демонстрировали определение символической константы:

```
#define CH_NEP 2.718271828
```

Чем отличается эти константы? Здесь следует иметь в виду два момента.

Во-первых, типизированная константа по своему смыслу относится к конкретному типу данных, поэтому компилятор генерирует совершенно определенное представление для ее значения, а представление символической константы не определено.

Во-вторых, имя символической константы значимо только на этапе препроцессорной обработки исходного кода, поэтому компилятор не включает ее в отладочную информацию объектного модуля. Поэтому нельзя использовать это имя в выражениях при отладке. Напротив, типизированные константы являются по существу переменными, и их имена доступны отладчику. В силу этих причин предпочтительнее применять для представления постоянных величин типизированные константы, а не макросы #define.

**Операции и выражения.** Как в математике, так и на языке С, а также и в др. алгоритмических языках из переменных, функций и констант можно составлять формулы, которые называют *выражениями.*

Единственное отличие выражений С++ от конвенциональных формул заключается в том, что набор *операций,* соединяющих члены выражения, отличается от применяемого, скажем, в алгебре. Вот один пример выражения:

```
aResult = (first - second * RATE) <<3
```

Операции характеризуются своим *приоритетом,* определяющим порядок, в котором производится оценка выражения, и правилом *ассоциации,* задающим направление последовательных оценок, идущих друг за другим операций одного приоритета.

Как и в обычных формулах, для изменения порядка оценки выражения могут применяться круглые скобки (кстати, в приведенном выражении они излишни и введены только для наглядности). Знак равенства здесь также является *операцией присваивания,* которая сама (и, соответственно, все выражение в целом) возвращает значение. В этом отличие С от других языков, в частности Pascal, где присваивание является оператором, а не операцией. Оператором выражение станет, если поставить после него точку с запятой.

В таблице 2.4. мы знакомились с операциями языка С в порядке убывания приоритета. Пополним наши знания некоторыми подробностями.

**Семантика операций.** Несколько слов об операциях, перечисленных в таблице 2.4. Смысл и представление некоторых из них будет проясняться в дальнейшем при выполнении заданий и изучении массивов, структур и указателей.

**Операции присваивания.** Операция присваивания (=) не представляет особых трудностей. При ее выполнении значением переменной в левой части становится результат оценки выражения справа. Как уже говорилось, эта операция сама возвращает значение, что позволяет, например, написать: a = b = c = someVar;

После исполнения такого оператора все три переменных a, b, c получат значение, равное someVar. Что касается остальных десяти операций присваивания, перечисленных в таблице 2.4., то они просто служат для сокращенной нотации присваивании определенного вида. Например,

```
s += i;  эквивалентно    s = s + i;
x *= 10; эквивалентно    x = x * 10.
```

**Приведение типа.** Если в операторе присваивания тип результата, полученного при оценке выражения в правой части, отличен от типа переменной слева, компилятор выполнит автоматическое *приведение типа* (по-английски typecast или просто cast) результата к типу переменной. Например, если оценка выражения дает вещественный результат, который присваивается целой переменной, то дробная часть результата будет отброшена, после чего будет выполнено присваивание. Ниже показан и обратный случай приведения:

```
int p;
double pReal = 2.718281828;
p = pReal; // p получает значение 2
pReal = p; // pReal теперь равно 2.0
```

Возможно и принудительное приведение типа, которое выполняется посредством *операции приведения* и может применяться к любому операнду в выражении, например:

```
p = p0 + (int)(pReal + 0.5); // Округление pReal
```

Следует иметь в виду, что операция приведения типа может работать двояким образом:

Во-первых, она может производить действительное преобразование данных, как это происходит при приведении целого типа к вещественному типу и наоборот. Получаются совершенно новые данные, физически отличные от исходных;

Во-вторых, операция может никак не воздействовать на имеющиеся данные, а только изменять их интерпретацию. Например, если переменную типа **short** со значением -1 привести к типу **unsigned short**, то данные останутся теми же самыми, но будут интерпретироваться по-другому (как целое без знака), в результате чего будет получено значение 65535.

Приведение типов в алгоритмических языках C, C++ и C++ Builder является одним из «скользких камней», если нет особой надобности, его следует его избегать.

**Смешанные выражения.** В арифметическом выражении могут присутствовать операнды различных типов — как целые, так и вещественные, а кроме того, и те, и другие могут иметь различную длину (**short, long** и т. д.), в то время как оба операнда любой арифметической операции должны иметь один и тот же тип. В процессе оценки таких выражений компилятор следует алгоритму т. н. *возведения типов,* который заключается в следующем.

На каждом шаге оценки выражения выполняется одна операция, и имеются два операнда. Если их тип различен, операнд меньшего «ранга экстенсивности» приводится к типу более «экстенсивного». Под экстенсивностью понимается диапазон значений, который поддерживается данным типом. По возрастанию экстенсивности типы следуют в очевидном порядке:

**char short**
**int, long**
**float**
**double**
**long double**

Кроме того, если в операции участвуют знаковый и беззнаковый целочисленные типы, то знаковый операнд приводится к беззнаковому типу. Результат тоже будет беззнаковым.

Во избежание ошибок нужно точно представлять себе, что при этом происходит, и при необходимости применять операцию приведения, явно преобразующую тот или иной операнд.

**Функции.** Функция, как уже говорилось, является основным структурным элементом языков C, C++ C++ и Builder. Выше мы уже показывали синтаксис *определения функции:*

*Возвращаемый_тип имя_функции* (*список_параметров)*
*{*
*тело_функции*
*}*.

Подобная форма описания синтаксиса является использование форм Бэкуса-Наура. Это нечто вроде метаязыка для формализации правил, определений и др. конструкций языков программирования. Надеемся, что смысл написанного достаточно ясен. *Курсивом_без_пробелов* обозначаются синтаксические элементы, имеющие самостоятельное значение. Например, *список_параметров* является отдельной синтаксической единицей, хотя он обладает собственной внутренней структурой. Можно было бы раскрыть его определение примерно так:

*Список__параметров:*
**Void**
*объявление_параметра[, объявление_параметра...]*

Далее требовалось бы раскрыть смысл элемента объявление_параметра и т. д. (Несколько строк под определяемым понятием показывают различные варианты его раскрытия.)

Необязательные элементы помещаются в квадратные скобки. Взаимоисключающие варианты отделяются друг от друга вертикальной чертой (например, [+ | -] означает: “здесь может стоять либо плюс, либо минус, либо вообще ничего"). Многоточие показывает, что последний синтаксический элемент может повторяться произвольное число раз.

Тело функции состоит из операторов, каждый из которых завершается точкой с запятой. Заметьте, что сам заголовок функции (его иногда называют ***сигнатурой)*** не содержит точки с запятой.

*Оператор в С может занимать одну или несколько строк, переход на следующую строку с точки зрения компилятора эквивалентен простому пробелу.*

Помимо определения для функции обычно пишется также ее *объявление,* или *прототип,* который размещается в заголовочном файле и служит для проверки корректности обращений к функции при раздельной компиляции исходных файлов. Прототип идентичен заголовку функции, но заканчивается точкой с запятой. Тело функции отсутствует:

```
возвращаемый тип имя функции (список параметров);
```

Функции пишутся для того, чтобы можно было их *вызывать* в различных местах программы. *Вызов функции* является выражением и принадлежит к типу, указанному в ее определении; он имеет вид

```
имя_функции(параметры)
        параметры:
         пусто
      параметр[, параметр...]
```

Параметры, при вызове функции, называют аргументами.

Значение, возвращаемое функцией, можно игнорировать, т.е. использовать функцию в качестве процедуры:

DoSomething(argi, arg2);

Мы так и поступали, когда выводили на экран сообщения функцией printf().

*Функция, в С может иметь переменное или, точнее, неопределенное число параметров. В этом случае за последним обязательным параметром в заголовке функции следует многоточие (...). Подобным образом объявляется функция printf*:

```
int printf(const char *format, ...);
```

*Неопределенное число параметров означает, что количество и тип действительных аргументов в вызове должно так или иначе ей сообщаться, как это и происходит в случае printf() — там число аргументов определяется по числу спецификаторов в строке формата. Тело функции с переменным числом параметров должно быть реализовано на языке ассемблера или, возможно, при помощи каких-то не вполне "законных" ухищрений.*

Пока мы имели дело всего с тремя функциями: ***main(), printf() и getch().*** Давайте поближе познакомимся с `printf()` и другими функциями ввода-вывода стандартной библиотеки С.

**Ввод и вывод в С.** Printf () является функцией стандартной библиотеки с переменным числом аргументов. Она всегда имеет, по крайней мере, один аргумент — *строку формата,* чаще всего строковый литерал. Строка может содержать *спецификаторы преобразования.*

Функция сканирует строку и передает ее символы на *стандартный вывод* программы, по умолчанию консоль, пока не встретит спецификатор преобразования. В этом случае `printf()` ищет дополнительный аргумент, который форматируется и выводится в соответствии со спецификацией. Таким образом, вызов `printf()` должен содержать столько дополнительных аргументов, сколько спецификаторов преобразования имеется в строке формата.

**Спецификация преобразования.** Синтаксис спецификатора преобразования имеет такой вид:

***%[флаги] [поле][.точность][размер]символ типа***

Как видите, обязательными элементами спецификатора являются только начальный знак процента и символ, задающий тип преобразования. В таблице 5.2 приведены возможные варианты элементов спецификации.

Таблица 5.2.

| Элемент | Символ | Аргумент | Описание |
|---|---|---|---|
| флаг | - | | Выровнять вывод по левому краю поля. |
| | 0 | | Заполнить свободные позиции нулями вместо пробелов. |
| | + | | Всегда выводить знак числа. |
| | пробел | | Вывести пробел на месте знака, если число положительное. |
| | # | | Вывести 0 перед восьмеричным или Ox перед шестнадцатеричным значением. |
| поле | число | | Минимальная ширина поля вывода. |
| точность | число | | Строки—максимальное число выводимых символов; для целых — минимальное число выводимых цифр; для вещественных — число цифр дробной части. |
| размер | h | | Аргумент -- короткое целое. |
| | 1 | | Аргумент — длинное целое. |
| | L | | Аргумент имеет тип long double. |
| символ типа | d | целое | Форматировать как десятичное целое со знаком. |
| | i | целое | То же, что и d. |
| | o | целое | Форматировать как восьмеричное без знака. |
| | U | целое | Форматировать как десятичное без знака. |
| | x | целое | Форматировать в шестнадцатеричном в нижнем регистре. |

| | | | |
|---|---|---|---|
| | X | целое | Форматировать в шестнадцатеричном в верхнем регистре. |
| | f | вещественное | Вещественное в форме [-]dddd.dddd. |
| | e | вещественное | Вещественное в форме [-]d.ddde[+\|-]dd. |
| | E | вещественное | То же, что и e, с заменой e на E. |
| | *ё* | вещественное | Использовать форму f или e в зависимости от величины числа и ширины поля. |
| | G | вещественное | То же, что и g — но форма f или E. |
| | c, | символ | Вывести одиночный символ. |
| | s | строка | Вывести строку. |
| | n | указатель | Аргумент — указатель на переменную типа int. Куда записывается количество выведенных символов. |
| | p | указатель | Вывести указатель в виде шестнадцатеричного числа XXXXXXXX. |

Как видите, *флаги* задают «стиль» представления чисел на выводе, *поле* и *точность* определяют характеристики поля, отведенного под вывод аргумента, *размер* уточняет тип аргумента и *символ_типа* задает собственно тип преобразования. Следующий пример показывает возможности форматирования функции printf(). Советую не полениться и поэкспериментировать с этим кодом, меняя флаги и параметры поля вывода.

**Задание 5.1.** Постройте самостоятельно блок – схему алгоритма к задаче 5.2.

**Задача 5.2. Возможности функции** printf ()

```
/*
** Printf.c: Демонстрация форматирования вывода на консоль
** функцией printf().
* /
#pragma hdrstop
#include <stdio.h>
#include <conio.h>
#pragma argsused
int main(int argc, char *argv[])
{
double p = 21982.91028;
int j = 255,a=6,b=-90,c=0xab82,d=01756;
short t=32,u=-32767,r=-1;
long l=-31L,m=-567,n=12345643;
unsigned int ua=225, ub=0xfa, uc=-5764;
char s1[] = "Type of integer formatting:";
char s[] = "Press any key...";
/*Вывести 4 цифры; вывести обязательный знак: */
printf("Test integer formatting: %13.4d %+8d\n", j, j);
```

```
   /*Вывести по левому краю со знаком; заполнить нулями: */
   printf("More integer formatting: %-+13d % 08d\n", j, j);
   printf("Test octal and hex: %#13o %#8.6x\n", j, j);
   printf("\nTest e and f conversion: %13.7e %8.2f\n", p, p)
;
   /*Печать переменных типа int a=6,b=-90,c=0xab82,d=01756,
   как десятичных целых */
   printf("\n%s", s1); /* Вывести строку подсказки. */
   printf("\nShow int: a=%d\t b=%d\t c=%d\t d=%d\n", a, b, c,
d);
   /*Печать переменных типа short t=32,u=-32767,r=-1*/
   printf("Show short: t=%d\t u=%d\t r=%d\n", t, u, r);
   /*Печать переменных типа long l=-31L,m=-567,n=12345643*/
   printf("Show long: l=%ld\t m=%ld\t n=%ld\n", l, m, n);
   /*Печать переменных типа unsigned int ua=225, ub=0xfa,
uc=-5764,
   как десятичных целых */
   printf("Show unsigned int: ua=%u\t ub=%u\t uc=%u\n", ua,
ub, uc);
   printf("\n%s", s); /* Вывести строку подсказки. */
   getch () ;
          return 0;
   }
```

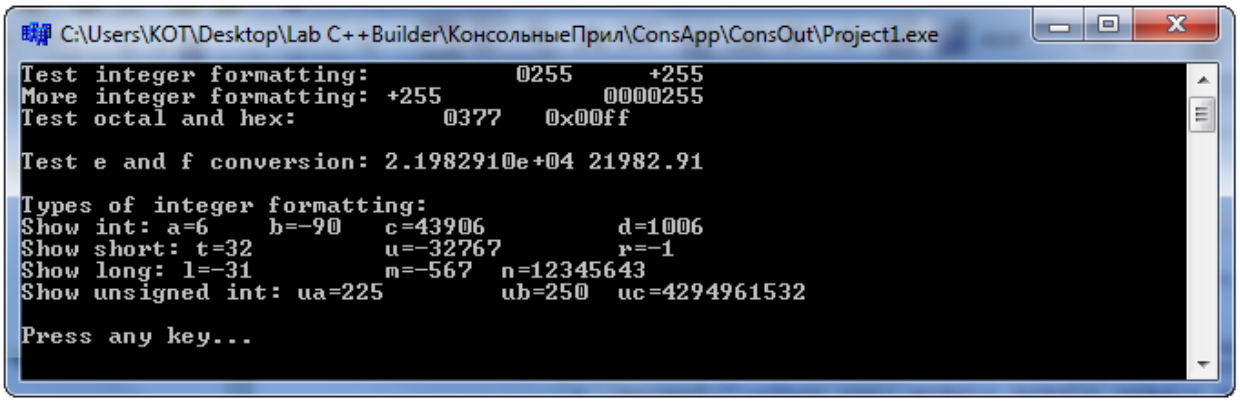



Рис.5.8. Демонстрация возможностей функции printf().

**Escape-последовательности.** В строках языка С для представления специальных (например, непечатаемых) символов используются *escape-последовательности,* состоящие из обратной дробной черты, за которой следует один или несколько символов. (Название появилось по аналогии с командами управления терминалом или принтером, которые действительно представляли собой последовательности переменной длины, начинающиеся с кода ESC.) В приведенных примерах функции printf() вы уже встречались с одной такой последовательностью — \n. Сама обратная косая черта называется *escape-символом.*

В таблице 3 перечислены возможные esc-последовательности.

Таблица 5.3.

| Последовательность | Название | Описание |
|---|---|---|
| **\a** | Звонок | Подает звуковой сигнал. |
| **\b** | Возврат на шаг | Возврат курсора на одну позицию назад. |
| **\f** | Перевод страницы | Начинает новую страницу. |
| **\n** | Перевод строки | Начинает новую строку. |
| **\r** | Возврат каретки | Возврат курсора к началу текущей строки. |
| **\t** | Табуляция | Переход к следующей позиции табуляции. |
| **\v** | Вертикальная табуляция | Переход на несколько строк вниз. |
| \\ | | Выводит обратную дробную черту. |
| \' | | Выводит апостроф (одинарную кавычку). |
| \" | | Выводит кавычку (двойную). |

Кроме того, esc-последовательности могут представлять символы в ASCII-коде — в восьмеричном или шестнадцатеричном формате:

| | |
|---|---|
| \000 | От одной до трех восьмеричных цифр после esc-символа. |
| \xHH или \XHH | Одна или две шестнадцатеричных цифры после esc-символа. |

**Функции ввода строки — `scanf()` и `gets()`.** В языке С для ввода имеется «зеркальный двойник» printf() — функция scant(). Функция читает данные со *стандартного ввода,* по умолчанию — клавиатуры. Она так же, как и printf(), принимает строку формата с несколькими спецификаторами преобразования и несколько дополнительных параметров, которые должны быть *адресами* переменных, куда будут записаны введенные значения.

В языке С функция не может изменять значение передаваемых ей аргументов, поскольку ей передается только временная копия содержимого соответствующей переменной, т.к. нет автоматического механизма передачи по ссылке. Это называется передачей параметра по значению. Чтобы передать из функции некоторое значение через параметр, ее вызывают с указателем на переменную (грубо говоря, ее адресом), подлежащую модификации. Функция не может изменить переданный ей аргумент, т.е. сам адрес, но она может записать информацию в память по этому адресу. Адрес получают с помощью операции &, например, SaVar. Подробнее мы обсудим это, когда будем говорить об указателях.

Примером вызова `scanf ()` может служить следующий фрагмент кода: **int** age;

printf("*Enter your age:* ");//*Запросить ввод возраста пользователя.*
scanf *("%d"*, &age); // *Прочитать введенное число.*

Функция возвращает число успешно сканированных полей, которое в приведенном фрагменте игнорируется. При необходимости вы можете найти полную информацию по scanf () в оперативной справке C++Builder. Однако следует сказать, что программисты не любят эту функцию и пользуются ей очень редко. Причина в том, что опечатка при вводе (скажем, наличие буквы в поле, предполагающем ввод числа и т. п.) может привести к непредсказуемым результатам. Контролировать корректность ввода и обеспечить адекватную реакцию программы на ошибку при работе со scanf () довольно сложно. Поэтому часто предпочитают прочитать целиком всю строку, введенную пользователем, в некоторый буфер, а затем самостоятельно декодировать ее, выделяя отдельные лексемы и преобразуя их в соответствующие значения. В этом случае можно контролировать каждый отдельный шаг процесса преобразования.

Ввод строки с клавиатуры производится функцией `gets ()`:

```
char s[80] ;
gets (s) ;
```

Буфером, в который помещается введенная строка, является здесь символьный массив s[ ]. О массивах чуть позже, пока же скажем, что в данном случае определяется буфер, достаточный для хранения строки длиной в 79 символов — на единицу меньше, чем объявленная длина массива. Одна дополнительная позиция необходима для хранения признака конца строки; все строки в С должны оканчиваться нуль-символом \0, о котором программисту, как правило, заботиться не нужно. Функции обработки строк сами распознают эти символы или, как gets(), автоматически добавляют нуль-символ в конец строки-результата. Функция gets() возвращает данные через параметр, поэтому, как говорилось выше, ей нужно передать в качестве параметра адрес соответствующего символьного массива. Операция взятия адреса, однако, здесь не нужна, поскольку имя массива (без индекса) само по себе является указателем на начало массива. Забегая вперед, скажем, что показанная нотация эквивалентна

```
     gets(&s[0]);
// Аргумент - указатель на начальный элемент массива
s.
```

Для преобразования строк, содержащих цифровое представление чисел, в численные типы данных могут применяться функции atoi(), ato1() и atof(). Они преобразуют строки соответственно в целые, длинные целые и вещественные числа (типы **`int`**, **`long`** и **`double`**). Входная строка может содержать начальные пробелы; первый встреченный символ, который не может входить в число, завершает преобразование.

Прототипы этих функций находятся в файле stdlib.h.

**Задача 5.3. Пример создания функции.** Пользователю предлагается ввести имя (в произвольной форме — только имя, имя и фамилию и т. п.), а затем номер телефона, просто как 7-значное число без пробелов или дефисов. После этого программа распечатывает полученные данные, выводя номер телефона в более привычном формате (рис.5.9).

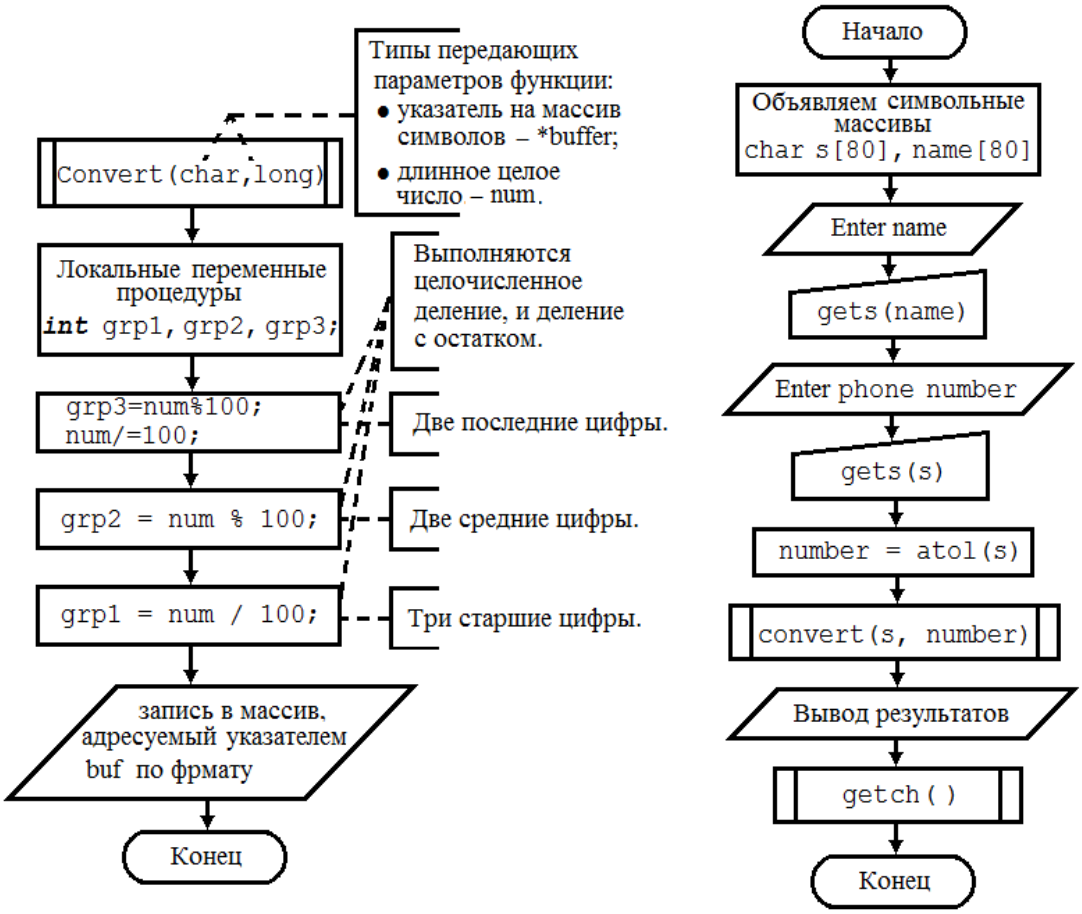



Рис. 5.9. Функция `convert(`***`char, long`***`)` для задачи 5.3.

```
  /*
  ** Convert.h: Пример функции, преобразующей число
  ** в строку специального вида. */
#pragma hdrstop
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
/* Прототип функции */
```

```
  void Convert(char *buffer, long num);
  //---------------------------------------------------------
  #pragma argsused
  int main(int argc, char* argv[])
  {
     long number;
     char s[80], name[80] ;
     printf("Enter name: ");
     gets(name) ;
     printf("Enter phone number: ");
     gets (s) ;
     number = atol(s);
  /* Преобразовать номер в обычную форму. */
     Convert(s, number);
  /* Вывести результат. */
     printf("\n%-30s %10s\n", name, s);
     getch () ;
    return 0;
  }
  /* Определение функции */
  void Convert(char *buffer, long num)
  {
  int grp1, grp2, grp3;
  grp3 = num % 100; // Две последние цифры.
num /= 100;
  grp2 = num % 100; // Две средние цифры
  grp1 = num / 100; //Три старшие цифры преобразовать в строку.
  sprintf (buffer, "%03d-%02d-%02d", grp1, grp2, grp3) ;
  }
```

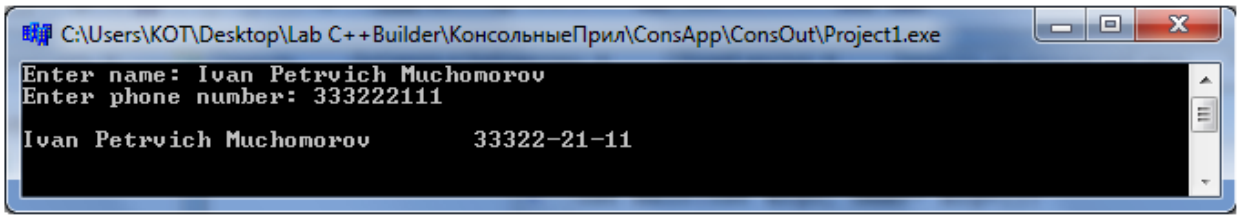



Рис 5.10. *Программа Convert.* Вывод номера телефона в более привычном формате.

Функция **Convert()** описана как **void** и не возвращает значения, вследствие чего в ее теле можно опустить оператор return. (оператор return служит для возврата значения функции в вызывающую программу.) Она преобразует переданный ей телефонный номер (второй параметр) и записывает его в указанный строковый буфер (первый параметр). Центральный момент преобразования — разбиение номера на группы — является довольно характерным примером применения операций деления с остатком.

Для преобразования полученных групп в строку вызывается функция sprintf (). Она совершенно аналогична функции printf () за

исключением того, что вместо вывода на консоль записывает результат в строковый буфер, указанный первым параметром.

В основной программе, т. е. в функции main (), использована функция ato1(), преобразующая строку в длинное целое.

В верхней части файла мы поместили прототип функции Convert() Определение функции мы поместили *после* ***main()***, поэтому прототип в данном случае необходим — без него компилятор не сможет корректно генерировать вызов Convert().

*Подытожим некоторые правила относительно прототипов и определений функций:*

- *Функции стандартные, и определяемые пользователем может возвращать значение практически любого типа или не возвращать его вообще. В последнем случае функция описывается как* ***void.***
- *Функция может не иметь параметров. В этом случае на месте списка параметров в прототипе или определении также ставится ключевое слово void или список оставляют пустым; в вызове функции на месте списка аргументов также ничего не пишется (однако скобки необходимы).*
- *В прототипе, в отличие от определения, нет необходимости указывать имена параметров; список параметров может состоять из перечисления только их типов, разделенных запятыми, например: void Convert(****char*, long****);*
- *Прототип не обязателен, если определение функции расположено в тексте программы выше того места, где она вызывается (точнее говоря, в этом случае прототипом служит само определение функции).*

**Задание 5.2.** Выполните свои варианты заданий в виде консольных приложения из темы 2 на языке С.

### 5.3. Область действия переменных и связанные с ней понятия

Теперь, когда мы более-менее разобрались с принципами функциональной организации программы, следует обсудить некоторые весьма важные вопросы относительно переменных.

Переменные в С могут быть *локальными* и *глобальными, статическими* и *автоматическими, регистровыми, внешними* и даже *нестабильными.* Они различаются своей *областью действия, видимостью* и *временем жизни.* Попробуем как-то сориентироваться во всем этом многообразии.

**Область действия.** Область действия — это та часть программы, где переменная в принципе доступна для программного кода (что

означает это “в принципе”, выяснится чуть позже). По своей области действия переменные делятся на локальные и глобальные.

Локальные переменные объявляются внутри функции и вне ее тела недоступны. Вернитесь к последнему примеру. Там программа состоит из двух функций. В ***main()*** объявляются переменные number, s и name (две последних — не простые переменные, а массивы, но это несущественно). В функции Convert объявлены grp1, grp2 и grp3.

Все эти переменные являются локальными. К каждой из них можно обращаться только в пределах объявляющей ее функции. Поэтому, кстати, имена локальных переменных не обязаны быть уникальными. Если две функции описывают переменную с одним и тем же именем, то это две совершенно различные переменные и никакой неоднозначности не возникает.

*Параметры в определении функции (формальные параметры) можно рассматривать как локальные переменные, инициализируемые значениями аргументов при ее вызове.*

В противоположность локальным глобальные переменные не относятся ни к какой функции и объявляются совершенно независимо. В пределах текущего модуля имя глобальной переменной, естественно, должно быть уникальным. Областью действия глобальных переменных является по умолчанию вся программа.

Довольно интересная проблема возникает, казалось бы, когда имя локальной переменной функции совпадает с именем некоторой глобальной переменной. Это вполне допустимая ситуация, и одноименные переменные здесь на самом деле различны. Если мы входим внутрь определения функции, то оказываемся в области действия сразу двух переменных. Однако локальная переменная в этом случае *скрывает,* как говорят, глобальную переменную с тем же именем. Тут речь идет об их *области видимости,* которая не совпадает с областью действия. Эти два понятия часто путают.

**Задача 5.4.** Пример кода, иллюстрирующий вышесказанное.

Самостоятельно начертить блок схему.

```
 ** Область действия и видимость переменных.*/
//---------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include <stdio.h>
#include <conio.h>
int iVar = 111; // Глобальная переменная.
/* Прототипы функции */
void Func1(void);
```

```
void Func2(void);
//---------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{int iVar = 111; // Глобальная переменная.
int main(void)
{
printf ("Initialize iVar: %d.\n", -iVar);
// Печатает 111. Funci();//Печатает 222, но не изменяет
// глобальную iVar.
printf("After call Func1(): %d.\n", iVar);
Func2 (); // Печатает 111 и изменяет iVar на 333.
printf ("After call Func2(): %d.\n", iVar) ;
return 0;
}
//---------------------------------------------------------------------
void Func1(void) {
int iVar = 222; //Локальная переменная Funci().
/* Локальная переменная скрывает глобальную. */
printf("Initial iVar in Func1() = %d.\n", iVar);}
void Func2(void){
/*Глобальная переменная доступна.*/
printf("Result iVar in Func2 () = %d.\n", iVar);
                iVar = 333; //Изменяет глобальную переменную.}
```


Рис. 5.11. Результаты задачи 5.4.

**Время жизни.** Время жизни переменной в известной мере определяется ее областью действия. Память под глобальную переменную отводится, можно сказать, еще на этапе компиляции программы; во всяком случае, переменная существует все время, пока программа выполняется.

Локальная переменная создается при входе в функцию и уничтожается при возврате из нее. Она является *автоматической* переменной. Поэтому никак нельзя ожидать, например, что локальная переменная будет сохранять свое значение в промежутках между вызовами объявляющей ее функции. Память под переменную выделяется на стеке программы.

Если же вы хотите, чтобы локальная переменная сохраняла значение между вызовами функции, ее следует объявить с модификатором ***static***, как в следующем:

```
int AFunc(void){
```

```
    /* Так можно организовать счетчик вызовов
функции. */
    static int callCount = 0;
    // Здесь что-то делается...
        return ++callCount;
    }
```

*На тот случай, если у кого-то этот код вызвал сомнения, скажу, что инициализация локальной статической переменной производится всего один раз — при запуске программы, а не при каждом входе в функцию.*

Такая переменная будет существовать все время, пока программа выполняется. Память под статическую переменную отводится в той же области, где располагаются глобальные переменные. Таким образом, статическая локальная переменная очень похожа на глобальную, за исключением того, что ее областью действия является все-таки объявляющая функция; вне ее переменная недоступна.

**Модификаторы переменных.** Помимо static, в C имеются и другие модификаторы, применяемые к объявлениям переменных. Опишем их вкратце.

- **static**. Его воздействие на локальную переменную описано выше. Примененный к глобальной переменной, он ограничивает область ее действия текущим файлом (модулем компиляции).
- **auto**. Специфицирует локальную переменную как создаваемую автоматически и подразумевается по умолчанию.
- **register**. Этот модификатор рекомендует компилятору разместить локальную переменную в регистре процессора, если это возможно.
- **extern**. Модификатор говорит компилятору, что переменная является внешней, т. е. объявлена в другом файле.

**Модификатор volatile.** Об этом модификаторе следует сказать отдельно. Он применяется для объявления переменных, которые можно назвать *нестабильными.* Модификатор volatile сообщает компилятору, что значение переменной может изменяться как бы само по себе, например, некоторым фоновым процессом или аппаратно. Поэтому компилятор не должен пытаться как-либо оптимизировать выражения, в которые входит переменная, — предполагая, например, что ее значение не менялось с предыдущего раза и потому выражение не нужно заново пересчитывать.

Есть и другой момент. В программе могут быть так называемые *критические участки кода,* во время исполнения которых изменение

значения нестабильной переменной приведет к абсурдным результатам. (Возьмите случай оценки «А или не-А», если А нестабильно) Для таких критических участков компилятор должен создавать копию, например, в регистре, и пользоваться этой копией, а не самой переменной.

*Можно написать такое объявление:*

```
volatile const int vciVar = 10;
```

*Другими словами, «нестабильная константа» типа* ***int****. В этом нет никакого противоречия — компилятор не позволит программному коду изменять переменную, но, и не будет предполагать ее значение априори известным, так как оно может меняться в силу внешних причин.*

**Массивы.** Массивы и указатели довольно тесно связаны между собой. Имя массива можно разыменовывать, как указатель. В свою очередь, указатель можно индексировать, как массив, если это имеет смысл.

*Массив* по существу является совокупностью однотипных переменных *(элементов* массива), объединенных под одним именем и различающихся своими *индексами.* Массив объявляется подобно простой переменной, но после имени массива указывается число его элементов в квадратных скобках: `int myArray[8];`

Массив, как и переменную, можно инициализировать при объявлении. Значения для последовательных элементов массива отделяются друг от друга запятыми и заключаются в фигурные скобки:

```
int iArray[8] = {7, 4, 3, 5, 0, 1, 2, 6);
```

Обращение к отдельным элементам массива производится путем указания индекса элемента в квадратных скобках, например:

```
myArray[3] = 11;
myArray[i] = iArray[7-i];
```

Индекс должен быть целым выражением, значение которого не выходит за пределы допустимого диапазона. Поскольку индексация массивов начинается в С всегда с нуля (т. е. первый элемент имеет индекс 0), то, если массив состоит из *N* элементов, индекс может принимать значения *0*..N-1.

В языке С не предусмотрена автоматическая проверка допустимости значений индекса времени выполнения, поэтому при индексации массивов нужно быть внимательным. Выход индекса за границы массива может приводить к совершенно непредсказуемым результатам.

Массивы естественным образом сочетаются с циклами ***for***.

Приведем пример программы, работающей с массивом целых чисел.

**Задача 5.5.** Выполнить так называемую “пузырьковую сортировку” введенных пользователем чисел в порядке возрастания. Работу программы 5.5. иллюстрирует блок - схема рис. 5.12. и.5.13 результаты.

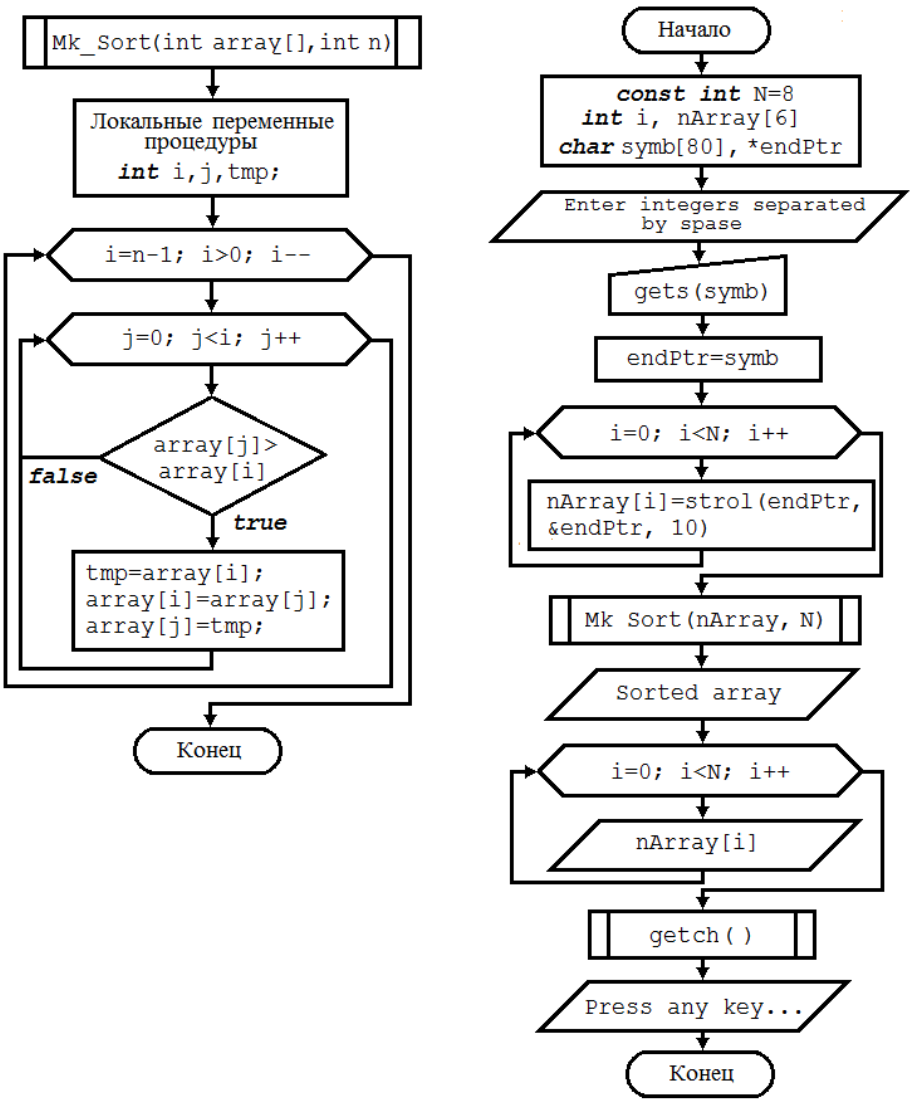



Рис. 5.12. Пузырьковая сортировка целых чисел в массиве.

**Листинг кода программы 5.5.** Программа пузырьковой сортировки

```
      /*** Loop.c: Программа пузырьковой сортировки. ***/
      #pragma hdrstop
      #include <stdio.h>
      #include <stdlib.h>
      #include <conio.h>
      void Mk_Sort(int array[ ], int n);     //Прототип
функции
      /***** Процедура сортировки ******/
 void Mk_Sort(int array[ ], int n)
 {
  int i, j, tmp;
   for (i = n-1; i > 0; i--)
     for (j = 0; j < i; j++)
      if (array[j] > array[i]) {
```

```
        tmp = array[i];
        array[i] = array[j];
        array [j] = tmp;
      }
} /* Конец Mk_SortO */
      #pragma argsused
      int main(int argc, char* argv[])
      {
      const int N = 8;
      int i, nArray[8];
      char symb[80], *endPtr;
printf("Enter %d integers separated by spaces:\n", N);
gets(symb); // Прочитать строку пользователя.
           endPtr = symb; // Инициализировать указатель
    строки.
           for (i =0; i < N; i++) // Преобразование чисел.
           nArray[i] = strtol(endPtr, &endPtr, 10);
           Mk_Sort(nArray, N); // Вызов программы сортировки.
           printf("Sorted array:\n");
      for (i =0; i < N; i++)
      // Вывод отсортированного массива.
      printf("%8d ", nArray[i]);
           printf("\n\nPress a key...");
      getch() ;
      return 0;
      }
```

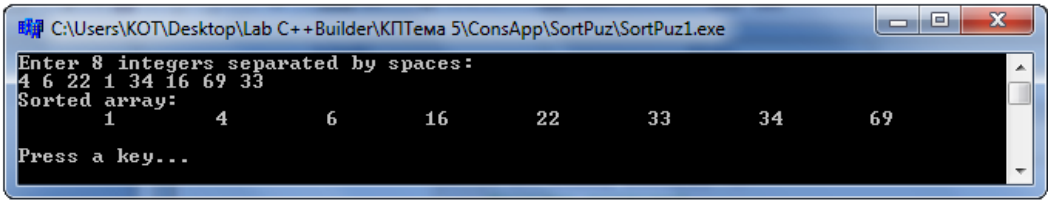



Рис. 5.13. Робота программы сортировки

Как видите, функция Mk_Sort (), выполняющая сортировку массива из целых чисел, содержит двойной вложенный цикл **for**. Операторы внутри цикла обменивают значения двух соседних элементов массива, если первый из них больше второго. В главной функции циклы for использованы для чтения и вывода элементов массива nArray.

Строка, содержащая все введенные пользователем числа, считывается целиком в символьный массив s. После этого для преобразования символьного представления чисел в данные целого типа в цикле вызывается функция strtol (). Ее прототип (в stdlib.h) имеет вид

```
long strtol(const char *str, char **endptr, int radix);
```

Эта функция действует подобно atol(), преобразуя строку str в значение типа long, однако обладает более широкими возможностями.

Параметр radix задает основание системы счисления (8, 10 или 16). В параметре endptr функция возвращает указатель на непреобразованную часть строки str (т.е. на строку, оставшуюся после исключения из нее первого из чисел). Таким образом, в цикле мы последовательно вычленяем из строки все восемь чисел и записываем их в элементы целого массива.

Функция позволяет довольно просто контролировать корректность ввода. Если при ее вызове происходит ошибка, например, строка содержит меньше чисел, чем их должно быть, возвращаемый в endptr указатель будет совпадать с аргументом str.

Ключевое слово **const** в объявлении первого параметра говорит компилятору, что функция не должна изменять элементы строки, на которую указывает str. Попытка модифицировать строку в теле функции вызовет ошибку при ее компиляции.

## 5.4. Спагетти коды в программировании. Решение системы линейных алгебраических уравнений (СЛАУ) методом Жордана-Гаусса с использованием спагетти кодов

Спагетти-код — плохо спроектированная, слабо структурированная, запутанная и трудная для понимания программа, особенно содержащая много операторов GOTO (особенно переходов назад), исключений и других конструкций, ухудшающих структурированность. Спагетти-код назван так, потому что ход выполнения программы похож на миску спагетти, то есть извилистый и запутанный. Иногда называется «кенгуру-код» (kangaroo code) из-за множества инструкций «jump».

В настоящее время термин применяется не только к случаям злоупотребления GOTO, но и к любому «многосвязному» коду, в котором один и тот же небольшой фрагмент исполняется в большом количестве различных ситуаций и выполняет очень много различных логических функций.

Спагетти-код обычно возникает:

1. От неопытности разработчиков;
2. От серьёзного давления по срокам, как установленного руководством (например, в принятой в компании системе мотивации на «работу быстрее»), так и установленного разработчиком самому себе (желание всё сделать наиболее быстрым способом), при этом не является результатом преднамеренного запутывания.

Спагетти-код может быть отлажен и работать правильно и с высокой производительностью, но он крайне сложен в сопровождении и развитии. Правка спагетти для добавления новой функциональности иногда несёт такой огромный потенциал внесения новых ошибок, что рефакторинг (главное лекарство от спагетти) становится неизбежным.

Язык С допускает использование спагетти кода наряду с парадигмой структурного программирования, где в частности применяется использование оператора goto.

Формат оператора: `goto метка`

*Важно помнить, при использовании этого оператора, программа становиться трудно читаемой, и трудно воспринимаемой по смыслу.*

Приведем пример, где используем операторы безусловного перехода в частности для решения системы линейных алгебраических уравнение (СЛАУ)методом Жордана-Гаусса.

Суть метода заключается в том, рассматривают первое уравнение, как уравнение с разрешающим элементом (должен быть максимальным элементом в столбце по абсолютной величине).

$$\begin{bmatrix} a_{11} + a_{12} + \cdots + a_{1n} = b_1 \\ a_{21} + a_{22} + \cdots + a_{2n} = b_2 \\ \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \\ a_{n1} + a_{n2} + \cdots + a_{nn} = b_n \end{bmatrix}$$

Делают для этого выборку. Затем разделив, на этот коэффициент получаем единицу, т.е. $a_{11}=1$. Затем с помощью этого уравнения исключаем это неизвестное из всех уравнений, используя алгебраические преобразования над матрицами, кроме первого уравнения. Выбрав во втором уравнении неизвестное с коэффициентом отличное от нуля, и разделив аналогично, как в первом, исключаем неизвестные из всех уравнений кроме второго, и т.д. Процесс продолжаем до тех пор, пока не получаем СЛАУ вида

$$\begin{bmatrix} 1 + 0 + \cdots + 0 = b_1^* \\ 0 + 1 + \cdots + 0 = b_2^* \\ \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \\ 0 + 0 + \cdots + 1 = b_n^* \end{bmatrix}$$

В ходе решения СЛАУ выясняем случаи, когда нет решений, множество решений и единственное решение. Указанные утверждения можем сделать при обстоятельствах:

- если число уравнений в СЛАУ меньше неизвестных – множество решений;

- если все коэффициенты в СЛАУ при неизвестных, какого либо уравнения равны нулю, свободный член отличен от нуля – система не совместимо (нет решений);
- если система не удовлетворяет предыдущим пунктам, то имеет единственное решение.

**Задача 5.6.** Пусть задана система линейных алгебраических уравнений (СЛАУ)

$$\begin{bmatrix} x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 = 13 \\ 2x_1 + x_2 + 2x_3 + 3x_4 + 4x_5 = 10 \\ 2x_1 + 2x_2 + x_3 + 2x_4 + 3x_5 = 11 \\ 2x_1 + 2x_2 + 2x_3 + x_4 + 2x_5 = 6 \\ 2x_1 + 2x_2 + 2x_3 + 2x_4 + x_5 = 3 \end{bmatrix}$$

необходимо найти решение СЛАУ методом Жордана-Гаусса. Исследовать все случаи.

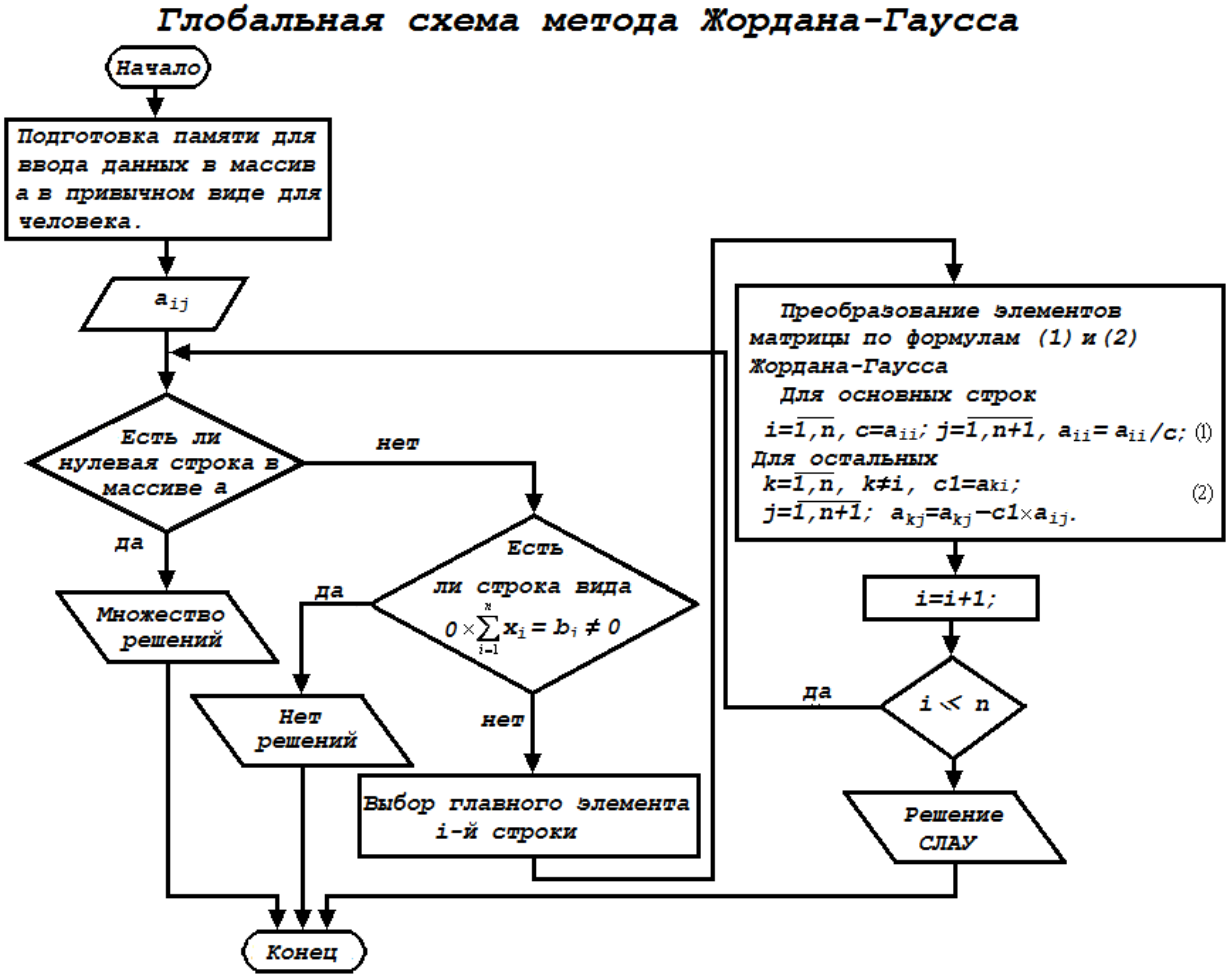



Рис.5.14. Глобальная схема метода Жордана-Гаусса.

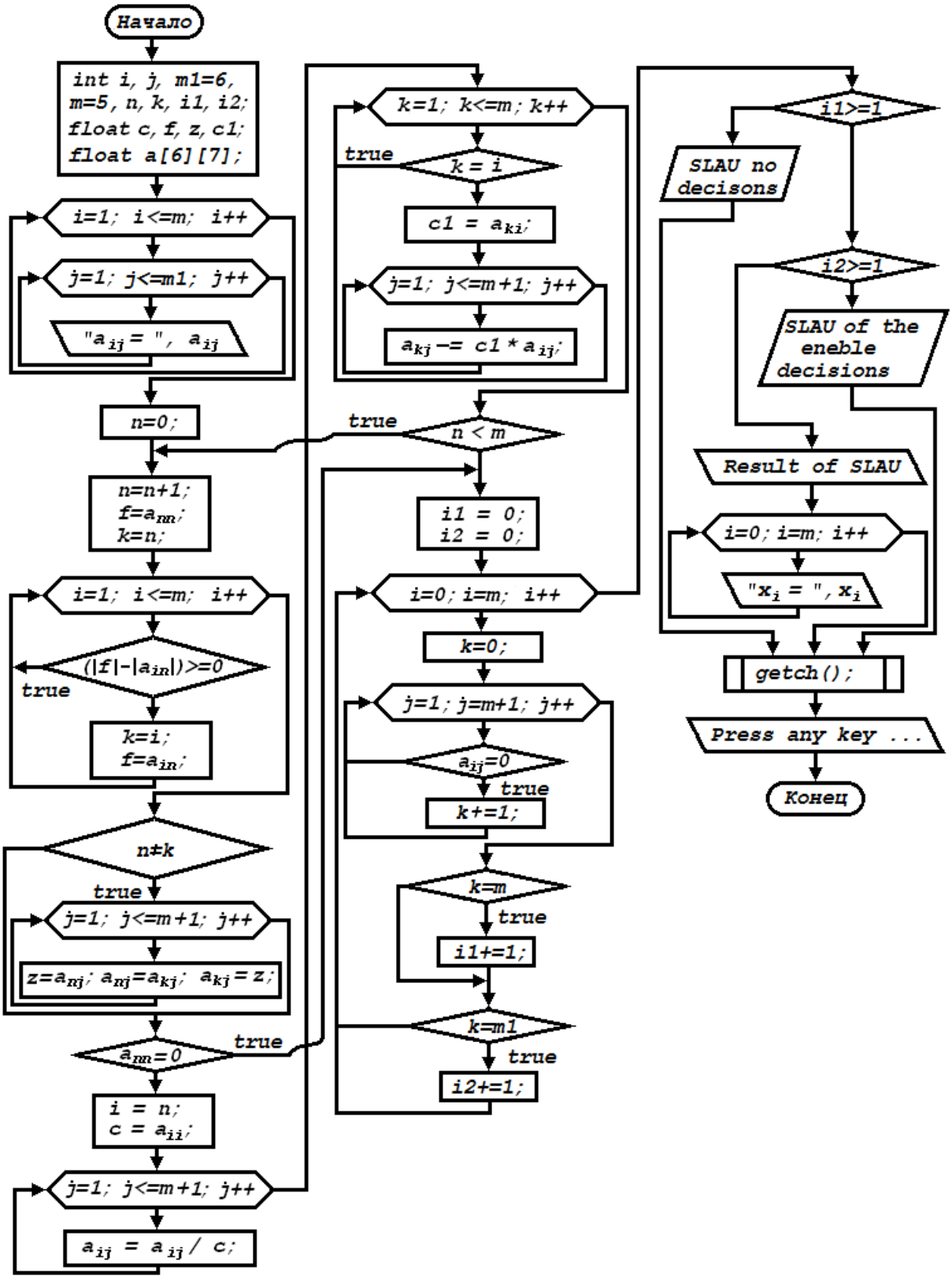



Рис.5.15. Детальная схема блок-схема алгоритма метода Жордана-Гаусса.

```
 //---------------------------------------------------------
#pragma hdrstop
#include "math.h"
#include "stdio.h"
```

```
#include "conio.h"
#pragma argsused
//---------------------------------------------------------
int main(int argc, char* argv[])
{
   int i, j, m1=6, m=5, n, k, i1, i2;
   float c, f, z, c1;
/*** Объявляем массив строк и столбцов на один больше,
чтобы  избавиться от непривычного 0 строки и 0 столбца.***/
  float a[6][7];
/***    Ввод элементов aij и свободных членов СЛАУ    ***/
      for(i=1;i<=m;i++)
       for(j=1;j<=n+1;j++)
        {
          printf("a[%d][%d] = ",i,j);
          scanf("%f", &a[i][j]);
        }
         n=0;
  BeginT: n=n+1;
/***Устанавливаем диагональный элемент максимальным в
столбце ***/
          f=a[n][n];
          k=n;
        for(i=n;i<=m;i++)
        if ((fabs(f)-fabs(a[i][n]))>=0)
           {
             continue;
           }
        else
           {
             f=a[i][n]; k=i;
           }
        if (n!=k)
          for(j=1;j<=m+1;j++)
            {
              z = a[n][j]; a[n][j]=a[k][j]; a[k][j]=z;
            }
/*** Проверяем диагональный элемент равен 0 или нет ***/
         if (a[n][n]==0) goto entry1;
           i=n;
           c=a[i][i];
/***     Строку делим на диагональный элемент     ***/
         for (j=1; j<=m+1; j++)
           a[i][j]=a[i][j]/c;
/***     Производим вычисления по формуле (2)     ***/
        for (k=1; k<=m; k++)
           {
             if (k==i) continue;
             c1=a[k][i];
             for (j=1; j<=m+1; j++)
                a[k][j]-=c1*a[i][j];
           }
        if (n<m) goto BeginT;
```

```
entry1: i1=0; i2=0;
        for (i=n; i<=m; i++)
          {
            k=0;
            for(j=1; j<=m1; j++)
              if (a[i][j]==0) k=k+1;
            if (k==m) i1+=1;
            if (k==m1)i2+=1;
           }
         if (i1>=1) {
           printf("SLAU no decisions");
           }
         else if (i2>=1){
            printf("SLAU of the eneble decisions");
           }
         else
           {
              printf("\nResult of sytem");
               for(i=1;i<=m;i++)
                 printf("\n x[%d] = %f",i,a[i][n+1]);
           }
       endmet: printf("\nPress any key ...");
        getch();
         return 0;
}
//---------------------------------------------------------------
```

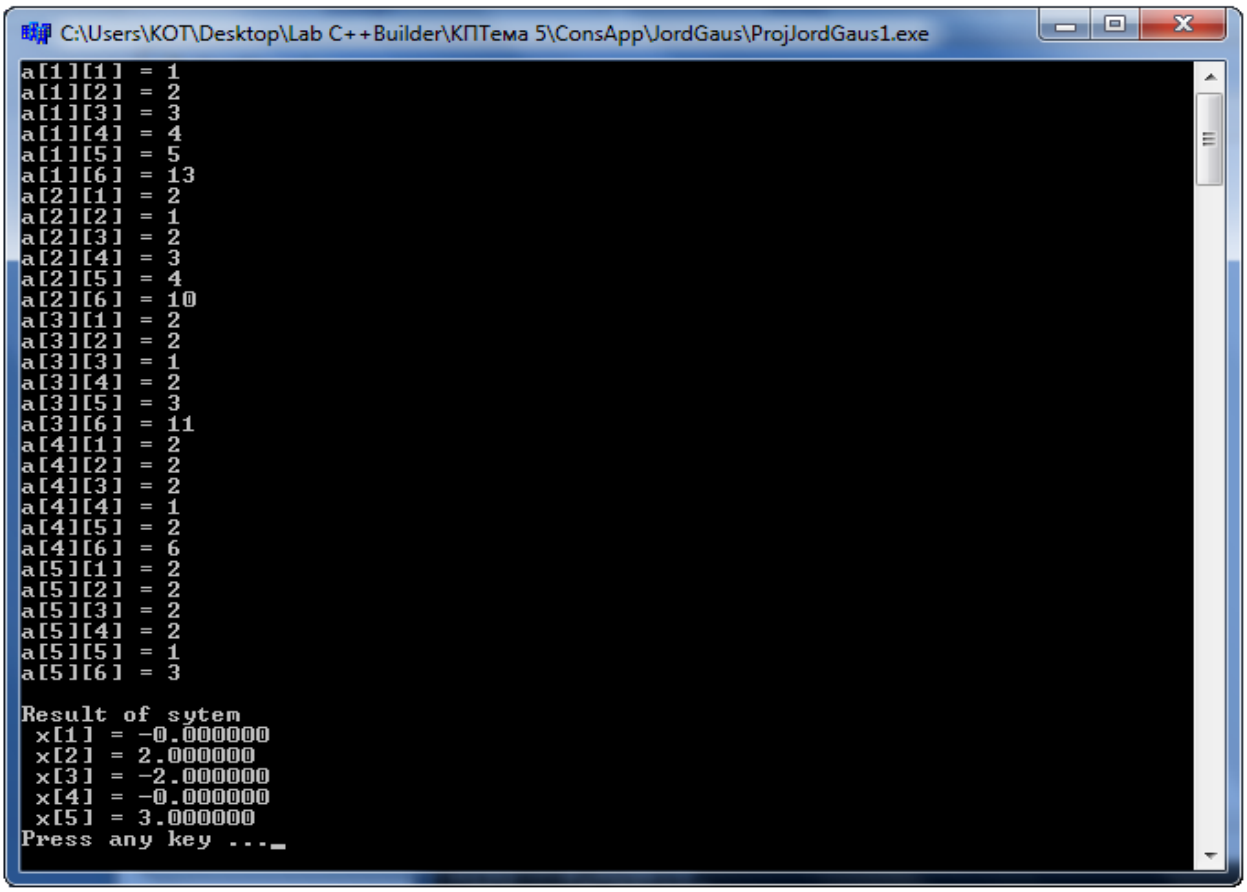



Рис. 5.16. Решение СЛАУ методом Жордана-Гаусса, система линейных алгебраических уравнений совместима, и имеет единственное решение.

Результаты решения СЛАУ приведены на рисунке 5.16.

Рассмотрим случай, когда СЛАУ имеет множество решений.

$$\begin{bmatrix} x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 = 13 \\ 2x_1 + 4x_2 + 6x_3 + 8x_4 + 10x_5 = 26 \\ 2x_1 + 2x_2 + x_3 + 2x_4 + 3x_5 = 11 \\ 2x_1 + 2x_2 + 2x_3 + x_4 + 2x_5 = 6 \\ 2x_1 + 2x_2 + 2x_3 + 2x_4 + x_5 = 3 \end{bmatrix}$$

Как уже вы уже заметили, 2 строка является линейной комбинацией первой строки, и, очевидно, СЛАУ имеет множество решений, можно в ручном режиме не проверять. Результаты компьютерной обработки программного кода приведены на рисунке 5.17.

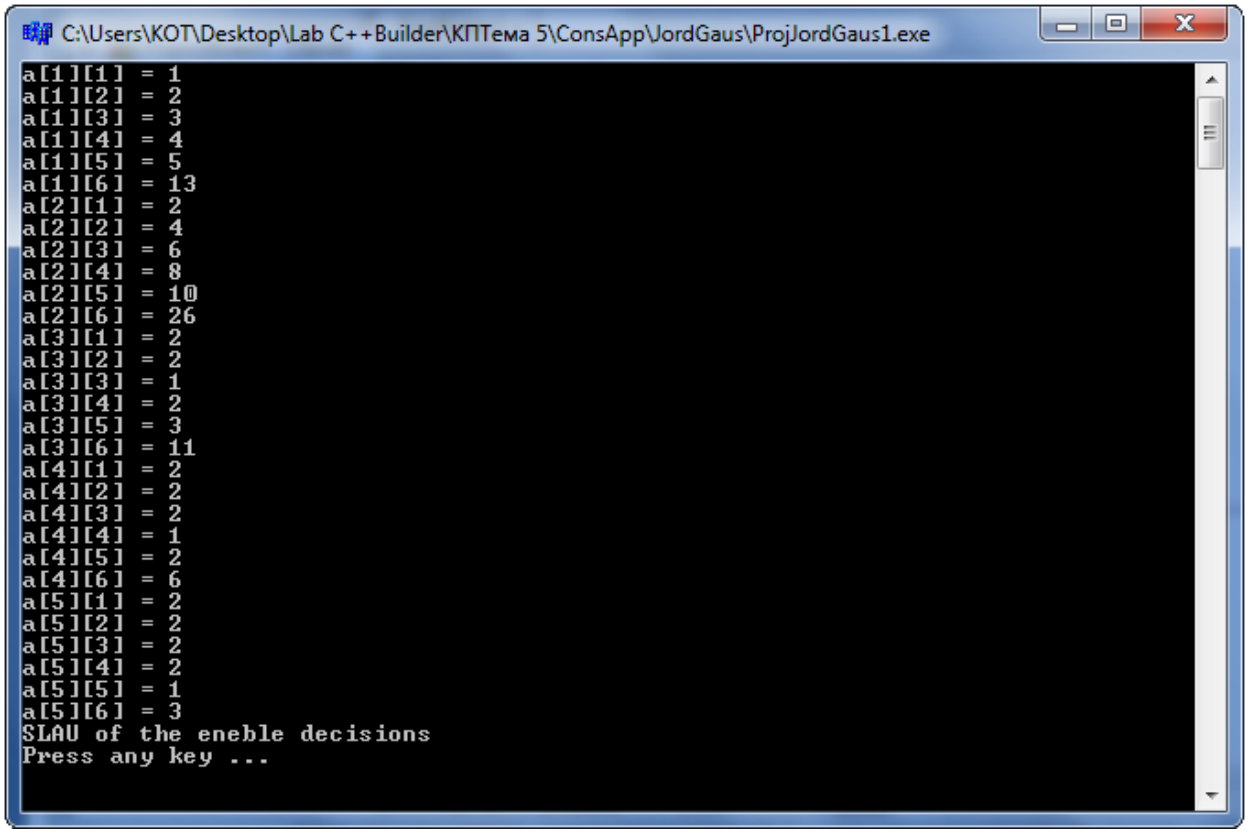



Рис. 5.17. Решение СЛАУ методом Жордана-Гаусса, система линейных алгебраических уравнений совместима, и имеет множество решений.

Приведем теперь случай несовместимости СЛАУ.

$$\begin{bmatrix} x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 = 13 \\ 2x_1 + 4x_2 + 6x_3 + 8x_4 + 10x_5 = 20 \\ 2x_1 + 2x_2 + x_3 + 2x_4 + 3x_5 = 11 \\ 2x_1 + 2x_2 + 2x_3 + x_4 + 2x_5 = 6 \\ 2x_1 + 2x_2 + 2x_3 + 2x_4 + x_5 = 3 \end{bmatrix}$$

В данном случае, очевидно, первые две строки являются линейными комбинациями, а свободные члены нет, следовательно, система не

совместима, т.е. не имеет решений. Результаты приводятся на рисунке 5.18.

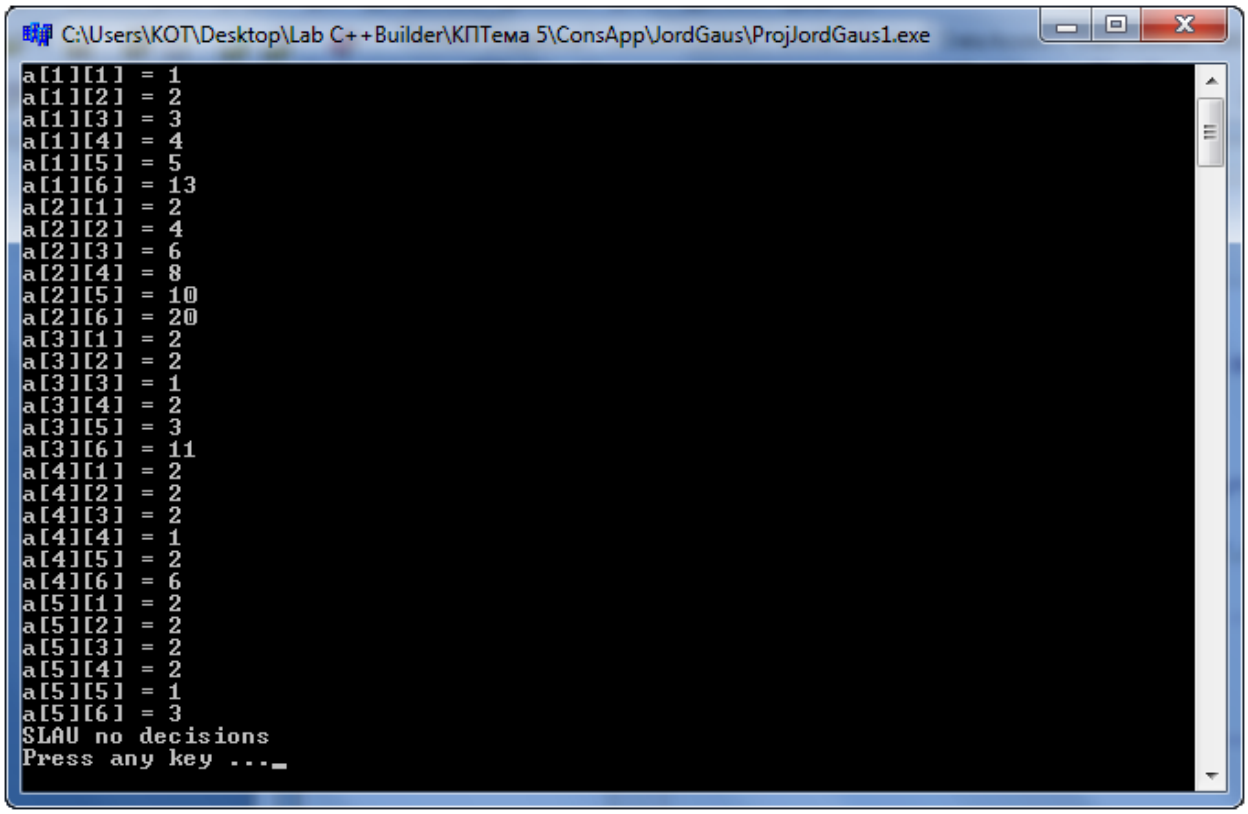



Рис. 5.18. Решение СЛАУ методом Жордана-Гаусса, система линейных алгебраических уравнений не совместима, и не имеет решений.

## 5.5. Структурное программирование

**Пример решения системы линейных алгебраических уравнений методом Гаусса.** В основе структурного программирования лежит методология представления программного кода в виде блоков имеющие иерархическую структуру. Парадигма была предложена в Э. Дейкстрой, и допольнена Н.Виртом.

Основные концепции парадигмы:

1. Всякий программный код может быт представлено в виде структуры, построенной из трёх типов базовых конструкций:

- последовательного исполнения — принципа движения двухколесного велосипеда, т.е. одноразовое выполнение операций в том порядке, в котором они записаны в тексте программы;
- ветвление — альтернативный выбор двух или более условий в выполнении заданий, в зависимости от выполнения некоторого заданного условия;

- цикл — многократное исполнение одной и той же операции до тех пор, пока выполняется некоторое заданное условие (условие продолжения цикла).

1. В коде программы допускаются вложенность базовых конструкций друг в друга произвольным образом, однако средств управления последовательностью выполнения операций не предусматривается.
2. Любая часть программного кода (повторяющиеся части особенно) представляющий логический блок может быт представлено в виде функции или процедуры (общее название подпрограмма). При этом в программный код вставляется вызов подпрограммы, которая при выполнении такого кода инструкции выполняется вызванная подпрограмма, после чего исполнение программы продолжается со следующей за командой вызова подпрограммы инструкции программного кода.

Вот, пожалуй, и все основные концепции парадигма структурного программирования.

Приведем укрупнению схему для решения системы линейных алгебраических уравнений (СЛАУ) методом Гаусса.

Идея метода Гаусса (иногда его называют методом последовательного исключения неизвестных) заключается в том, чтобы исходную линейную систему $\mathbf{A}\vec{\mathrm{X}} = \vec{\mathrm{B}}$ в общем виде его можно изобразить

$$\begin{bmatrix} a_{11} + a_{12} + \cdots + a_{1n} = b_1 \\ a_{21} + a_{22} + \cdots + a_{2n} = b_2 \\ \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \\ a_{n1} + a_{n2} + \cdots + a_{nn} = b_n \end{bmatrix}$$

Затем с помощью преобразований, не изменяющих решение СЛАУ:

- обе части любого уравнения СЛАУ можно умножить или разделить на число отличное от нуля;
- любое уравнение в СЛАУ можно сложить с другим уравнением или из него вычесть другое уравнения той же системы и записать вместо первого, привести к виду $\mathrm{V}\vec{\mathrm{X}} = \vec{\mathrm{B}}^*$ (5.1), где V – верхняя треугольная матрица, $\vec{\mathrm{B}}^*$ -измененный в процессе преобразований вектор правых частей СЛАУ $\mathbf{A}\vec{\mathrm{X}} = \vec{\mathrm{B}}$.

Запишем подробно систему, каким его получим после преобразований приведенных выше

$$\begin{array}{ccccccc} v_{11}x_1 & + & v_{12}x_2 & + & \cdots & + v_{1n}x_n & = b_1^* \\ 0 & + & v_{22}x_2 & + & \cdots & + v_{2n}x_n & = b_2^* \\ \cdots & & \cdots & & \cdots & \cdots & \cdots \\ 0 & + & 0 & + & \cdots & + v_{nn}x_n & = b_n^* \end{array} \quad (1)$$

Система (1) удобно тем, что из нее непосредственно можно определить корни СЛАУ. Действительно из последнего уравнения находим $x_n$ затем из предпоследнего $x_{n-1}$ подставив полученное значение $x_n$ из последнего уравнения и т.д. доходим до $x_1$.

Сведение исходной матрицы к треугольной называют прямым ходом гауссовского исключения, а определение решений системы обратным ходом.

**Задача 5.7.** Решите СЛАУ методом Гаусса

$$\begin{bmatrix} x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 = 13 \\ 2x_1 + x_2 + 2x_3 + 3x_4 + 4x_5 = 10 \\ 2x_1 + 2x_2 + x_3 + 2x_4 + 3x_5 = 11 \\ 2x_1 + 2x_2 + 2x_3 + x_4 + 2x_5 = 6 \\ 2x_1 + 2x_2 + 2x_3 + 2x_4 + x_5 = 3 \end{bmatrix}$$

Приведем сначала укрупненную блок - схему

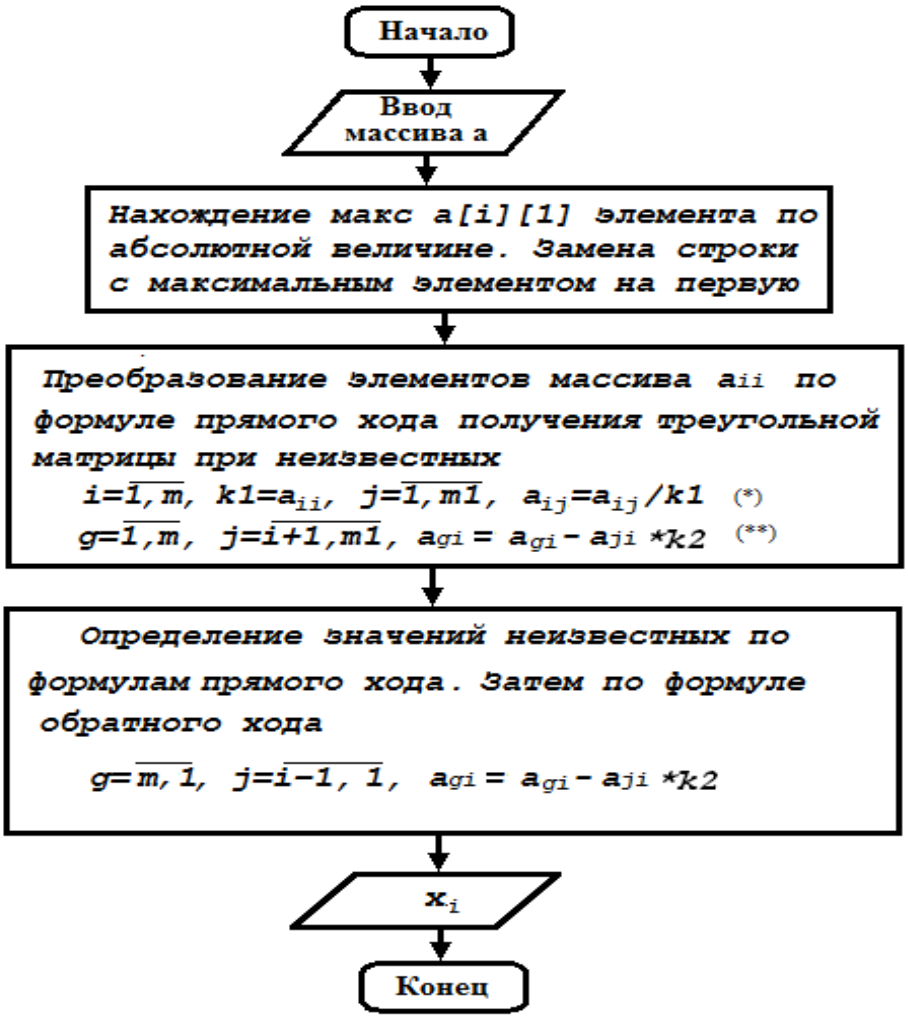



Рис. 5.19. Укрупненная блок схема метода Гаусса

Приведем подробную блок-схему, где используем методы структурного программирования на рис. 5.20.

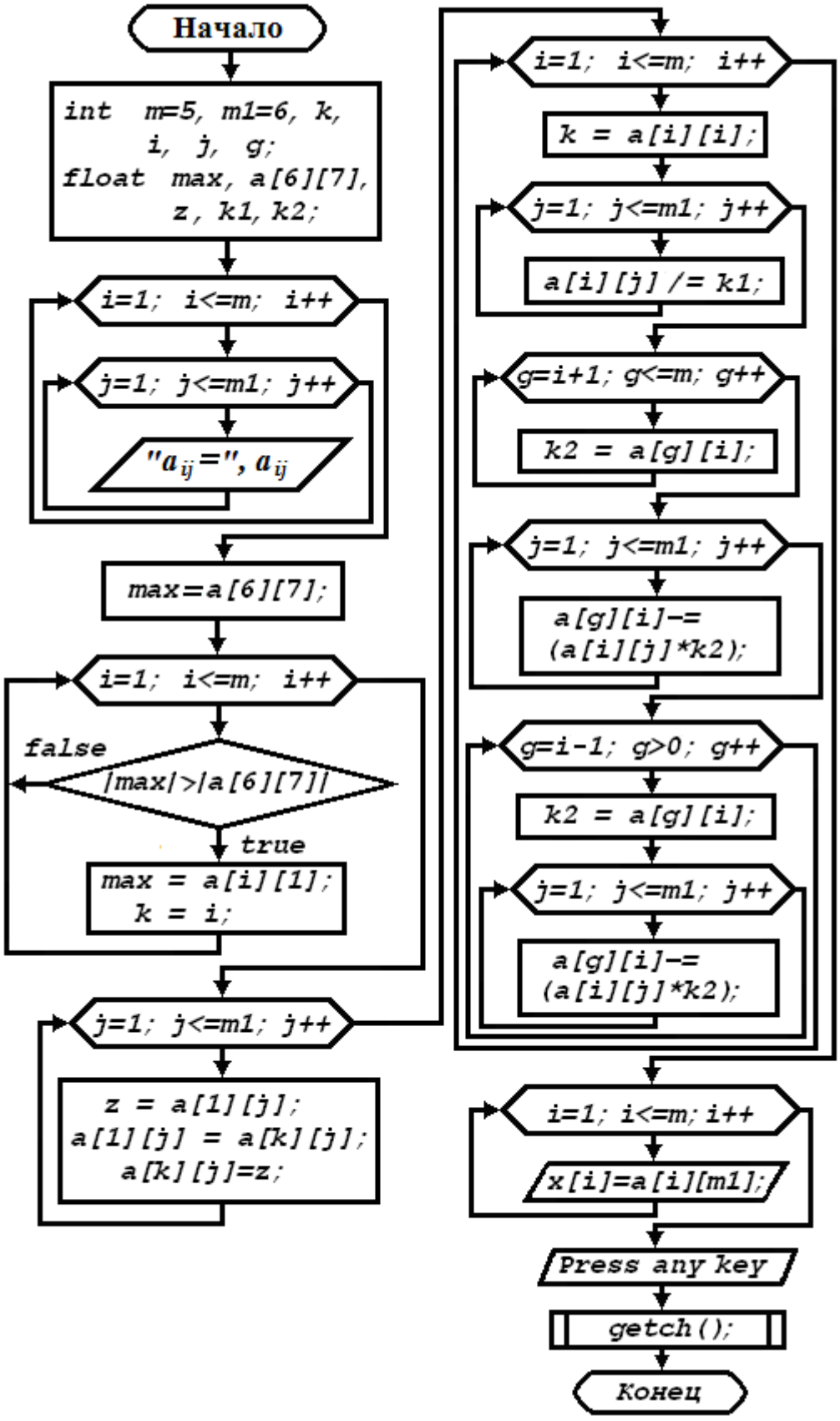



Рис.5.20. Подробная схема решения СЛАУ методом Гаусса для структурного стиля программирования.

```
//---------------------------------------------------------
#pragma hdrstop
#include "math.h"
#include "conio.h"
#include "stdio.h"
#pragma argsused
int main(int argc, char* argv[])
{
   int i,j,g, m=5,m1=6, k;
   float z, k1,k2, max;
   float a[6][7];
    for (i=1;i<=m;i++)
       for (j=1; j<=m1; j++)
        {
            printf(" a[%d][%d]= ",i,j);
            scanf("%lf", &a[i][j]);
        }
    max = a[1][1];
    for (i=1; i<=m; i++)
        if (fabs(a[i][1])> max)
        {
            max = a[i][1];
            k = i;
        }
    for (j=1;j<=m1;j++)
    {
        z = a[1][j];
        a[1][j] = a[k][j];
        a[k][j] = z;
    }
    for (i=1;i<=m;i++)
    {
        k1 = a[i][i];
        for (j=1; j<=m1; j++)
            a[i][j] = a[i][j] / k1;
        for (g=i+1;g<=m;g++)
        {
            k2 = a[g][i];
            for (j=1; j<=m1; j++)
                a[g][j] = a[g][j] - (a[i][j] * k2);
        }
        for (g=i-1;g>0;g--)
        {
            k2 = a[g][i];
            for (j=1; j<=m1; j++)
                a[g][j] = a[g][j] - (a[i][j] * k2);
        }
    }
    for (i=1;i<=m; i++)
        printf("\nX%d = %lf ",i,a[i][m1]);
getch();
        return 0;
}
```

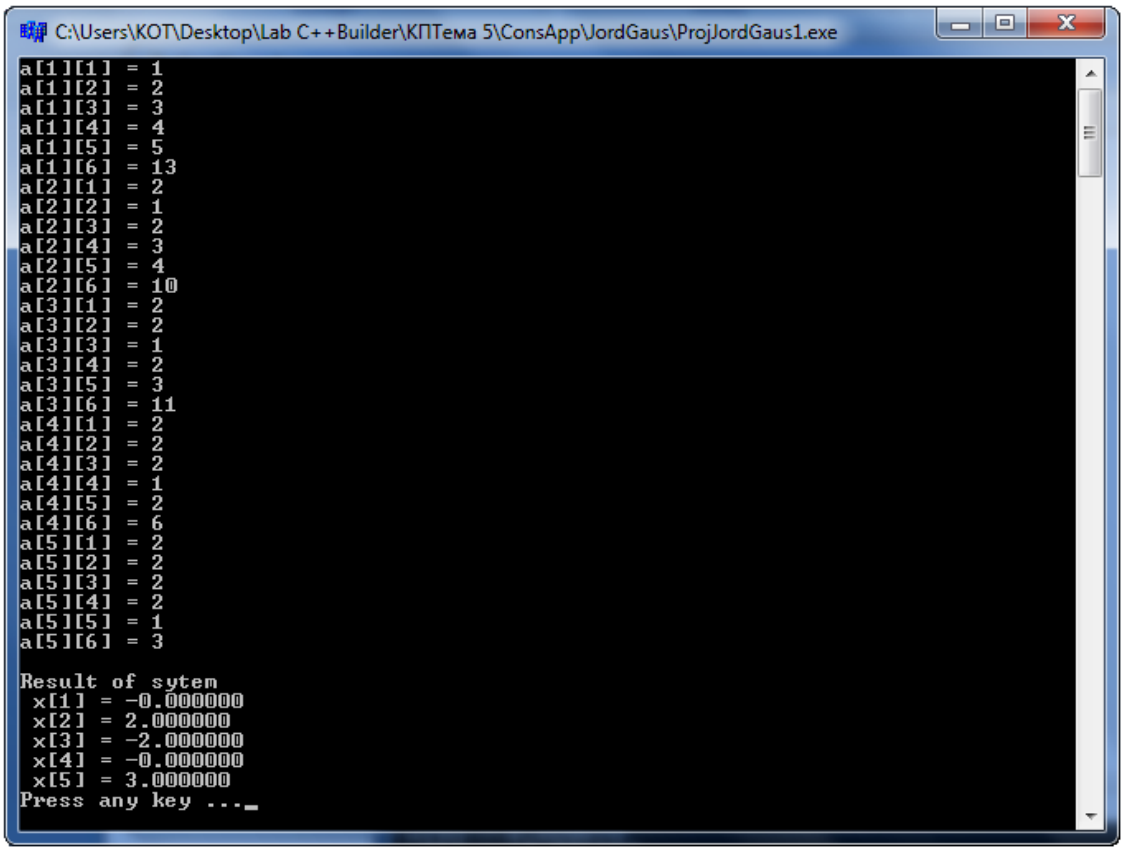



Рис. 5.21. Решение СЛАУ методом Жордана-Гаусса, система линейных алгебраических уравнений совместима, и имеет единственное решение.

**Указатели и ссылки.** Напоминаем, об указателях мы говорили при изучении лабораторной работы №2, и в этой теме также, теперь остановимся более подробно. Обычно начинающие пользователи их избегают, попытаемся искоренить этот пробел.

Все объекты программы, будь то переменная базового или же производного типа, занимает в памяти определенную область памяти, которая определяется ее адресом местоположения.

Вышеуказанный, процесс происходит следующим образом, при объявлении переменной для нее резервируется место в памяти, размер которого зависит от типа данной переменной, а для доступа к содержимому объекта служит его имя (идентификатор). Для того чтобы узнать адрес конкретной переменной, служит унарная операция взятия адреса. При этом перед именем переменной ставится знак амперсанда (&).

**Задача 5.8.** Следующий ниже пример программы выведет на печать сначала значение переменной Var, а затем ее адрес:

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

```
#include "stdio.h"
#pragma argsused
int main(int argc, char* argv[])
{
unsigned int Var = 69;

int main(int argc, char* argv[])
{
   long int var = 69;
   long int *p_var =&var;
   printf("var = %d \n", var);
   printf("Adres = %d\n", &var);
   printf("Press any key...");
   getch();
        return 0;}
//---------------------------------------------------------------
```

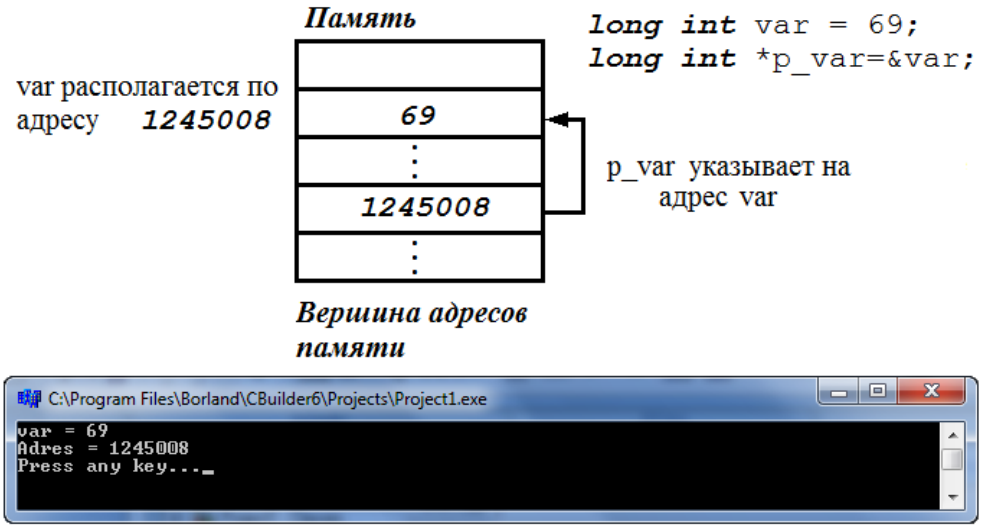



Рис. 5.22. Переменные в памяти

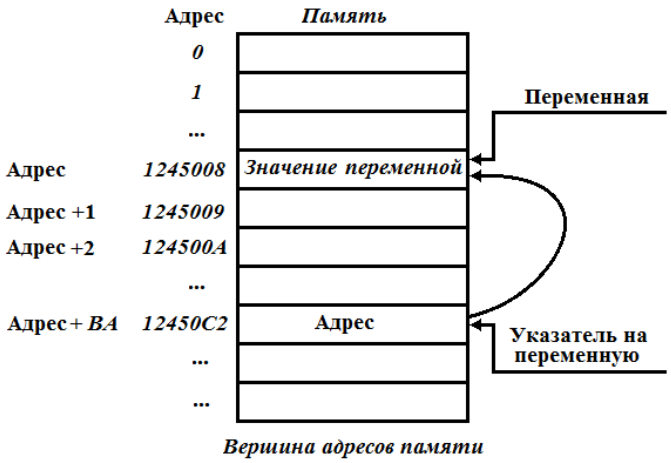



Рис. 5.23. Суть указателя

Результат адреса может отличаться даже при повторном запуске программы, так как невозможно предугадать, по какому адресу начнут размещаться переменные - это зависит от многих факторов. Важно другое: разница в адресах первой, второй и других последующих переменных в коде программы всегда будет одинаковым, в зависимости от размера занимаемой им памяти (см. рис. 5.23).

***Указатель*** *— это переменная, которая содержит адрес другого объекта.*

Этим объектом может быть некоторая переменная, *динамический объект* или функция. Говорят, что указатель *ссылается* на соответствующий объект. Хотя адрес, по существу — 32-битное целое число, определяющее положение объекта в виртуальной памяти программы, указатель является не просто целым числом, а специальным типом данных. Он «помнит», на какого рода данные ссылается. Напоминаем, формат объявления указателя выглядит так:

***тип_указываемого_объекта*** **идентификатор_указателя [= значение];*

Здесь ***тип_указываемого_объекта*** определяет тип данных, на которые ссылается указатель с именем идентификатор. Символ «звездочка» (*) сообщает компилятору, что объявленная переменная является указателем и не зависимо от того, сколько памяти требуется отвести под сам объект, для указателя резервируется два или четыре байта в зависимости от используемой модели памяти.

Вот примеры объявлений:

```
int *pIntVarable; // Указатель на целое.
double *pDouble = doubleVarable; // Инициализация указателя
// на double.
char *arrStr[10]; // Массив указателей на char.
char (*arrStr) [10][10]; // Указатель на матрицу char.
```

Последний пример довольно специфичен. Подобные конструкции применяются в объявлении параметров функций, передающих многомерные массивы неопределенного размера.

Чтобы получить доступ к объекту, на который указатель ссылается, где последний *разыменовывают,* применяя операцию-звездочку. Например. *pDouble будет представлять значение переменной, на которую ссылается

```
pDouble:
double doubleVarable = 2.718281828;
double *pDouble = SdoubleVarable;
printf("Значение самого указателя (адрес): %p",
pDoubie) ;
printf("Число, на которое он ссылается: %f",
*pDouble);
```

Как мы уже говорили в предыдущих разделах, указатели используются при обработке строк, а также для передачи функциям параметров, значения которых могут ими изменяться (передача по ссылке). Но главная «прелесть» указателей в том, что они позволяют

создавать и обрабатывать ***динамические структуры данных***. В языке С можно выделить память под некоторый объект не только с помощью оператора объявления, но и динамически, во время исполнения программы. Объект создается в свободной области виртуальной памяти функцией `malloc()`. Вот пример (предполагается, что переменные объявлены так, как выше):

```
    pDouble = malloc(sizeof(double));   //Динамическое выделение памяти.
    *pDouble = doubleVar;  //Присвоение значения динамическому объекту.
    printf("Значение динамического объекта: %f", *pDouble);
    free(pDouble);// Освобождение памяти.
```

Аргументом `malloc()` является размер куска памяти, которую нужно выделить; для этого можно применить операцию ***sizeof***, возвращающий, размер (в байтах) переменной или типа, указанного в качестве операнда.

Функция **`malloc()`** возвращает значение типа ***`void*`*** — «пустой указатель». Это указатель, который может указывать на данные любого типа. Такой указатель нельзя разыменовывать, поскольку неизвестно, на что он указывает — сколько байтов занимает его объект и как их нужно интерпретировать. В данном случае операция присваивания автоматически приводит значение `malloc()` к типу ***`double*`***. Можно было бы написать в явном виде

```
pDouble = (double*)malloc(sizeof(double));
```

Если выделение памяти по какой-то причине невозможно, malloc () возвращает NULL, нулевой указатель. На самом деле эта константа определяется в stdlib.h как целое — «длинный нуль»:

```
#define NULL OL
```

Хорошо ли вы поняли смысл различия двух последних примеров? В первом из них указателю pDouble присваивается адрес переменной doubleVariable. Во втором указателю присваивается адрес динамически созданного объекта типа double; после этого объекту, на который ссылается pDouble, присваивается значение переменной doubleVariable. Создается динамическая копия значения переменной.

Память, выделенную `malloc ()`, следует освободить функцией `free()`, если динамический объект вам больше не нужен. Возьмите это себе за правило и не полагайтесь на то, что система Windows

автоматически уничтожает все динамические объекты программы по ее завершении.

Перед тем, как разыменовывать указатель, его нужно обязательно инициализировать, либо при объявлении, либо путем присвоения ему адреса какого-либо объекта, возможно, динамического — как в последнем примере. Аналогично, если к указателю применяется функция free(), он становится недействительным и не ссылается больше ни на какие осмысленные данные. Чтобы использовать его повторно, необходимо снова присвоить ему адрес некоторого объекта.

Разыменование нулевого указателя также приводит к ошибке. Поэтому при работе с объектами, создаваемыми с помощью malloc (), обычно всегда проверяют, не возвратила ли эта функция нулевое значение.

**Указатель на функцию.** Можно объявить, инициализировать и использовать указатель на функцию. В вызовах API Windows часто применяют, например, “возвратно-вызываемые функции”. В вызове API в качестве аргумента в этом случае употребляется указатель на соответствующую функцию.

Вот пример, из которого все станет ясно.

```
/************************ Некоторая функция:******/
void ShowString(char *s)
     {
          printf (s);
     }
/********************** Главная функция:*********/
int main(void) {
void (*pFunc)(char*); // Объявление указателя на функцию.
          pFunc = ShowString; // Инициализация указателя
       адресом
          // функции.
          (*pFunc)("Calling a pointer to function!\n");
     return 0;
     }
```

Вы, возможно, обратили внимание, что в примере указателю присваивается значение, представленное просто именем функции без скобок со списком параметров. То есть «значение», представленное именем функции, имеет тот же тип, что и объявленный здесь указатель. Поэтому и вызвать функцию через указатель можно было бы проще:

```
pFunc("Calling a pointer to function!\n");
```

Соотношение между указателями и функциями примерно такое же, как между указателями и массивами, о чем говорится в следующем разделе.

**Указатели и массивы.** Между указателями и массивами в С существует тесная связь. Имя массива без индекса эквивалентно указателю на его первый элемент. Поэтому можно написать:

```
int iArray[4];
int *piArr;
piArr = iArray;//piArr указывает на начальный
элемент iArray.
```

Последнее эквивалентно

```
piArr = &iArray[0];
```

И наоборот, указатель можно использовать подобно имени массива, г. е. индексировать его. Например, piArr [3] представляет четвертый элемент массива iArray [ ] .

К указателю можно прибавлять или отнимать от него целочисленные выражения, применять операции инкремента и декремента. При этом значение указателя изменяется в соответствии с размером объектов, на которые он указывает. Так, `(piArr + 2)` указывает на третий элемент массива. Это то же самое, что и `&iArray [2]`. Когда мы прибавляем к указателю единицу `(piArr++)`, адрес, который в нем содержится, в действительности увеличивается на 4 — размер типа int.

*Таким образом, в выражениях с указателями и массивами можно обращаться одинаково. Следует только помнить, что объявление массива выделяет память под соответствующее число элементов, а объявление указателя никакой памяти не выделяет, вернее, выделяет память для хранения значения указателя — некоторого адреса. Компилятор по-разному рассматривает указатели и массивы, хотя внешне они могут выглядеть очень похоже.*

Однако возможности, которые раскрываются перед программистом благодаря указателям, выявляются в полной мере лишь при работе с динамическими структурами данных, о которых мы поговорим в следующем разделе.

**Типы, определяемые пользователем.** Встроенные типы данных, указатели и массивы образуют основу для представления и обработки информации на языке С. Подлинная же сила языка состоит в том, что он позволяет пользователю (под «пользователем» в подобного рода выражениях понимается программист) самому определять наиболее подходящие для конкретной задачи типы, способные адекватно представлять сложно структурированные данные реального мира. Эти средства С мы начнем изучать со сравнительно простого вспомогательного оператора typedef.

**Переименование типов.** Любому типу в С можно присвоить простое имя или переименовать его. Это делается с помощью ключевого слова typedef:

*typedef тип новое_имя_типа;*

или

*typedef тип новое_имя_типа [размер_массива][...];*

для типов-массивов. (Квадратные скобки здесь означают не необязательность синтаксического элемента, а «настоящие» скобки.) Кроме того, можно вводить имена для типов указателей на функцию и т. п. Формально описать все возможные typedef довольно сложно, поэтому мы этого делать не будем. Вообще следует руководствоваться таким правилом: если вы объявляете объект как принадлежащий к определенному в typedef типу, имя объекта нужно подставить вместо *нового_имени_типа.* Убрав typedef, вы получите эквивалентное объявление объекта. Вот примеры:

```
typedef short Arrlndex;
                //Псевдоним для short.
typedef char MessageStr[80];
                //Имя типа для массивов char[80]
typedef int *IPtrFunc(void);
                //Функция, возвращающая указатель на int
typedef int (*IFuncPtr)(void);
                //Указатель на функцию, возвращающую int
```

В общем, typedef является просто средством упрощения записи операторов объявления переменных.

**Перечислимые типы.** Ключевое слово enum позволяет описать *перечислимый* тип, представляющий переменные, которые могут принимать значения из заданного набора целых именованных констант. Определение перечислимого типа выглядит так:

***enum имя-имена {имя_константы [= значение], ...};***

Значение равно по умолчанию нулю для первого из перечислителей (так обычно называют определяемые в enum константы). Любая другая константа, для которой значение не указано, принимается равной значению предыдущей константы плюс единица.

Например:

```
enum Status
{
Success = 1,
Wait, Proceed,
Error = -1
};
```

*В операторе* ***enum*** *после закрывающей фигурной скобки можно сразу объявить несколько переменных данного типа:*

***enum*** *имена {список_констант} переменная[, ...];*

Нужно иметь в виду, что *имя-имена* не является настоящим именем типа. Именем типа будет в вышеприведенном примере enum Status. Соответственно переменные должны объявляться как

```
enum Status ProclStatus, Proc2Status;
```

Однако всегда можно воспользоваться ключевым словом typedef и ввести для перечисления подлинное новое имя. Обычно это делается сразу:

*typedef enum имена {список_констант } имя_типа;*

Предыдущее объявление можно переписать так:

```
typedef enum _Status {
Success = 1,
Wait, Proceed,
Error = -1 } Status;
```

Тогда Status будет полноценным именем перечислимого типа. (Обратите внимание, что для имени мы указали имя _Status. Это обычная практика.)

**Структуры.** Массивы позволяют обращаться с набором логически связанных однотипных элементов как с единым целым. Если же требуется хранить набор разнородных, но логически связанных данных, описывающих, например, состояние некоторого объекта реального мира, используются *структуры.* Синтаксис структуры имеет такой вид:

*struct имена [список элементов] [переменные];*

*Список_элементов* состоит из объявлений, аналогичных объявлениям переменных. Объявления элементов оканчиваются точкой с запятой. Вот простой пример структуры, предназначенной для хранения основных сведений о человеке:

```
struct Person (
     char lastName[32]; // Фамилия.
     char firstName[32]; // Имя.
     Sex sex;
     // Пол: перечислимый тип
     // (male, female}.
     short age; // Возраст.
long phoneNum; // Телефон как длинное целое.
}
aPerson; // Объявляет переменную типа
// struct Person.
```

Как видите, все довольно просто. Структура группирует различные данные, относящиеся к конкретному человеку. Как и в случае перечислений, в определении структуры можно сразу объявить переменные структурного типа, указав их имена после закрывающей фигурной скобки. Аналогично именем типа является struct *имена,* и его можно сразу переопределить с помощью ключевого слова typedef.

Для доступа к отдельным элементам структуры имеются две операции: точка и стрелка, за которыми следует имя элемента. Какую из них следует применять, зависит от того, имеете ли вы дело с самой переменной-структурой или у вас есть только указатель на нее, как это имеет место в случае динамических объектов. С именем переменной применяется точка, с указателем — стрелка. Имея в виду предыдущее определение, можно было бы написать:

```
struct Person *pPerson - SaPerson;
// Указатель на структуру.
aPerson.age = atol(ageStr);
// Записать в структуру
// возраст aPerson.sex - male; и т.д.
pPerson->phoneNum = atol(phoneStr); //
/* Напечатать имя и фамилию (предполагается, что они уже
инициализированы).*/
printf("%s %s\n", pPerson->firstName, pPerson->lastName);
```

**Битовые поля.** В качестве элементов структуры можно определять *битовые поля.* Для них задается *ширина поля* в битах, и компилятор отводит под элемент ровно столько бит, сколько указано. Несколько битовых полей может быть таким образом упаковано в одном слове. Синтаксис битового поля:

*тип [имя поля]: ширина поля;*

*Тип* поля может быть ***int*** или ***unsigned int***. Доступ к битовым полям осуществляется так же, как и к регулярным элементам структуры. Если *имя_поля* отсутствует, место под поле отводится, но оно остается недоступным. Это будут просто «заполняющие» биты.

Битовые поля применяются либо там, где необходима плотная упаковка информации (как это бывает при передаче функции некоторого набора логических флагов), либо, например, для отображения регистров внешнего устройства, которые часто бывают организованы как совокупность небольших полей и отдельных битов.

**Объединения.** *Объединения,* определяемые с помощью ключевого слова union, похожи по своему виду на структуры:

*union имена {список_элементов] [переменные];*

Отличие состоит в том, что все элементы объединения занимают одно и то же место в памяти, они перекрываются. Компилятор отводит под объединение память, достаточную для размещения наибольшего элемента.

Объединения полезны, когда требуется обеспечить своего рода «полиморфное поведение» некоторого объекта. Например, вы хотите определить тип, реализующий представление различных геометрических фигур — прямоугольников, окружностей, линий, многоугольников. В зависимости от того, чем конкретно является данная фигура, для ее описания необходимы различные наборы значений. Круг описывается иначе, чем многоугольник и т. п. И вот из структур, описывающих различные фигуры, можно в свою очередь составить обобщенный тип-объединение, который будет обрабатываться различно в зависимости от значения специального поля, определяющего род фигуры. В то же время на все фигуры можно будет ссылаться через указатели одного и того же типа, что, в частности, позволит составлять динамические связанные списки из любых фигур.

Доступ к элементам объединения (их иногда называют *разделами)* осуществляется так *же,* как и в структурах, — посредством точки или стрелки, за которыми следует имя раздела.

В заключение разговора о типах, определяемых пользователем, приведем пример законченной программы. В ней определяется тип структуры, способной хранить данные различных “графических объектов”. В программе реализованы всего два их вида — прямоугольник и текстовая метка.

Листинг 5.9. Демонстрация работы со структурами

```
/* Struct.c: Структуры и объединения. */
#pragma hdrstop
#include <stdio.h>
#include <conio.h>
#include <string.h>
/* Тип для определения вида объекта */
typedef enum {rect=1, labi} Type;
/*Структура для хранения прямоугольников и текстовых
меток.*/
typedef struct _GForm { Type type;
struct _GForm *next;
/* Указатель для связанного списка. */
union {
/* Анонимное объединение. */
struct {
/* Прямоугольник. */
```

```
int left, top;
int right, bottom;
} rect;
struct {
/* Текстовая метка. */
int x, y;
char text [20];
} labi;
} data;
} Gform;

/* Функция вывода данных объекта. */
void ShowForm(Gform *f)
{
switch (f->type) {
case rect:
/* Прямоугольник. */
printf("Rectangle: (%d, %d) (%d, %d)\n",
f->data.rect.left, f->data.rect.top,
f->data.rect.right, f->data.rect.bottom);
break;
case labi:
/* Метка. */
printf('Text label: (%d, %d) \"%s\"\n",
f->data.labi.x, f->data.labi.y,
f->data.labi.text);
}
}
int main(void)
{
GForm form1, form2;
          /* Инициализация первого объекта. */
form1.type = rect;
form1.data.rect.left = 50;
form1.data.rect.top = 25;
form1.data.rect.right = 100;
form1.data.rect.bottom = 75;
          /* Инициализация второго объекта. */
form2.type = labi;
form2.data.labi.x = 60;
form2.data.labi.y = 40;
strcpy(form2.data.labi.text, "This is a Label!");
                  /* Распечатка... */
ShowForm(&form1);
ShowForm(&form2);
printf("\nPress any key...");
getch() ;
return 0;
}
```

Работу программы иллюстрирует рис. 5.24.

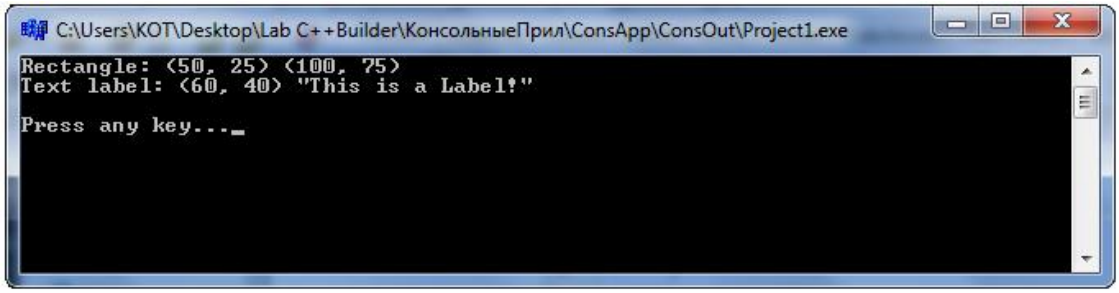



Рис. 5.24. Программа Struct.c (Project2)

Обратите внимание, что перечисления, структуры и объединения могут быть анонимными, т е. не иметь имен-этикеток.

Внимательно рассмотрите определение типа Gform:

```
typedef struct _GForm { Type type;
  struct Gform *next;
          /* Указатель для связанного списка. */
    union {
                /* Анонимное объединение. */
      struct {
                    /* Прямоугольник. */
         int left, top;
         int right, bottom;
         } rect;
      struct {
                   /* Текстовая метка. */
         int x, y;
         char text[20] ;
         } labl;
    } data;
} GForm;
```

Структура _Gform имеет, как таковая, три элемента: type, next (не используется) и data. Последний является анонимным объединением разделов rect и labl, каждый из которых, в свою очередь, является анонимной структурой. Элементы первой хранят значения координат верхнего левого и правого нижнего углов прямоугольника; элементами второй являются координаты, задающие положение текста, и сама текстовая строка. Получаются довольно длинные выражения для доступа к элементам данных (forml.data.rect.bottom).

## 5.6. Индивидуальные задания
## Одномерные массивы

Составить блок- схему алгоритма и программу на языке Borland C.

Результаты вывести на монитор.

1) В массиве X[10] найти количество отрицательных элементов и заменить их значениями по абсолютной величине.
2) Найти сумму и среднеарифметическую сумму значение положительных элементов, и номер первого положительного элемента массива X[10].
3) Найти минимальный элемент массива X[10] и его порядковый номер, а также отпечатать последний отрицательный элемент.
4) Найти порядковые номера элементов и их сумму в массиве X[10] удовлетворяющих условию $0 \le X_i < 10$.
5) Для массива A[10], где $A_i=1/X_i$ найти номера и сумму элементов I ,при котором $A_i<5$, если X меняется от 0.1 до 1 с шагом 0.1.
6) Вычислить массив чисел $Z_i=(X_i+A_i)/2$, если I=1…20; $A_i=\mathrm{Sin}X_i$, а $X_i$ изменяется от 0.1 до 2 с шагом 0.1.
7) Вычислить массив чисел, $z_i = \dfrac{e^{x_i} + y^{2_i}}{e^{y_i}}$ если $X_i$ изменяется от 2 с шагом 0.5, а $Y_i$- от 1 с шагом 0.25 (I=1….10).
8) Переписать подряд положительные элементы- массива X[10] в массив Y.
9) Записать подряд в массив N четные элементы целочисленного массива X[10].
10) Записать в массив Y пять первых положительных элементов массива X[10].
11) Вычислить $a = \dfrac{e^{x_i}}{x_i}$ $i$=1…10, Найти i, при которой элемент массива A(10) $a_i < k$, где некоторое число вводимое с клавиатуры.
12) Найти сумму элементов массива A[10], имеющих нечетные индексы, а элементы с четными индексами записать в массив A.
13) Элементы массива A[10], стоящие на четных местах заменяя их по абсолютной величине, записать подряд в массив Y, а стоящие на нечетных местах - в массив Z.
14) Переписать положительные элементы массива X[15] в массив Y , а отрицательные- в массив Z.
15) Элементы целочисленного массива M[10], кратные 3, переписать в массив L подряд.

16) Элементы массива X[20] записать в массив Y[20] в обратном порядке.
17) Вычислить сумму положительных элементов массива X[10] и их среднеарифметическое.
18) Вычислить среднеарифметическую сумму элементов массива A[10], удовлетворяющих условию $1<A\leq 2$.
19) Вычислить среднеарифметическую сумму элементов целочисленного массива N[10], кратных 7.
20) Для массива X[10] определить количество элементов, равных 0, и их порядковые номера.
21) Найти максимальный элемент массива X[10] и его порядковый номер.
22) Найти минимальный элемент массива X[10] и его порядковый номер.
23) Вычислить среднеарифметическую сумму положительных элементов массива X[10], стоящих на четных местах.
24) Для массивов X[10] и Y[10] найти наибольшее значение $(X_i+Y_i)$ и значение I при этом.
25) Элементы целочисленного массива M[15]; кратные 13, переписать подряд в массив Y.
26) Найти сумму и количество элементов массива X[10] таких, что $a<X_i<b$.
27) Найти номера элементов массива X[10], для которых $X_i>b$ и записать их в целочисленный массив N.
28) Дан массив X[10], найти среднеарифметическую сумму элементов, удовлетворяющих условию $0<X_i<2$.
29) Переписать массив X[10] в массив Y[10] таким образом, чтобы сначала стояли элементы с четными индексами, потом- с нечетными.
30) Дан массив X[10], найти $X[I]_{max}$ среднеарифметические числа положительных и отрицательных элементов отдельно.

**Функции**

По заданным вещественным значениям α, β и целому n получить:

а) значения $y_i=\Psi(\alpha,\beta,I)$ $(I=1,2,\ldots,n)$, где ψ-заданная функция;

б) значения $u=(y_1,y_{2,\ldots}y_n)$, где φ-заданная функция.

Функции ψ и φ, а также α, β и n определяются вариантом задания.

Предусмотреть в программе построчный вывод массива $\{y_i\}$, а также значение u на монитор и в файл в текущем каталоге с именем Res.txt.

**Варианты задания.**

| № | Ψ | φ | α | β | n | № | Ψ | φ | α | β | n |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 10 | 1 | 12.3 | 2.4 | 9 | 16 | 1 | 1 | 3.5 | 4.2 | 13 |
| 2 | 9 | 2 | 1.23 | 6.4 | 10 | 17 | 4 | 3 | 6.2 | 4.52 | 12 |
| 3 | 8 | 3 | 14.5 | 8.3 | 15. | 18 | 2 | 2 | 7.5 | 5.8 | 6 |
| 4 | 7 | 4 | 10.5 | 5.2 | 10 | 19 | 3 | 4 | 3.3 | 6.6 | 7 |
| 5 | 6 | 5 | -3.5 | 4.5 | 5 | 20 | 5 | 1 | 5.6 | 7.4 | 20 |
| 6 | 5 | 6 | 6.8 | 2.2 | 8 | 21 | 7 | 5 | 5.3 | 5.8 | 8 |
| 7 | 4 | 7 | -3.2 | 3.5 | 6 | 22 | 9 | 7 | 6.8 | 3.4 | 11 |
| 8 | 3 | 8 | -1.6 | 1.6 | 14 | 23 | 2 | 9 | 8.3 | 4.6 | 16 |
| 9 | 2 | 9 | 2.5 | 5.3 | 20 | 24 | 4 | 8 | 4.2 | 6.8 | 14 |
| 10 | 1 | 10 | 5.7 | 2.4 | 10 | 25 | 10 | 10 | 5.8 | 9.2 | 9 |
| 11 | 3 | 6 | -5.7 | 9.3 | 13 | 26 | 8 | 8 | 8.3 | 4.9 | 7 |
| 12 | 4 | 9 | -8.4 | 4.5 | 10 | 27 | 6 | 9 | 9.5 | 7.9 | 10 |
| 13 | 5 | 3 | 5.7 | 6.3 | 7 | 28 | 3 | 2 | 7.3 | 8.5 | 16 |
| 14 | 6 | 5 | 0.8 | 5.3 | 4 | 29 | 5 | 7 | 5.8 | 7.3 | 15 |
| 15 | 7 | 3 | 7.9 | 4.5 | 8 | 30 | 2 | 9 | 3.1 | 6.5 | 12 |

## Функции Ψ (α,β,İ).

1) $$Y=\begin{cases}\dfrac{i^2(\alpha+i\beta)\sqrt{\alpha\beta}}{(\beta-\alpha)^2+i^3} & при \quad \alpha\ge\beta\\ \dfrac{(i+2)\sqrt[5]{\beta+i\alpha^3}}{(\alpha-\beta)^2+i\alpha} & при \quad \alpha<\beta\end{cases}$$

2) $$Y=\begin{cases}\dfrac{(\alpha-e^{\beta})^2}{(\alpha-\beta)^2+i^2} & при \quad \alpha\ge\beta\\ \dfrac{(\alpha^2-e^{\beta})^2}{(\beta-\alpha)^2+i^2} & при \quad \alpha<\beta\end{cases}$$

3) $$Y=\begin{cases}\dfrac{(\alpha+(-1)^1 i\beta)^2}{i\sqrt{\alpha^2+\beta}} & при \quad \alpha>0\\ \dfrac{(i+i\beta)^2}{\sqrt{i+\beta^2}} & при \quad \alpha\le 0\end{cases}$$

4) $$Y=\begin{cases}\dfrac{\alpha+\beta}{i\sqrt{\alpha^2+\beta}} & при \quad \beta>0\\ \dfrac{(\alpha+i)^2}{i+\beta} & при \quad \beta\le 0\end{cases}$$

5) $$Y=\begin{cases}\dfrac{\alpha}{i\sqrt{\alpha^2+\beta}} & при \quad \beta>0\\ \dfrac{\alpha+i\beta}{\sqrt{i^2+\beta^2}} & при \quad \beta\le 0\end{cases}$$

6) $$Y=\begin{cases}\dfrac{(\alpha+(-1)^1 i\beta)^2}{i^2\sqrt{\beta}} & при \quad \beta>0\\ \dfrac{(i+i\beta)^2}{\sqrt{i+\cos\beta^2}} & при \quad \beta\le 0\end{cases}$$

7) $$Y=\begin{cases}\dfrac{(\alpha+i\beta)^2}{i\sqrt{\alpha+\beta^2}} & при\ \beta>0\\ \dfrac{(i+2)\sqrt[5]{\beta+i\alpha^3}}{(\alpha-\beta)^2+i\alpha} & при\ \beta\le 0\end{cases}$$

8) $$Y=\begin{cases}\dfrac{Sin(\alpha+\beta)}{i\sqrt{\alpha^2+\beta^2}} & при \quad \alpha>0\\ \dfrac{(\alpha+i\beta)^2}{\sqrt{i+\beta^2}} & при \quad \alpha\le 0\end{cases}$$

9) $$Y=\begin{cases}\dfrac{e^{\alpha-\beta}}{\sqrt{\alpha^2+\beta^2}} & при\ \alpha>0\\ \dfrac{(\alpha+i\beta)^2}{\sqrt{\alpha^3}} & при \quad \alpha\le 0\end{cases}$$

10) $$Y=\begin{cases}\dfrac{i\beta}{i\sqrt{\alpha+\beta^3}} & при \quad \alpha>\beta\\ \dfrac{(\alpha+i\beta)^2}{\sqrt{\alpha^2}} & при\alpha\le 0\end{cases}$$

## Функция φ($y_1$, $y_2$, …$y_n$)

1. $$\varphi=\min_{1\le i\le n}\sqrt{\frac{\alpha^2+\beta^2}{|\alpha\beta|}}$$

2. $$\varphi=\min_{1\le i\le n}\frac{\left|y_i^{\,2}-\beta\right|}{\alpha+\beta}$$

3. $$\varphi=\max_{1\le i\le n}|y_i|-\min_{1\le i\le n}|y_i|$$

4 . $$\varphi=\min_{1\le i\le n}(\sqrt{|\alpha\beta|}+y_i)$$

5. $\varphi=\underset{1\le i\le n}{Max}\ |y_i| +\cos(\alpha)$ 6. $\varphi=\underset{1\le i\le n}{Min}\sqrt{y_i e^{\alpha+\beta}}$

7. $\varphi= \mathrm{Ln}|\beta|+\underset{1\le i\le n}{Min}|y_i|$ 8. $\varphi=\underset{1\le i\le n}{Min}\sqrt{|\alpha\beta|}$

9. $\varphi=\sum_{i=1}^{n}\sqrt{e^{z_1}+1}$, где $z_1=\begin{cases} y_1 & при\ y_1\in[0,10] \\ 7 & при\ y_1\notin[0,10] \end{cases}$

10. $\varphi=\sum_{i=1}^{n}\sqrt{|\alpha|^{z_1}}$, где $z_1=\begin{matrix} y_1 & при\ y_1\le 10 \\ \sin\beta & при\ y_1>10 \end{matrix}$

**Выполнить условия задания 5.3.2. с исходными данными:**

$$Y=\begin{cases} \dfrac{(i+1)(2\alpha+i\beta)}{(\beta-\alpha)^2+\alpha i+i^2} & при\ \alpha>\beta \\ \dfrac{(i-1)\,(2\beta^3+(2i)^3)}{(\beta-\alpha)^2+\alpha i+i^2} & при\ \alpha\le\beta \end{cases} \qquad \varphi=\underset{1\le i\le n}{Min}\sqrt{\frac{y_i^2+\alpha^2}{2}}$$

**α=1,5; β=2,7; N=10.**

**Двумерные массивы.** Составить блок- схему и программу на языке Borland C, результаты работы вывести на монитор и в файл в текущем каталоге с именем Pr.res.

1. Вычислить построчные наименьшие элементы массива A(6,6)
2. Транспортировать матрицу A(5,5), и найти след матрицы.
3. В массиве A(4,4) найти строку с наименьшим элементом и заменить эту строку первым столбцом.
4. В целочисленном массиве A(4,5) найти сумму всех кратных 3 элементов и разделить на нее элементы второстепенной диагонали.
5. В целочисленном массиве A(4,4) все элементы четных строк поменять местами с элементами нечетных строк.
6. В массиве A(5,5) найти след матрицы и разделить на него элементы 3 строки.
7. Отсортировать элементы второстепенной диагонали массива A(6,6) по возрастанию.
8. Создать массив B(4) из среднеарифметических положительных элементов каждой строки массива A(4,6).
9. В массиве A(5,5) найти все одинаковые элементы и их координаты.
10. Выбрать в массиве A(6,6) наибольший элемент и разделить на него элементы второстепенной диагонали.

11. Выбрать в массиве A(6,6) наибольший элемент и разделить на него элементы главной диагонали.
12. Из построчных сумм элементов массива A(6,6) создать массив B(6).
13. Вычислить сумму отрицательных элементов массива A(4,4).
14. Вычислить количество четных элементов массива A(4,4).
15. Найти наименьший элемент в массиве A(5,5), и в ту строку и столбец записать нули.
16. Найти минимальный и максимальный элементы матрицы A(5,5), и поменять местами строки матрицы. Если они расположены, в разных строках, если в одной строке, то поменять столбцы.
17. Дана матрица A(m,m) и вектор B(m). Записать на второстепенную диагональ элементы вектора, а элементы главной диагонали в вектор размер m, размер m выбрать произвольно.
18. Дана матрица A(m,m) и вектор B(m). Записать на второстепенную диагональ элементы главной диагонали в вектор размер m, размер m выбрать произвольно.
19. Выбрать максимальный элемент матрицы C(m,n). Найти номер столбца, элементы четных строк разделить на максимальный элемент.
20. Дана матрица D(m,n). Найти номер столбца, элементы которого образуют возрастающую последовательность.
21. Перемножить A(m,n) и B(m,n) элементы результирующей матрицы вычислить с помощью выражения $c_{ik} = \sum_{j=1}^{m} a_{jk} b_{jk}$
22. Вычислить значения полинома $y_i = a_{i1}x^9 + a_{i2}x^8 + \ldots + a_{xi} + a_{i10}$ при различных значениях коэффициентов, используя формулу Горнера $y_i = (\ldots(((a_{i1}+a_{i2})\ x+a_{i3})\ x+\ldots+a_{i9})+a_{i10}$. Коэффициенты сведены в матрицу A(5,10).
23. Найти среднеарифметическое положительных элементов при условии, что в каждом столбце есть хотя бы один положительный элемент.
24. Создать массив Z(6,5), если $z_{i,j} = \sum_{i=1}^{6} \sum_{j=1}^{5} \frac{\sin^j(x)}{j}$
25. Определить является ли целая квадратная матрица $5^{10}$ порядка магическим квадратом, т. е. такой, в которой суммы элементов во всех строках и столбцах одинаковы.
26. Элементы главной диагонали матрицы A(5,5) расположить в порядке возрастания. Каждый элемент полученной матрицы умножить на сумму элементов главной диагонали.

27. Вычислить номер строки матрицы A(5,6) сумма элементов, которой по абсолютной величине минимальна.
28. Заменит отрицательные числа в массиве A(5,5) их квадратами. Посчитать нулевые элементы матрицы.
29. целочисленном массиве A(4,4) найти суму всех отрицательных элементов и заменить их значениями по модулю.
30. В массиве A(20) найти три наибольших и 3 наименьших числа.

## 5.7. Контрольные вопросы

1. Основные элементы и структура программы на языке С.
2. Представление данных в С.
3. Встроенные типы данных (стандартные типы С).
4. Операции и выражения в языке С.
5. Понятия функции в языке С.
6. Спецификация преобразования в яязыке С.
7. Escape-последовательности в языке С.
8. Область действия переменных и связанные с ней понятия.
9. Спагетти коды в программировании.
10. Решение системы линейных алгебраических уравнений (СЛАУ) методом Жордана-Гаусса
11. Структурное программирование.
12. Пример решения системы линейных алгебраических уравнений методом Гаусса.
13. Указатели и ссылки в языке С.
14. Типы, определяемые пользователем в языке С.
15. Понятие объединения в языке С..

# ЛАБОРАТОРНАЯ РАБОТА №6.
# Создание консольных приложений на языке С++ в среде С ++ Builder

**Цель лабораторной работы**: изучить и тестировать парадигмы ООП для консольных программ на языке С++ в среде С ++Builder, а также научиться создавать UML - диаграммы.

## 6.1. Примеры подготовки и создания консольного приложения на алгоритмическом языке С++

При изучении консольных приложений на языке С, в работе №5, мы познакомились, как стартовать их, *File* ⇒ *Close All,* т.е. закрываем окно Форм (Form) и Редактора кода (Units).

Следующая команда *File* ⇒ *New* ⇒ *Other* позволяет открыть нам различные группы, свойственные С++ Builder.

Нас интересует вопрос создания консольного приложения, и мы выбираем команду (объект) *Console Wizard* ⇒ *Ok*, при этом получаем окно мастера создания консольных приложений.

Для тех, кто знаком с языками С и С++ тему можно пропустить.

**Простейший ввод и вывод на языке С++.** Для вывода информации на экран монитора используется объект **cout** (**consol** - это давнишний термин, который является эквивалентом экрана и клавиатуры одновременно) и пишут его перед операцией **consol out** эквивалентен **cout <<**,а **consol in** ввод информации с клавиатуры эквивалентен **cin >>**.

Например, **cout** << "Hello World!";
**cin >> x;**

Для форматированного вывода используют и манипуляторы, см. примеры ниже, которые показывают действия различных манипулоторов.

**Манипуляторы hex** и **oct** соответственно используют для печати шестнадцатеричных и восьмеричных чисел.

**Задача 6.1.** Нарисовать UML-диаграмму, и написать программный код, для демонстрации операций ввод **cin >> x** и вывод **cout** << информации на монитор, а также манипуляторы hex и oct для печати шестнадцатеричных и восьмеричных чисел.

UML-диаграмму для простейшего случая вывода данных с манипуляторами hex и oct нарисована на рис. 6.1. а результаты приведены на рис. 6.2.

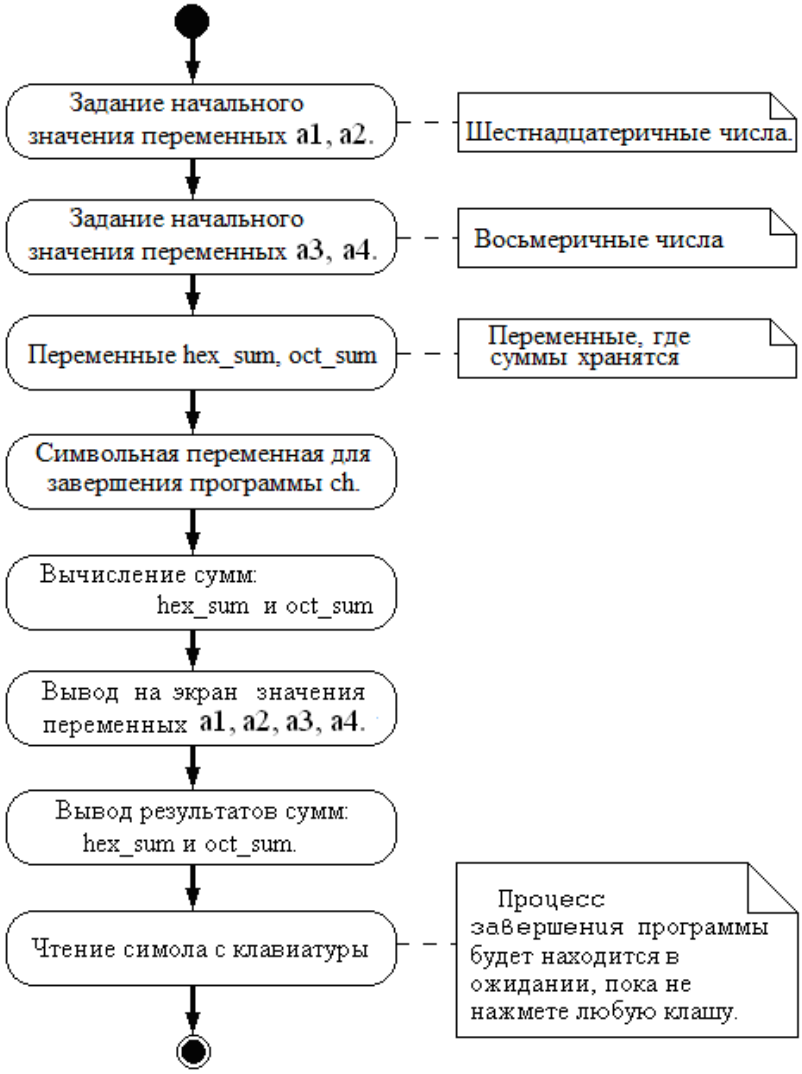



Рис. 6.1. UML-диаграмма для задачи 6.1.

## Программный код задачи 6.1

```
//---------------------------------------------------------
    #include <vcl.h>
    #pragma hdrstop
    #include <iostream.h>
    //---------------------------------------------------------
    #pragma argsused
      using namespace std;
    int main(int argc, char* argv[])
    {
      int a1=0x1a, a2=0xff; // Шестнадцатеричные числа.
      int a3=054, a4=077; //Восьмеричные числа
      int hex_sum, oct_sum; // Переменные где хранятся суммы
      char ch;
    cout << "The hexadecimal a1 and a2: a1= "
         << hex << a1 << "; a2= " << hex << a2 << endl;
         cout << "The octal numbers a3 and a4: a3= "
         << oct << a3 << "; a4= " << oct << a4 <<endl;
```

```
    hex_sum=a1+a2;
    oct_sum=a3+a4;
    cout << "The hexadecimal add up to:"
         << hex << hex_sum<< endl;
    cout << "The octal numbers add up to:"
         << oct << oct_sum<< endl;
    cout << "\nPress any key...";
    cin >> ch;
         return 0;
}
//---------------------------------------------------------
```

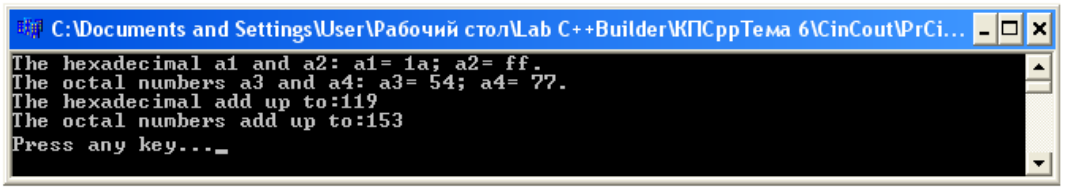



Рис 6.2. Операции ввода/вывода и манипуляторы hex и oct.

**Задача 6.2.** Используя, манипуляторы `setw()` и `setprecilion()` выполнить форматированный вывод данных, тест примера и UML – диаграммы на рис. 6.3. и 6.4.

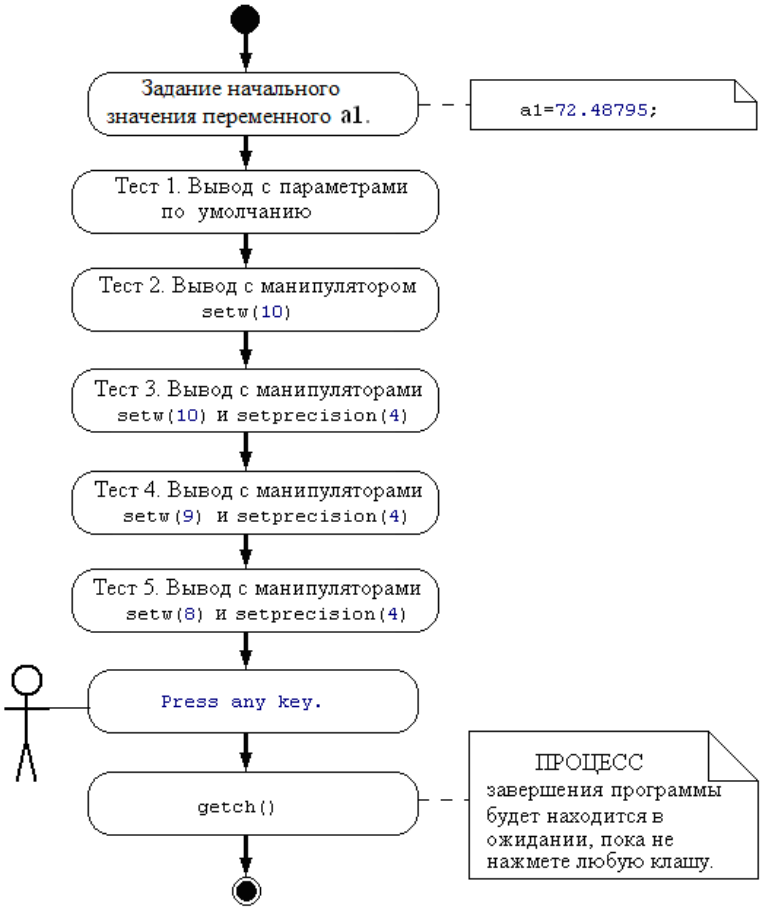



Рис. 6.3. UML –диаграмма вывода данных с манипуляторами `setw()` и `setprecilion()`

Программный код и результаты задачи 6.2.

Fnsetw.cpp

```
#include <vcl.h>
#pragma hdrstop
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
 double a1=72.48795;
    cout << " a1=72.48795  Test 1   a1="
         << a1  <<"\n";
    cout << " a1=72.48795  Test 2   a1="
         << setw(10) << a1 <<"\n";
    cout << " a1=72.48795  Test 3   a1="
         << setw(10) << setprecision(4)<< a1 <<"\n";
    cout << " a1=72.48795  Test 4   a1="
         << setw(9) << setprecision(4)<< a1 <<"\n";
    cout << " a1=72.48795  Test 5   a1="
         << setw(8) << setprecision(4)<< a1 <<"\n";

    cout << "\n  Press any key. ";
  getch();
       return 0;
}
```

```
C:\Documents and Settings\User\Рабочий стол\Lab C++Builder\КПСppТема 6\FunSetw...
a1=72.48795  Test 1   a1=72.4879
a1=72.48795  Test 2   a1=   72.4879
a1=72.48795  Test 3   a1=     72.49
a1=72.48795  Test 4   a1=    72.49
a1=72.48795  Test 5   a1=   72.49

 Press any key.
```

Рис. 6.4. Манипуляторы форматирования `setw()` и `setprecilion()`

Для форматированного вывода данных используется также и манипуляторы setiosflags для установленния глобальных флагов и resetiosflsgs для их отмены, используя класс C++ iostream. Допускается комбинированное использование флагов. В таблице 6.1. приведены наиболее часто используемые флаги.

Таблица 6.1.

| **Значение** | **Результат, если зачение установленно** |
|---|---|
| ios::skipws | Игнорирует пустое пространство при вводе. |
| ios::left | Вывод с выравниванием слева. |
| ios::right | Вывод с выравниванием справа. |
| ios::dec | Вывод в десятичном формате. |
| ios::oct | Вывод в восмеричном формате. |
| ios::hex | Вывод в шестнадцатеричном формате. |
| ios::scientific | Вывод чисел в научном с плавающей точкой формате. |
| ios::fixset | Вывод чисел в формате с фиксированной точкой. |
| ios::showpoint | Выводить десятичную точку. |

## 6.2. Общие сведения о классах

Мы говорили о классах при выполнения работ №3 и №4. Класс - составной тип данных, элементами которого являются функции и переменные. В основу понятия класс положен тот факт, что «над объектами можно совершать различные операции». Свойства объектов описываются с помощью полей классов (члены-данные), а действия над объектами описываются с помощью функций (членов-функций), которые называются ***методами класса***. Следовательно, *класс* имеет *имя*, и состоит из *полей*, так называемых *члены-данные* и *методов* (членов-функций) класса.

Напоминаем, класс имеет следующий формат:

```
class name  // name – имя класса
{
                    private:        //Описание закрытых членов-данных
                и методов класса, доступных только в классе.
                    protected: // Описание защищенных членов и
                методов класса, которые передаются потомкам.
 public: //Описание открытых членов и методов
  класса
}
```

Приведем резюмирующие фундаментальные правила для разделов класса:

1. *Порядок следования разделов произвольный, и один и тот же раздел можно определять несколько раз.*
2. *Если разделы не определены, компилятор (по умолчанию) объявляет все элементы закрытыми.*
3. *Помещать данные-элементы в открытый раздел следует только в том случае, если в этом есть необходимость, например, если это упрощает вашу задачу. Обычно элементы-данные помещаются в защищенный раздел, чтобы к ним имели доступ функции-элементы классов-потомков.*
4. *Желательно использовать для изменения значений данных и доступа к ним функции-элементы. При использовании функции можно осуществлять проверку данных и, если нужно, изменять другие данные.*
5. *Класс допускает возможность иметь несколько конструкторов.*
6. *Класс допускает обявление деструкторов, которые обычно объявляются в открытом разделе класса, но можно и в закрытом.*

7. *Функции-элементы (в том числе конструкторы и деструкторы), состоящие из нескольких операторов, желательно определяться вне объявления класса.*

**Открытые методы и члены – данных класса.**

**Внимание!** В отличие от полей структуры доступных всегда, доступ к членам-данным и методам в классах могут быть различного уровня:

- ***Открытые*** **`public:`** *(публичные) доступно всем. Вызов открытых членов и методов класса осуществляется с помощью оператора "." ("точка");*
- ***Закрытые*** **`private:`** *(приватные), доступ, к которым возможен только в классе через общий интерфейс вместо прямого доступа. Классам-потомкам запрещен доступ к закрытым данным своих базовых классов;*
- ***Защищенные*** **`protected:`** *(наследуемые), доступны напрямую только они потомкам, а остальные нет.*

После описания класса необходимо описать переменную типа ***`class`***.

Например,

```
name_class name;
```

здесь `name_class` – имя класса, `name` – имя переменной.

*В дальнейшем переменную типа* ***`class`*** *будем называть «объект» или «экземпляр класса». Объявление переменной типа* ***`class`*** *(в нашем примере переменная* `name` *типа* `name_class`*) называется созданием (инициализацией) объекта (экземпляра класса).*

После описания переменной можно обращаться к членам и методам класса.

Обращение к членам и методам класса, напоминаем, осуществляется аналогично обращению к полям структуры с помощью оператора «.» (точка).

```
/*Обращение к полю p1 экземпляра класса name. */
name.p1;
/* Обращение к методу f1 экземпляра класса name,par1, par2, …, parn – список формальных параметров функции f1.*/
name.f1(par1,par2,…parn);
```

Члены класса доступны из любого метода класса, и их не надо передавать, в качестве параметров функций-методов.

**Задача 6.3**. Составить UML диаграмму для класса complex, и рассмотреть в задаче элементарные операции над комплексными числами по выбору пользователя.

В классе complex будут ***члены класса***:

- ***double*** `x` – действительная часть комплексного числа;
- ***double*** `y` – мнимая часть комплексного числа
- ***double*** `z` – комплексное число

***Методы класса***:

- ***double*** `modul()` – функция вычисления модуля комплексного числа;
- ***double*** `argument()` – функция вычисления аргумента комплексного числа;
- ***void*** `show_complex()` – функция выводит комплексное число на экран.
- ***void*** `comadd()` - функция выводит на экран сложение двух комплексных чисел;
  - ***void*** `comsubst()`- функция выводит на экран вычитание двух комплексных чисел;
  - ***void*** `commult()`- функция выводит на экран умножение двух комплексных чисел;
  - ***void*** `comdiv()`- функция выводит на экран деление двух комплексных чисел.

На рисунках 6.5 -6. 9. приведены UML – диаграммы класса complex.

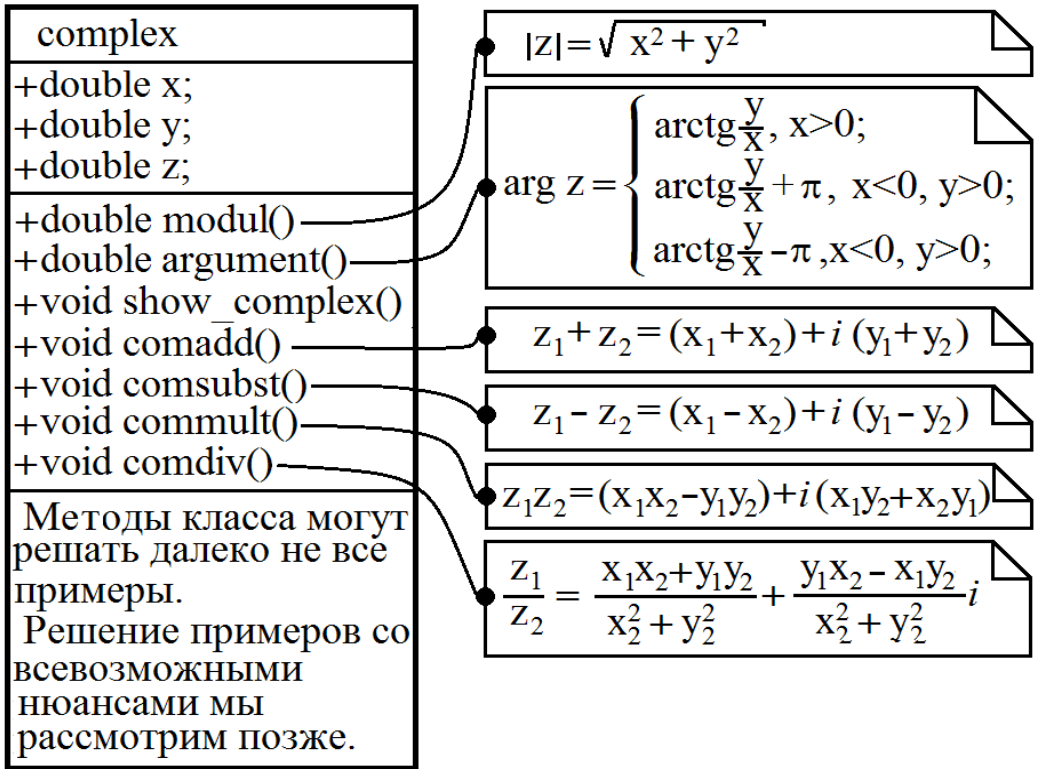



Рис. 6.5. Диаграмма (UML) класса complex.

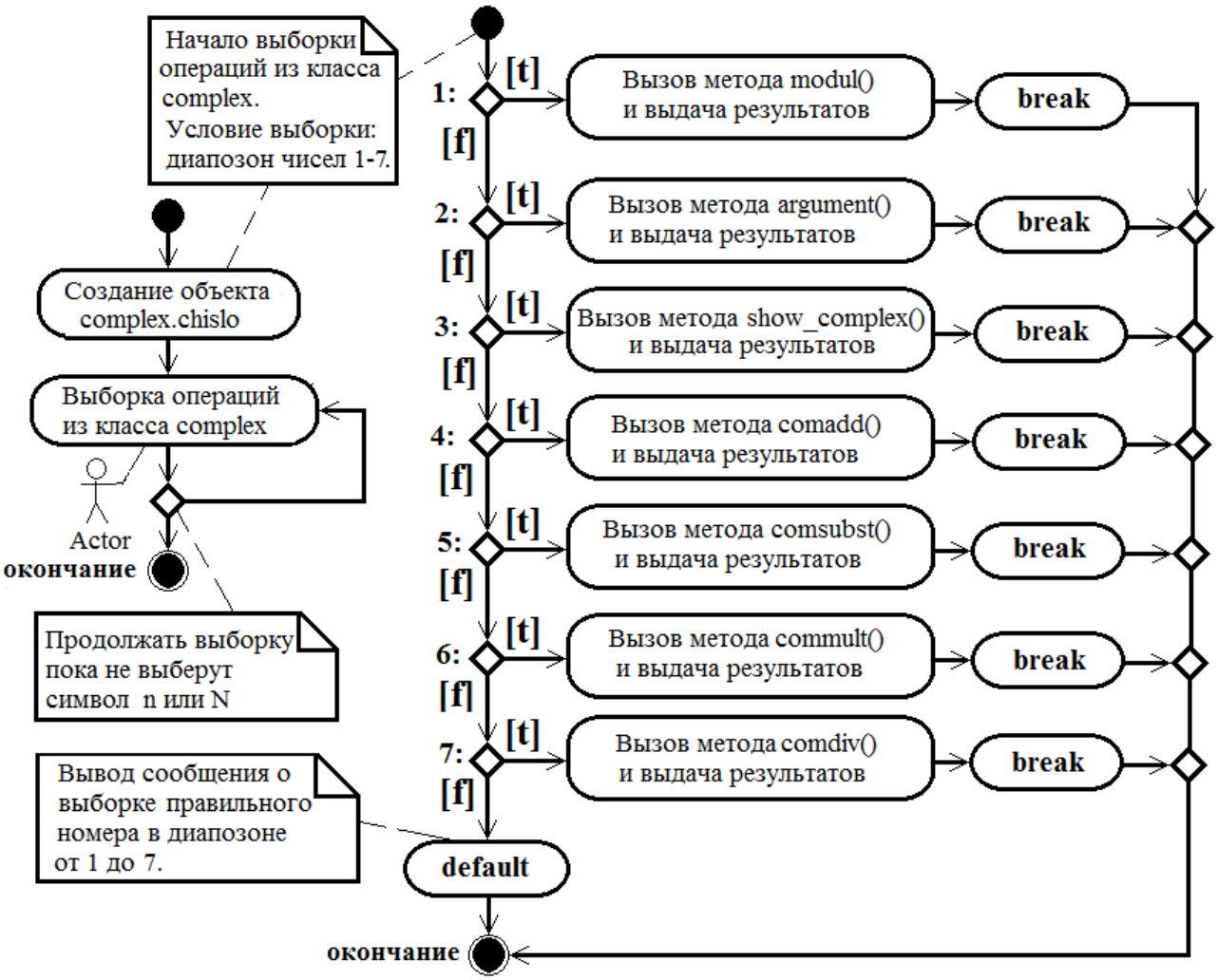



Рис. 6.6. Диаграмма деятельности (UML) для задачи 6.3.

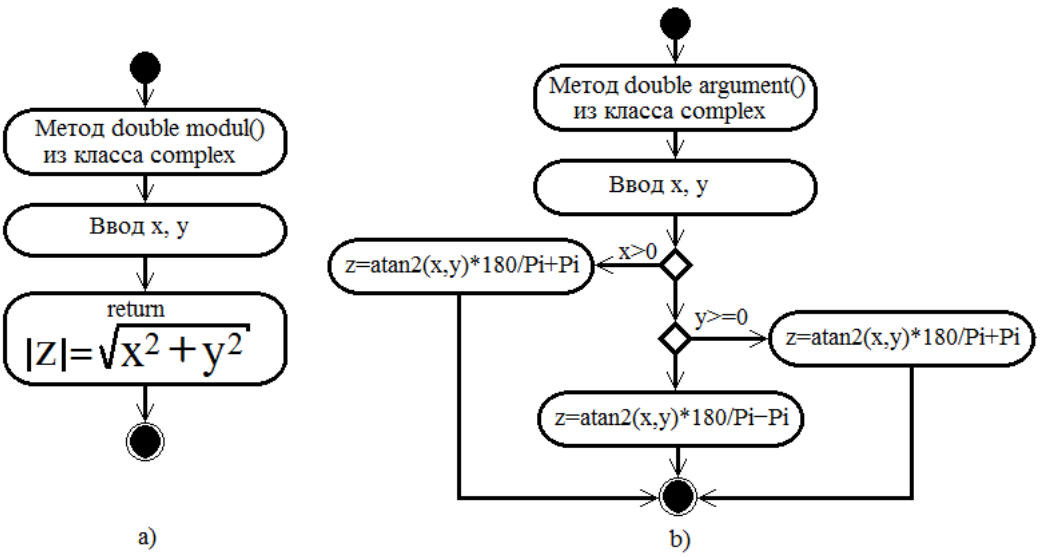



Рис. 6.7. Диаграмма деятельности (UML) для задачи 6.3.

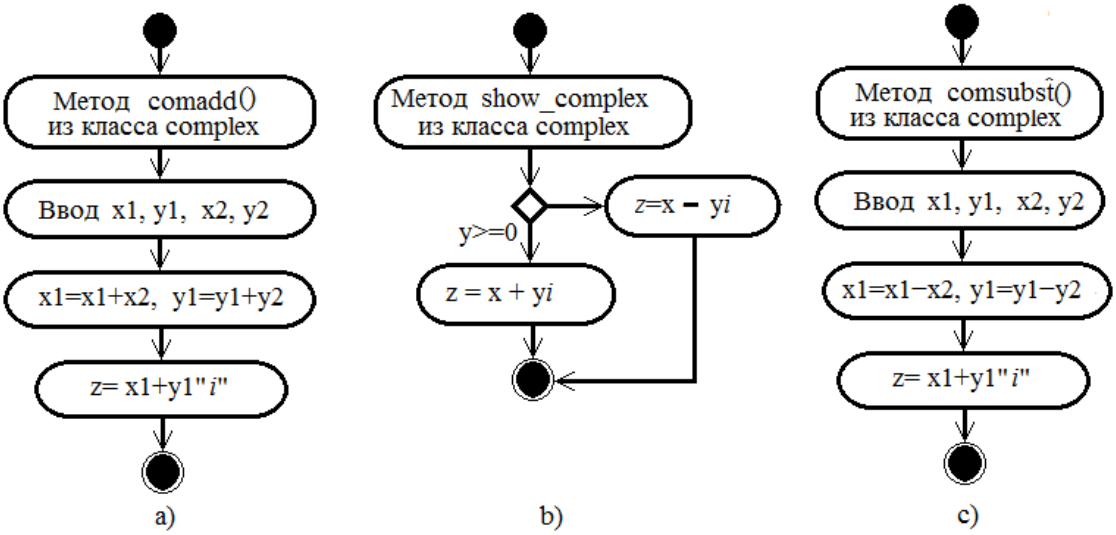



Рис. 6.8. Диаграмма деятельности (UML) для задачи 6.3.

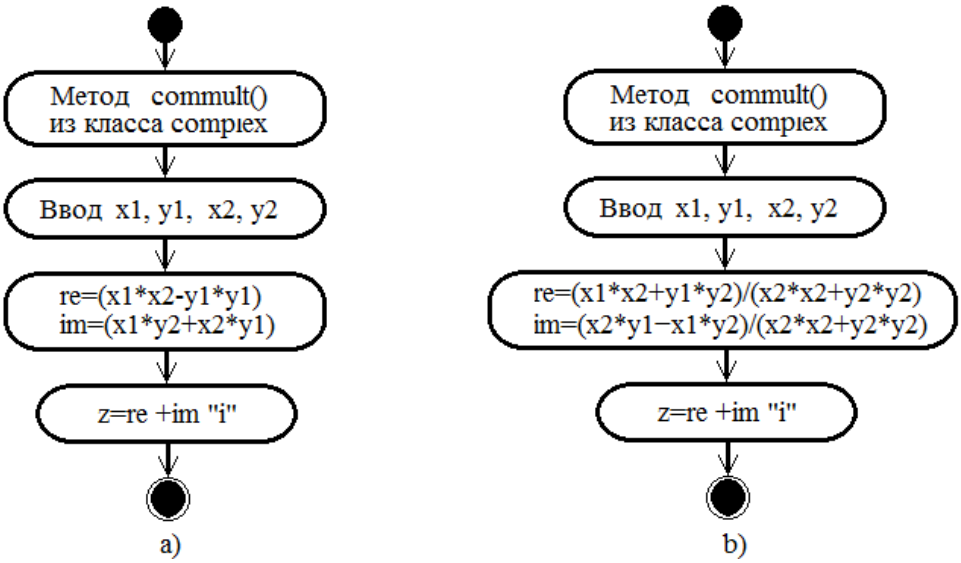



Рис. 6.9. Диаграмма деятельности (UML) для задачи 6.3.

Ниже приведен пример, где описывает класс и функция main, демонстрирующая работу с классом `complex`. На рис 6.10. результаты тестирования программного кода.

```
#include <vcl.h>
#pragma hdrstop
#include <string>
#include <iostream.h>
#include <math.h>
#define PI 3.14159
using namespace std;
class complex //Определяем класс complex
{
 public:
   double x; //Действительная часть комплексного
числа.
```

```
        double y; //Мнимая часть комплексного числа.
    /* Метод класса или член-функция complex – функция
modul, для вычисления модуля комплексного числа.*/
        double modul()
      {
       cout << "Re x= "; cin >> x;
       cout << "Im y= "; cin >> y;
       return pow(x*x+y*y,0.5);
      }
    /*Метод класса complex – функция argument, для
вычисления аргумента комплексного числа.*/
    double argument()
    {
       double z;
       cout << "Re x= "; cin >> x;
       cout << "Im y= "; cin >> y;
       if (x>0) z= atan2(y,x)*180/PI;
       else if (y>=0) z=atan2(y,x)*180/PI+PI;
       else z=atan2(y,x)*180/PI-PI;
     return z;
    }
    /*Метод класса complex – функция show_complex, для
вывода комплексного числа.*/
    void show_complex()
    {
    if (y>=0)
    /*Вывод комплексного числа с положительной мнимой
частью.*/
    cout<<x<<"+"<<y<<"i"<<endl;
    else
    /* Вывод комплексного числа с отрицательной мнимой
частью. */
    cout<< " Z = "<<x <<y <<"i"<<endl;
    }
    /* Операции над комплексными числами: сложение;
вычитание; умножение деление. */
     void comadd ()  //Сложение.
      {
        float x1,x2,y1,y2;
         cout<<"x1="; cin >> x1;
         cout<<"y1="; cin >> y1;
         cout<<"x2="; cin >> x2;
         cout<<"y2="; cin >> y2;
          x1+=x2;
          y1+=y2;
      cout <<" Z="<<x1<<" + "<<y1<<"i  -Add"<<endl;
      }
     void comsubst()   //Вычитание.
      {
        float x1,x2,y1,y2;
         cout<<"x1="; cin >> x1;
         cout<<"y1="; cin >> y1;
         cout<<"x2="; cin >> x2;
```

```
     cout<<"y2="; cin >> y2;
      x1-=x2;
      y1-=y2;
  cout<<"  Z="<<x1<<" + "<<y1<<"i  -Subs"<<endl;
  }
  void commult()   //Умножение.
  {
    float x1,x2,y1,y2,re,im ;
     cout<<"x1="; cin >> x1;
     cout<<"y1="; cin >> y1;
     cout<<"x2="; cin >> x2;
     cout<<"y2="; cin >> y2;
      re=(x1*x2-y1*y2);
      im=(x1*y2+x2*y1);
  cout<<"  Z="<<re<<" + "<<im<<"i  -Mult"<<endl;
  }
   void comdiv()   //Деление.
  {
    float x1,x2,y1,y2,re,im ;
     cout<<"x1="; cin >> x1;
     cout<<"y1="; cin >> y1;
     cout<<"x2="; cin >> x2;
     cout<<"y2="; cin >> y2;
      re=(x1*x2+y1*y2)/(x2*x2+y2*y2);
      im=(x2*y1-x1*y2)/(x2*x2+y2*y2);
  cout<<"  Z="<<re<<" + "<<im<<"i  -Div"<<endl;
  }
};
int main(int argc, char* argv[])
{
//Определяем переменную chislo типа complex.
  complex chislo;
    char ans;
    int num;
 do
  {
     cout << " Please enter a number " <<endl;
     cout << " 1 - a method modul() " << endl;
     cout << " 2 - a method argument() " << endl;
     cout << " 3 - a method show_complex() " << endl;
     cout << " 4 - a method comadd() " << endl;
     cout << " 5 - a method comsubst() " << endl;
     cout << " 6 - a method commult() " << endl;
     cout << " 7 - a method comdiv() " << endl;
     cin >> num;
   switch (num)
     {
      /* Вывод модулей комплексного числа, и
        обращения к методам класса.*/
       case 1: cout<<"Modul' chisla="<<chislo.modul();
               break;
       case 2: cout<<endl<<"Argument chisla="
               <<chislo.argument() <<endl;
```

```
                    break;
               case 3: chislo.show_complex();
                    break;
               case 4: cout<<"Z = z1 + z2 " << endl;
chislo.comadd();
                    break;
               case 5: cout <<" Z = z1 - z2"<<endl;
chislo.comsubst();
                    break;
               case 6: cout <<"Z = z1 * z2 "<<endl;
chislo.commult();
                    break;
               case 7: cout <<"Z = z1 / z2 " <<endl;
chislo.comdiv();
                    break;
          default: cout <<"Please select the number from
diapason 1-7";
             }
            cout <<" You must type a Y or an N" << endl;
            cout <<" Do you want to continue (Y/N)?" << endl;
            cin >> ans;
          }
         while (ans!='n');
        return 0;
        }
```

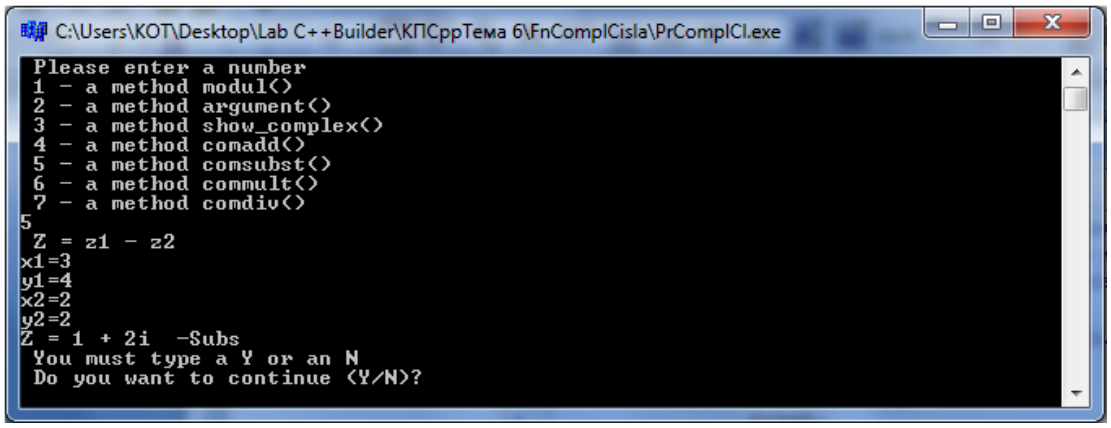



Рис. 6.10. Комплексные числа. Результаты работы.

## 6.3. Конструкторы и деструкторы

**Общие положения о конструкторах и деструкторах.** В ООП для классов, конструкторы являются специфическим типом функций-элементов, *тип возвращаемого значения для которых не указывается, а имя должно совпадать с именем своего класса*. Вызываются они при создании нового представителя класса. Если пользователь не вызвал, как в примере 6.3, то невидимые пользователям, и носящий имя класса системой создается всегда при объявлении класса

автоматически. Однако система допускает, наличие нескольких конструкторов и деструкторов, хотя и один деструктор достаточно, чтобы уничтожить, все конструкторы, т.е. освободить память и разрушить класс. Конструкторы используются для процесса инициализации членов данных и других целей, о которых мы укажем позже в примерах. Напоминаем, конструкторы и деструкторы являются членами - функции класса, однако при их объявлении тип не указывается, и, следовательно, они ничего не возвращают.

***Рекомендация!*** *В конструкторах не желательно выполнять операции, подвергающиеся проверке, и выдачи результатов проверки (например, выделение памяти функция* `void *malloc(size_t)` *и др.), т.к. если какие-то действия не выполнены, то об этом невозможно сообщить задаче.*

Резюмируя, выше сказанное утверждаем.

1. *Конструкторы и деструкторы на языке С++по умолчанию, вызываются автоматически, и они являются гарантами правильного создание и разрушение объектов класса.*

Общий вид (синтаксис) объявления конструктора:

```
class className
    {
    public:
      className();                  // конструктор по
умолчанию
      className(const className &c);   // конструктор копии
      className(<список параметров>); // остальные
конструкторы
    };
```

2. *Конструкторы и деструктор отличаются от других объектных методов следующими особенностями:*

- *Имя совпадает с именем своего класса.*
- *Не имеют возвращаемого значения, а также не указывают тип void.*
- *Не могут наследоваться, однако производный класс, если вид доступа не* **`private`***, то может вызывать конструкторы и деструктор базового класса.*
- *По умолчанию автоматически генерируются компилятором как public, если не были объявлены иначе.*
- *Автоматически вызываются компилятором, чтобы гарантировать надлежащее создание и уничтожение объектов классов.*

• *Могут содержать неявные обращения к операторам new и delete, если объект требует выделения и уничтожения динамической памяти.*

**Демонстрация объявления конструкторов.** Приведем текст программного кода с разлиными конструкторами для демонстраии их видов.

**Задача 6.4.** Написать программнй код, демонстрирующий, форматы и возможности конструктора и их инициализацию.

```
//---------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include <iostream>
#include <conio.h>
//---------------------------------------------------------
#pragma argsused
  using namespace std;
class TestConstructor {
public:
      /***   Конструктор без параметров – конструктор по
             умолчанию № 1.                          ***/
  TestConstructor() { cout << "TestConstructor()" << endl;
}
      /***   Конструктор с одним параметром – остальные
             конструкторы № 2.                       ***/
  TestConstructor(int)
       { cout << "TestConstructor(int)" << endl; }
      /***   Конструктор с с двумя параметрами – остальные
             конструкторы № 3.                       ***/
  TestConstructor(int, int)
       { cout << "TestConstructor(int, int)" << endl; }
      /***   Конструктор с одним параметром – конструктор
             копирования № 4.                        ***/
  TestConstructor(const TestConstructor&)
       { cout << "TestConstructor(const test&)" << endl; }
      /***   Конструктор с одним параметром – конструктор
             копирования № 5.                        ***/
  void operator=(const TestConstructor&)
       { cout << "operator=(const TestConstructor&)" <<
endl; }
};
//---------------------------------------------------------
int main(int argc, char* argv[])
{
 cout << "TestConstructor test_1        : ";
           TestConstructor test_1;
 cout << "TestConstructor test_2(69)    : ";
           TestConstructor test_2(69);
 cout << "TestConstructor test_3 = 69   : ";
           TestConstructor test_3 = 69;
```

```
     cout << "TestConstructor test_4 = TestConstructor(4) : ";
                TestConstructor test_4 = TestConstructor(4);
     cout << "TestConstructor test_5(13, 17)   : ";
                TestConstructor test_5(13, 17);
     cout << "TestConstructor test_6 = TestConstructor(13, 17):
";
                TestConstructor test_6 = TestConstructor(13,
17);
     cout << "TestConstructor test_7; test_7 = 1   : ";
                TestConstructor test_7; test_7 = 1;
     cout<<"TestConstructor test_8; test_8 =
TestConstructor(1):";
               TestConstructor test_8; test_8 =
TestConstructor(1);
     cout << "TestConstructor test_9(test_8)       : ";
                TestConstructor test_9(test_8);
     cout << "TestConstructor test_10 = 'a'    : ";
                TestConstructor test_10 = 'a';
      cout <<"Press any Key ...";
         getch();          return 0;
    }
    //---------------------------------------------------------
```

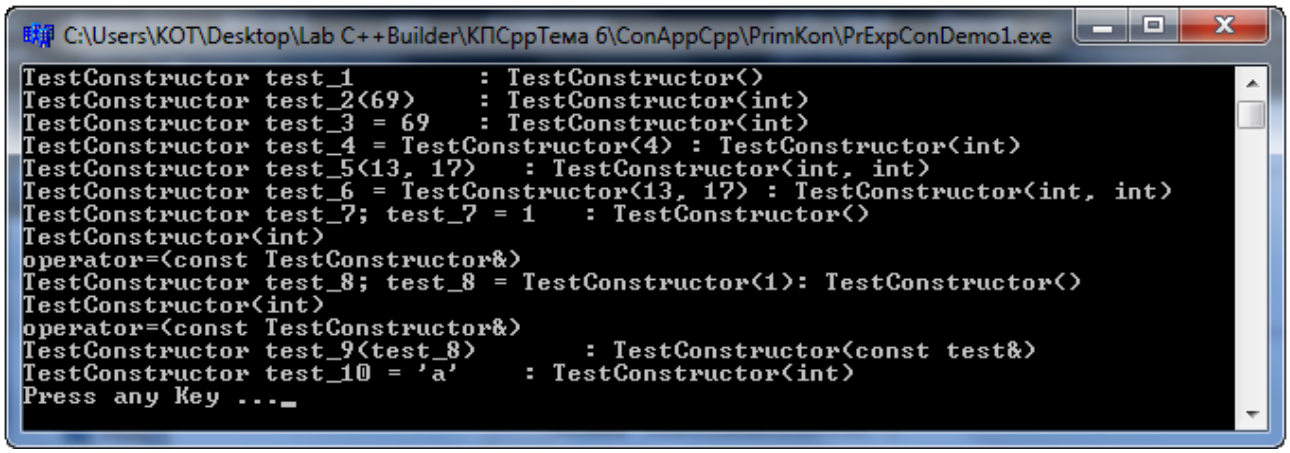



Рис. 6.11. Демонстрация вариантов инициализации конструкторов

Если нужно проинициализировать новый объект существующим (т.е. сделать точную копию объекта), используется конструктор копирования. Здесь рациональное зерно такого конструктора как раз в том, что работает быстрее, чем `operator=`, т.к. позволяет проинициализировать данные объекта во время их конструирования. Также немаловажно: если в процессе инициализации нового объекта будет выброшено исключение, например, `bad_alloc`, то некорректный объект создан не будет (впрочем, то же самое касается любого конструктора).

Далее мы в примерах более подробно рассмотрим работу с конструкторами и деструкторами в следующих пунктах.

**Пример функционирования конструтора и деструктора .**

Приведем пример, где у класса имеются и конструктор, и деструктор.

**Задача 6.5.** Заданы координаты правильных фигур и сферы, а также радиус сферы или вписанный радиус поверхность основания правильных фигур. Вычислить объем и поверхности фигур. Для задачи использовать конструктор и деструктор. Используя, конструктор инициализируйте переменную.

***class*** `My_Figure` – **Класс,** который объявлен в задаче.

Данный класс содержит ***члены класса***:

- ***float*** `r;` Радиус вписанной окружности основания правильных фигур или сферы;
- ***float*** `x,y,z;` Координаты фигуры.

Функции – класса:

- `My_Figure(`***float*** `x_coord,` ***float*** `y_coord,` ***float*** `z_coord,` ***float*** `radius)` Конструктор, которого создает пользователь;
- `~My_Figure ()` Деструктор.

Функции (методы) – класса:

- ***float*** `volume_sphere` Объем сферы;
- ***float*** `surface_sphere()` Поверхность сферы;
- ***float*** `volume_tetraedr()` Объем тетраэдра;
- ***float*** `surface_tetraedr()` Поверхность тетраэдра;
- ***float*** `surface_cube()` Поверхность куба;
- ***float*** `volume_cube()` Объем куба.

***Задание №1***. Постройте диаграммы класса и деятельности (UML).

```
/*Пример, где все члены-данные и методы общедоступны и имеет конструктор и деструктор*/
 #include <vcl.h>
#pragma hdrstop
#include <iostream>
#include <math.h>
#include <conio.h>
#define PI 3.14159
#define KS 10.3923
#define KV 2.44949
#pragma argsused
  using namespace std;
class My_Figure
 {
  public:
   /***Члены-данные они все открытые ***/
   float r;    // Радиус вписанной окружности
   float x,y,z; // Координаты
```

```
       /*** Члены-функции они все открытые, т.е. общедоступны
***/
    My_Figure(float  x_coord,  float  y_coord,  //Конструктор,
которого
              float   z_coord,   float   radius)      //создает
пользователь.
     {
          x=x_coord; y=y_coord; z=z_coord; r=radius;
      }
      ~My_Figure () {}       // Деструктор пользователя
      float volume_sphere ()      // Объем сферы
      {  return (r*r*r*4*PI/3);  }
      float  surface_sphere()   // Поверхность сферы
      {
        return (4*r*r*PI);
      }
      float volume_tetraedr()     // Объем тетраэдра
      {
        return (r*r*r/KV);
      }
      float surface_tetraedr()    // Поверхность тетраэдра
      {
        return (r*r*KS);
      }
      float surface_cube()         // Поверхность куба
      {
        return (24*r*r);
      }
       float volume_cube()         // Объем куба
       {
        return(8*r*r*r);
       }
     };
    int main(int argc, char* argv[])
    {
     /*** Объвляем переменную s типа класса My_Figure, и через
          конструктор выполняем процесс инициализации объекта
***/
         My_Figure s(1.0, 2.0, 3.0, 4.0);
          cout << " X="<<s.x << "\n"   // Обращение
               << " Y="<<s.y << "\n"   // к переменным
               << " Z="<<s.z << "\n"   // типа класс My_Figure
               << " R="<<s.r << endl;  // через s
     /*** Вывод объема и поверхности сферы  ***/
   cout << "The volume_sphere is "<<s.volume_sphere()<<endl;
   cout << "The surface_sphere is "<<s.surface_sphere()
<<endl;
      /*** Вывод объема и поверхности тетраэдра  ***/
   cout << "The volume_tetraedr is "<<s.volume_tetraedr()
<<endl;
   cout <<"The surface_tetraedr is "<<s.surface_tetraedr()
<<endl;
      /*** Вывод объема и поверхности куба  ***/
```

```
    cout << "The volume_cube is "<<s.volume_cube() <<endl;
    cout << "The surface_cube is "<<s.surface_cube() <<endl;
            cout <<"Press any key ...";
            getch();
             return 0;
    }
```

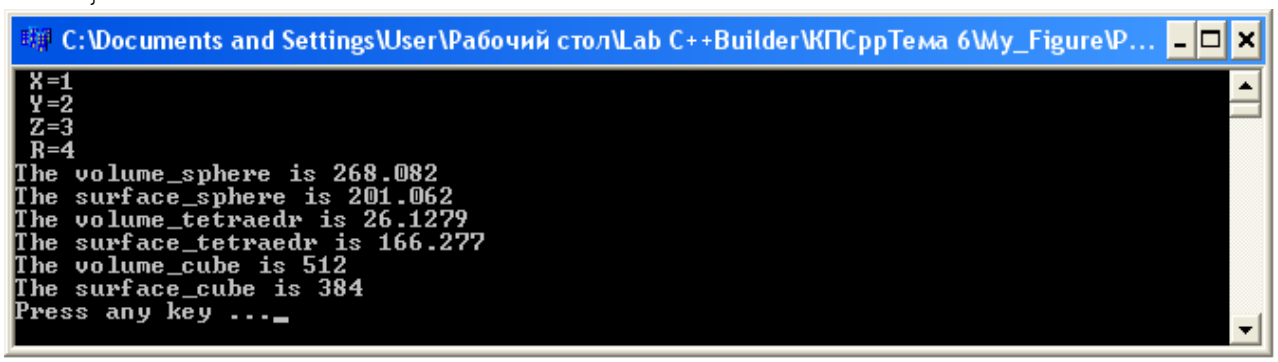



Рис.6.12. Поверхности и объемы фигур при заданных координатах.

## 6.4. Открытые и закрытые доступы (видимость)

В примерах 6.3. - 6.5. все члены были объявлены, как открытые. Использование открытых членов и методов позволяет получить полный доступ к элементам класса, однако это не всегда хорошо. Если все члены класса объявить открытыми, то при непосредственном обращении к ним появится потенциальная возможность внести ошибку в функционирование взаимосвязанных между собой методов класса. Поэтому, общим принципом является следующее: «Чем меньше открытых данных о классе используется в программе, тем лучше». Уменьшение количества публичных членов и методов позволит минимизировать количество ошибок. Желательно, взвесить все за и против. Количество открытых методов следует минимизировать разумно, т.к. в их использовании есть и свои хорошие стороны, например, они способствуют созданию ***общего интерфейса***.

***Внимание!.*** *Хорошо разработанный интерфейс должен удовлетворять следующим требованиям:*

- *Конструкторы и деструкторы всегда должны быть публичными (открытыми), хотя допускаются и остальные;*
- *Предусмотреть процесс инициализации объекта;*
- *Производить установку значений закрытых (частных) переменных.*

Если закрытие данные описать в начале класса, то ключевое слово private можно не писать. По умолчанию метод доступа обращения к членам и методам считаются закрытыми (private).

**Задача 6.6.** Видоизменим задачу 6.4. Заданы координаты правильных фигур и сферы, а также радиус сферы или круга. Вычислить объем и поверхности фигур. Для задачи использовать конструктор и деструктор, а также общий интерфейс.

***class*** `My_Figure` – **Класс,** который объявлен в задаче.

Данный класс содержит как закрытие, так и открытие ***члены данные и методы класса***. Сначала рассмотрим закрытие (приватные) элементы.

***private:***

- ***float*** `x,y,z;` Координаты фигуры;
- ***float*** `r;` Радиус только для сферы и круга;
- ***float*** `cube ()` Число в кубе (метод);
- ***float*** `square()` Число в квадрате (метод).

Далее рассмотрим публичные (открытые) члены-данные и члены функции. ***public:***

1. ***Функции – класса***:
   - `My_Figure(`***float*** `x_coord,` ***float*** `y_coord,` ***float*** `z_coord,` ***float*** `radius)` Конструктор, которого создает пользователь;
   - `~My_Figure ()` Деструктор
2. ***Функции (методы) – класса***:
   - ***float*** `volume_sphere` Объем сферы;
   - ***float*** `surface_sphere()` Поверхность сферы;
   - ***float*** `volume_tetraedr()` Объем тетраэдра;
   - ***float*** `surface_tetraedr()` Поверхность тетраэдра;
   - ***float*** `surface_cube()` Поверхность куба;
   - ***float*** `volume_cube()` Объем куба.
3. **Функция обслуживания** (позволяющие отыскать и изменять значения приватных переменных ):
   - ***void*** `SetX (`***float*** `newX);`
   - ***void*** `SetY (`***float*** `newY);`
   - ***void*** `SetZ (`***float*** `newZ);`
   - ***void*** `SetRadius (`***float*** `newRadius)`
   - ***float*** `GetX()`
   - ***float*** `GetY ()`
   - ***float*** `GetZ ()`
   - ***float*** `GetRadius ().`

```
// Пример для отображения скрытия данных
#include <vcl.h>
#pragma hdrstop
#include <iostream>
```

```
#include <math.h>
#include <conio.h>
#define PI 3.14159
#pragma argsused
  using namespace std;
class My_Figure
 {
  /***  Члены - данные и члены функции они все закрытые ***/
 private:
  float r;     // радиус
   float cube () { return (r*r*r);}     // число в кубе
   float square(){ return (r*r);}       // число в квадрате
  float x,y,z; // Координаты
   /*** Члены - функции они открытые т.е. общедоступные ***/
 public:
 My_Figure( float x_coord, float y_coord, // Конструктор,
которого
            float z_coord, float radius)  // создает
пользователь.
 {
      x=x_coord;
      y=y_coord;
      z=z_coord;
      r=radius;
  }
  ~My_Figure () {}       // Деструктор созданный пользователем
  float volume_sphere ()       // Объем шара
  {
    return (cube()*4*PI/3);
  }
  float  surface_sphere()   // Поверхность сферы
  {
    return (4*square()*PI);
  }
  float volume_tetraedr()      // Объем тетраэдра
  {
    return (cube()*sqrt(2)/12);
  }
  float surface_tetraedr()     // Поверхность тетраэдра
  {
    return (sqrt(3)*square());
  }
  float volume_cube()          // Поверхность куба
  {
    return (cube());
  }
  float surface_cube()          // Поверхность куба
  {
    return (6*square());
  }
  void SetX (float newX)
    {
       x=newX;
```

```
    }
  void SetY (float newY)
    {
       y=newY;
    }
  void SetZ (float newZ)
    {
       z=newZ;
    }
  void SetRadius (float newRadius)
    {
       r=newRadius;
    }
  float GetX()
    {
      return(x);
    }
  float GetY ()
    {
       return(y);
    }
  float GetZ ()
    {
       return(z);
    }
  float GetRadius ()
    {
       return(r);
    }
 };
int main(int argc, char* argv[])
{
    /*** Объявляем переменную типа класса My_Figure. Выполняем
         процесс инициализации объекта s***/
    My_Figure s(1.0, 2.0, 3.0, 4.0);
       cout<< " X="<<s.GetX()          // Обащение к
переменным
           << " Y="<<s.GetY()          // типа класс My_Figure
           << " Z="<<s.GetZ()          // через s
           << " R="<<s.GetRadius()     // при закрытых членах
           <<"\n";
     /*** Вывод объема и поверхности сферы  ***/
     cout<< "The volume_sphere is "<<s.volume_sphere()<<endl;
     cout<< "The surface_sphere is "<<s.surface_sphere()
<<endl;
      /*** Вывод объема и поверхности тетраэдра  ***/
     cout<< "The volume_tetraedr is
"<<s.volume_tetraedr()<<endl;
     cout<< "The surface_tetraedr is
"<<s.surface_tetraedr()<<endl;
      /*** Вывод объема и поверхности куба  ***/
     cout << "The volume_cube is "<<s.volume_cube() <<endl;
     cout << "The surface_cube is "<<s.surface_cube() <<endl;
```

```
      /*** Изменяем значения в закрытых членах ***/
         s.SetX(4.2);
         s.SetY(5.3);
         s.SetZ(6.7);
         s.SetRadius(6.9);
     /***Выводим на экран измененные значения в закрытых членах
***/
      cout   << " X="<<s.GetX()         // Обращение  к
переменным
             << " Y="<<s.GetY()          // типа класс My_Figure
             << " Z="<<s.GetZ()          // через s
             << " R="<<s.GetRadius()     // при закрытых членах
             <<"\n";
      /*** Вывод объема и поверхности сферы  ***/
      cout << "The volume_sphere is "<<s.volume_sphere()<<endl;
      cout << "The surface_sphere is "<<s.surface_sphere()
endl;
       /*** Вывод объема и поверхности тетраэдра  ***/
      cout << "The volume_tetraedr is
"<<s.volume_tetraedr()<<endl;
      cout <<"The surface_tetraedr is
"<<s.surface_tetraedr()<<endl;
       /*** Вывод объема и поверхности куба  ***/
      cout << "The volume_cube is "<<s.volume_cube() <<"\n";
      cout << "The surface_cube is "<<s.surface_cube() <<"\n";
      cout <<"Press any key ...";
        getch();
     return 0;
}
```

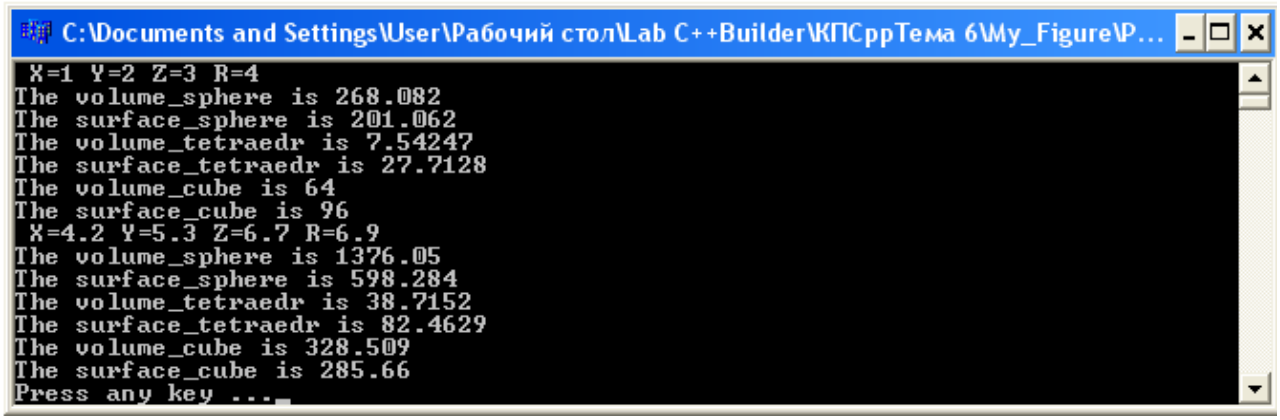



Рис. 6.13. Демонстрация инициализацию общего интерфейса.

**Рекомендация.** Хороший стиль программирования, когда описывают методы за пределами класса.

**Задача 6.7.** Изменим рассмотренный ранее пример класса complex. Добавим метод input_z_com(), предназначенный для ввода действительной и мнимой части числа, члены класса и метод show_complex сделаем закрытыми, а остальные методы открытыми. Текст кода программы будет иметь вид:

```
#include <vcl.h>
#pragma hdrstop
#include <string>
#include <iostream.h>
#include <math.h>
#define PI 3.14159
using namespace std;
class complex //Определяем класс complex
{
   /*** Прототипы функции описываем в классе ***/
 public:    //Открытые методы (прототипы функции)
      complex (double x_re, double y_im) //Конструктор
       { x = x_re; y=y_im; }
      ~complex () {}                     //Деструктор
      void input_z_com(); // Ввод данных
    /*** Метод класса complex - прототип функция modul,
        для вычисления модуля комплексного числа. ***/
      double modul();
    /*** Метод класса complex - прототип функции argument, для
        вычисления аргумента комплексного числа. ***/
      double argument();
    /*** Прототипы функций операции над комплексными числами:
         сложение; вычитание; умножение деление.***/
      void comadd ();   //Сложение
      void comsubst();  //Вычитание
      void commult();   //Умножение
      void comdiv();    //Деление
    /*** Функции, позволяющие осуществить поиск и изменение
         значений (членами не являющимися элементами класса)
         в приватных переменных ***/
     private: //Закрытие методы (прототипы функции) и члены
данных
      double x; // Действительная часть комплексного числа.
      double y; // Мнимая часть комплексного числа.
    /*** Метод класса complex - прототип функции show_complex,
         для вывода комплексного числа.***/
      void show_complex();
};
     //-------------------------------------------------------
     /*** Описание самих функций за пределами объвления класса
***/
     //-------------------------------------------------------
/*** Метод класса complex - функция input_z_com(), для
     вывода данных  ***/
      void complex::input_z_com()
       {
           cout << "Re x= "; cin >> x;
           cout << "Im y= "; cin >> y;
           show_complex();
       }
//-------------------------------------------------------
/*** Метод класса или член-функция complex - функция modul,
     для вычисления модуля комплексного числа. ***/
```

```
 double complex::modul()
{
 input_z_com();
 return pow(x*x+y*y,0.5);
 }
//-------------------------------------------------------
/*** Метод класса complex – функция argument, для вычисления
     аргумента комплексного числа. ***/
 double complex::argument()
 {
   double z;
   cout << "Re x= "; cin >> x;
   cout << "Im y= "; cin >> y;
   if (x>0) z= atan2(y,x)*180/PI;
   else if (y>=0) z=atan2(y,x)*180/PI+PI;
   else z=atan2(y,x)*180/PI-PI;
 return z;
 }
//-------------------------------------------------------------
/*** Метод класса complex – функция show_complex, для вывода
     комплексного числа.  ***/
 void complex::show_complex()
 {
     if (y>=0)
     /* Вывод комплексного числа с положительной мнимой
        частью.*/
    cout << " Z = " << x <<"+"<<y<<"i"<<endl;
     else
     /* Вывод комплексного числа с отрицательной мнимой
        частью. */
    cout<< " Z = "<<x<<y<<"i"<<endl;
 }
   /***  Операции над комплексными числами: сложение;
         вычитание; умножение деление.                ***/
//-------------------------------------------------------------
 void complex::comadd ()  //Сложение
  {
    float x1,x2,y1,y2;
     cout<<"x1="; cin >> x1;
     cout<<"y1="; cin >> y1;
     cout<<"x2="; cin >> x2;
     cout<<"y2="; cin >> y2;
      x1+=x2;
      y1+=y2;
     cout <<"Z = " << x1 << " + " << y1 <<"i  -Add"<<  endl;
  }
//-------------------------------------------------------
 void complex::comsubst()   //Вычитание
  {
    float x1,x2,y1,y2;
     cout<<"x1="; cin >> x1;
     cout<<"y1="; cin >> y1;
     cout<<"x2="; cin >> x2;
```

```
     cout<<"y2="; cin >> y2;
      x1-=x2;
      y1-=y2;
     cout <<"Z = " << x1 << " + " << y1 <<"i  -Subs"<<  endl;
  }
//-------------------------------------------------------------
 void complex::commult()   //Умножение
  {
    float x1,x2,y1,y2,re,im ;
     cout<<"x1="; cin >> x1;
     cout<<"y1="; cin >> y1;
     cout<<"x2="; cin >> x2;
     cout<<"y2="; cin >> y2;
      re=(x1*x2-y1*y2);
      im=(x1*y2+x2*y1);
     cout <<"Z = " << re << " + " << im <<"i  -Mult"<< endl;
  }
//-------------------------------------------------------------
 void complex::comdiv()   //Деление
  {
    float x1,x2,y1,y2,re,im ;
     cout<<"x1="; cin >> x1;
     cout<<"y1="; cin >> y1;
     cout<<"x2="; cin >> x2;
     cout<<"y2="; cin >> y2;
      re=(x1*x2+y1*y2)/(x2*x2+y2*y2);
      im=(x2*y1-x1*y2)/(x2*x2+y2*y2);
     cout <<"Z = " << re << " + " << im <<"i  -Div"<<  endl;
  }
//-------------------------------------------------------------

int main(int argc, char* argv[])
{
//Определяем переменную chislo типа complex.
 complex chislo (1, 1);
    char ans;
    int num;
 do
  {
     cout << " Please enter a number " <<endl;
     cout << " 1 - a method modul() " << endl;
     cout << " 2 - a method argument() " << endl;
     cout << " 3 - a method show_complex() " << endl;
     cout << " 4 - a method comadd() " << endl;
     cout << " 5 - a method comsubst() " << endl;
     cout << " 6 - a method commult() " << endl;
     cout << " 7 - a method comdiv() " << endl;
     cin >> num;
   switch (num)
     {
    /* Вывод модуля комплексного числа, chislo.modul() -
       обращение к методу класса.*/
   case 1: cout<<"Modul' chisla="<<chislo.modul();
```

```
            break;
     /* Вывод аргумента комплексного числа,
        chislo.argument() - обращение к методу класса.*/
    case 2: cout<<"Argument chisla="<<chislo.argument()<<endl;
            break;
     /* Вывод комплексного числа, chislo.show_complex() -
        обращение к методу класса. */
    case 3: chislo.input_z_com();
            break;
    case 4: cout << " Z = z1 + z2 " << endl; chislo.comadd();
            break;
    case 5: cout << " Z = z1 - z2 " << endl;
   chislo.comsubst();
            break;
    case 6: cout << " Z = z1 * z2 " << endl; chislo.commult();
            break;
    case 7: cout << " Z = z1 / z2 " << endl; chislo.comdiv();
            break;
    default: cout << "Please select the number from diapason
                     1-7";
      }
     cout <<" You must type a Y or an N" << endl;
     cout <<" Do you want to continue (Y/N)?" << endl;
     cin >> ans;
   }
  while (ans!='n');
  return 0;
}
//-------------------------------------------------------------
```

В предыдущих примерах было показано совместное использование открытых и закрытых элементов класса. Разделение на открытые и закрытые в этом примере несколько искусственное, оно проведено только для иллюстрации механизма совместного использования закрытых или открытых элементов класса. Если попробовать обратиться к методу show_complex() или к членам класса x, y из функции main, то компилятор выдаст сообщение об ошибке (доступ к элементам класса запрещен). При создании в программе экземпляра класса, формируется указатель this, в котором хранится адрес переменной-экземпляра класса. Указатель this выступает в роли указателя на текущий объект.

В самом примере пока есть недоработки, пытливый читатель пусть не осудит критически, более полную и окончательную версию мы приведем позже в следующей теме, при построении калькулятора для комплексных чисел. Пока мы изучаем, только возможности класса, не вдаваясь скурпулезно в подробности математического аппарта комплексного анализа.

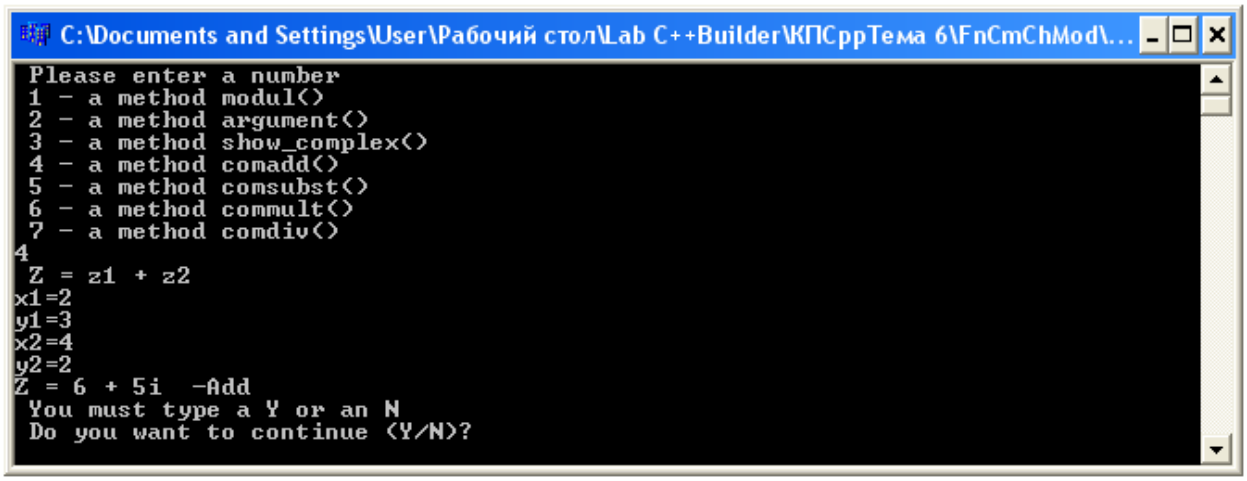



Рис. 6.14. Операция над комплексными числами.

## 6.5. Выделение памяти для массива в конструкторе

Если члены класса, являются массивами (указателями), то в конструкторе логично предусмотреть выделение памяти для него.

**Задача 6.8.** С использованием классов решить следующую задачу. Заданы координаты k точек в n-мерном пространстве. Найти точки, расстояние между которыми наибольшее и наименьшее.

Для решения задачи создадим класс dim_of_space.

***Члены класса:***

- ***int k*** – количество точек;
- ***int n*** – размерность пространства;
- ***double*** `**a` – матрица, в которой хранятся координаты точек, `a[i][j]` - i-я координата точки с номером j.
- ***double*** `min` – минимальное расстояние между точками в n-мерном пространстве;
- ***double*** `max` – максимальное расстояние между точками в n-мерном пространстве;
- ***int*** `imin`, ***int*** `jmin` – точки, расстояние между которыми минимально;
- ***int*** `imax`, ***int*** `jmax` – точки, расстояние между которыми максимально.

***Методы класса:***

- `dim_of_space()`– конструктор класса, в котором определяются k – количество точек, n – размерность пространства, выделяется память для матрицы a координат точки, и вводятся координаты точек;

- `poisk_max()` – функция нахождения точек, расстояние между которыми наибольшее;
- `poisk_min()` – функция нахождения точек, расстояние между которыми наименьшее;
- `output_result()` – функция вывода результатов: значений min, max, imin, jmin, imax, jmax;
- `delete_a()` – освобождение памяти выделенной для матрицы a.

В главной программе необходимо будет описать экземпляр класса и последовательно вызвать методы `poisk_min()`, `poisk_max()`, `output_result()`, `delete_a()`.

UML – диаграммы для задачи 6.8. приведены на рис. 6.15 – 6.19.

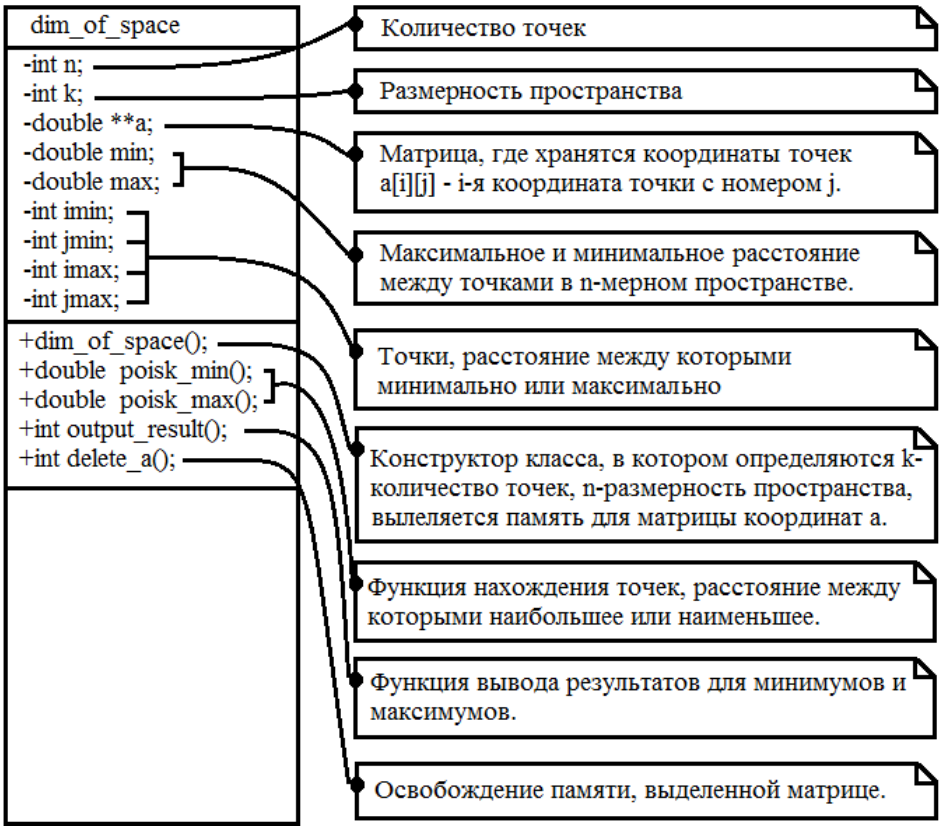



Рис. 6.15. UML диаграмма класса `dim_of_space`.

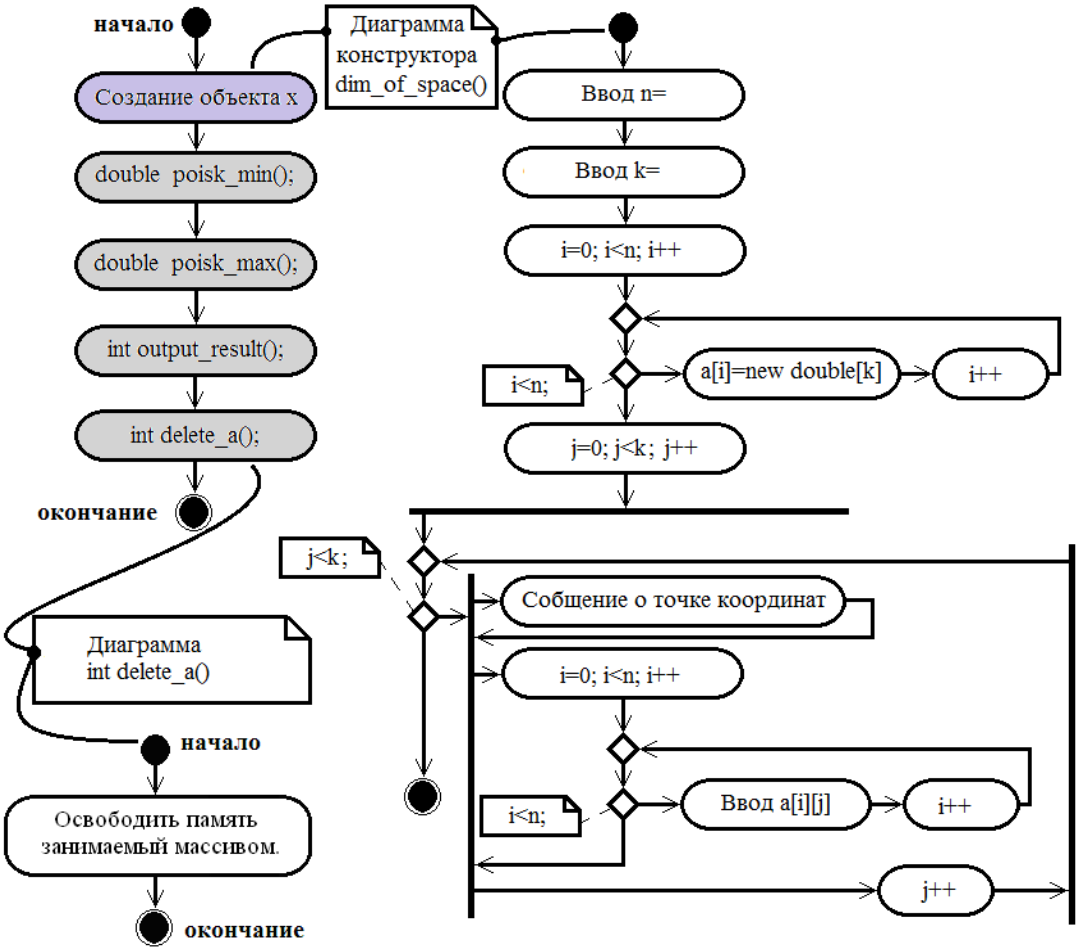



Рис. 6.16. UML диаграммы главной функции, конструктора
и функции delete_a().

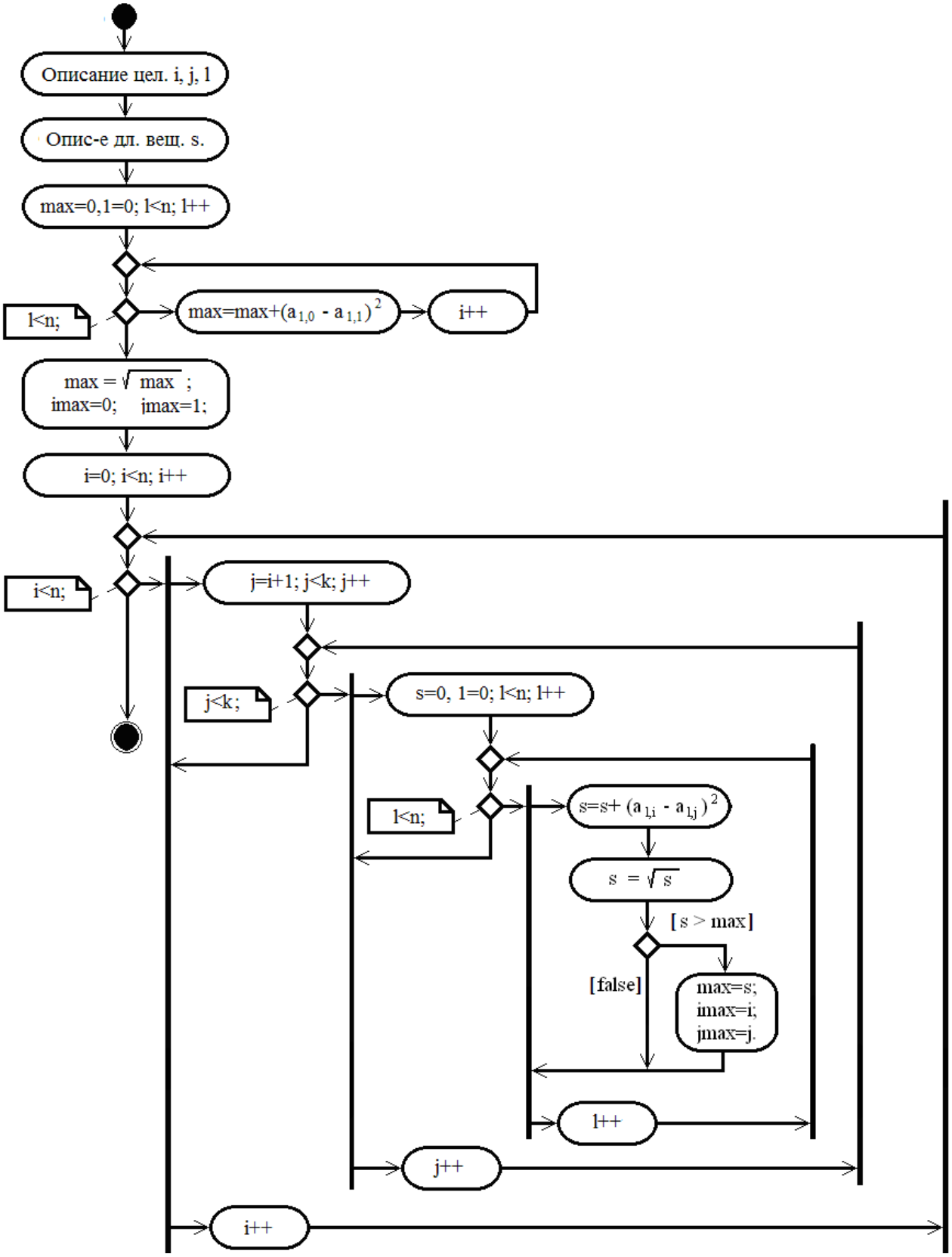



Рис. 6.17. UML диаграмма функции poisk_max.

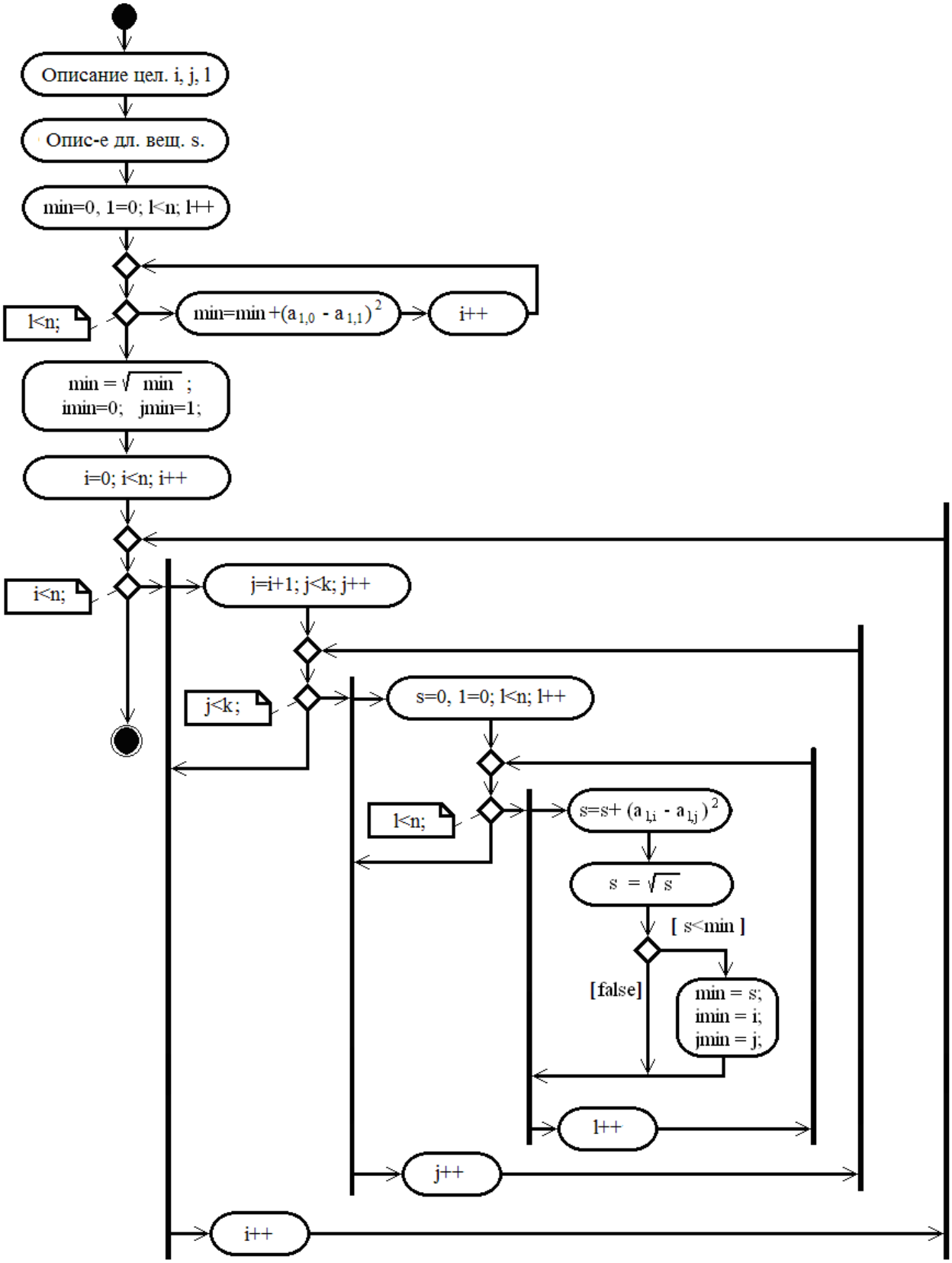



Рис. 6.18. UML диаграмма функции poisk_min.

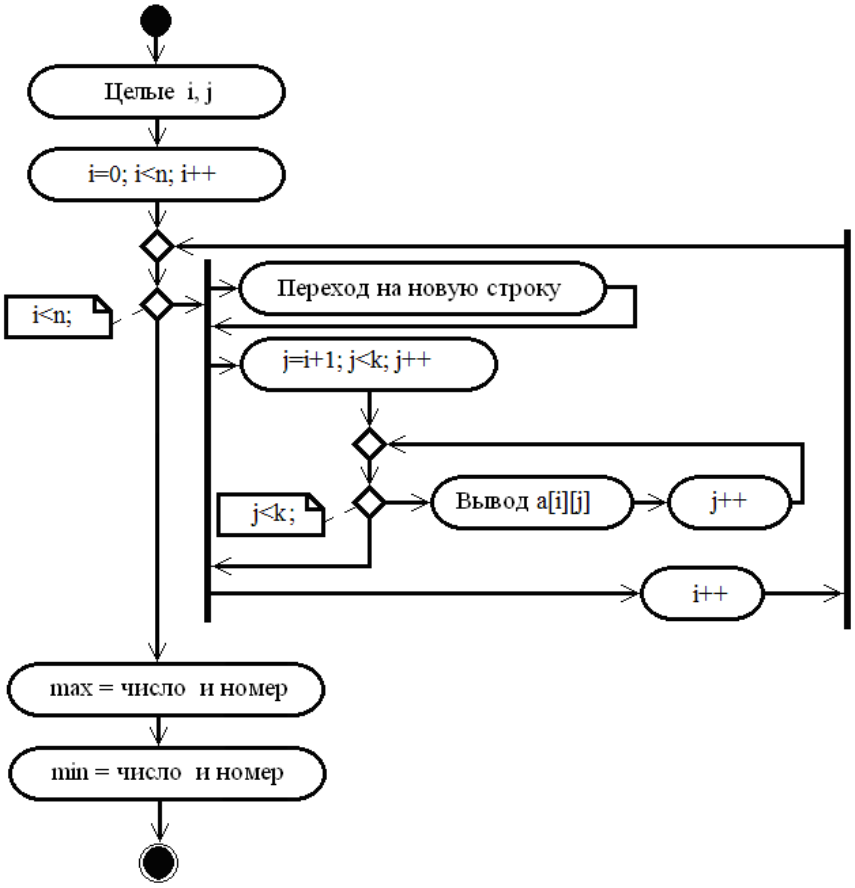



Рис.6.19. UML диаграмма функции output_result.

Текст кода программы:

```
//---------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include <iostream>
#include <math.h>
#include <conio.h>
#pragma argsused
using namespace std;
  // Описываем класс dim_of_space n-мерное пространство
 class dim_of_space{

 public:                        // Открытые методы класса.
    dim_of_space();             // Конструктор класса
    double poisk_min();
    double poisk_max();
    int output_result();
    int delete_a();
 private:                       // Закрытые члены класса.
  int n, k, imin, jmin, imax, jmax;
  double **a;
  double min, max;
};
```

```
int main(int argc, char* argv[])    // Главная функция
{
/*** Описание переменной экземпляра класса dim_of_space, -
     объекта x. ***/
  dim_of_space x;
 /*** Вызов метода poisk_max для поиска максимального
расстояния
      между точками в n-мерном пространстве.   ***/
  x.poisk_max();
 /*** Вызов метода poisk_min для поиска максимального
расстояния
      между точками в n-мерном пространстве.   ***/
  x.poisk_min();
 // Вызов метода output_result для вывода результатов
  x.output_result();
 // Вызов функции delete_a.
  x.delete_a();
  cout <<"Press any key ...";
       getch();
 return 0;
}
//-------------------------------------------------------------
/***       Текст функции конструктор класса dim_of_space
***/
  dim_of_space::dim_of_space()
  {
     int i,j;
     cout<<"Ukagite razmernost prostrantva n="; cin>>n;
     cout<<"Ukagite kolichestvo tochek k="; cin>>k;
     a=new double*[n];
   for(i=0;i<n;i++)
       a[i]=new double[k];
    for(j=0;j<k;j++)
    {
      cout<<"VVedite koordinaty "<<j<<" tochki"<<endl;
      for(i=0;i<n;i++)
       {
        // cout <<" a["<<i<<"]["<<j<<"] = ";
         cin>>a[i][j];
       }
    }
  }
//-------------------------------------------------------------
/***        Текст метода poisk_max класса dim_of_space.
***/
double dim_of_space::poisk_max()
  {
     int i,j,l;
     double s;
    for(max=0,l=0;l<n;l++)
       max+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
    max=pow(max,0.5);
    imax=0;jmax=1;
```

```
    for(i=0;i<k;i++)
      for(j=i+1;j<k;j++)
      {
         for(s=0,l=0;l<n;l++)
           s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
           s=pow(s,0.5);
           if (s>max)
             {
               max=s;
               imax=i;
               jmax=j;
             }
      }
   return 0;
  }
//-------------------------------------------------------
/*** Текст метода poisk_min класса dim_of_space  ***/
  double dim_of_space::poisk_min()
  {
       int i,j,l;
       double s;
     for(min=0,l=0;l<n;l++)
       min+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
     min=pow(min,0.5);
     imin=0;jmin=1;
     for(i=0;i<n;i++)
       for(j=i+1;j<k;j++)
         {
           for(s=0,l=0;l<n;l++)
             s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
           s=pow(s,0.5);
           if (s<min)
             {
               min=s;
               imin=i;
               jmin=j;
             }
         }
   return 0;
  }
//-------------------------------------------------------
/***Текст метода output_result класса dim_of_space  ***/
  int dim_of_space::output_result()
  {
      int i,j;
    for(i=0;i<n;cout<<endl,i++)
      for (j=0;j<k;j++)
         cout<<a[i][j]<<"\t";
    cout<<"max="<<max<<"\t nomera "<<imax<<"\t"<<jmax<<endl;
    cout<<"min="<<min<<"\t nomera "<<imin<<"\t"<<jmin<<endl;
   return 0;
  }
//-------------------------------------------------------
```

```
/*** Текст функции деструктор класса dim_of_space. ***/
int dim_of_space::delete_a
{
  delete [] a;
  return 0;
}
//---------------------------------------------------------
```

***Результаты работы программы:***

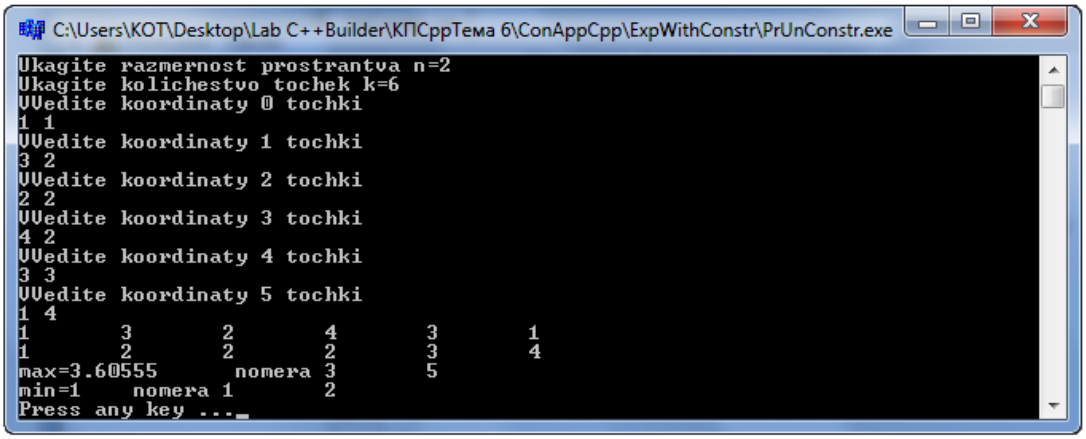



Рис.6. 20. Тест программного кода 6.8.

**Конструктор копирования (перегрузка конструктора).** Выше мы немного говорили о конструкторе копирования (см. рис. 11.). Дадим определение конструктору копирования.

*Конструктор, принимающий ссылку на собственный класс, называется конструктором копирования.*

Общая форма конструктора копирования имеет вид:

```
1. Имя_класса (const и/или volatile имя_класса
&ссылка_на_объект)
            {
              // тело конструктора
            }
```

или

```
2. Имя_класса (имя_класса &ссылка_на_объет)
            {
              // тело конструктора
            }
```

*Конструктор копирования может иметь также дополнительные параметры, если для них определены значения по умолчанию. Однако в*

*любом случае первым параметром должна быть ссылка на объект, выполняющий инициализацию.*

Напоминаем, инициализация возникает в трех случаях:

- один объект инициализирует другой;
- копия объекта передается в функцию;
- создается временный объект, обычно, как возвращаемым значением.

Например, любая из следующих инструкций вызывает инициализацию:

```
my_class x = y; // инициализация
func (x); // передача параметра
x = func (); // получение временного объекта
```

Второй формат конструктора позволяет изменить копируемый объект. Это довольно редкая ситуация, однако она предусмотрена в стандартной библиотеке вызовом std::auto_ptr.

***Внимание!*** *Если есть два вида конструктора копирования, тогда первый вид неуместен!*

Теперь, для наглядности приведем возможные и допустимые форматы конструкторов копирования класса `CopyConsr`:

```
 CopyConsr (CopyConsr&);                          //(1)
CopyConsr (CopyConsr const&);                     //(2)
     CopyConsr (CopyConsr const volatile&);              //(3)
     CopyConsr (CopyConsr volatile&);                    //(4)
     CopyConsr (CopyConsr const&, int = 69);             //(5)
     CopyConsr  (CopyConsr  const&,  double=3.14,  int=33);
                                                         //(6)
```

Следующие конструкторы копирования (или постоянные конструкторы) некорректны:

```
CopyConsr (CopyConsr )
CopyConsr (CopyConsr const),
```

так как вызов этих конструкторов потребует очередного копирования, что приведет к бесконечному рекурсивному вызову (т.е. бесконечный цикл).

Напоминаем, конструктор копирования, является необходимым членом –функции, т.к. по умолчанию, его может генерировать компилятор.

Как известно по умолчанию при инициализации одного объекта другим интерпретатор или компилятор C++ выполняет побитовое копирование, однако при этом точная копия инициализирующего объекта создается в целевом объекте. Хотя в большинстве случаев такой способ инициализации объекта является вполне приемлемым, (в

предыдущих примерах мы не обращали на это внимание) существует варианты ситуаций, когда побитовое копирование не может использоваться. Например, такой вариант ситуации имеет место, когда объект выделяет память при своем создании. Рассмотрим в качестве примера два объекта `rjevsky` и `pouchik_rjevsky` класса `Anegdot_Person`, выделяющего память при создании объекта. Если объект `rjevsky` уже существует, следовательно, ему уже выделил память. И предположим, что `rjevsky` используется для инициализации объекта `pouchik_rjevsky`, как показано ниже в задаче 6.9.:

**Задача 6.9.** Выполнить копирование используя, конструктор копирования по умолчанию.

Задание 2. Самостоятельно создать UML диаграмму для задачи 6.9.

Текст кода программы

```
//---------------------------------------------------------------
#include <vcl.h>
    #pragma hdrstop
    #include <iostream.h>
    #include <conio.h>
    #pragma argsused
    using namespace std;
    class Anegdot_Person
    {
     public:
        int age;
        Anegdot_Person(int age): age(age) {}
        ~Anegdot_Person(){}
    };
    int main(int argc, char* argv[])
    {
    /*** Интерпретатор сгенировал по умолчанию примерно такой
         конструктор:Anegdot_Person(Anegdot_Person       const&
copy):age(copy.age){}     ***/
       Anegdot_Person rjevsky(30);
       Anegdot_Person nataly(15);
       Anegdot_Person poruchik_rjevsky = rjevsky;
       cout << rjevsky.age << " " << nataly.age
           << "  "<< poruchik_rjevsky.age << endl;
       rjevsky.age = 33;
       cout << rjevsky.age <<" " << nataly.age
            << " " << poruchik_rjevsky.age << endl;
       cout <<"Press any Key ...";
      getch();
     return 0;
    }
```

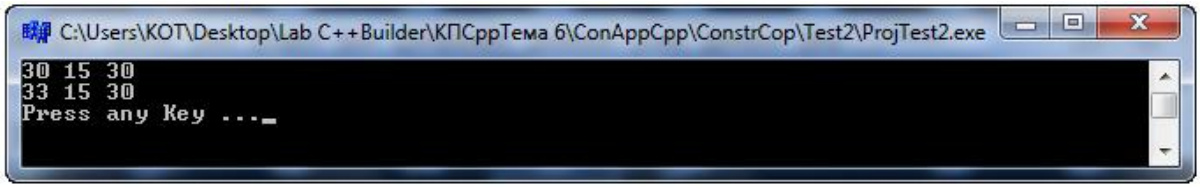



Рис. 6.2121. Результаты работы конструктора по умолчанию.

При побитовом копировании `pouchik_rjevsky` станет точной копией `rjevsky`. Это означает, что `pouchik_rjevsky` будет использовать тот же самый участок выделенной памяти, что и `rjevsky`, вместо того, чтобы выделить свой собственный. Ясно, что такая ситуация нежелательна. Например, если класс `Anegdot_Person` включает в себя деструктор, освобождающий память, то тогда одна и та же память будет освобождаться дважды при уничтожении объектов `rjevsky` и `pouchik_rjevsky`!

Проблема того же типа может возникнуть еще в двух случаях. Первый из них возникает, когда копия объекта создается при передаче в функцию объекта в качестве аргумента. Второй случай возникает, когда временный объект создается функцией, возвращающей объект в качестве своего значения. (Не надо забывать, что временные объекты автоматически создаются, чтобы содержать возвращаемое значение функции, и они могут также создаваться в некоторых других ситуациях.)

Для решения подобных проблем язык C++ позволяет создать конструктор копирования, который используется компилятором, когда один объект инициализирует другой. При наличии конструктора копирования побитовое копирование не выполняется.

**Задача 6.10**. Написать собственный конструктор копирования. Скопировать данные массива используя, глубокое копирование.

Текст кода программы

```
//---------------------------------------------------------
#include <vcl.h>
    #pragma hdrstop
    #include <iostream.h>
    #include <algorithm>
    #include <conio.h>
    //---------------------------------------------------------
    #pragma argsused
    using namespace std;
    class Array
    {
      public:
        int size;
```

```
    int* data;
     Array(int size)
        : size(size), data(new int[size]) {}
             //  для std::copy добавьте #include <algorithm>
     Array(Array const& copy)
        : size(copy.size), data(new int[copy.size])
        {
         std::copy(copy.data, copy.data + copy.size, data);
        }
     ~Array()
    {
        delete[] data;
    }
};
int main(int argc, char* argv[])
{
/***   Вызываем наш собственный конструктор копирования,
       Выполняющий глубокое копирование                  ***/
   Array first(20);
    first.data[0] = 25;
    {
        Array copy = first;
       cout << first.data[0] << " " << copy.data[0] << endl;
    }    // (1)
    first.data[0] = 10;    // (2)
     cout <<"Press any Key ...";
     getch();
          return 0;
}
```


Рис. 6.22. Результаты работы задачи 6.10.

***Рекомендация.*** *К построению* ***UML*** *диаграммы задачи 6.10. вернитесь после изучения параграфов наследование и полиморфизм.*

Для четкого представления сути функционирования конструктора копирования, принципиально, важно понять следующие два вопроса.

1. *Какое копирование осуществляет стандартный конструктор копирования?*

*Стандартный конструктор копирования* вызывает конструкторы копирования членов-данных и членов-функций. Неглубокое копирование, если по умолчанию или глубокое копирование по требованию (*глубокое копиование* - копирование значений полей класса, даже если они заданы указателями).

***Внимание!*** *Определять конструктор копирования, желательно для классов, моделирующих динамические структуры данных. Конструкторы копирования должны выполнять т.н. глубокое копирование, которое подразумевает копирование динамических данных. Если вы не определили конструктор копирования, напоминаем, компилятор создаст конструктор копирования по умолчанию, который будет создавать поверхностную копию, копируя только элементы-данные. При этом будет скопировано содержимое данных-элементов, содержащих указатели на другие, данные, но сами эти данные скопированы не будут.*

**Не полагайтесь** на поверхностный конструктор копирования для классов имеющих данные-указатели.

Иногда путают копирование и присваивание.

**Чем отличается копирование от присваивания?**

Копирование и присваивание это разные операции. При копировании – работает конструктор копирования, при этом создается новый объект, а при операции присваивание – поля одного объекта заполняются значениями другого(operator= уничтожает объект), и новый объект не создается.

Ранее мы указывали, чтобы конструкторы объявлялись в открытом разделе, однако могут быть и в закрытом разделе. Конструкторы, не являющиеся открытыми, в реальных программах C++ чаще всего используются для:

- предотвращения копирования одного объекта в другой объект того же класса;
- указания на то, что конструктор должен вызываться только в случае, когда данный класс выступает в роли базового в иерархии наследования, а не для создания объектов, которыми программа может манипулировать напрямую (см. обсуждение наследования ).
- На рис. 11. мы изобразили различные конструкторы и их форматы. Теперь более основательно Копирование объектов выполняется за счет использования конструктора копирования и копирования оператора присваивания. Конструктор копирования в качестве первого параметра (типа const или volatile) принимает ссылку на собственный тип класса. Он может еще принимать аргументы, но все остальные тогда должны иметь связанные с этими аргументами значения по умолчанию.

## 6.6. Перегрузка операторов и функций

Встречаются задачи, когда один и тот же алгоритм необходимо выполнять для различных типов данных. И для удобства его определить одним именем для весех типов. Если это имя мнемонично, то есть несет нужную информацию, это делает программу более понятной, поскольку для каждого действия требуется помнить только одно имя. *Использование нескольких функций с одним и тем же именем, но с различными типами параметров, называется перегрузкой функций.*

Разрешение (перевод английского слова resolution в смысле «уточнение») процесса перегрузки определяет компилятор, т.е. какую именно функцию требуется вызвать, по типу фактических параметров. Тип возвращаемого функцией значения в разрешении не участвует. Механизм разрешения основан на достаточно сложном наборе правил, смысл которых сводится к тому, чтобы использовать функцию с наиболее подходящими аргументами и выдать сообщение, если такой не найдется. Допустим, имеется четыре варианта функции, определяющей наибольшее значение:

```
   int max(int, int ); // Возвращает max из двух целых
   char* max(char*, char*); //Возвращает подстроку max длины
// Возвращает max, из первого параметра и длины второго:
   int max (int, char*);
// Возвращает max из второго параметра и длины первого:
   int max(char*, int);
   void f(int a, int b, char* c, char* d)
   {
      cout « max(a, b) « max(c, d) « max(a, c) « max(c, b);
   }
```

При вызове функции max компилятор выбирает соответствующий типу фактических параметров вариант функции (в приведенном примере будут последовательно вызваны все четыре варианта функции).

Если точного соответствия не найдено, выполняются продвижения порядковых типов в соответствии с общими правилами, например, ***bool*** и ***char*** в ***int***, ***float*** в ***double*** и т.п. Далее выполняются стандартные преобразования типов, например, ***int*** в ***double*** или указателей в ***void****. Следующим шагом является выполнение преобразований типа, заданных пользователем, а также поиск соответствий за счет переменного числа аргументов функций. Если соответствие на одном и том же этапе может быть получено более чем одним способом, вызов считается неоднозначным и выдается сообщение об ошибке.

***Неоднозначность может появиться при:***

- *преобразовании типа;*
- *использовании параметров-ссылок;*

- *использовании аргументов по умолчанию.*

***Правила описания перегруженных функций.***

- *Перегруженные функции должны находиться в одной области видимости, иначе произойдет сокрытие аналогично одинаковым именам переменных во вложенных блоках.*
- *Перегруженные функции могут иметь параметры по умолчанию, при этом значения одного и того же параметра в разных функциях должны совпадать.*
- *В различных вариантах перегруженных функций может быть различное количество параметров по умолчанию.*
- *Функции не могут быть перегружены, если описание их параметров отличается только модификатором const или использованием ссылки (например,* `int` *и* `const int` *или* `int` *и* `int&`*).*

Перегрузка операций C++ позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные функции.

Эта дает возможность использовать собственные типы точно так же, как стандартные. Обозначения собственных операций вводить нельзя. Можно перегружать любые операции, существующие в C++, за исключением:

```
.* ?: # ## sizeof
```

Перегрузка операций осуществляется с помощью методов специального вида (функций-операций), и подчиняется следующим правилам:

- при перегрузке операций сохраняются параметры, приоритеты операций и правила ассоциации (справа налево или слева направо), используемые в стандартных типах данных;
- для стандартных типов данных переопределение операции не допустимо;
- в функции-операции не допускаются параметры по умолчанию;
- функции-операции допускает наследование (за исключением =*);
- функции-операции нельзя определять типом static.

Функцию-операцию можно определить тремя способами: она должна быть либо методом класса, либо дружественной функцией класса, либо обычной функцией.

В двух последних случаях функция должна принимать хотя бы один аргумент, имеющий тип класса, указателя или ссылки на класс.

Функция-операция содержит ключевое слово ***operator***, за которым следует знак переопределяемой операции:

тип operator операция (список параметров) { тело функции }

Также как операторы и любые другие функции, конструкторы могут перегружаться.

В C++ существует возможность перегрузки операции внутри класса, например, можно добиться того, что операция * при работе с матрицами осуществляла умножение матриц, а при работе с комплексными числами – умножение комплексных чисел.

***Внимание!*** *Для перегрузки операций внутри класса нужно написать специальную функцию – метод класса. При перегрузке операций следует помнить следующее:*

- *при перегрузке нельзя изменить приоритет операций;*
- *нельзя изменить тип операции (из унарной операции нельзя сделать бинарную или наоборот);*
- *перегруженная операция является членом класса и может использоваться только в выражениях с объектами своего класса;*
- *нельзя создавать новые операции;*
- *запрещено перегружать операции: . (доступ к членам класса), унарную операцию *(значение по адресу указателя), ::(расширение области видимости) , ?: (операция if);*
- *допустима перегрузка следующих операций: `+, -, *, /, %, =, <, >, +=, -=, *=, /=, <, >, &&, ||, ++, --, (), [], new, delete`.*
- *нельзя перегружать **`sizeof, ., .*, typeid`**, т.к. их операнд – это имя типа.*

Для перегрузки бинарной операции внутри класса необходимо создать функцию-метод:

```
 type operator symbols(type1 parametr)
{
      операторы;
   }
```

где ***type*** – *тип возвращаемого операцией значения,*

- ***operator*** – *служебное слово,*
- *symbols – перегружая операция,*
- ***type1*** – *тип второго операнда, первым операндом является экземпляр класса,*
- *parametr – имя переменной второго операнда.*

**Задача 6.11.** Создать класс для работы с матрицами, перегрузив операции сложения, вычитания и умножения.

Приведем сначала UML – диаграммы для решаемой задачи. Прежде всего, опишем класс matrix, котрый является основой всей программы. Затем диаграмму главного модуля и методов, которыми пользуется данный класс.

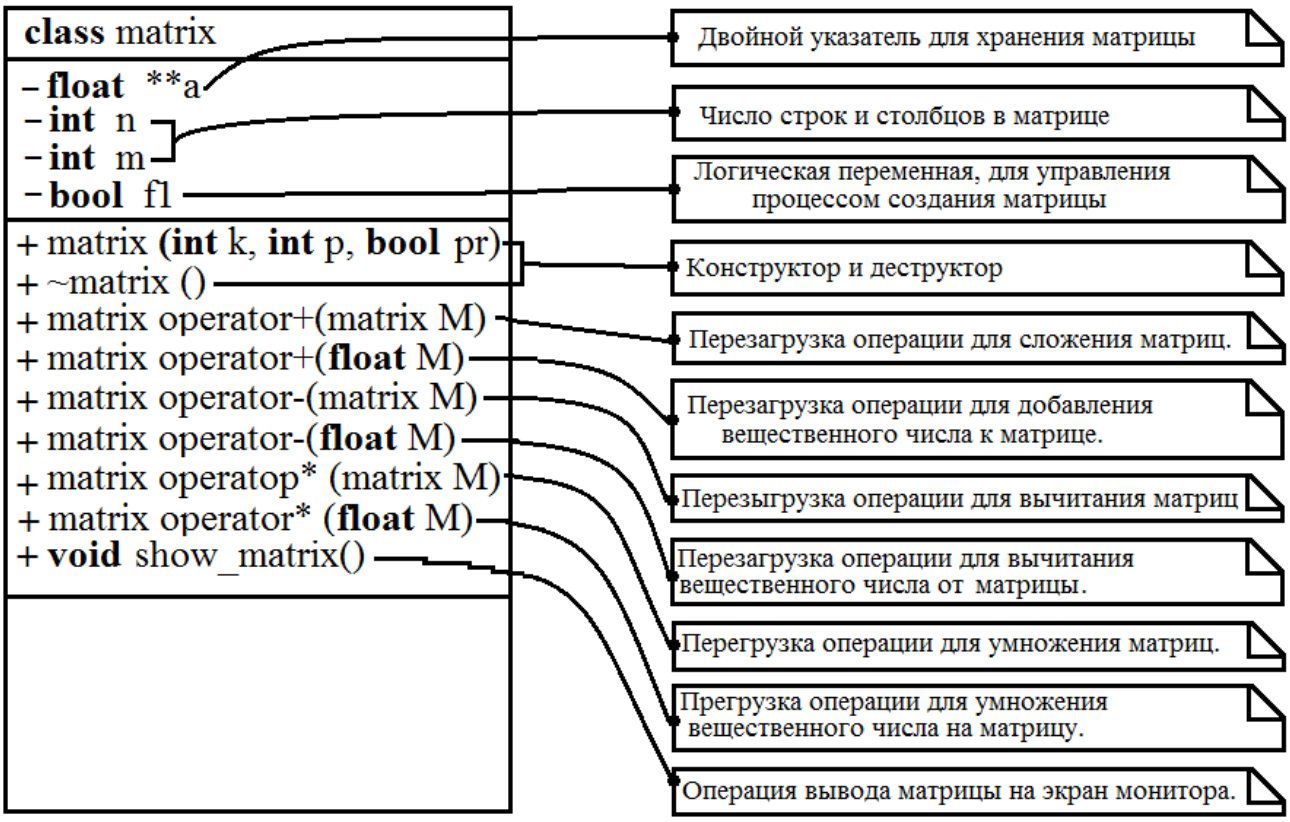



Рис.6.22. Класс matrix.

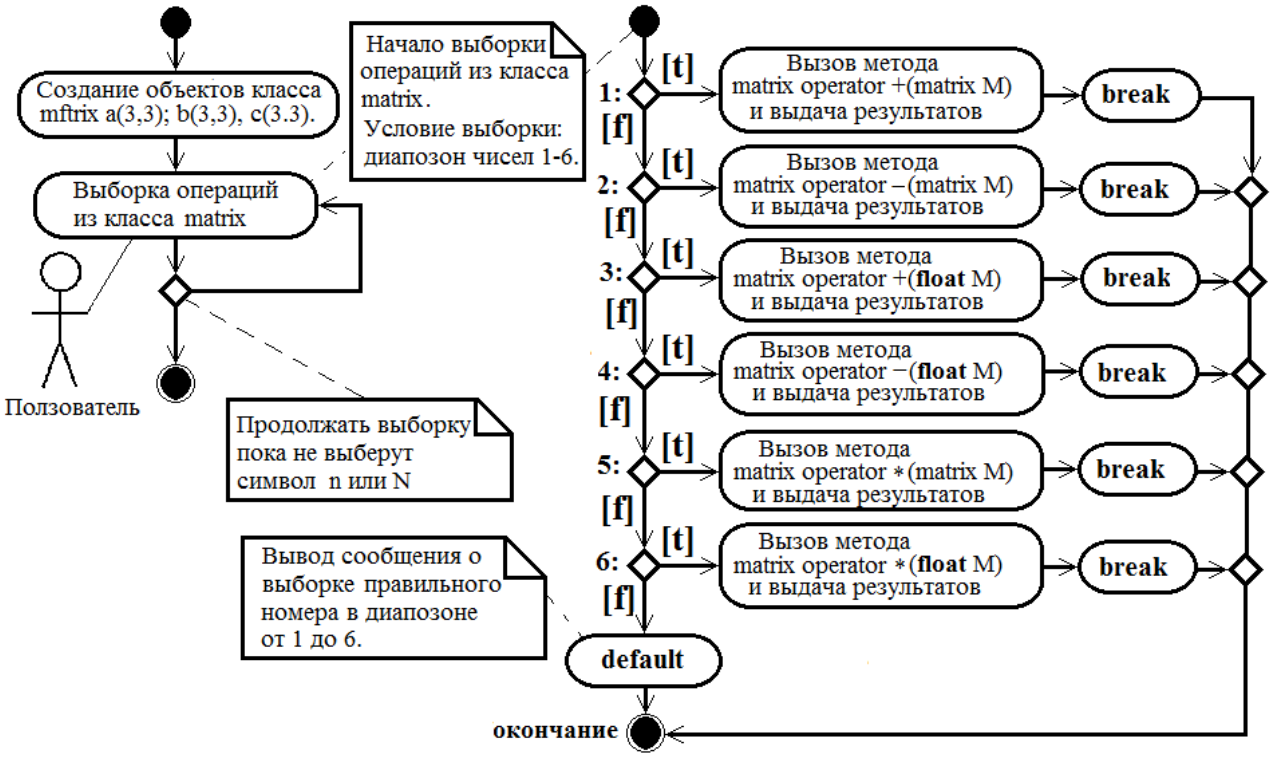



Рис.6. 23. UML – диаграмма для главной функции задачи 6.11.

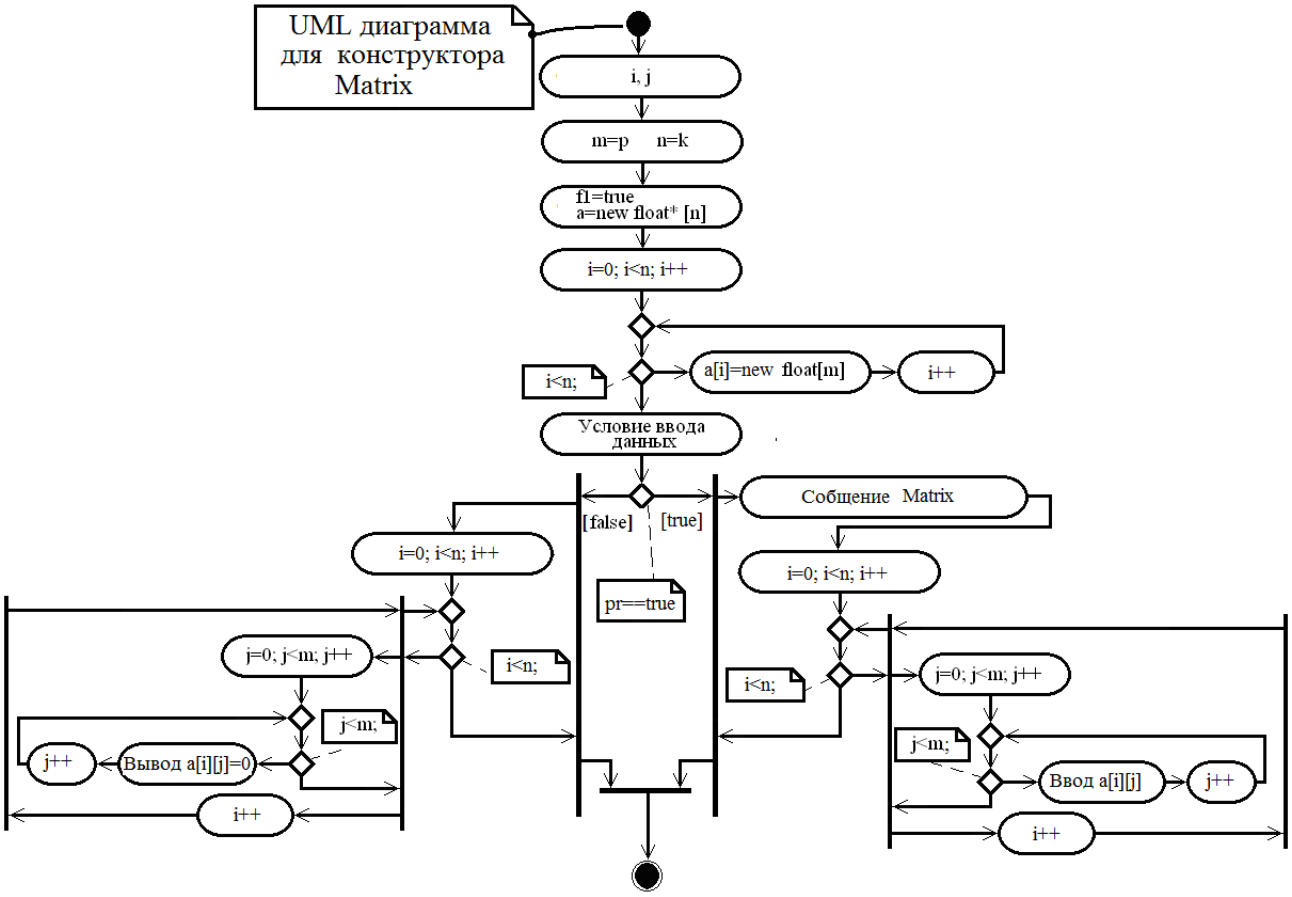



Рис.6. 24. UML – диаграмма конструктора класса matrix.

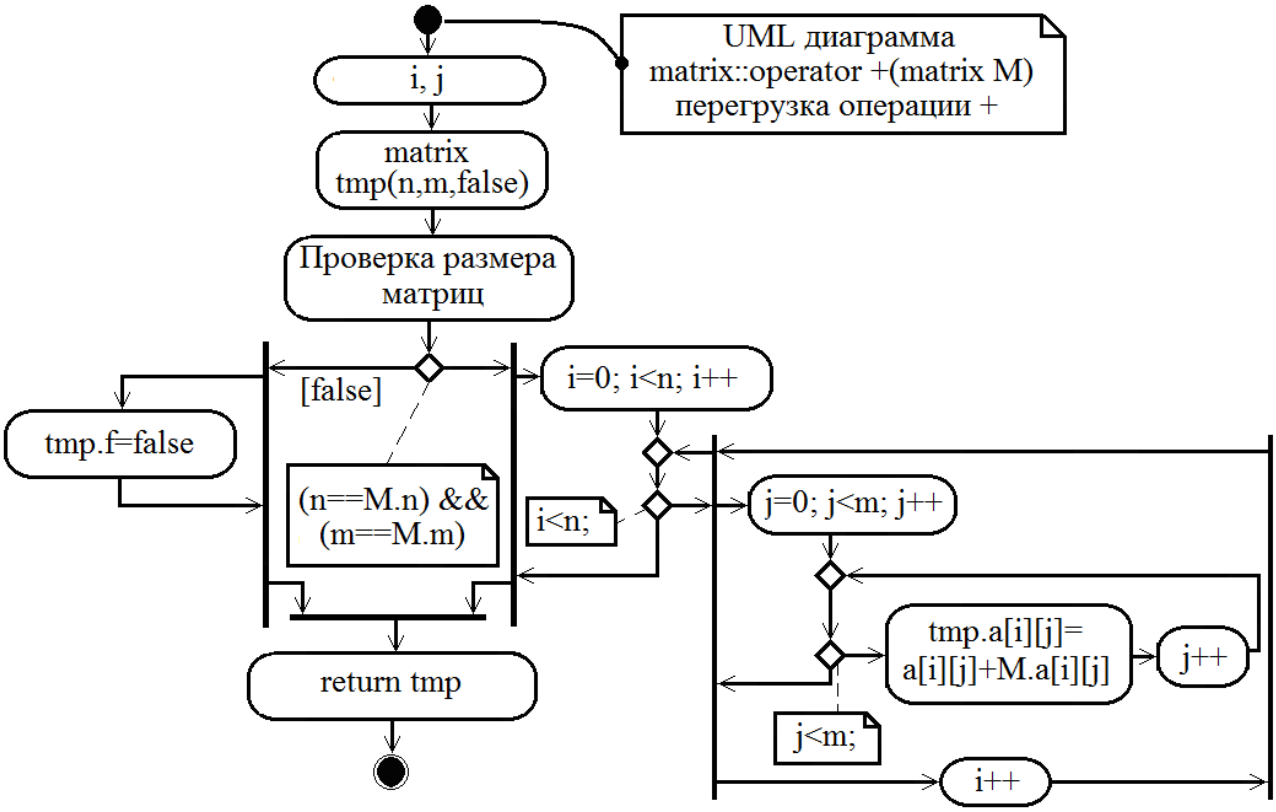



Рис.6.25. UML – диаграмма для пергрузки операции сложения
двух массивов в классе matrix.

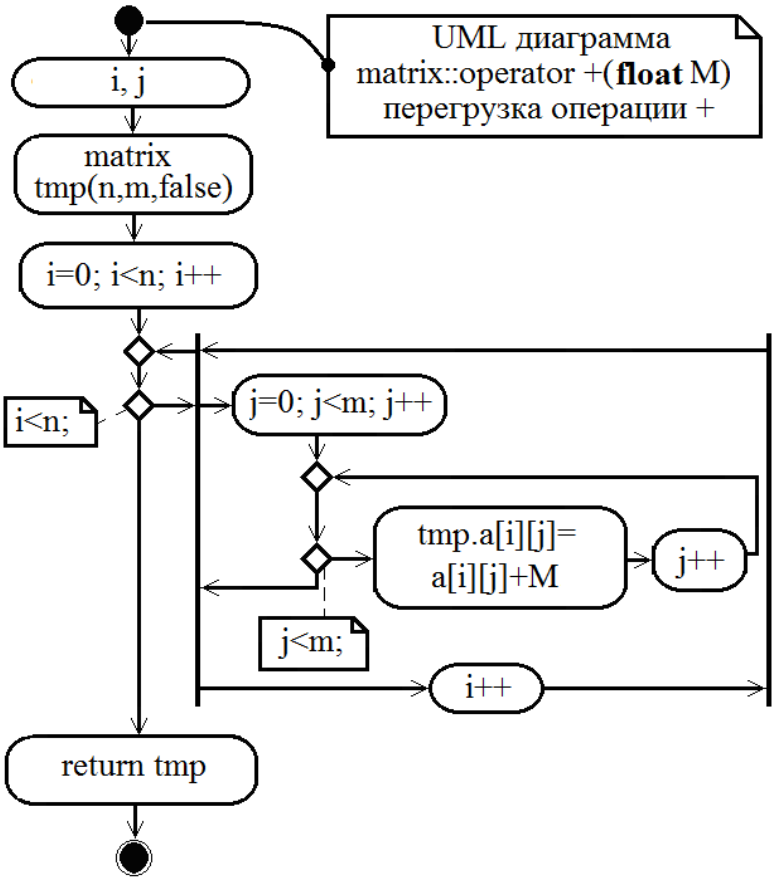



Рис.6.26. UML – диаграмма, для пергрузки операции сложения - массива с вещественным числом в классе matrix.

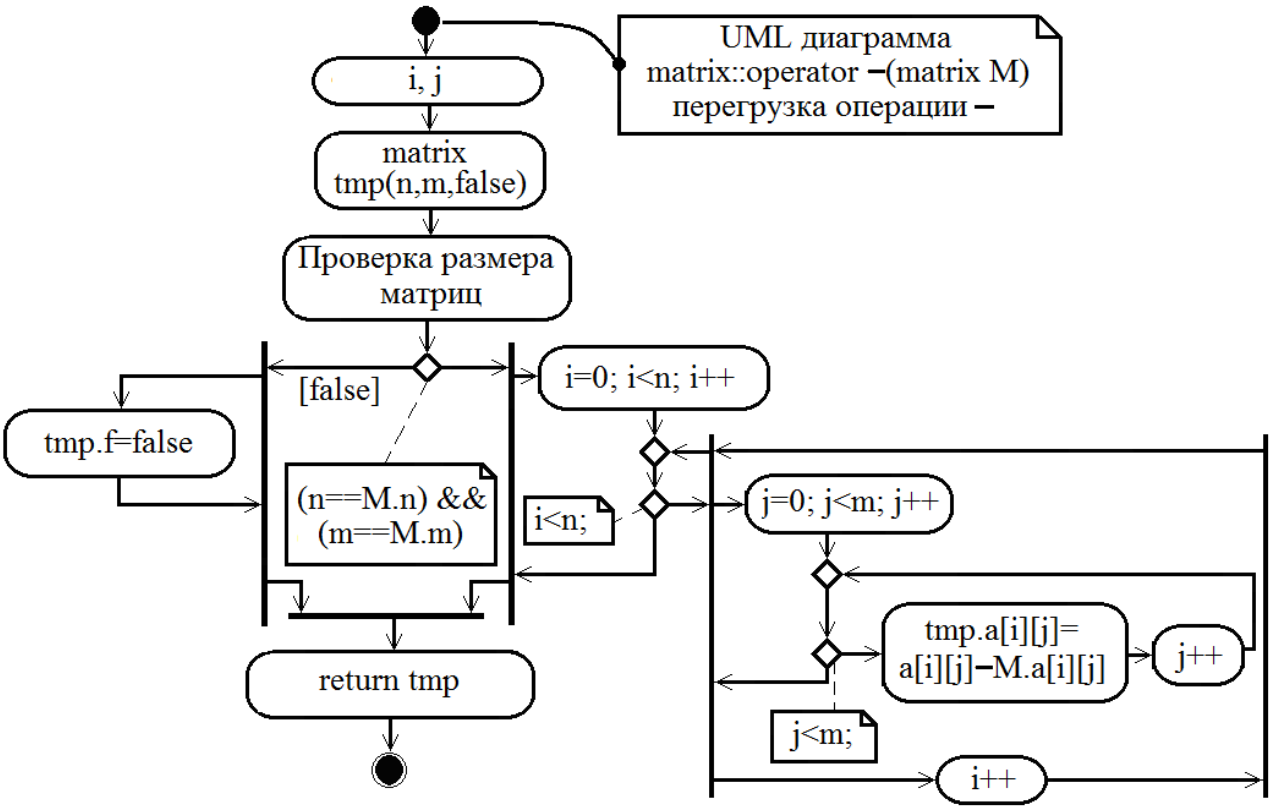



Рис.6.27. UML – диаграмма для пергрузки операции вычитания двух массивов в классе matrix.

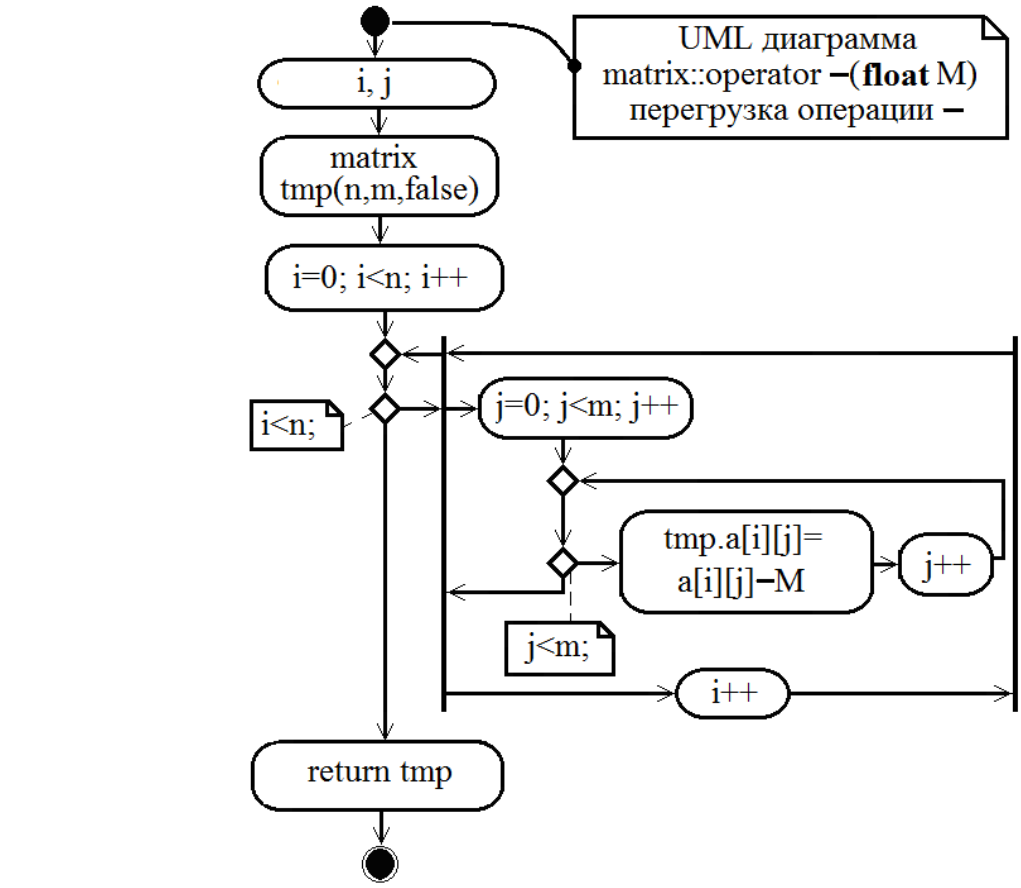



Рис. 6.28. UML –диаграмма, для пергрузки операции вычитания - вещественного числа от массива в классе matrix.

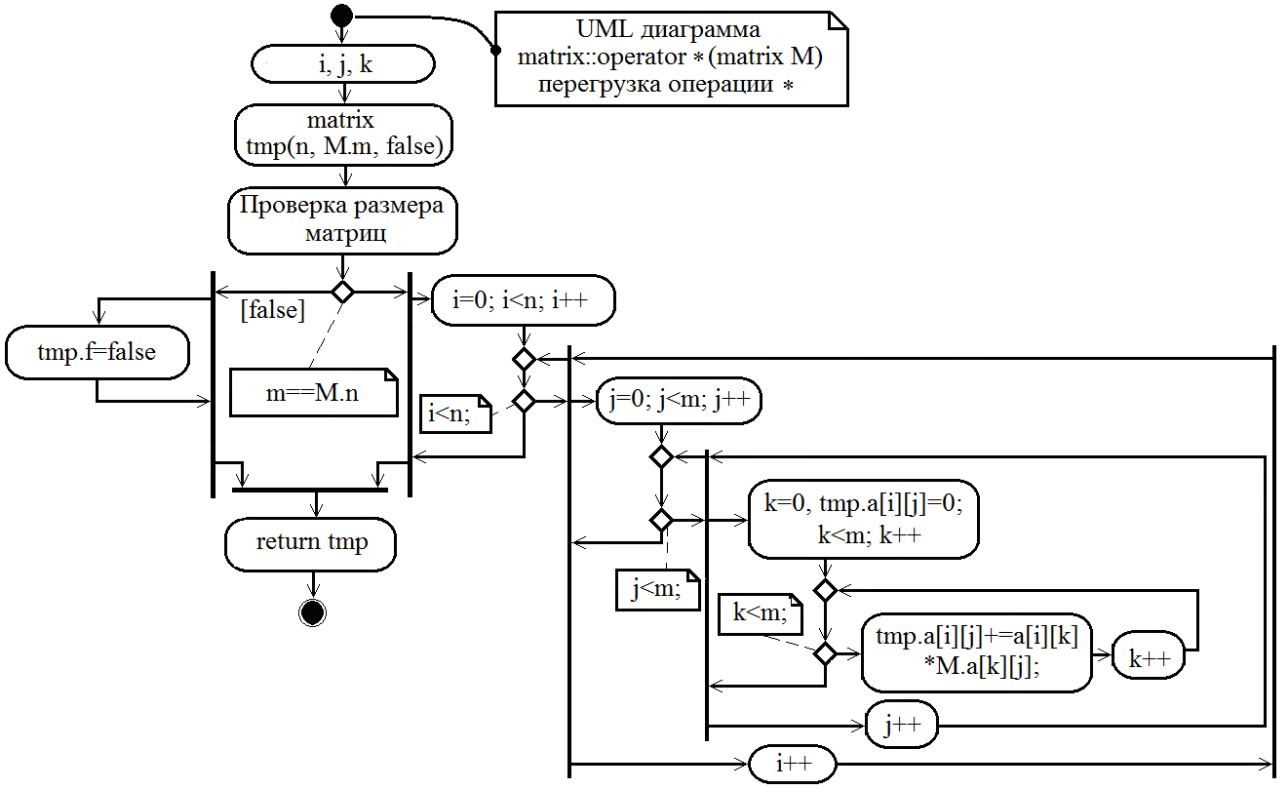



Рис. 6.29. UML – диаграмма для пергрузки операции умножения двух массивов в классе matrix.

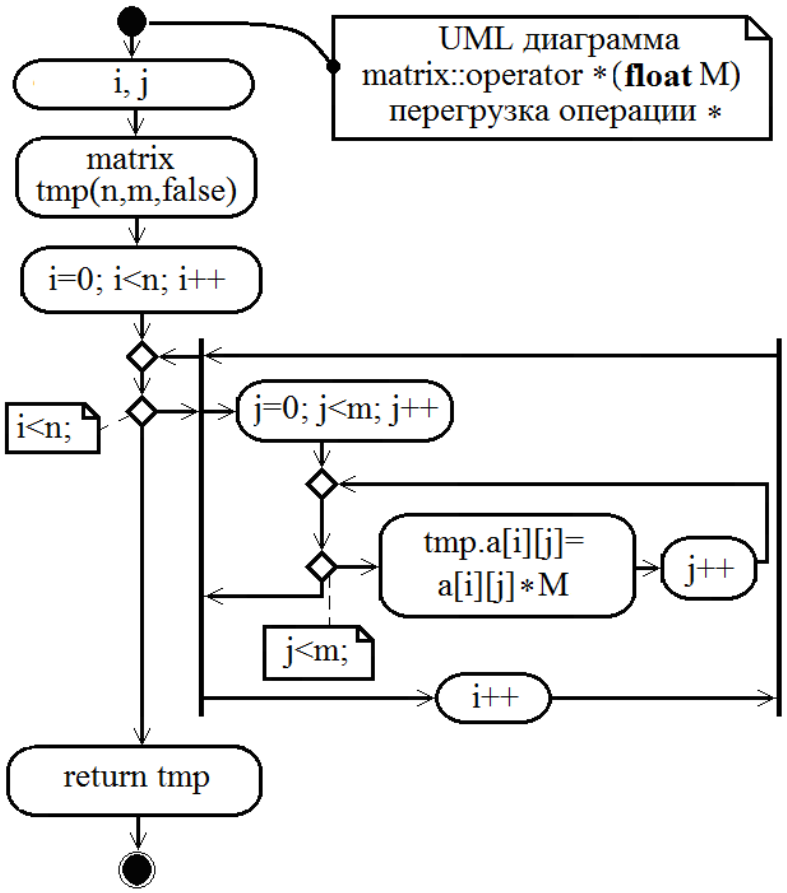



Рис. 6.30. UML – диаграмма, для пергрузки операции умножения - массива на вещественное число в классе matrix.

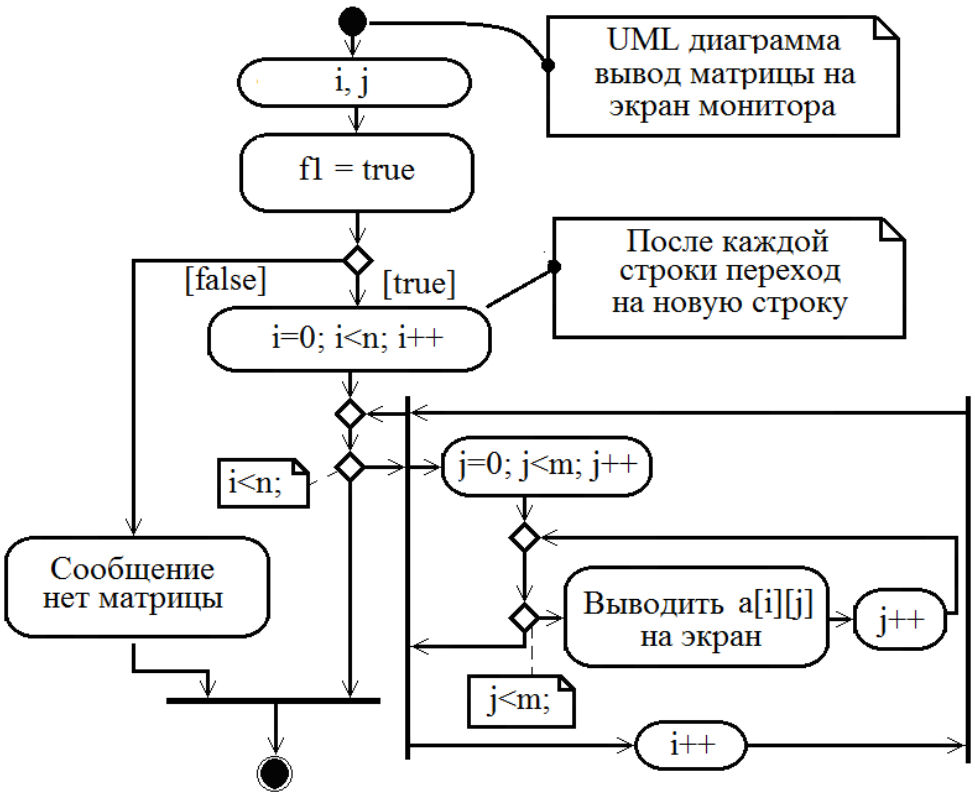



Рис. 6.31. UML – диаграмма, позволяющая, вывести результаты операций перегрузки на экран монитора в классе matrix.

Программный код для задачи 6.11.

```
//---------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include <iostream>
#include <math.h>
#include <conio.h>
//---------------------------------------------------------
#pragma argsused
using namespace std;
class matrix {
public:
//Конструктор
  matrix (int k, int p, bool pr=true);
/*Перегрузка операции + для выполнения операции сложения
матриц.*/
  matrix operator+(matrix M);
/*Перегрузка операции + для выполнения операции добавления к
матрице вещественного числа.*/
  matrix operator+(float M);
/*Перегрузка операции - для выполнения операции вычитания
матриц.*/
  matrix operator-(matrix M);
/*Перегрузка операции - для выполнения операции вычитания из
матрицы вещественного числа.*/
  matrix operator-(float M);
/*Перегрузка операции * для выполнения операции умножения
матриц.*/
  matrix operator*(matrix M);
/* Перегрузка операции * для выполнения операции умножения
матрицы на вещественное число. */
matrix operator*(float M);
  ~matrix();
/* Метод вывода матрицы на экран построчно. */
  void show_matrix();
private:
  float **a; //Двойной указатель для хранения матрицы.
  int n;     //Число строк в матрице.
  int m;     //Число столбцов в матрице.
  bool fl;   //Логическая переменная fl принимает значение
false,
             //если матрицу сформировать не удалось.
};
/***                    Главная функция.
***/
int main()
{
   matrix a1(3,3), b1(3,3), c1(3,3,false);
   c1=a1+b1;
   c1.show_matrix();
   c1=a1-100;
```

```
   c1.show_matrix();
   c1=b1*5;
   c1.show_matrix();
   c1=a1*b1;
   c1.show_matrix();
}
/***              Конструктор.                          ***/
matrix::matrix(int k, int p,bool pr)
{
 int i,j;
 n=k;
 m=p;
 fl=true;
 a=new float*[n];
 for(i=0;i<n;i++)
    a[i]=new float[m];
 if (pr)
    {
      cout<<"Matrix"<<endl;
      for(i=0;i<n;i++)
        for(j=0;j<m;j++)
         {
           cout <<" a["<<i<<"]["<<j<<"]= ";
           cin>>a[i][j];
         }
    }
 else
    for(i=0;i<n;i++)
      for(j=0;j<m;j++)
       {
            cout <<" a["<<i<<"]["<<j<<"]= ";
            a[i][j]=0;
       }
}
/***     Перегрузка операции сложения матриц.      ***/
matrix matrix::operator+(matrix M)
{
   int i,j;
      /* Временная матрица tmp для хранения результата
сложения
         двух матриц. */
   matrix tmp(n,m,false);
    /*   Если обе матрицы одинакового размера, то формируем
матрицу
         tmp, как сумму матриц.    */
   if ((n==M.n)&&(m==M.m))
     {
      for(i=0;i<n;i++)
       for(j=0;j<m;j++)
         tmp.a[i][j]=a[i][j]+M.a[i][j];
     }
   else
    /*   Если размеры матриц не совпадают, то fl=false
```

```
        (результирующую матрицу сформировать не удалось).
*/
   tmp.fl=false;
return tmp;   //Возвращаем результат операциq - матрицу tmp.
}
/***   Перегрузка операции + для выполнения операции
      добавления к матрице вещественного числа.       ***/
matrix matrix::operator+(float M)
{
   int i,j;
   matrix tmp(n,m,false);
   for(i=0;i<n;i++)
     for(j=0;j<m;j++)
         tmp.a[i][j]=a[i][j]+M;
return tmp;   //Возвращаем результат операциq - матрицу tmp.
}
/*       Перегрузка операции вычитания матриц.       ***/
matrix matrix::operator-(matrix M)
{
   int i,j;
      /*   Временная матрица tmp для хранения результата
           вычитания двух матриц.          ***/
   matrix tmp(n,m,false);
    /*    Если обе матрицы одинакового размера, то формируем
          матрицу tmp, как разность матриц.      ***/
   if ((n==M.n)&&(m==M.m))
     {
      for(i=0;i<n;i++)
        for(j=0;j<m;j++)
        tmp.a[i][j]=a[i][j]-M.a[i][j];
     }
   else
    /*    Если размеры матриц не совпадают, то fl=false
    (результирующую матрицу сформировать не удалось).   ***/
   tmp.fl=false;
//Возвращаем матрицу tmp, как результат операции.
return tmp;   //Возвращаем результат операциq - матрицу tmp.
}
/***        Перегрузка операции - для выполнения операции
    вычитания из матрицы вещественного числа.       ***/
matrix matrix::operator-(float M)
{
   int i,j;
   matrix tmp(n,m,false);
   for(i=0;i<n;i++)
     for(j=0;j<m;j++)
       tmp.a[i][j]=a[i][j]-M;
return tmp;   //Возвращаем результат операциq - матрицу tmp.
}
/***     Перегрузка операции умножения матриц        ***/
matrix matrix::operator*(matrix M)
{
   int i,j,k;
```

```
    /*   Временная матрица tmp для хранения результата
         умножения двух матриц.                */
   matrix tmp(n,M.m,false);
    /*    Если количество столбцов в первой матрицы совпадает
      с количеством строк во второй матрицы, то     */
   if ((m==M.n))
     {          //  выполняем умножение матриц
    for(i=0;i<n;i++)
      for(j=0;j<M.m;j++)
        for(k=0,tmp.a[i][j]=0;k<m;k++)
          tmp.a[i][j]+=a[i][k]*M.a[k][j];
     }
    /*    Если количество столбцов в первой матрицы не
совпадает  с количеством строк во второй матрице, то fl=false
(результирующую матрицу сформировать не удалось).   */
   else
     tmp.fl=false;
return tmp;   //Возвращаем результат операциq - матрицу tmp.
}
/***         Перегрузка операции * для выполнения операции
     умножения матрицы на вещественное число.       ***/
matrix matrix::operator*(float M)
{
   int i,j;
   matrix tmp(n,m,false);
   for(i=0;i<n;i++)
     for(j=0;j<m;j++)
        tmp.a[i][j]=a[i][j]*M;
return tmp;   //Возвращаем результат операции - матрицу tmp.
}
/***               Деструктор                  ***/
matrix::~matrix()
{
}
/***             Метод вывода матрицы.          ***/
void matrix::show_matrix()
{
   int i,j;
    /*  Если матрица сформирована, то выводим ее на экран.
***/
   if (fl)
     {
      cout<<"Matrix"<<endl;
      for(i=0;i<n;cout<<endl,i++)
         for(j=0;j<m;j++)
         cout<<a[i][j]<<"\t";
     }
    /*    Если матрицу сформировать не удалось, то выводим
          сообщение об этом на экран.      */
   else
     cout<<"No Matrix"<<endl;
}
```

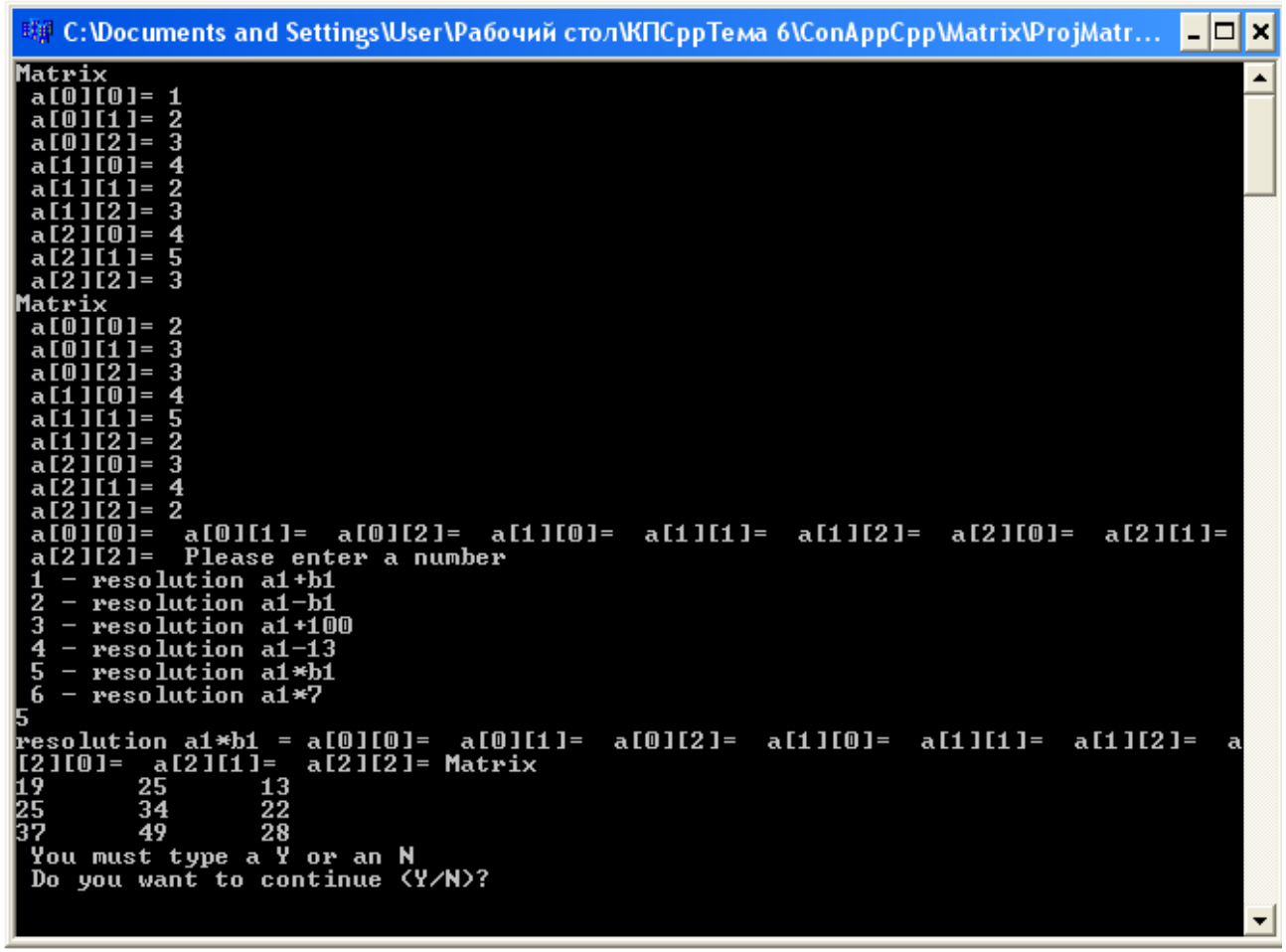



Рис. 6.32. Перегрузки операций. Операции над массивами.

## 6 . 7. Наследование классов

При выполнения работы №3 и №4, мы слегка коснулись наследования, дали основные фундаментальные определения, а также привели некоторые примеры, описывающие, наследование. Можем авторитетно заявить: «Одной из основных особенностей ООП является возможность наследования». Наследование – это один из способов повторного использования программного обеспечения, при котором новые производные классы (наследники) создаются на базе уже существующих базовых классов (родителей) и от них получают некоторые характерные черты, определяющие непосредственно через члены-данные и члены-функции (методами). При этом созданный новый класс является наследником членов и методов ранее определенного базового класса. Создаваемый путем наследования класс является производным (derived class), который в свою очередь может выступать в качестве базового класса (based class) для создаваемых классов (см. рис 12. Работа №4). Если имена методов

производного и базового классов совпадают, то методы производного класса перегружают методы базового класса.

**Процесс реализации наследования (*inheritance*).** Вернемся к ранее рассмотренным нами примерам в частности к задаче 6.6.

Изменим эту задачу в лучшую сторону, используя аксиоматику принципа наследования в ООП. Покажем, каким образом, если мы имеем некоторые готовые классы (родительские), как их можно использовать для получения новых классов (потомков).

**Задача 6.12.**

```
// Пример для демонстрации наследования классов
#include <vcl.h>
#pragma hdrstop
#include <iostream>
#include <math.h>
#include <conio.h>
#define PI 3.14159
#pragma argsused
  using namespace std;
   //Базовый класс Shape
class Shape
 {
  /***  Члены - данные они все защищенные  ***/
 protected:
     float x,y,z; // Координаты
 public:
  /***  Конструктор, родительского класса  ***/
    Shape ( float x_coord, float y_coord, float z_coord)
     {
      x=x_coord;
      y=y_coord;
      z=z_coord;
     }
  /***  Деструктор родительского класса  ***/
    ~ Shape () {}       //
  void SetX (float newX);
  void SetY (float newY);
  void SetZ (float newZ);
  float GetX();
  float GetY ();
  float GetZ ();
};
//Класс потомок My_Figure
class My_Figure:public Shape
 {
  /***  Члены - данные и члены функции они все закрытые ***/
 private:
  float r;     // радиус
   float cube () { return (r*r*r);}     // число в кубе
   float square(){ return (r*r);}       // число в квадрате
```

```
   /*** Члены - функции они открытые, т.е. общедоступные ***/
 public:
  /***  Конструктор, которого создает пользователь.  ***/
My_Figure( float x_coord, float y_coord, float z_coord, float
radius): Shape(x_coord, y_coord, z_coord)
 {
      x=x_coord;
      y=y_coord;
      z=z_coord;
      r=radius;
      cout << "My_Figure(" << x << ", " << y <<", "
           << z <<", " << r<<") created" << endl;
  }
  ~My_Figure () {}       // Деструктор созданный пользователем
  float volume_sphere ();      // Объем шара
  float  surface_sphere();     // Поверхность сферы
  float volume_tetraedr();     // Объем тетраэдра
  float surface_tetraedr();    // Поверхность тетраэдра
  float volume_cube();         // Поверхность куба
  float surface_cube();        // Поверхность куба
  void SetRadius (float newRadius);
  float GetRadius ();
};
int main(int argc, char* argv[])
{
    /*** Объявляем переменную типа класса My_Figure. Выполняем
         процесс инициализации объекта s***/
    My_Figure s(1.0, 2.0, 3.0, 4.0);
       cout<< " X="<<s.GetX()           // Обащение к
переменным
           << " Y="<<s.GetY()           // типа класс My_Figure
           << " Z="<<s.GetZ()           // через s
           << " R="<<s.GetRadius()      // при закрытых членах
           <<"\n";
     /*** Вывод объема и поверхности сферы  ***/
     cout<< "The volume_sphere is "<<s.volume_sphere()<<endl;
     cout<< "The surface_sphere is "<<s.surface_sphere()
<<endl;
      /*** Вывод объема и поверхности тетраэдра  ***/
     cout<< "The volume_tetraedr is
"<<s.volume_tetraedr()<<endl;
     cout<< "The surface_tetraedr is
"<<s.surface_tetraedr()<<endl;
      /*** Вывод объема и поверхности куба  ***/
     cout << "The volume_cube is "<<s.volume_cube() <<endl;
     cout << "The surface_cube is "<<s.surface_cube() <<endl;
     /*** Изменяем значения в закрытых членах ***/
        s.SetX(4.2);
        s.SetY(5.3);
        s.SetZ(6.7);
        s.SetRadius(6.9);
  /***Выводим на экран измененные значения в закрытых членах
***/
```

```
     cout   << " X="<<s.GetX()        // Обращение  к
переменным
            << " Y="<<s.GetY()          // типа класс My_Figure
            << " Z="<<s.GetZ()          // через s
            << " R="<<s.GetRadius()     // при закрытых членах
            <<"\n";
     /*** Вывод объема и поверхности сферы  ***/
     cout << "The volume_sphere is "<<s.volume_sphere()<<endl;
     cout << "The surface_sphere is
"<<s.surface_sphere()<<endl;
      /*** Вывод объема и поверхности тетраэдра  ***/
     cout << "The volume_tetraedr is
"<<s.volume_tetraedr()<<endl;
     cout <<"The surface_tetraedr is
"<<s.surface_tetraedr()<<endl;
      /*** Вывод объема и поверхности куба  ***/
     cout << "The volume_cube is "<<s.volume_cube() <<"\n";
     cout << "The surface_cube is "<<s.surface_cube() <<"\n";
     cout <<"Press any key ...";
       getch();
    return 0;
}
/***      class Shape      ***/
  void Shape::SetX (float newX)
    {
       x=newX;
    }
  void Shape::SetY (float newY)
    {
       y=newY;
    }
  void Shape::SetZ (float newZ)
    {
       z=newZ;
    }
  float Shape::GetX()
    {
      return(x);
    }
  float Shape::GetY ()
    {
       return(y);
    }
  float Shape::GetZ ()
    {
       return(z);
    }
/***       class My_Figure          ***/
  float My_Figure::volume_sphere ()      // Объем шара
  {
    return (cube()*4*PI/3);
  }
  float  My_Figure::surface_sphere()   // Поверхность сферы
```

```
  {
    return (4*square()*PI);
  }
  float My_Figure::volume_tetraedr()      // Объем тетраэдра
  {
    return (cube()*sqrt(2)/12);
  }
  float My_Figure::surface_tetraedr()     // Поверхность
тетраэдра
  {
    return (sqrt(3)*square());
  }
  float My_Figure::volume_cube()          // Поверхность куба
  {
    return (cube());
  }
  float My_Figure::surface_cube()          // Поверхность куба
  {
    return (6*square());
  }
  void My_Figure::SetRadius (float newRadius)
    {
       r=newRadius;
    }
  float My_Figure::GetRadius ()
    {
       return(r);
    }
```

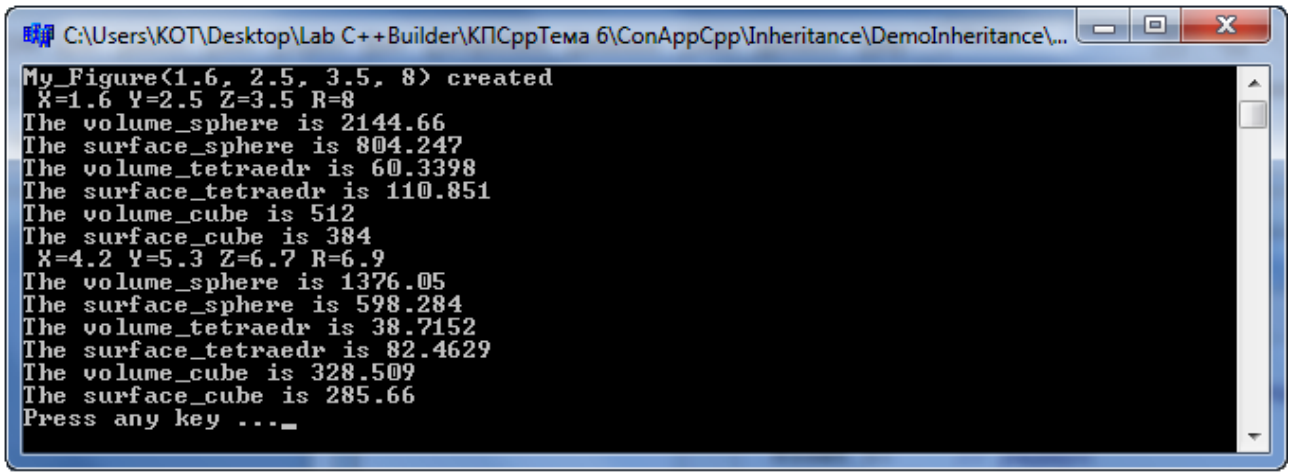



Рис.6.33. Результаты программного кода исследования демонстрация классов наследования.

При использовании наследования члены и методы кроме свойств public и private могут иметь свойство protected. Для одиночного класса описатели protected и private равносильны. Разница между protected и private проявляется при наследовании, закрытые члены и методы, объявленные в базовом классе, как protected, в производном могут

использоваться, как открытые (public). Защищенные(protected) члены и методы являются чем-то промежуточным между public и private.

При создании производного класса используется следующий синтаксис.

```
class name_derived_class:
type_inheritance base_class
{ // закрытые члены и методы класса
   ...
  public: // открыте члены и методы класса
   ...
  protected: // защищенные члены и методы класса для потомков
   ...
};
```

Здесь

- name_derived_class – имя создаваемого производного класса,
- type_inheritance – способ наследования, возможны следующие способы наследования public, private и protected.
- base_class – имя базового типа.

Следует различать тип доступа к элементам в базовом классе и тип наследования.

Типы наследования и типа доступа

| Способ доступа | Спецификатор в базовом классе | Доступ в производном классе |
|---|---|---|
| Private | private<br>protected<br>public | нет<br>private<br>private |
| Protected | private<br>protected<br>public | нет<br>protected<br>protected |
| Public | private<br>protected<br>public | нет<br>protected<br>public |

При порождении производного класса из базового, имеющего конструктор, конструктор базового типа необходимо вызывать из конструктора производного класса. Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Если в конструкторе производного класса явный вызов конструктора базового типа отсутствует, то он вызывается автоматически без параметров. В случае нескольких уровней наследования конструкторы вызываются, начиная с верхнего уровня. В случае нескольких базовых типов их конструкторы вызываются в порядке объявления.

Если в описании класса есть ссылка на описываемый позже класс, то его надо просто объявить с помощью оператора class new_class;

Это описание аналогично описанию прототипов функции.

**Полиморфизм.** Теперь рассмотрим последнюю парадигму ООП *полиморфизм*. Слово ***полиморфизм*** это сложное слово сотоящее из трех слов «поли» - много, «морф» - превод или преобразование одной вещи или предмета в другой, а «изм» - процесс. Данная парадигма чем - то схоже с перегрузкой, однако здесь речь идет о классах.

Полиморфизм дает возможность общаться с объектами различного типа так, как если бы они были объектами одного тапа. Для реализации полиморфизма зачастую используют виртуальные функции – члены.

Виртуальные функции – члены допускают при наследовании методы, которые в различных производных классах работают по различным алгоритмам, но имеют одинаковые выходные параметры и возвращаемое значение. Такие методы называются виртуальными и описываются с помощью служебного слова ***virtual***.

**Задача 6.13**. Рассмотрим абстрактный базовый класс figure (фигура), на базе которого можно построить производные классы для реальных фигур (эллипс, окружность, квадрат, ромб, прямоугольник, треугольник и т.д.). На листинге приведен базовый класс figure и производные классы _circle (окружность) и RecTangle (прямоугольник).

```
//-------------------------------------------------------------
#include "stdafx.h"
#include <iostream>
#include <math.h>
#define PI 3.14159
using namespace std;
class figure                //Базовый класс figure.
{
public:
  /*** n – количество сторон фигуры, для окружности n=1.  ***/
   int n;
   /*** p – массив длин сторон фигуры, для окружности в
        p хранится радиус.                ***/
   float *p;
   figure();                //Конструктор.
   float perimetr();        //Метод вычисления периметра
фигуры.
   virtual float square(); //Метод вычисления площади фигуры.
   /***  Метод вывода информации о фигуре: ее название,
         периметр, площадь и т.д.              ***/
   virtual void show_parametri();
};
//-------------------------------------------------------//
/***              Функции класса figure          ***/
   figure::figure()         //Конструктор класса фигуры.
    {
      cout<<"This is abstract constructor"<<endl;
```

```
    }
   /*** Метод вычисления периметра, он будет перегружаться
        только в классе _circle.              ***/
   float figure::perimetr()
    {
      int i;
      float psum;
      for(psum=0,i=0;i<n;psum+=p[i],i++);
     return psum;
    }
   /*** Метод вычисления площади, пока он абстрактный, в
каждом классе будет перегружаться реальным методом. ***/
   float figure::square()
    {
      cout<<"No square abstract figure"<<endl;
      return 0;
    }
   /*** Метод вывода информации о фигуре, которая будет
        перегружаться в каждом производном классе.    ***/
   void figure::show_parametri()
    {
      cout<<"Abstract figure";
    }
//-------------------------------------------------------------
/***    Производный класс _circle (окружность), основанный
        на классе figure.              ***/
class _circle:public figure
{
public:
   _circle();               // Конструтор
   /*** Перегружаемые методы класса _circle()
        perimetr(),square(),show_parametri().        ***/
   float perimetr();
   virtual float square();
   virtual void show_parametri();
};
//-------------------------------------------------------------
/*** Производный класс RecTangle (прямоугольник),
     основанный на классе figure.          ***/
class RecTangle:public figure
{
public:
   RecTangle();           //Конструктор.
   /***  Перегружаемые методы square(),show_parametri().  ***/
   virtual float square();
   virtual void show_parametri();};
//-------------------------------------------------------------//
/***               ГЛАВНАЯ ФУНКЦИЯ             ***/
void main()
{
   _circle RR;
    RR.show_parametri();
    RecTangle PP;
```

```
    PP.show_parametri();
}
//-------------------------------------------------------------
/***               Методы класса _circle          ***/
   _circle::_circle()      //Конструктор класса.
     {
        cout<<"Parametri okruzhnosti"<<endl;
   /*** В качестве сторон окружности выступает единственный
        параметр радиус.                    ***/
      n=1;
      p=new float[n];
      cout<<"Vvedite radius";
      cin>>p[0];
     }
                    //Метод вычисления периметра окружности.
   float _circle::perimetr()
     {
      return 2*PI*p[0];
     }
                    //Метод вычисления площади окружности.
   float _circle::square()
     {
      return PI*p[0]*p[0];
     }
                    //Метод вывода параметров окружности.
   void _circle::show_parametri()
     {
             //Вывод сообщения о том, что это окружность.
      cout<<"This is a circle"<<endl;
             //Вывод радиуса окружности.
      cout<<"Radius="<<p[0]<<endl;
             //Вывод периметра окружности.
      cout<<"Perimetr="<<perimetr()<<endl;
             //Вывод площади окружности.
      cout<<"Square="<<square()<<endl;
     }
//-------------------------------------------------------------
/***              Методы класса RecTangle              ***/
   RecTangle::RecTangle()  //Конструктор класса RecTangle.
    {
     cout<<"Parametri rectangle"<<endl;
     n=4;                   //Количество сторон =4.
     p=new float[n];
            //Ввод длин сторон прямоугольника.
     cout<<"Vvedite dlini storon";
     cin>>p[0]>>p[1];
     p[2]=p[0];
     p[3]=p[1];
    }
          //Метод вычисления площади прямоугольника.
   float RecTangle::square()
     {
      return p[0]*p[1];
```

```
    }
        //Метод вывода параметров прямоугольника.
  void RecTangle::show_parametri()
    {
          //Вывод сообщения о том, что это прямоугольник.
     cout<<"This is a Rectangle"<<endl;
          //Вывод длин сторон прямоугольника.
     cout<<"a="<<p[0]<<" b="<<p[1]<<endl;
         /***   Вывод   периметра   прямоугольника.   Классе
RecTangle
          вызывает метод perimetr() базового класса
(figure). */
     cout<<"Perimetr="<<perimetr()<<endl;
         //Вывод площади прямоугольника.
     cout<<"Square="<<square()<<endl;
    }
```

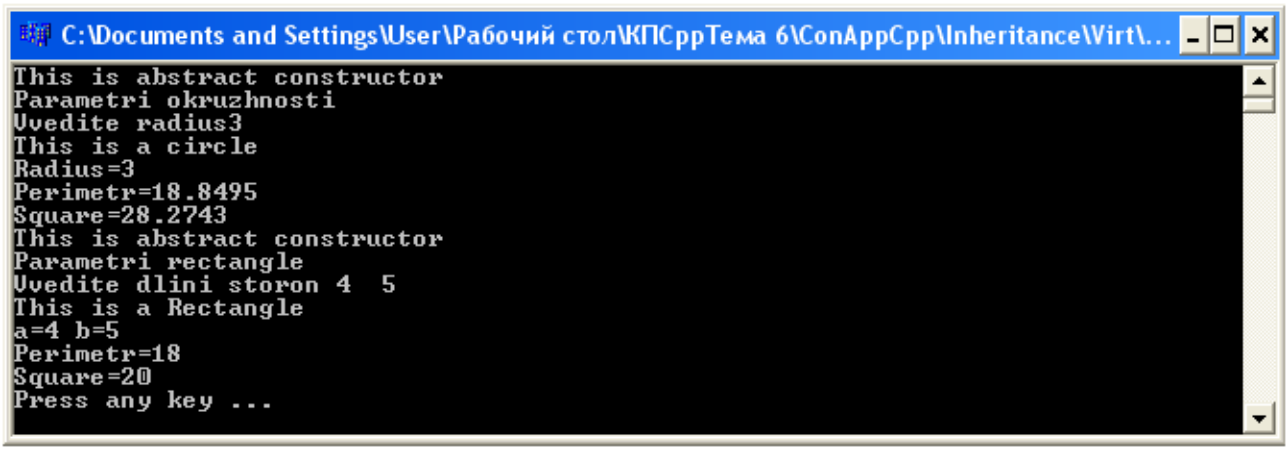



Рис. 6.34. Результаты работы программы.

## 6.8. Индивидуальные задание

Индивидуальные задание взять из работы №5 и выполнить на языке С++ в среде  С ++ Builder

## 6.9.  Контрольные вопросы

1. Понятия о классах.
2. Понятия открытые методы и члены – данных класса.
3. Понятия  о конструкторах  и деструкторах.
4. Открытые и закрытые доступы (видимость) к элементов класса.
5. Выделение памяти для массива в конструкторе.
6. Понятия о конструкторе копирования (перегрузка конструктора).
7. Понятия  о перегрузка операторов и функций.
8. Наследование классов.
9. Полиморфизм.

## Лабораторная работа №7.
## Массивы и консольные приложения

***Цель работы:*** изучить составной тип данных – массив, основные свойства компоненты *StringGrid*. Написать и отладить программу с использованием одномерных массивов, и понятие «указатель», правила создания и приемы обработки динамических массивов на примере двухмерного массива.

### 7.1. Обработка нескольких окон. Диаграммы взаимодействия (interaction diagram)

Для описания взаимодействия объектов в UML предусмотрены следующие виды диаграмм:

- Диаграмма последовательности – моделирует последовательность обмена сообщениями между объектами;
- Диаграмма коммуникаций – модулирует структуру взаимодействующих компонентов (для данного вида диаграммы в UML 1 используется наименование «диаграмма кооперацийй»);
- Временные диаграммы – моделирует изменение состояния нескольких объектов в момент взаимодействия;
- Диаграмма обзора взаимодействия – сочетание диаграммы деятельности и диаграммы последовательности.

**Диаграммы коммуникации и последовательности.** *Диаграммы коммуникации и последовательности* транзитивны, выражают взаимодействие, но показывают его различными способами и с достаточной степенью точности могут быть преобразованы одна в другую.

*Диаграмма коммуникации (Communication diagram, в UML 1.x — диаграмма кооперации или сотрудничества (collaboration diagram***)** — диаграмма, на которой изображаются взаимодействия между частями композитной структуры или ролями кооперации. В отличие от диаграммы последовательности, на диаграмме коммуникации явно указываются отношения между элементами (объектами), а время как отдельное измерение не используется (применяются порядковые номера вызовов).

**Диаграмма коммуникации** — способ описания поведения, семантически эквивалентный диаграмме последовательности. Фактически, это такое же описание последовательности обмена

сообщениями взаимодействующих экземпляров классификаторов, только выраженное другими графическими средствами.

Вместо того чтобы рисовать каждого участника в виде линии жизни и показывать последовательность сообщений, располагая их по вертикали, как это делается в диаграммах последовательности, коммуникационные диаграммы допускают произвольное размещение участников, позволяя рисовать связи, показывающие отношения участников, и использовать нумерацию для представления последовательности сообщений см. рис 7.1.

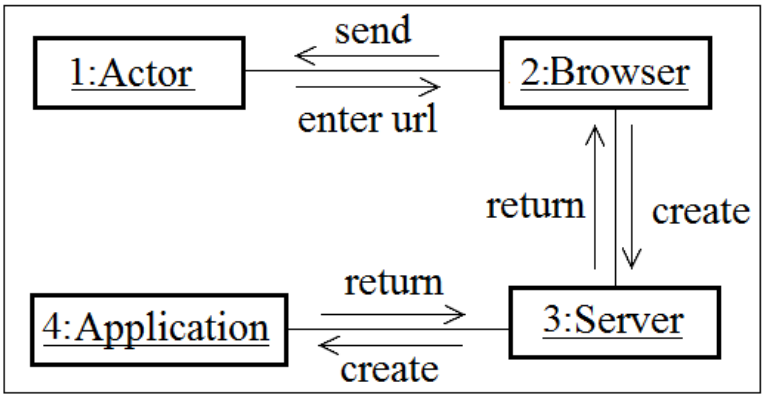



Рис. 7.1. Прием и передача сообщений.

Таким образом, на диаграмме коммуникации также, как и на диаграмме последовательности применяют один основной тип сущностей — экземпляры взаимодействующих классификаторов и один тип отношений — связи. Однако здесь акцент делается не на времени, а на структуре связей между конкретными экземплярами.

Диаграмма коммуникаций не имеет точно определенных нотаций для управляющей логики. Она допускает применение маркеров итерации и защиты, но не позволяет полностью определить алгоритм управления. Не существует также специальных обозначений для создания и удаления объектов, но ключевые слова «create» и «delete» соответствуют общепринятым соглашениям.

В UML 1.x эти диаграммы назывались *Диаграммами кооперации или сотрудничества.* Прежде всего, они отображает структуру взаимодействия и содержит следующие элементы:

- экземпляры действующих лиц и классов, участвующих в реализации варианта использования;
- ассоциации между экземплярами действующих лиц и классов;
- сообщения, передаваемые между экземплярами действующих лиц и классов.

Следовательно, *Диаграмма кооперации или сотрудничества* — это тип диаграмм, которая позволяет описать взаимодействия объектов, абстрагируясь от последовательности передачи сообщений. На этом типе диаграмм в компактном виде отражаются все принимаемые и передаваемые сообщения конкретного объекта, и типы этих сообщений.

На диаграмме эти элементы отображаются стандартно (экземпляр действующего лица – человечком, экземпляр класса (объект) – прямоугольником или графическим стереотипом класса анализа см. тему 3). В то же время следует помнить, что экземпляр – это конкретная реализация соответствующей сущности (действующего лица, класса, узла и т. д.). Чтобы учесть этот нюанс на диаграммах, имя экземпляра подчеркивается и может отображаться в следующих вариантах:

- Имя объекта: через имя класса (например, :Хинкал) или :Actor;
- Имя класса (например, Хинкал) – анонимный объект, Хинкал может быть, указывая атрибуты, :Аварский Хинкал, :Кумыкский Хинкал и т.д., и в единицах порции;
- Имя объекта (например, :Красная Площадь) – предполагается, что имя класса известно (Площадь);
- Имя объекта (например, Александр:) объект - спортсмен. Имя класса неизвестно в данном случае.

Взаимодействие между экземплярами действующих лиц и объектами моделируется посредством сообщений, отображаемых над ассоциациями. *Сообщение* (англ. message) – это спецификация факта передачи информации между сущностями с ожиданием выполнения определенных действий со стороны принимающей сущности. Сущность, отправляющую сообщение, называют ***клиентом,*** а принимающую – *сервером*. Таким образом, сообщения не только передают некоторую информацию, но и требуют или предполагают выполнения сервером определенных действий или передачу (возврат) клиенту необходимой информации. Если принимающей сообщение сущностью является объект, то оно представляет собой операцию (метод) объекта-сервера. Прием сообщения обычно трактуется, как возникновение события на сервере см. рис.7.1**.**

По причине того, что диаграммы Sequence и Collaboration являются разными взглядами на одни и те же процессы, Rational Rose позволяет создавать из Sequence диаграммы диаграмму Collaboration и наоборот, а также производит автоматическую синхронизацию этих диаграмм.


Рис. 7.2. Пример диаграммы сотрудничества уровня спецификаций

Диаграммы кооперации или сотрудничества уровня спецификаций оперируют классами, пользователями, кооперациями и ролями, которые играют пользователи и классы. Пример диаграммы сотрудничества уровня спецификаций приведен на рис. 7.2. Окружность – это кооперация, пунктирная линия – роль пользователя в кооперации, стрелка– отношение обобщения, общее для всех UML-диаграмм. Кооперация определяет взаимодействие классов. Участвуя в кооперации, классы совместно производят некоторый кооперативный результат. Диаграммы сотрудничества уровня примеров оперируют экземплярами классов (объектами), связями между ними и сообщениями, которыми обмениваются объекты более подробно выше.

Объекты на диаграммах сотрудничества обозначаются так же, как и на других UML-диаграммах – прямоугольниками. Однако на диаграммах

данного вида имя объекта может дополняться его ролью в сотрудничестве. На рис. 7.3, а показан образец, на рис. 7.3, б – пример обозначения имени объекта, любая из трех частей которого может отсутствовать.


Рис. 7.3. Графическое изображение объектов на диаграммах сотрудничества.

Объекты, которые могут управлять другими объектами, называются активными (active object) и помечаются словом {active}.

Для обозначения группы объектов, которым адресован один и тот же сигнал, вводится понятие мультиобъекта (рис. 7.4.).

Между объектами диаграммы сотрудничества существуют связи (links см. работу №3), по которым объекты посылают друг другу сообщения. Связи не имеют названий (в терминах UML они являются анонимными), но могут быть специфицированы ключевыми словами (стереотипами см. тему 3) они изображены в комментариях на рис 7.2.

Приведенные стереотипы требуют пояснения. Связь между объектами в информационной системе на уровне программирования на определенном языке осуществляется посредством передачи параметров (переменных) от одного объекта другому объекту. Например, объект «Отдел продаж» передает объекту «Склад» некоторый принятый в организации документ (переменную), в котором сообщает о необходимости выделения продукции той или иной номенклатуры. Значение передаваемых параметров является содержанием передаваемого посредством связи сообщения.

Сообщения на диаграммах сотрудничества изображаются стрелками вдоль связей. Порядок передачи сообщений может быть определен явным указанием номера сообщения возле стрелки. Вид сообщения несет дополнительную смысловую нагрузку в виде определения ролей взаимодействующих объектов.

Сообщение записывается в определенном формате. Например, показанная ниже запись означает, что данное сообщение будет передано только после сообщений с номерами 1 и 2 (предшествующие сообщения), при условии истинности введенного пароля (сторожевое условие). В потоке последовательных сообщений оно будет занимать место между сообщениями 3.1 и 3.3, при этом возможна параллельная передача сообщения с другими сообщениями, имеющими номер 3.2. Само сообщение вызывает метод нахождения сведений о человеке по фамилии,

имени и отчеству (список аргументов), предполагая предоставление карточки по форме 1А на запрашиваемое лицо (возвращаемое значение): 1,2/[пароль] 3.2 Форма_1А:= найти_сведения (Фамилия, Имя, Отчество)

Создавая диаграмму сотрудничества, наиболее важные объекты, участвующие во взаимодействии, следует помещать в середину диаграммы. Это способствует более точному представлению отношений между взаимодействующими объектами.

При проектировании диаграмм сотрудничества с нуля (в отличие от генерирования их автоматически из диаграмм последовательности), следует придерживаться такого пошагового порядка:

- Определение области диаграммы (область диаграммы сотрудничества в зависимости от задачи определяет пользователь);
- Размещение на диаграмме объектов, участвующих во взаимодействии (желательно наиболее важные объекты размещать ближе к центру диаграммы);
- Для наиболее важных свойств и состояний сотрудничества, установите начальные значения этих свойств или состояния;
- Создайте связи между объектами;
- Создайте сообщения, соответствующие каждой связи;
- Добавьте порядковые номера к каждому сообщению в соответствии с их временным порядком во взаимодействии.

*Когда применяются коммуникационные диаграммы (*приводится цитата из книги М. Фаулер. UML Основы*).* Основной вопрос, связанный с коммуникационными диаграммами, заключается в том, в каких случаях надо предпочесть их, а не более общие диаграммы последовательности. Ведущую роль в принятии такого решения играют личные предпочтения: у людей разные вкусы. Чаще всего именно это определяет тот или иной выбор…

Более рациональный подход утверждает, что диаграммы последовательности удобнее, если вы хотите подчеркнуть последовательность вызовов, а коммуникационные диаграммы лучше выбрать, когда надо акцентировать внимание на связях. Многие специалисты считают, что коммуникационные диаграммы проще модифицировать на доске, используя CRC-карточки.

**Диаграмма последовательности (Sequence diagram)** — диаграмма, на которой показаны взаимодействия объектов, упорядоченные по времени их проявления, на которой изображено упорядоченное во времени взаимодействие объектов. В частности, на ней изображаются участвующие во взаимодействии объекты и последовательность

сообщений, которыми они обмениваются, другими словами это способ описания поведения системы "на примерах".

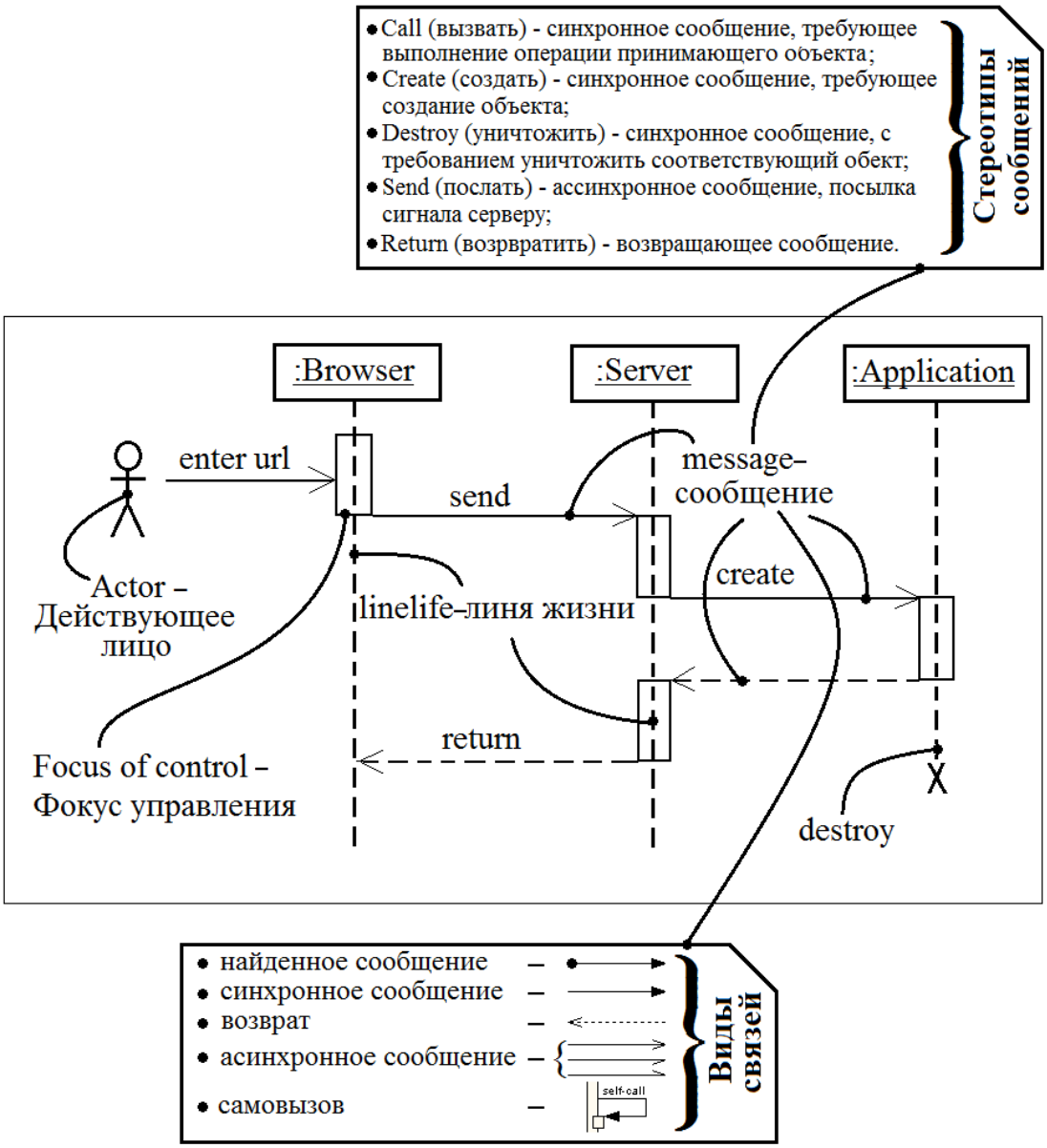



Рис .7.4. UML диаграмма последовательности функционирования операции отправки сообщения для уничтожения приложений.

Фактически, диаграмма последовательности — это запись протокола конкретного сеанса работы системы (или фрагмента такого протокола). В ООП важное значение имеет при работе процесс выполнения пересылки сообщений между взаимодействующими объектами, которая является одним из основополагающих концепций. Именно последовательность посылок сообщений отображается на данной диаграмме, отсюда и название см. рис. 7.4, где приведен фрагмент протокола выполненной используя способ диаграммы последовательности и виды сообщений

На диаграмме последовательности применяют один основной тип сущностей — экземпляры взаимодействующих классификаторов (в основном классов, компонентов и действующих лиц), и один из типов отношений — связи, по которым происходит обмен сообщениями, напоминаем, о связях мы говорили при при выполнение работя №3, некоторые наиболее часто используемые приводим на рис. 7.4. в виде комментарий.

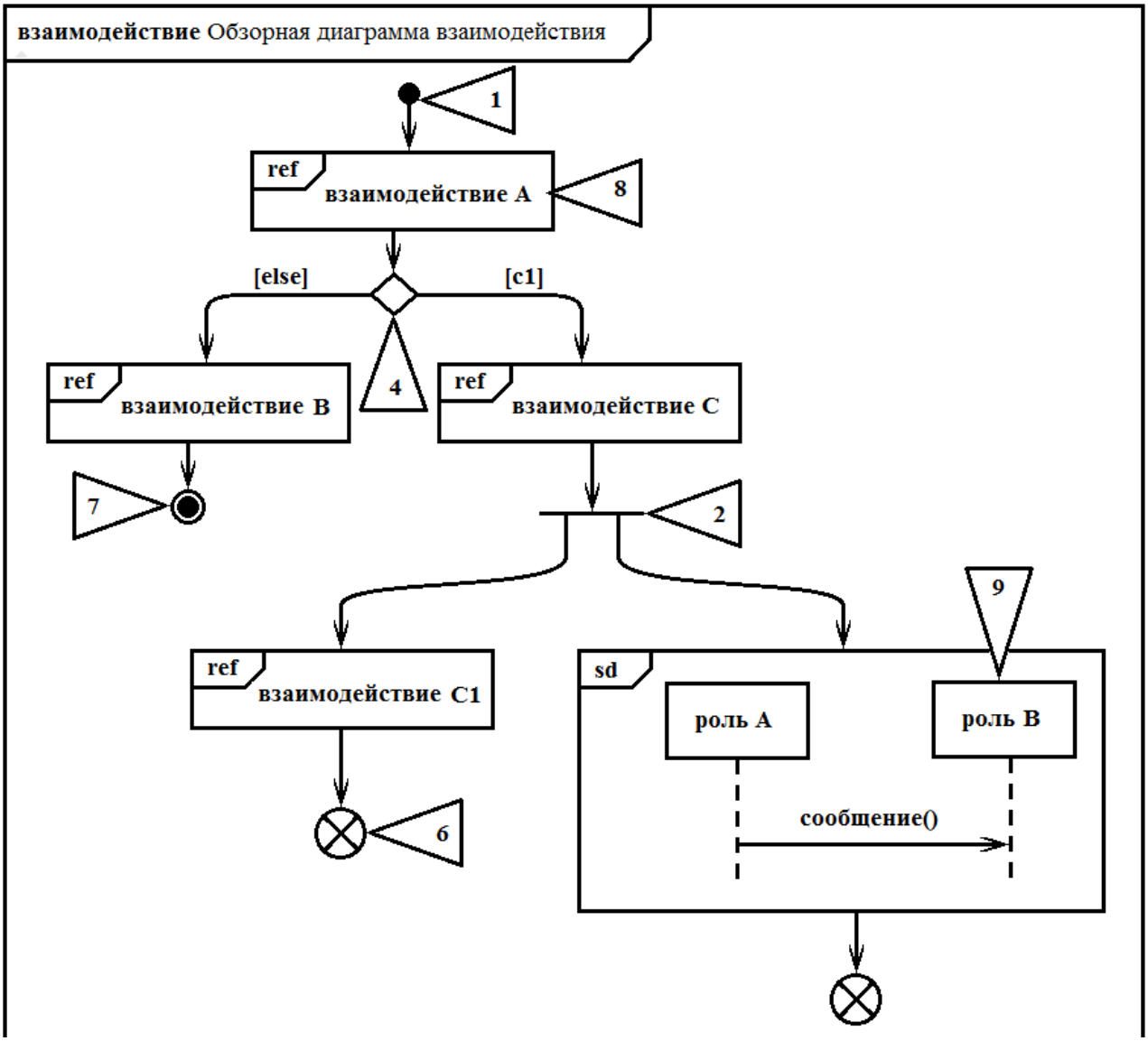



Рис 7.5. Диаграмма обзора взаимодействия.

**Диаграмма обзора взаимодействия.** *Диаграмма обзора взаимодействия (Interaction overview diagram)* — разновидность диаграммы деятельности, включающая фрагменты диаграммы последовательности действий (Sequence diagram) и конструкции потока

управления, так называемые диаграммы сотрудничества (Collaboration diagram), которые позволяют с разных точек зрения рассмотреть взаимодействие объектов в создаваемой системе. Обычно обзорная диаграмма взаимодействия используется как дополнение к диаграмме деятельности, для того, чтобы связать последовательность выполняемых действий с взаимодействиями, описываемыми диаграммами последовательности.

Обзорные диаграммы взаимодействия декларируются в спецификации UML как вариант диаграмм активностей, и определяет их семантику декларативно. Для процесса сравнения поведений желательно определить их семантику деятельности - как вариант семантики диаграмм активности. Семантика же диаграмм активностей определяется через поток маркеров (tokens) по графу, состоящий из соединяющих дуг и переходов. Для определения поведения каждому объекту сопоставляется набор маркеров. Каждому маркеру ставится в соответствие экземпляр некоторой роли, выполняемой объектом, где маркера соответствуют экземплярам роли и отображают одни и те же понятия. Нахождение маркера для экземпляра роли в некоторой активности, означает, что объект, которому соответствует этот экземпляр роли, участвует во взаимодействии, описываемом сопоставленной этой активности диаграммой взаимодействия. Состояние объекта определяется набором ролей, которые он исполняет, т.е. соответствующим множеством экземпляров ролей и их состояниями. Состояние всей системы определяется текущей разметкой диаграмм маркерами.

Диаграммы обзора взаимодействия, вместо узлов действий и объектов диаграмм деятельности, имеют фреймы, каждый из которых может соответствовать взаимодействию или использованию взаимодействия. Альтернативные комбинированные фрагменты представляются узлом решения и соответствующим узлом слияния.

Параллельные комбинированные фрагменты представляются узлом разделения и соответствующим узлом соединения. Комбинированные фрагменты типа Цикл представляются простыми циклами. Ветвление и слияния ветвлений на диаграммах обзора взаимодействия должны быть должным образом вложенными. Диаграммы обзора взаимодействия заключаются во фрейм, аналогично другим видам диаграмм взаимодействия с тегом sd и ref.

Так в приведенном примере на рис. 7.5. поток управления на диаграмме показывается стандартным для диаграммы деятельности способом – через многочисленные узлы управления, в которые входят:

начальный узел (1); развилка (2) и слияние (3); разветвление (4) и соединение (5); узел завершения потока (6) и заключительный узел (7). Узлы деятельности представляют собой либо ссылки на взаимодействия (8), либо вложенные диаграммы последовательности (9).

**Диаграмма синхронизации (временные диаграммы).** Диаграмма синхронизации (Timing diagram) (см. рис. 7.6.) используется как дополнение или альтернатива к диаграммам автомата и последовательности, позволяющая:

- связать изменение состояний элементов с сообщениями, которыми элементы обмениваются;
- привязать эти события к временной шкале;
- показать значения атрибутов и других свойств линий жизни.

Для объекта, который описывает диаграмма синхронизации, указывается имя (1 Name), тип (2 Type), а также возможные состояния (3 State). Изменение состояния показывается либо ломаной линией (4), либо (если число состояний достаточно большое) используют альтернативную нотацию (5 Lifeline).

Линии жизни на временной диаграмме изображаются в отдельных секциях, слева от которых указываются их имена, и упорядочены по вертикали. На диаграмме также могут быть показаны значения некоторого атрибута отдельной линии жизни как некоторая функция времени. Значение представляется в форме специального символа и указывается явно в форме текста. Если на одной диаграмме требуется показать несколько объектов, то каждый из них показывается в рамках своей линии жизни (6). Линия жизни может перемещаться по диаграмме вверх и вниз, что отражает изменение ее состояния или может быть изображена горизонтально с целью изображения на ней отдельных состояний или значений.

Взаимодействие между объектами показывается с помощью посылки/приема сообщений (7 massage), которые изображаются между линиями жизни, которые располагаются вертикально. Если сообщения пересекают несколько линий жизни, то можно воспользоваться метками продолжения (8), которые также и являются графическим сокращением, и еще используются в том случае, когда соединяемые сообщением линии жизни располагаются далеко друг от друга.

Временная шкала (9) позволяет привязать линии жизни к определенным промежуткам времени, задаваемым засечками (10), где текущее время увеличивается в направлении слева направо, и в некоторых

случаях — дискретные моменты изменений. Время для всех линий жизни синхронизовано и течет одинаково.

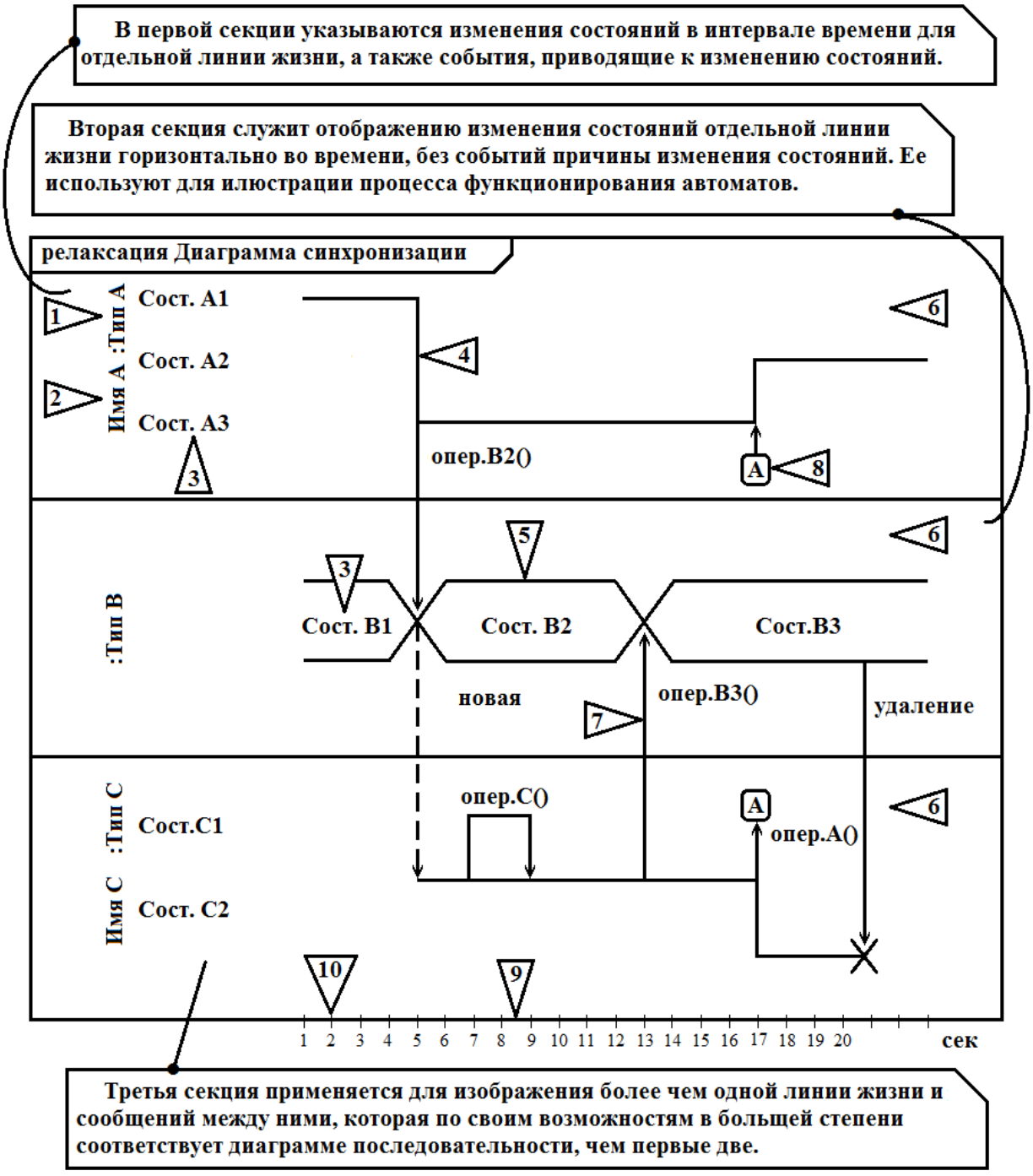



Рис. 7.6. Диаграмма синхронизации (временная диаграмма).

**Диаграмма профилей.** Диаграмма профилей рис.7.7 (profile) берет часть UML и расширяет его с помощью связанной группы стереотипов для определенной цели, например для бизнес-моделирования

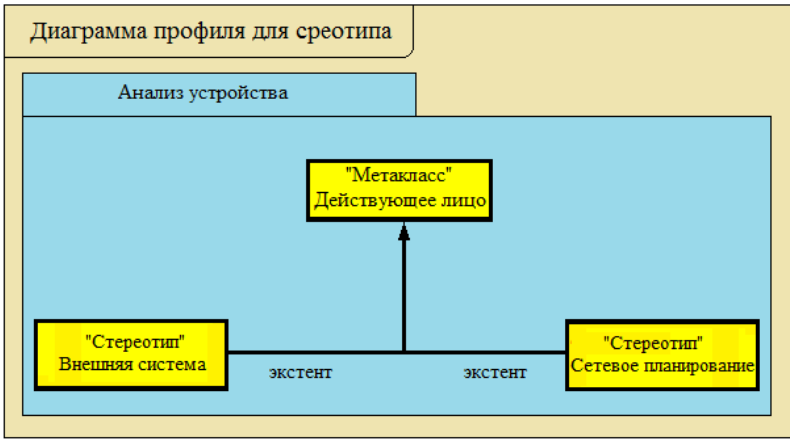



Рис. 7.7. Диаграмма профилей.

Напомним что массив – конечная последовательность данных одного типа. Массив – объект сложного типа, каждый элемент которого определяется именем (*ID*) и целочисленным значением индекса (номера), по которому к элементу массива производится доступ. Рассмотрим одномерные массивы. Начальные сведения о массивах рассмотренны в работе №3.

***Внимание*! *Индексы массивов в языке C/C++ начинаются с 0.***

В программе одномерный массив декларируется следующим образом:

```
тип ID массива [размер];
```

где *размер* – указывает количество элементов в массиве. Размер массива может задаваться константой или константным выражением. Для использования массивов переменного размера существует отдельный механизм – динамическое выделение памяти.

Примеры декларации массивов:

```
int a[5];
double b[4] = {1.5, 2.5, 3.75};
```

в целочисленном массиве ***a*** первый элемент ***a***[0], второй – ***a***[1], …, пятый – ***a***[4]. Для массива ***b***, состоящего из действительных чисел, выполнена инициализация, причем элементы массива получат следующие значения: ***b***[0]=1.5, ***b***[1]=2.5, ***b***[2]=3.75, ***b***[3]=0.

В языке C/C++ не проверяется выход индекса за пределы массива. Корректность использования индексов элементов массива должен контролировать программист.

Примеры описания массивов:

```
const Nmax=10;                  – задание максимального значения;
typedef double mas1[Nmax*2];    – описание типа одномерного массива;
```

```
mas1 a;                    – декларация массива a типа mas1;
int ss[10];                – массив из десяти целых чисел.
```

Элементы массивов могут использоваться в выражениях так же, как и обычные переменные, например:

```
f = 2*a[3] + a[Ss[i] + 1]*3;
a[n] = 1 + sqrt(fabs(a[n-1]));
```

**Создание оконного приложения. Компонента StringGrid.** При работе с массивами ввод и вывод значений обычно организуется с использованием компоненты *StringGrid*, предназначенной для отображения информации в виде двухмерной таблицы, каждая ячейка которой представляет собой окно однострочного редактора (аналогично окну *Edit*). Доступ к информации осуществляется с помощью элемента *Cells*[*ACol*][*ARow*] типа *AnsiString*, где целочисленные значения *ACol*, *ARow* указывают позицию элемента.

***Внимание! Первый индекс ACol определяет номер столбца, а второй ARow – номер строки*** в отличие от индексов массива.

В инспекторе объектов значения *ColCount* и *RowCount* устанавливают начальные значения количества столбцов и строк в таблице, а *FixedCols* и *FixedRows* задают количество столбцов и строк фиксированной зоны. Фиксированная зона выделена другим цветом и обычно используется для надписей.

**Пример выполнения задания на массивы .** Удалить из массива *A* размером *N*, состоящего из целых чисел (положительных и отрицательных), все отрицательные числа. Новый массив не создавать. Для заполнения массива использовать фунцию *random*(*kod*) – генератор случайных равномерно распределенных целых чисел от 0 до (*int*)*kod*.

**Пример создания оконного приложения.** Значение *N* вводить из *Edit*, значения массива *A* – из компоненты *StringGrid*. Результат вывести в компоненту *StringGrid*.

Панель диалога и результаты выполнения программы приведена на рис. 7.8.

**Настройка компоненты StringGrid.** На закладке *Additional* выберите пиктограмму  , установите компоненты *StringGrid*1 и *StringGrid*2 и отрегулируйте их размеры. В инспекторе объектов для обоих компонент установите значения *ColCount* равными 2, *RowCount* равными 1, т.е. по два столбца и одной строке, а значения *FixedCols* и *FixedRows* равными 0. Значение ширины клетки столбца *DefaultColWidth* равным 40.

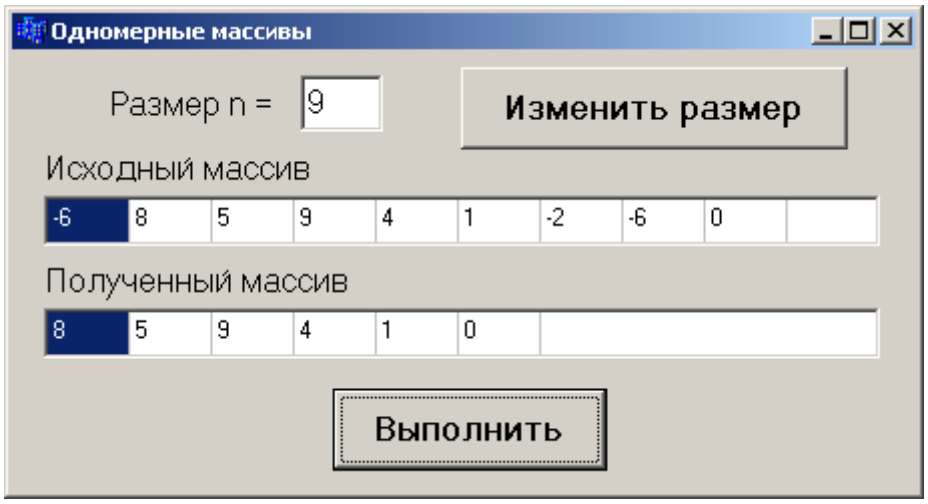



Рис. 7.8. Ввод одномерного массива.

По умолчанию в компоненту *StringGrid* ввод данных разрешен только программно. Для разрешения ввода данных с клавиатуры необходимо в свойстве ***Options*** строку ***goEditing*** для компоненты *StringGrid*1 установить в положение ***true***.

Текст функций-обработчиков может иметь следующий вид:

```
        ...
int n = 4;
//---------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
       randomize();   // Изменение начального адреса для
random()
  Edit1->Text=IntToStr(n);
  StringGrid1->ColCount=n;
  for(int i=0; i<n;i++) // Заполнение массива A случайными
числами
             StringGrid1->Cells[i][0] = IntToStr(random(21)-
10);
        Label3->Hide();    // Скрыть компоненту
        StringGrid2->Hide();
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        n=StrToInt(Edit1->Text);
        if(n>10){
                ShowMessage("Максимальное количество 10!");
                n=10;
                Edit1->Text = "10";
        }
        StringGrid1->ColCount=n;
        for(int i=0; i<n;i++)
                StringGrid1->Cells[i][0]=IntToStr(random(21)-
10);
        Label3->Hide();
```

```
        StringGrid2->Hide();
}
//---------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
       int i, kol = 0, a[10];   // Декларация одномерного
массива
//Заполнение массива A элементами из таблицы StringGrid1
        for(i=0; i<n;i++)
              a[i]=StrToInt(StringGrid1->Cells[i][0]);
//Удаление отрицательных элементов из массива A
        for(i=0; i<n;i++)
                if(a[i]>=0) a[kol++] = a[i];
        StringGrid2->ColCount = kol;
        StringGrid2->Show();        // Показать компоненту
        Label3->Show();
//Вывод результата в таблицу StringGrid2
        for(i=0; i<kol;i++) StringGrid2-
>Cells[i][0]=IntToStr(a[i]);
}
```

**Пример создания консольного приложения.** Текст программы может иметь следующий вид (обратите внимание на то, что функция *main* используется в простейшей форме – без параметров и не возвращает результатов):

```
      ...
#include <stdio.h>
#include <conio.h>
void main()
{
          int a[10],n, i, kol=0;
    randomize();    // Изменение начального адреса для random()
        printf("Input N (<=10) ");
        scanf("%d", &n);
        puts("\n Massiv A");
        for(i=0; i<n;i++) {
   a[i] = random(21)-10; //Заполнение массива A случайными
числами
              printf("%4d", a[i]);
        }
//Удаление отрицательных элементов из массива A
        for(i=0; i<n;i++)
               if(a[i]>=0)  a[kol++] = a[i];
        puts("\n Rezult massiv A");
        for(i=0; i<kol;i++) printf("%4d", a[i]);
        puts("\n Press any key ... ");
        getch();
}
```

С заполненным случайными числами массивом *A* результат программы может быть следующим:

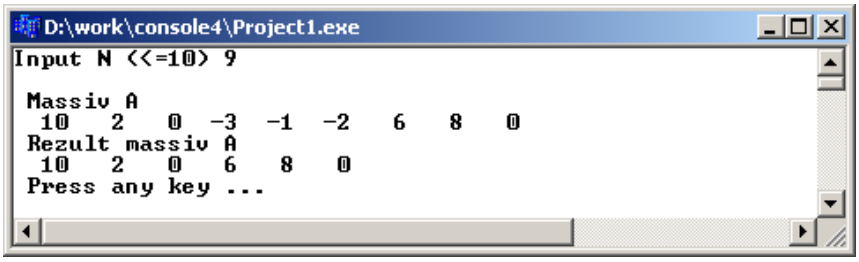



Рис.7. 9. Создание консольного приложения для массива *A*.

## 7. 2. Индивидуальные задания

Написать программу по обработке одномерных массивов. Размеры массивов вводить с клавиатуры. В консольном приложении предусмотреть возможность ввода данных как с клавиатуры, так и с использованием функции *random*(). При создании оконного приложения скалярный (простой) результат выводить в виде компоненты *Label*, а массивы вводить и выводить с помощью компонент *StringGrid*.

В одномерном массиве, состоящем из *n* вводимых с клавиатуры целых элементов, вычислить:

1. Произведение элементов массива, расположенных между максимальным и минимальным элементами;
2. Сумму элементов массива, расположенных между первым и последним нулевыми элементами;
3. Сумму элементов массива, расположенных до последнего положительного элемента;
4. Сумму элементов массива, расположенных между первым и последним положительными элементами;
5. Произведение элементов массива, расположенных между первым и вторым нулевыми элементами;
6. Сумму элементов массива, расположенных между первым и вторым отрицательными элементами;
7. Сумму элементов массива, расположенных до минимального элемента;
8. Сумму модулей элементов массива, расположенных после последнего отрицательного элемента;
9. Сумму элементов массива, расположенных после последнего элемента, равного нулю;
10. Сумму модулей элементов массива, расположенных после минимального по модулю элемента;

11. Сумму элементов массива, расположенных после минимального элемента;

12. Сумму элементов массива, расположенных после первого положительного элемента;

13. Сумму модулей элементов массива, расположенных после первого отрицательного элемента;

14. Сумму модулей элементов массива, расположенных после первого элемента, равного нулю;

15. Сумму положительных элементов массива, расположенных до максимального элемента;

16. Произведение элементов массива, расположенных между первым и последним отрицательными элементами.

### 7.3. Особенности применения указателей

В работе №3 рассмотренны указатели. Обращение к объектам любого типа в языке С может проводиться по имени, как мы до сих пор делали, и по ***указателю*** (косвенная адресация).

Указатель – это переменная, которая может содержать адрес некоторого объекта в памяти компьютера, например, адрес другой переменной. Через указатель, установленный на переменную, можно обращаться к участку оперативной памяти (ОП), отведенной компилятором под ее значение.

Указатель объявляется следующим образом:

```
тип * ID указателя;
```

Перед использованием указатель должен быть инициирован либо конкретным адресом, либо значением *NULL* (0) – отсутствие указателя.

С указателями связаны две унарные операции: & и *. Операция & означает «взять адрес», а операция разадресации * – «значение, расположенное по адресу», например:

```
int x, *y;   // x – переменная типа int , y – указатель типа int
y = &x;      // y – адрес переменной x
*y = 1;      // по адресу y записать 1, в результате x = 1
```

При работе с указателями можно использовать операции сложения, вычитания и сравнения, причем выполняются они в единицах того типа, на который установлен указатель.

Операции сложения, вычитания и сравнения (больше/меньше) имеют смысл только для последовательно расположенных данных – массивов. Операции сравнения «==» и «!=» имеют смысл для любых указателей, т.е.

если два указателя равны между собой, то они указывают на одну и ту же переменную.

**Связь указателей с массивами.** Указатели и массивы тесно связаны между собой. Идентификатор массива является указателем на его первый элемент, т.е. для массива int ***a***[10], выражения ***a*** и ***a***[0] имеют одинаковые значения, т.к. адрес первого (с индексом 0) элемента массива – это адрес начала размещения его элементов в ОП.

Пусть объявлены – массив из 10 элементов и указатель типа *double*:

```
double a[10],  *p;
```

если `p = a;` (установить указатель *p* на начало массива *a*), то следующие обращения: ***a***`[i], *(a+i)` *и* `*(p+i)` эквивалентны, т.е. для любых указателей можно использовать две эквивалентные формы доступа к элементам массива: *a*[i] и *(*a*+i). Очевидна эквивалентность следующих выражений:

`&a[0]` ↔ `&(*p)` ↔ `p`

**Декларация многомерного массива:** *тип ID*`[`*размер* `1][`*размер* `2]…[`*размер* `N];` причем быстрее изменяется последний индекс, т.к. многомерные массивы размещаются в ОП в последовательности столбцов, например, массив целого типа, состоящий из двух строк и трех столбцов (с инициализацией начальных значений)

```
int a[2][3] = {{0,1,2},{3,4,5}};
```

в ОП будет размещен следующим образом:

`a[0][0]=0, a[0][1]=1, a[0][2]=2, a[1][0]=3, a[1][1]=4, a[1][2]=5.`

Если в списке инициализаторов данных не хватает, то соответствующему элементу присваивается значение 0.

**Указатели на указатели.** Связь указателей и массивов с одним измерением справедливо и для массивов с большим числом измерений.

Если рассматривать предыдущий массив (`int a[2][3];`) как массив двух массивов размерностью по три элемента каждый, то обращение к элементу `a[i][j]` соответствует эквивалентное выражение `*(*(a+i)+j)`, а объявление этого массива с использованием указателей будет иметь вид

```
int **a;
```

Таким образом, имя двухмерного массива – *ID* указателя на указатель.

**Динамическое размещение данных.** Для создания массивов с переменной размерностью используется динамическое размещение данных, декларируемых указателями.

Для работы с динамической памятью используются стандартные функции библиотеки *alloc.h*:

*void* ***malloc**(size)* и *void* ***calloc**(n, size)* – выделяют блок памяти размером ***size*** и ***n*×*size*** байт соответственно; возвращают указатель на выделенную область, при ошибке – значение ***NULL***;

`void` ***free***`(tt);` – освобождает ранее выделенную память с адресом `tt`.

Другим, более предпочтительным подходом к динамическому распределению памяти является использование операций языка C++ *new* и *delete*.

Операция ***new*** возвращает адрес ОП, отведенной под динамически размещенный объект, при ошибке – *NULL*, а операция ***delete*** освобождает память.

*Минимальный набор действий, необходимых для* динамического размещения одномерного массива *действительных чисел размером* n*:*

```
double *a;
     ...
a = new double[n]; // Захват памяти для n элементов
                              ...
delete []a;          // Освобождение памяти
```

*Минимальный набор действий, необходимых для* динамического размещения двухмерного массива *действительных чисел размером* n×m*:*

```
int i, n, m;              // n, m – размеры массива
double **a;
a = new double *[n];      // Захват памяти под указатели
for(i=0; i<n; i++)
     a[i] = new double [m];   // и под элементы
                ...
for(i=0; i<n; i++) delete []a[i];  // Освобождение памяти
delete []a;
```

Для современных компиляторов (версий старше «6») для освобождения памяти достаточно записать только *delete []a*;

**Пример выполнения задания.** Рассчитать значения вектора $\vec{Y} = A \cdot \vec{B}$, где $A$ – квадратная матрица размером $N \times N$, а $Y$ и $B$ – одномерные массивы размером $N$. Элементы вектора $Y$ определяются по формуле

$$Y_i = \sum_{j=0}^{N-1} A_{ij} \cdot B_j .$$

Рассмотрим пример создания интерфейса диалого оконного

приложения. Значение *N* вводить из *Edit*, *A* и *B* – из компонент *StringGrid*. Результат вывести в компоненту *StringGrid*.

Панель диалога и результаты выполнения программы приведена на рис. 7.10.

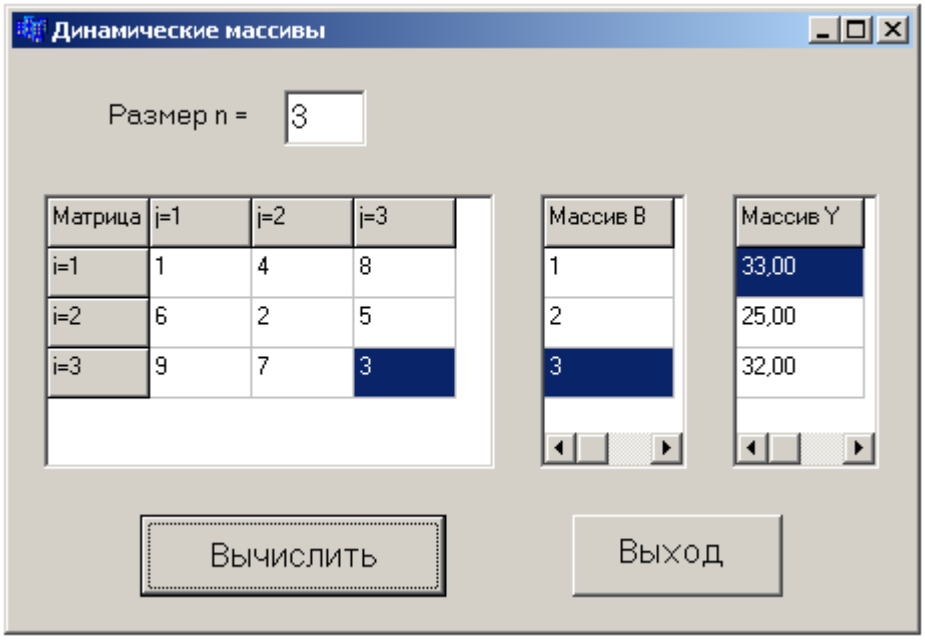



Рис. 7.10. Интерфейс диалового окна для приложения.

**Настройка компонент StringGrid интерфейса диалого окна приложения.** Для компоненты *StringGrid*1 значения *ColCount* и *RowCount* установите равными, например, 3 – три столбца и три строки, а *FixedCols* и *FixedRows* – 1.

Так как компоненты *StringGrid*2 и *StringGrid*3 имеют только один столбец, то у них *ColCount* = 1, *RowCount* = 3, а *FixedCols* = 0 и *FixedRows* = 1.

В свойстве ***Options*** строку ***goEditing*** для компонент *StringGrid*1 и *StringGrid*2 установите в положение ***true***.

Для изменения размера *n* используется функция-обработчик *EditChange*, полученная двойным щелчком по компоненте *Edit*.

Текст программы может иметь следующий вид:

```
                ...
//---------------- Глобальные переменные  -------------------
        int n = 3;
        double **a, *b;              // Декларации указателей
//---------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
        Edit1->Text=IntToStr(n);
        StringGrid1->ColCount = n+1;
        StringGrid1->RowCount = n+1;
```

```
        StringGrid2->RowCount = n+1;
        StringGrid3->RowCount = n+1;
// Ввод в левую верхнюю ячейку таблицы названия массивов
        StringGrid1->Cells[0][0] = "Матрица A";
        StringGrid2->Cells[0][0] = "Массив B";
        StringGrid3->Cells[0][0] = "Массив Y";
        for(int i=1; i<=n;i++){
         StringGrid1->Cells[0][i]="i="+IntToStr(i);
         StringGrid1->Cells[i][0]="j="+IntToStr(i);
        }
}
//---------------------------------------------------------
void __fastcall TForm1::Edit1Change(TObject *Sender)
{
        int i;
        n=StrToInt(Edit1->Text);
        StringGrid1->ColCount = n+1;        StringGrid1-
>RowCount = n+1;
        StringGrid2->RowCount = n+1;       StringGrid3-
>RowCount = n+1;
        for(i=1; i<=n;i++){
                StringGrid1->Cells[0][i]="i="+IntToStr(i);
                StringGrid1->Cells[i][0]="j="+IntToStr(i);
        }
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        double s;
        int i,j;
        a = new double*[n];          // Захват памяти под
указатели
        for(i=0; i<n;i++)  a[i] = new double[n];   // Захват
памяти
                                                   //под
элементы
        b = new double[n];
/* Заполнение массивов A и B элементами из таблиц StringGrid1
и StringGrid2 */
  for(i=0; i<n;i++) {
   for(j=0; j<n;j++)
       a[i][j]=StrToFloat(StringGrid1->Cells[j+1][i+1]);
     b[i]=StrToFloat(StringGrid2->Cells[0][i+1]);
  }
/* Умножение строки матрицы A на вектор B и вывод результата s
в StringGrid3 */
  for(i=0; i<n;i++){
            for(s=0, j=0; j<n;j++)  s += a[i][j]*b[j];
      StringGrid3->Cells[0][i+1] = FloatToStrF(s,
ffFixed,8,2);
  }
}
//---------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
```

```
      delete []a;
      delete []b;
      ShowMessage("Память освобождена!");
      Close();
}
```

### Пример создания консольного приложения

Текст программы может иметь следующий вид:

```
 void main()
   {
     double **a, *b, s;
     int i, j, n;
 printf(" Input size N : ");     scanf("%d",&n);
a = new double*[n];             // Захват памяти под указатели
for(i=0; i<n;i++)
      a[i] = new double[n];     // Захват памяти под элементы
b = new double[n];
 puts("\n Input Massiv A:");
for(i=0; i<n;i++)
           for(j=0; j<n;j++)     scanf("%lf", &a[i][j]);
puts("\n Input Massiv B:");
for( i=0; i<n;i++)        scanf("%lf", &b[i]);
puts("\n Massiv Y:");
for(i=0; i<n;i++){
          for(s=0, j=0; j<n;j++)  s+=a[i][j]*b[j];
           printf("  %8.2lf ", s);
}
delete []a;
delete []b;
puts("\n Delete !");
puts("\n Press any key ... ");
getch();
}
```

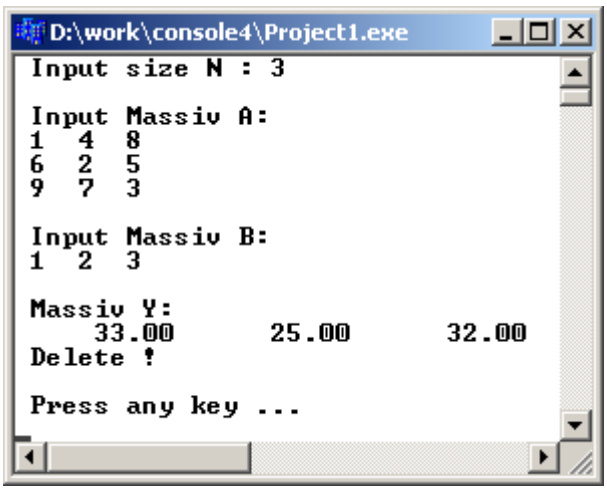



Рис. 7.11. Результаты консольного приложения

При вводе значений элементов массивов в одной строке через пробелы должен получиться следующий результат:

### 7. 4. Индивидуальные задания

Написать программу по обработке динамических массивов. Размеры массивов вводить с клавиатуры. При создании оконного приложения скалярный (простой) результат выводить в виде компоненты *Label*, а массивы вводить и выводить с помощью компонент *StringGrid*, в которых 0-й столбец и 0-ю строку использовать для отображения индексов массивов.

1. Из матрицы размером $N \times M$ получить вектор $B$, присвоив его $k$-му элементу значение 0, если все элементы $k$-го столбца матрицы нулевые, иначе 1.

2. Из матрицы размером $N \times M$ получить вектор $B$, присвоив его $k$-му элементу значение 1, если элементы $k$-й строки матрицы упорядочены по убыванию, иначе 0.

3. Из матрицы размером $N \times M$ получить вектор $B$, присвоив его $k$-му элементу значение 1, если $k$-я строка матрицы симметрична, иначе значение 0.

4. Задана матрица размером $N \times M$. Определить количество «особых» элементов матрицы, считая элемент «особым», если он больше суммы остальных элементов своего столбца.

5. Задана матрица размером $N \times M$. Определить количество элементов матрицы, у которых слева находится элемент больше его, а справа – меньше.

6. Задана матрица размером $N \times M$. Определить количество различных значений матрицы, т.е. повторяющиеся элементы считать один раз.

7. В матрице размером $N \times M$ упорядочить строки по возрастанию их первых элементов.

8. В матрице размером $N \times M$ упорядочить строки по возрастанию суммы их элементов.

9. В матрице размером $N \times M$ упорядочить строки по возрастанию их наибольших элементов.

10. Определить, является ли квадратная матрица симметричной относительно побочной диагонали.

11. Задана матрица размером $N \times M$. Определить количество элементов матрицы, у которых слева находится элемент меньше его, а справа – больше.

12. В квадратной матрице найти произведение элементов, лежащих выше побочной диагонали.

13. В квадратной матрице найти максимальный среди элементов, лежащих ниже побочной диагонали.

14. В матрице размером $N \times M$ поменять местами строку, содержащую элемент с наибольшим значением со строкой, содержащей элемент с наименьшим значением.

15. Из матрицы размером $n$ получить матрицу размером $n$-1 путем удаления строки и столбца, на пересечении которых расположен элемент с наибольшим по модулю значением.

16. В матрице размером $n$ найти сумму элементов, лежащих ниже главной диагонали, и произведение элементов, лежащих выше главной диагонали.

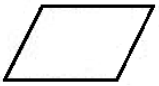
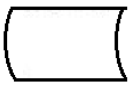
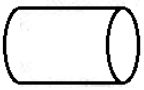
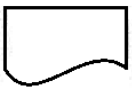
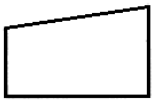
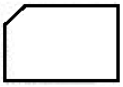
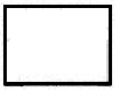
## 7.5. Контрольные вопросы

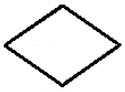
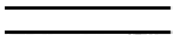
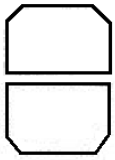
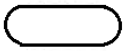
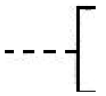
1. Назначения диаграммы взаимодействия (interaction diagram)
2. Назначения диаграмм коммуникации и последовательности.
3. Назначения диаграмма последовательности.
4. Назначения диаграмма обзора взаимодействия.
5. Назначения диаграмма синхронизации.
6. Назначения диаграмма профилей.
7. Настройка компоненты StringGrid.
8. Декларация многомерного массива.
9. Указатели на указатели.
10. Динамическое размещение данных.

## Литература


1. Эрик С. Реймонд Искусство программирования для Unix М: ЗАО «Издательство БИНОМ», 2005.
2. Бруно Бабе. Просто и ясно о Borland C++: Пер. с англ. - М.: БИНОМ, 1994.
3. Н. Б. Культин C++ Builder в задачах и примерах. – СПб.: БХВ-Петербург, 2005. - 336с.: ил.
4. Б. Керниган, Д. Ритчи Язык программирования Си
5. Б. Страуструп. Язык программирования C++, спец. изд. /Пер. с англ. М: «Издательство БИНОМ» - «Невский Диалект», 2002.
6. Архангельский А.Я Компоненты и классы М: ЗАО «Издательство БИНОМ», 2007.
7. Архангельский А.Я Язык C++ в C++ Builder М: ООО «Издательство БИНОМ», 2008.

# ПРИЛОЖЕНИЕ 1

| Символ | Наименование символа | Схема данных | Схема программы | Схема работы системы | Схема взаимодействия программ | Схема ресурсов системы |
|---|---|---|---|---|---|---|
| **Символы данных**<br>*Основные* | | | | | | |
|  | Данные | + | + | + | + | + |
|  | Запоминаемые данные | + | + | + | + | + |
| *Специфические* | | | | | | |
|  | Оперативное запоминающее устройство | + | - | + | + | + |
|  | Запоминающее устройство с последовательной выборкой | + | - | + | + | + |
|  | Запоминающее устройство с прямым доступом | + | - | + | + | + |
|  | Документ | + | - | + | + | + |
|  | Ручной ввод | + | - | + | + | + |
|  | Карта | + | - | + | + | + |
|  | Бумажная лента | + | - | + | + | + |
|  | Дисплей | + | - | + | + | + |
| **Символы процесса**<br>*Основные* | | | | | | |
|  | Процесс | + | + | + | + | + |
| *Специфические* | | | | | | |
|  | Предопределенный процесс | - | + | + | + | - |

| Символ | Наименование символа | Схема данных | Схема программы | Схема работы системы | Схема взаимодействия программ | Схема ресурсов системы |
|---|---|---|---|---|---|---|
|  | Ручная операция | + | - | + | + | - |
|  | Подготовка | + | + | + | + | - |
|  | Решение | - | + | + | - | - |
|  | Параллельные действия | - | + | + | + | - |
|  | Граница цикла | - | + | + | - | - |
| **Символы линий** *Основные* | | | | | | |
|  | Линия | + | + | + | + | + |
| *Специфические* | | | | | | |
|  | Передача управления | - | - | - | + | - |
|  | Канал связи | + | - | + | + | + |
|  | Пунктирная линия | + | + | + | + | + |
| **Специальные символы** | | | | | | |
|  | Соединитель | + | + | + | + | + |
|  | Терминатор | + | + | + | - | - |
|  | Комментарий | + | + | + | + | + |
|  | Пропуск | + | + | + | + | + |
| Примечание. Знак «+» указывает, что символ используют в данной схеме, знак «-» - не используют. | | | | | | |

# ПРИЛОЖЕНИЕ 2

## Грамматика С

Ниже приведены грамматические правила, которые мы уже рассматривали. Они имеют то же содержание, но даны в ином порядке.

Здесь не приводятся определения следующих символов-терминов: целая-константа, символьная-константа, константа-с-плавающей-точкой, идентификатор, строка и константа-перечисление. Слова, набранные полужирным латинским шрифтом, являются ключевыми.

```
единица-трансляции:
    внешнее-объявление
    единица-трансляции внешнее-объявление
внешнее-объявление:
    определение-функции
    объявление
определение функции:
    спецификаторы-объявления_необ объявитель
    список-объявлений_необ составная-инструкция
объявление:
    спецификаторы-объявления список-инициализаторов-
объявителей_необ
список-объявлений:
    объявление
    список-объявлений объявление
спецификаторы-объявления:
    спецификатор-класса-памяти спецификаторы-объявления_необ
    спецификатор-типа спецификаторы-объявления_необ
    квалификатор-типа спецификаторы-объявления_необ
спецификатор-класса-памяти: один из
    auto    register    static    extern    typedef
спецификатор-типа: один из
    void char short int long float double signed unsigned
    спецификатор-структуры-или-объединения
    спецификатор-перечисления
    typedef-имя

квалификатор-типа: один из
    const   volatile
спецификатор-структуры-или-объединения:
    структуры-или-объединения-идентификатор_необ { список-
объявлений-структуры }
    структуры-или-объединения идентификатор
структура-или-объединение: одно из
    struct   union
список-объявлений-структуры:
```

```
    объявление-структуры
    список-объявлений-структуры объявление-структуры
список-объявителей-иииициализаторов:
    объявитель-инициализатор
    список-объявителей-инициализаторов, объявитель-
инициализатор
объявитель-инициализатор:
    объявитель
    объявитель = инициализатор
объявление-структуры:
    список-спецификаторов-квалификаторов список-объявителей-
структуры
список-спецификаторов-квалификаторов:
    спецификатор-типа список-спецификаторов-квалификаторовнеоб
    квалификатор-типа список-спецификаторов-квалификаторовнеоб
список-структуры-объявителей:
    структуры–объявитель
    список-структуры-объявителей, структуры-объявитель
структуры-объявитель:
    объявитель
    объявительнеоб : константное-выражение
спецификатор-перечисления:
    enum идентификаторнеоб { список-перечислителей }
    enum идентификатор
список-перечислителей:
    перечислитель
    список-перечислителей перечислитель

перечислитель:
    идентификатор
    указательнеоб собственно-объявитель
собственно-объявитель:
    идентификатор
    ( объявитель )
    собственно-объявитель [ константное-выражениенеоб ]
    собственно-объявитель ( список-типов-параметров )
    собственно-объявитель ( список-идентификаторовнеоб )

указатель:
    * список-квалификаторов-типанеоб
    * список-квалификаторов-типанеоб указатель
список-квалификаторов-типа:
    квалификатор-типа
    список-квалификаторов-типа квалификатор-типа
список-типов-параметров:
    список-параметров
```

```
    список-параметров, ...
список-параметров:
    объявление-параметра
    список-параметров, объявление-параметра
объявление-параметра:
    спецификаторы-объявления объявитель
    спецификаторы-объявления абстрактный-объявительнеоб
список-идентификаторов:
    идентификатор
    список-идентификаторов, идентификатор
инициализатор:
    выражение-присваивания
    { список-инициализаторов }
    { список-инициализаторов, }
список-инициализаторов:
    инициализатор
    список-инициализаторов, инициализатор
имя-типа:
    список-спецификаторое-квалификаторов абстрактный-
объявительнеоб
абстрактный-объявитель:
    указатель
    указательнеоб собственно-абстрактный-объявитель
собственно-абстрактный-объявитель:
    ( абстрактный-объявитель )
    собственно-абстрактный-объявительнеоб [константное-
выражениенеоб]
    собственно-абстрактный-объявительнеоб (список-типов-
параметровнеоб)
typedef-имя:
    идентификатор
инструкция:
    помеченная-инструкция
    инструкция–выражение
    составная-инструкция
    инструкция-выбора
    циклическая-инструкция
    инструкция-перехода
помеченная-инструкция:
    идентификатор: инструкция
    case константное-выражение: инструкция
    default: инструкция
инструкция-выражение:
    выражениенеоб;
составная-инструкция:
    ( список-объявленийнеоб список-инструкцийнеоб)
```

```
список-инструкций:
    инструкция
    список-инструкций инструкция
инструкция-выбора:
    if ( выражение ) инструкция
    if ( выражение ) инструкция else инструкция
    switch ( выражение ) инструкция
циклическая-инструкция:
    while ( выражение ) инструкция
    do инструкция while ( выражение )
    return выражение_необ;
выражение:
    выражение-присваивания
    выражение, выражение-присваивания
выражение-присваивания:
    условное-выражение
    унарное-выражение оператор-присваивания выражение-
присваивания
оператор-присваивания: один из
    =   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=
условное-выражение:
    логическое-ИЛИ-выражение
    логическое-ИЛИ-выражение ? выражение : условное-выражение
константное-выражение:
    условное-выражение
логическое-ИЛИ-выражение:
    логическое-И-выражение
    логическое-ИЛИ-выражение || логическое-И-выражение
логическое-И-выражение:
    ИЛИ-выражение
    логическое-И-выражение && ИЛИ-выражение
ИЛИ-выражение:
    исключающее-ИЛИ-выражение
    ИЛИ-выражение | исключающее-ИЛИ-выражение
исключающее-ИЛИ-выражение:
    И-выражение
    исключающее-ИЛИ-выражение ^ И-выражение
И-выражение:
    выражение-равенства
    И-выражение & выражение-равенства
выражение-равенства:
    выражение-отношения
    выражение-равенства == выражение-отношения
    выражение-равенства != выражение-отношения
выражение-отношения:
    сдвиговое-выражение
```

```
    выражение-отношения < сдвиговое-выражение
    выражение-отношения > сдвиговое-выражение
    выражение-отношения <= сдвиговое-выражение
    выражение-отношения >= сдвиговое-выражение
сдвиговое-выражение:
    аддитивное-выражение
    сдвиговое-выражение >> аддитивное-выражение
    сдвиговое-выражение << аддитивное-выражение
аддитивное-выражение:
    мультипликативное-выражение
    аддитивное-выражение + мультипликативное-выражение
    аддитивное-выражение - мультипликативное-выражение
мультипликативное-выражение:
    выражение-приведенное-к-типу
    мультипликативное-выражение * выражение-приведенное-к-типу
    мультипликативное-выражение / выражение-приведенное-к-типу
    мультипликативное-выражение % выражение-приведенное-к-типу
выражение-приведенное-к-типу:
    унарное-выражение
    ( имя-типа ) выражение-приведенное-к-типу
унарное-выражение:
    постфиксное –выражение
    ++ унарное-выражение
    -- унарное-выражение
    унарный-оператор выражение-приведенное-к-типу
    sizeof унарное-выражение
    sizeof( имя-типа )
унарный-оператор: один из
    &   *   +   -   ~   !
постфиксное-выражение:
    первичное-выражение
    постфиксное-выражение[ выражение ]
    постфиксное-выражение( список-аргументов-выражений_необ )
    постфиксное-выражение, идентификатор
    постфиксное-выражение-> идентификатор
    постфиксное-выражение++
    постфиксное-выражение–
первичное -выражение:
    идентификатор
    константа
    строка
    ( выражение )
список-аргументов-выражений:
    выражение-присваивания
    список-аргументов-выражений , выражение-присваивания
константа:
```

целая-константа
символьная-константа
константа-с-плавающей-точкой
константа-перечисление

Ниже приводится грамматика языка препроцессора в виде перечня структур управляющих строк. Для механического получения программы грамматического разбора она не годится. Грамматика включает символ текст, который означает текст обычной программы, безусловные управляющие строки препроцессора и его законченные условные конструкции.

управляющая-строка:

***#define*** *идентификатор последовательность-лексем*
***#define*** *идентификатор ( идентификатор, ..., идентификатор) последовательность-лексем*
***#undef*** *идентификатор*
***#include*** *<имя-файла>*
***#include*** *"имя-файла"*
***#include*** *последовательность-лексем*
***#line*** *константа "идентификатор"*
***#line*** *константа*
***#error*** *последовательность-лексем*$_{необ}$
***#pragma*** *последовательность-лексем*$_{необ}$
***#***

условная-конструкция-препроцессора

условная-конструкция-препроцессора:

***if-****строка текст* ***elif****-части* ***else****-часть*$_{необ}$ ***#endif***

***if****-строка:*
***#if*** *константное-выражение*
***#ifdef*** *идентификатор*
***#ifndef*** *идентификатор*

***elif****-части:*
***elif****-строка текст*
***elif****-части*$_{необ}$

***elif****-строка:*
*#elif константное-выражение*

***else-часть:***
*else-строка текст*

***else****-строка:*
***#else***

# ПРИЛОЖЕНИЕ 3

**Примечание.** *Полный перечень всех компонент их свойств, методов и другие виды атрибутики приводятся в Help C++ Builder, здесь приводим используемые нами компоненты и некоторые свойства.*

**Label (вкладка Standart класс TLabel) .** Компонент предназначен для вывода текста на поверхность формы рис. П3.1. Свойства компонента, некоторые из которых представлены в табл. П3.1, определяют вид и расположение текста.


Рис. П3.1. Компонент Label

Свойства компонента Label

Таблица П3.1

| Свойство | Описание |
|---|---|
| **Name** | Имя компонента. Используется в программе для доступа к компоненту и его свойствам. |
| **Caption** | Отображаемый текст. |
| **Left** | Расстояние от левой границы поля вывода до левой границы формы. |
| **Top** | Расстояние от верхней границы поля вывода до верхней границы формы. |
| **Height** | Высота поля вывода. |
| **Width** | Ширина поля вывода. |
| **AutoSize** | Признак того, что размер поля определяется его содержимым. |
| **WordWrap** | Признак того, что слова, которые не помещаются в текущей строку автоматически переносятся на следующую строку (значение свойства **AutoSize** должно быть **false**). |
| **Alignment** | Задает способ выравнивания текста внутри поля. Текст может быть выровнен по левому краю (taLeftJustify), по центру ( taCenter) или по правому краю (taRightJustify). |
| **Font** | Шрифт, используемый для отображения текста. |
| **ParentFont** | Признак наследования компонентом характеристик шрифта формы, на которой находится компонент. Если значение равно true, то текст выводится шрифтом, установленным для формы. |
| **Color** | Цвет фона области вывода текста. |
| **Transparent** | Управляет отображением фона области вывода текста. Значение **true** делает область вывода текста прозрачной (область вывода не закрашивается цветом, заданным свойством Color). |
| Visible | Позволяет скрыть текст (false) или сделать его видимым (true). |

**Button (вкладка Standart, класс TButton).** Компонента представляет собой командную кнопку (рис. П3.2). Свойства компонента приведены в табл. П3.2.


Рис. П3.2. Компонент

Свойства компонента Button

Таблица П3.2

| Свойство | Описание |
|---|---|
| **Name** | Имя компонента, обычно используется в программе для доступа к компоненте, к его свойствам |
| **Caption** | Надпись на кнопке - заголовок (AnsiString), если перед некоторым символом в заголовке поставить амперсанд - &, символ подчеркивается, становится горячей клавишей. |
| **Left** | Расстояние от левой границы кнопки до левой границы формы. |
| **Top** | Расстояние от верхней границы кнопки до верхней границы формы. |
| **Height** | Высота кнопки. |
| **Width** | Ширина кнопки. |
| **Visible** | Позволяет скрыть кнопку (false) или сделать ее видимой (true). |
| **Hint** | Подсказка – текст, который появляется рядом с указателем мыши при позиционировании указателя на командной кнопке (для того, чтобы текст появился, надо, чтобы значение свойства ShowHint было true). |
| **ShowHint** | Разрешает (true) или запрещает (false) отображение подсказки при позиционировании указателя на кнопке. |
| **Font** | Позволяет установить шрифт, размеры, и другие атрибуты шрифта |
| **TabOrder** | Указывает позицию компоненты в списке, определяет порядок переключения фокуса между компонентами |
| **TabStop** | Определяет возможность доступа пользователя к кнопке с помощью клавиши Tab. |
| **Enabled** | Признак доступности кнопки. Если значение свойства равно **true**, то кнопка доступна. Если значение свойства равно **false**, то кнопка недоступна – например, в результате щелчка на кнопке, событие **Click** не возникает |

**Edit (вкладка Standart класс TEdit) .** Компонент представляет собой поле ввода-редактирования строки символов (рис. П3.3). Свойства компонента представлены в табл. П3.3.


Рис. П3.3. Компонент Edit

**Свойства компонента Edit**

Таблица П3.3

| Свойство | Описание |
|---|---|
| **Name** | Имя компонента, обычно используется в программе для доступа к компоненте, к его свойствам |
| Text | Текст, находящийся в поле ввода и редактирования. |

| | |
|---|---|
| Left | Расстояние от левой границы компонента до левой границы формы. |
| Top | Расстояние от верхней границы компонента до верхней границы формы. |
| Height | Высота компонента. |
| Width | Ширина компонента. |
| Font | Шрифт, используемый для отображения текста. |
| ParentFont | Признак наследования компонентом характеристик шрифт а формы, на которой находится компонент. Если значение равно true, то текст выводится шрифтом, установленным дл я формы. |
| Color | Цвет фона области вывода текста. |
| Visible | Позволяет скрыть текст (false) или сделать его видимым (true). |

**CGauge (вкладка Samples класс TCGauge).** Компонент предназначен для отображения хода процессов в виде линейки, текста или секторной диаграммы (рис. П3.4). Свойства компонента представлены в табл. П3.4.


Рис. П3.4. Компонент CGauge

Свойства компонента CGauge

Таблица П3.4

| Свойство | Описание |
|---|---|
| **Name** | Имя компонента, обычно используется в программе для доступа к компоненте, к его свойствам |
| **MaxValue** | Максимальное значение позиции, которое соответствует завершению отображаемого процесса. |
| **MinValue** | Начальное значение позиции, которое соответствует началу отображаемого процесса. |
| **Progress** | Позиция, которую можно задавать по мере протекания процесса, начиная со значения MinValue в начале процесса и кончая значением MaxValue в конце. Если минимальное и максимальное значения выражены в процентах, то позиция – это процент завершенной части процесса. |
| **ForeColor** | Цвет заполнения. |
| **ShowText** | Текстовое изображение процента выполнения на фоне диаграммы. |
| **Kind** | Тип диаграммы: gkHorisontalBar – горизонтальная полоса, gkVerticalBar – вертикальная полоса, gkPie – круговая диаграмма, grNeedle – секторная диаграмма, gkText – отображение текстом. |

**Chart (вкладка Additional класс TCGauge).** Компонент является панелью, на которой можно создавать диаграммы и графики (рис. П3.5). Свойства компонента представлены в табл. П3.5.


Рис. П3.5. Компонент Chart

Свойства компонента Chart

Таблица П3.5

| Свойство | Описание |
|---|---|
| **Name** | Имя компонента, обычно используется в программе для доступа к компоненте, к его свойствам |
| **AllowPanning** | Определяет возможность пользователя прокручивать наблюдаемую часть графика во время выполнения, нажимая правую кнопку мыши. Возможные значения: pmNone – прокрутка запрещена, pmHorizontal, pmVertical или pmBoth – разрешена соответственно прокрутка только в горизонтальном направлении, только в вертикальном или в обоих направлениях. |
| **AllowZoom** | Позволяет пользователю изменять во время выполнения масштаб изображения, вырезая фрагменты диаграмм или графика курсором мыши. |
| **Title** | Заголовок диаграммы. |
| **Foot** | Подпись под диаграммой. По умолчанию отсутствует. Текст подписи определяется под свойством Text. |
| **Frame** | Определяет рамку вокруг диаграммы. |
| **Legend** | Легенда диаграммы – список обозначений. |
| **View3d** | Разрешает или запрещает трехмерное отображение диаграммы. |
| **Visible** | Позволяет скрыть компонент (false) или сделать его видимым (true). |

**ComboBox (вкладка Standart класс TComboBox).** Компонент дает возможность ввести данные выбором из списка или путем набора на клавиатуре в поле редактирования (рис. П3.6). Свойства компонента представлены в табл. П3.6.

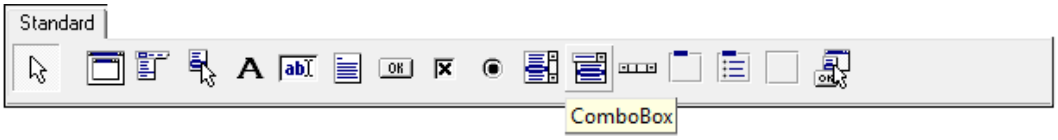



Рис. П3.6. Компонент ComboBox

Свойства компонента ComboBox

Таблица П3.6

| Свойство | Описание |
|---|---|
| **Name** | Имя компонента, обычно используется в программе для доступа к компоненте, к его свойствам |
| Text | Текст, находящийся в поле ввода/редактирования. |
| Items | Элементы списка – массива строк. |
| Count | Количество элементов списка. |
| ItemIndex | Номер элемента, выбранного в списке. Если ни один из элементов не был выбран, то значение свойства равно –1 (минус один). |
| Sorted | Признак необходимости автоматической сортировки (true) списка после добавления очередного элемента. |
| DropDownCount | Количество отображаемых элементов в раскрытом списке. Если количество элементов списка больше, чем DropDownCount, то появляется вертикальная полоса прокрутки. |

| | |
|---|---|
| Font | Шрифт, используемый для отображения элементов списка. |

**GroupBox (вкладка Standart класс TGroupBox).** Компонент представляет собой контейнер с рамкой и надписью, объединяющий группу управляющих компонентов (рис. П3.7). Свойства компонента представлены в табл. П3.7.

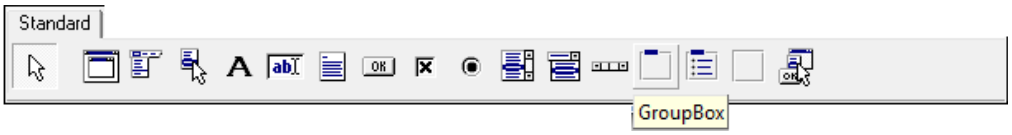



Рис. П3.7. Компонент GroupBox

Свойства компонента GroupBox

Таблица П3.7

| Свойство | Описание |
|---|---|
| **Name** | Имя компонента, обычно используется в программе для доступа к компоненте, к его свойствам |
| **Caption** | Заголовок контейнера. |
| **Color** | Цвет панели. |
| **Font** | Шрифт для отображения заголовка. «По умолчанию» используется компонентами находящимися в контейнере. При необходимости шрифт компонентов в контейнере можно изменить. |

**Image (вкладка Additional класс TImage) .** Компонент отображает графическое изображение и обеспечивает работу с ним (рис. П3.8). Свойства компонента представлены в табл. П3.8.

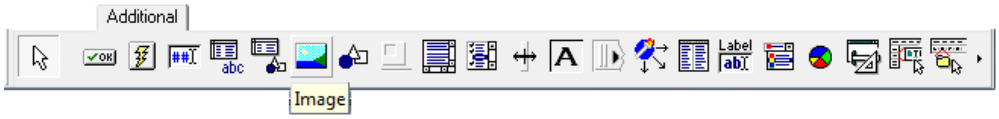



Рис. П3.8. Компонент Image

Свойства компонента Image

Таблица П3.8

| Свойство | Описание |
|---|---|
| **Name** | Имя компонента. Используется в программе для доступа к компоненту и его свойствам. |
| **AutoSize** | Указывает, изменяется ли автоматически размер компонента, подстраиваясь под размер изображения. |
| **Canvas** | Определяет поверхность (холст, канву) для рисования, для наложения друг на друга нескольких изображений. Доступно только, если в свойстве Picture хранится битовая матрица. |
| **Center** | Указывается, должно ли изображение центрироваться в поле компонента, если его размеры меньше размеров поля. При значении false изображение располагается в верхнем левом углу поля. Свойство не действует, если AutoSize установлено в true или Stretch установлено в true и Picture содержит не пиктограмму. |

| | |
|---|---|
| **Picture** | Определяет отображаемый графический объект. Может загружаться программно или во время проектирования с помощью Picture Editor. |
| **Stretch** | Указывает, должны ли изменяться размеры изображения, подгоняясь по размеры компонента. Учтите, что изменение размеров изображения приведет к его искажению, если соотношение сторон графического объект Image не одинако. |
| **Transparent** | Указывает, должен ли быть цвет фона изображения прозрачным, чтобы сквозь него было видно нижележащее изображение. |

**PageControl (вкладка Win32 класс TPageControl) .** Компонент представляет собой многостраничную панель, которая позволяет экономить пространство окна приложения, размещая на одном и том же месте страницы разного содержания (рис. П3.9). Свойства компонента представлены в табл. П3.9.


Рис. П3.9. Компонент PageControl

Свойства компонента PageControl

Таблица П3.9

| Свойство | Описание |
|---|---|
| Name | Имя компонента. Используется в программе для доступа к компоненту и его свойствам. |
| Style | Определяет стиль отображения компонента. |
| MultiLine | Определяет, будут ли закладки размещаться в несколько рядов, если все они не помещаются в один ряд. |
| TabPosition | Определяет место расположения ярлычков: tpBottom – внизу, tpLeft – слева, tpRight – справа и tpTop – вверху компонента. |
| TabHeigh<br>TabWidth | Высота и ширина ярлычков закладок в пикселах. |
| ActivePage | Имя активной страницы. |
| PageCount | Количество страниц. |

**PaintBox (вкладка System класс TPaintBox).** Компонент предназначен для рисования непосредственно на канве привозникновении события OnPaint (рис. П3.10). Свойства компонента представлены в табл. П3.10.


Рис. П3.10. Компонент PaintBox

Свойства компонента PaintBox

Таблица П3.10

| Свойство | Описание |
|---|---|

| | |
|---|---|
| Name | Имя компонента. Используется в программе для доступа к компоненту и его свойствам. |
| Align | Определяет способ выравнивания компонента внутри контейнера. |
| Canvas | Определяет поверхность (холст, канву) для рисования. |
| Color | Цвет закраски замкнутых фигур. |

**Panel (вкладка Standard класс TPanel) .** Компонент представляет собой контейнер для других компонентов (рис. П3.11). Может использоваться также для отображения текста. Свойства компонента представлены в табл. П3.11.


Рис. П3.11. Компонент Panel

Свойства компонента Panel

Таблица П3.11

| Свойство | Описание |
|---|---|
| **Name** | Имя компонента. Используется в программе для доступа к компоненту и его свойствам. |
| **Caption** | Отображаемый текст. |
| **AutoSize** | Признак того, что размер панели подстраивается под размещенные в ней компоненты. |
| **Alignment** | Задает способ выравнивания текста внутри поля. Текст может быть выровнен по левому краю (taLeftJustify), по центру (taCenter) или по правому краю (taRightJustify). |
| **BevelInner** | Определяет утопленный (bvLowered), выпуклый (bvRaised) или плоский вид (bvSpace) внутренней части компонента. |
| **BevelOuter** | Определяет выпуклый (bvLowered), утопленный (bvRaised) или плоский вид (bvSpace) обрамления компонента. Значение bvNone соответствует отсутствию обрамления. |
| **BevelWidth** | Определяет ширину обрамления компонента в пикселах. |
| **BorderStyle** | Указывает, ограничена ли клиентская область компонента одинарной бордюрной линией. |
| **BorderWidth** | Расстояние в пикселах между наружной и внутренней кромками обрамления. |
| **Color** | Цвет панели. |

## ПРИЛОЖЕНИЕ 4

**Методы класса tcanvas.** Любая картинка, чертеж или схема можно рассматривать, как совокупность графических примитивов: линий, точек, окружностей, дуг и других основополагающих фигур. Вычерчивание графических примитивов на поверхности (формы или компонентов Image, PaintBox) осуществляется применением соответствующих методов к свойству Canvas этой поверхности.

Методы рисования графических примитивов

Таблица П3.1.

| Метод | Описание |
|---|---|
| TextOut(x, y, s) | Выводит строку s от точки с координатами (x, y). |
| Draw(x, y, b) | Выводит от точки с координатами (x, y) битовый образ b. |
| LineTo(x, y) | Вычерчивает линию из текущей точки в точку с указанными координатами. Вид линии определяет свойство Pen. |
| MoveTo(x, y) | Перемещает указатель текущей точки в точку с указанными координатами. |
| PolyLine(pl) | Вычерчивает ломаную линию. Координаты точек перегиба задает параметр pl – массив структур типа TPoint. Если первый и последний элементы массива одинаковы, то будет вычерчен замкнутый контур. Вид линии определяет свойство Pen. |
| Polygon(pl) | Вычерчивает и закрашивает многоугольник. Координаты углов задает параметр pl – массив структур типа TPoint. Первый и последний элементы массива должны быть одинаковы. Вид границы определяет свойство Pen, цвет и стиль закраски внутренней области – свойство Brush. |
| Ellipse(x1, y1, x2, y2) | Вычерчивает эллипс, окружность или круг. Параметры x1, y1, x2, y2 задают размер прямоугольника, в который вписывается эллипс. Вид линии определяет свойство Pen. |
| Arc(x1, y1, x2, y2, x3, y3, x4, y4) | Вычерчивает дугу. Параметры xl, y1, x2, y2 определяют эллипс, из которого вырезается дуга, параметры x3, y3, x4 и y4 – координаты концов дуги. Дуга вычерчивается против часовой стрелки от точки (x3, y3) к точке (x4, y4). Вид линии (границы) определяет свойство Pen, цвет и способ закраски внутренней области – свойство Brush. |
| Rectangle(x1, y1, x2, y2) | Вычерчивает прямоугольник. Параметры xl, y1, x2 и y2 задают координаты левого верхнего и правого нижнего углов. Вид линии определяет свойство Pen, цвет и способ закраски внутренней области – свойство Brush. |
| RoundRec(xl, y1, x2, y2 x3, y3) | Вычерчивает прямоугольник со округленными углами параметры xl, y1, x2 и y2 задают координаты левого верхнего и правого нижнего углов, x3 и y3 – радиус округления. Вид линии определяет свойство Pen, цвет и способ закраски внутренней области – свойство Brush. |